Virtuelle Realitäten für die chirurgische Ausbildung: Strukturen, Algorithmen und ihre Anwendung

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

vorgelegt von
Clemens Wagner
aus Herrischried

Mannheim, 2003

Dekan:	Professor Dr. Jürgen Potthoff, Universität Mannheim
Referent:	Professor Dr. Reinhard Männer, Universität Mannheim
Korreferent:	Professor Dr.-Ing., Dr.-Ing. E.h. Wolfgang Straßer, Universität Tübingen

Tag der mündlichen Prüfung: 21. Oktober 2003

Für Alexander, Barbara und Donata Wagner

Vorwort

> Was kann ein Mensch erleben, der an den phantomatischen Generator angeschlossen ist? Alles.
>
> —Stanisław Lem, *Summa technologiae*

> Nothing can create the experience of factorizing the number 181.
>
> —David Deutsch, *The fabric of reality*

Ein Computer ist eine universelle Maschine. Mit diesem Satz habe ich vor einigen Monaten damit begonnen, die vorliegende Arbeit zu schreiben. Er hat es verdient, an dieser exponierten Stelle zu stehen – würde er nicht zutreffen, wäre die Informatik eine unvorstellbar langweilige Wissenschaft, die sich ausschließlich mit endlichen Automaten befassen müsste. Universalität dagegen ist die Grundlage ihrer beiden großen schöpferischen Träume: der Erschaffung künstlicher Intelligenz und des Baus virtueller Realitäten.

Um virtuelle Realität wird es im Folgenden gehen. Ich hoffe, dass der Augenoperationssimulator EyeSi und die Algorithmen der Softwarebibliothek VRM brauchbare Indizien dafür sind, dass man Träume verwirklichen kann.

Ich danke Herrn Prof. Dr. Reinhard Männer für die Betreuung meiner Dissertation und für die bemerkenswerte Freiheit, die er mir zugestanden hat. Besonders erwähnen möchte ich seine große Unterstützung bei meiner Robocup-Eskapade und die gute Zusammenarbeit bei der Organisation der PI2-Vorlesung.

Herrn Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer danke ich dafür, dass er sich als Korreferent zur Verfügung gestellt hat.

Prof. Dr. Dr. Hans-Joachim Bender war in allen Phasen des EyeSi-Projekts ein wertvoller Ratgeber.

Im November 2001 zeichnete die Stiftung Rhein-Neckar-Dreieck die Entwicklung des EyeSi-Systems mit ihrem Forschungs- und Innovationspreis aus. Diese große Anerkennung unserer Arbeit hat mich sehr gefreut und motiviert!

Einen erheblichen Anteil am Gelingen der Arbeit hatten die Teams von ViPA und der VRmagic GmbH. Ohne die visionäre und integrative Kraft von Dr. Markus Schill würden sie nicht existieren. Markus ist für mich mehr ein Freund als ein Kollege; der gemeinsame Blick, den wir für viele Probleme hatten und die produktiven Diskussionen zu den richtigen Zeitpunkten haben mir sehr geholfen.

Liu Bing, Johannes Grimm, Marc Hennen, Norbert Hinckers, Andreas Köpfle, Olaf Körner, Nikolaj Nock, Thomas Ruf, Michael Schneider und Stefan Sichler trugen jeder für sich und jeder auf seine eigene Weise zu der kreativen Atmosphäre bei, in der diese Arbeit entstand. Mit Johannes, Olaf und Andreas unternahm ich während der Entwicklung der VRM-Bibliothek häufige Reisen in die Feinheiten von C++. Andreas lieh mir während der Arbeit und hin und wieder auch am Wochenende sein Ohr, wenn ich über den seltsamen Geisteszustand beim Zusammenschreiben klagen musste.

Thomas hat mich mit seiner enormen Fachkompetenz immer dann gerettet, wenn ich in einem Hardware- oder Betriebssystemproblem gefangen war. Durch Stefans Diplomarbeit hat die Realitätsnähe der EyeSi-Visualisierungskomponente einen Quantensprung erfahren. Michael ist mit seiner kritischen Art und seinen neuen Ideen eine wertvolle Ergänzung für die EyeSi-Softwareentwicklung.

Unseren Sekretärinnen Christiane Glasbrenner und Andrea Seeger habe ich nicht nur die Bewältigung aller mir unverständlichen Verwaltungsvorgänge zu verdanken, sondern auch entspannende Gespräche über Themen jenseits der Computertechnik.

Einige Bücher sind es unbedingt wert, genannt zu werden. Douglas R. Hofstadters *Gödel, Escher, Bach* ist zu einem großen Teil für die Faszination verantwortlich, die die Informatik auf mich ausübt. Piergorgio Odifreddis *Classical Recursion Theory* liefert nicht nur auf anregende, anspruchsvolle Weise den mathematischen Hintergrund zu Hofstadters Gedanken, sondern besitzt überdies ein viel schöneres und traurigeres Vorwort, als man es in einem wissenschaftlichen Werk vermuten würde. Universalität und virtuelle Realität hat David Deutsch in *The fabric of reality* beschrieben. Er hat damit den Grundton des ersten Kapitels vorgegeben.

Nadja Nesselhauf hat zur gleichen Zeit wie ich mit dem Zusammenschreiben begonnen. Ohne unsere gegenseitigen, im Gleichtakt gebrochenen und

wiederaufgenommenen Ultimaten würde ich wahrscheinlich immer noch die Einführung überarbeiten.

Markus, Olaf, Eva Kleinknecht und Esther Krug haben die Arbeit korrekturgelesen. Ich verdanke ihnen viele wertvolle Anmerkungen.

Es ist sicherlich nicht möglich, an dieser Stelle alle wichtigen Menschen aufzuzählen, denen ich während meiner Dissertation begegnet bin. Einen möchte ich trotzdem erwähnen: Danke, Eva.

Dank gilt auch meiner Familie, bei der ich immer Rückhalt und Verständnis finde.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AABB	Axis-Aligned Bounding-Box
ACE	Adaptive Communication Environment
AGP	Accelerated Graphics Port
API	Application Programming Interface
CAD	Computer Aided Design
CAVE	CAVE Automatic Virtual Environment
CPU	Central Processing Unit
DMA	Direct Memory Access
EyeSi	Eye Surgery Simulation
FEM	Finite-Element Model
FPGA	Field Programmable Gate Array
GJK	Gilbert-Johnson-Keerthi
GLUT	Graphics Language Utility Toolkit
GUI	Graphical User Interface
HMD	Head-Mounted Display
I/O	Input/Output
ILM	Innere Laminarmembran
IZ	Impact Zone
k-DOP	k-Discrete Oriented Polytope
KI	Künstliche Intelligenz
LAN	Local Area Network
LOD	Level-Of-Detail
LP	Lineares Programm
MGE	Multigraphelement
NTSC	National Television System Committee
OLED	Organic Light-Emitting Diode
OpenGL	Open Graphics Library
OBB	Oriented Bounding-Box
OP	Operationssaal
PAL	Phase Alternation Line
PC	Personal Computer
PDA	Personal Digital Assistant
RAM	Random Access Memory

ROI	Region Of Interest
RT	Real Time
STL	Standard Template Library
USB	Universal Serial Bus
ViPA	Virtual Patient Analysis
VR	Virtuelle Realität
VRM	Virtuelle Realität in der Medizin
VRML	Virtual Reality Modeling Language
ZCOLL	Z-Buffer-based Collision Detection
ZCOLL/F	ZCOLL, frontflächenbeschränkt

1

Einführung

1.1 Fragestellung

Ziel der vorliegenden Arbeit war die Entwicklung der Software für den Simulator EyeSi (*Eye Surgery Simulation*), einer virtuellen Realität zum Training von Augenoperationen. Die Strukturen und Algorithmen wurden zu Modulen der Softwarebibliothek VRM (*Virtual Reality in Medicine*) verallgemeinert, mit der Operationssimulationen für Ausbildung und Planung aufgebaut werden können. In dieser Allgemeinheit werden sie im Folgenden beschrieben.

Zu den behandelten Themen gehören die Softwarearchitektur einer VRM-Applikation, ein Datenformat für deformierbares Gewebe sowie Verfahren für Kollisionserkennung, Gewebesimulation und Visualisierung.

Es gibt eine Randbedingung, die den vorgestellten Ansätzen gemeinsam ist und ihren Charakter entscheidend beeinflusst: alle Berechnungen müssen so schnell ausgeführt werden, dass die Realitätsnähe der Simulation nicht durch wahrnehmbare Latenz oder Wiederholrate verringert wird.

1.2 Überblick über die Arbeit

Abschnitt 1.3 dieses Kapitels diskutiert einige Begriffe und Sachverhalte im Zusammenhang mit virtueller Realität.

Kapitel 2 beschreibt die VRM-Bibliothek und die Architektur einer darauf aufbauenden VRM-Applikation. Bei dem Architekturentwurf wird besonderer Wert auf kurze Entwicklungszyklen und eine geringe Latenz der Applikation gelegt. Es werden verschiedene Quellen für Latenz unterschieden

und Synchronisationsmuster bei der nebenläufigen Ausführung diskutiert. Zuletzt wird anhand einiger Messungen diskutiert, unter welchen Bedingungen das Zeitverhalten von Windows XP und Linux geeignet für VRM-Applikationen ist.

Kapitel 3 stellt eine graphbasierte Datenstruktur für die Modellierung der simulierten Welt vor, die die Ausdrucksmächtigkeit der üblicherweise verwendeten Szenegraphen übersteigt und besonders für deformierbare Objekte geeignet ist. Sie verbindet Flexibilität, Geschwindigkeit und Typsicherheit und ist die Grundlage der in den VRM-Applikationen eingesetzten Simulations- und Visualisierungsmodelle.

Im Laufe der Arbeit wurde das neue bildbasierte Verfahren ZCOLL zur schnellen Kollisionserkennung bei Gewebe-Instrument-Interaktionen entwickelt. Es baut auf bereits existierenden Verfahren auf, daher geht Kapitel 4 nach einer allgemeinen Darstellung von Algorithmen zur Kollisionserkennung besonders auf bildbasierte Verfahren ein. ZCOLL wird in zwei Varianten vorgestellt, die sich in der Strenge ihrer Randbedingungen und in ihrem Zeitverhalten unterscheiden.

Die folgenden beiden Kapitel 5 und 6 beschäftigen sich mit Gewebesimulation und Echtzeit-Visualisierung. Die Darstellung beschränkt sich dabei auf diejenigen Aspekte, die für EyeSi und die Entwicklung der entsprechenden VRM-Module relevant waren.

Das Kapitel 5 zur Gewebesimulation geht nach einer Klassifizierung deskriptiver und physikalischer Modellierung auf Eigenschaften und Erweiterungen von Feder-Masse-Modellen ein. Zu den wichtigsten Gesichtspunkten gehören die Beziehung zwischen numerischer Stabilität und Berechnungsaufwand, die Erweiterung des Stabilitätsbereichs auf weniger elastische Materialen sowie die lokale Gitterverfeinerung.

Im – recht kurzen – Visualisierungskapitel 6 werden vor allem Erweiterungen von 3D-Oberflächengrafik angesprochen, mit denen die Eigenschaften moderner PC-Grafikkarten zur Erhöhung von Visualisierungsqualität und -geschwindigkeit genutzt werden können – Vertex-Arrays, Vertex- und Pixel-Shader.

In Kapitel 7 wird die Hard- und Software des Augenoperationssimulators EyeSi beschrieben. Es wird gezeigt, dass die vorgestellten Verfahren zu einer überzeugenden virtuellen Realität integriert werden können.

Eine Zusammenfassung der Arbeit gibt Kapitel 8.

Abb. 1.1 stellt die Abhängigkeiten der einzelnen Kapitel grafisch dar. Wie im Titel der Arbeit wird zwischen Strukturen, Algorithmen und ihrer Anwendung unterschieden. Auf der MGE-Datenstruktur bauen die Algorithmen zu Kollisionserkennung, Gewebesimulation und Visualisierung auf; im

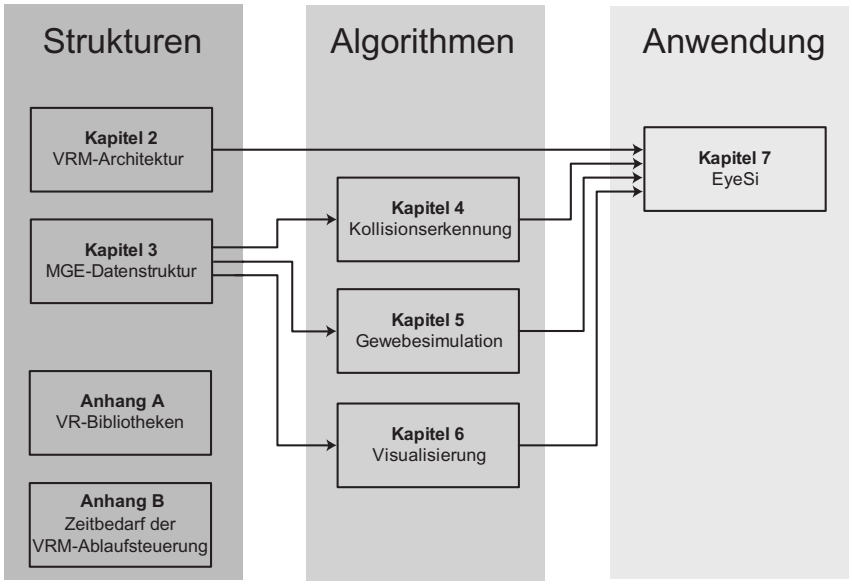


Abbildung 1.1: Abhängigkeiten zwischen den Kapiteln. Die Pfeile stellen eine „wird-verwendet-von"-Relation dar.

Augenoperationssimulator EyeSi werden die Ergebnisse der vorangegangenen Kapitel spezialisiert.

Einbettung in die Arbeitsgruppe

Die Dissertation wurde in der Arbeitsgruppe ViPA (*Virtual Patient Analysis*) des Lehrstuhls für Informatik V angefertigt. ViPA entwickelt Simulationsalgorithmen und VR-Techniken für medizinische Anwendungen. Produktentwicklung und -vermarktung wird von der VRmagic GmbH durchgeführt, einer Ausgründung des Lehrstuhls für Informatik V.

Abb. 1.2 stellt dar, welche Dissertationen und Diplomarbeiten Anteil an der Entwicklung der VRM-Bibliothek und des EyeSi-Systems haben.

Eigenheiten der Darstellung

Spezialbegriffe werden bei ihrer ersten Erwähnung *kursiv hervorgehoben*; falls es sich dabei um englische Begriffe handelt, werden sie in ihrer Originalschreibweise eingeführt und bei allen weiteren Erwähnungen in einer eingedeutschten Version verwendet (Beispiel: ein *vertex array* wird bei der zweiten Erwähnung zum Vertex-Array).

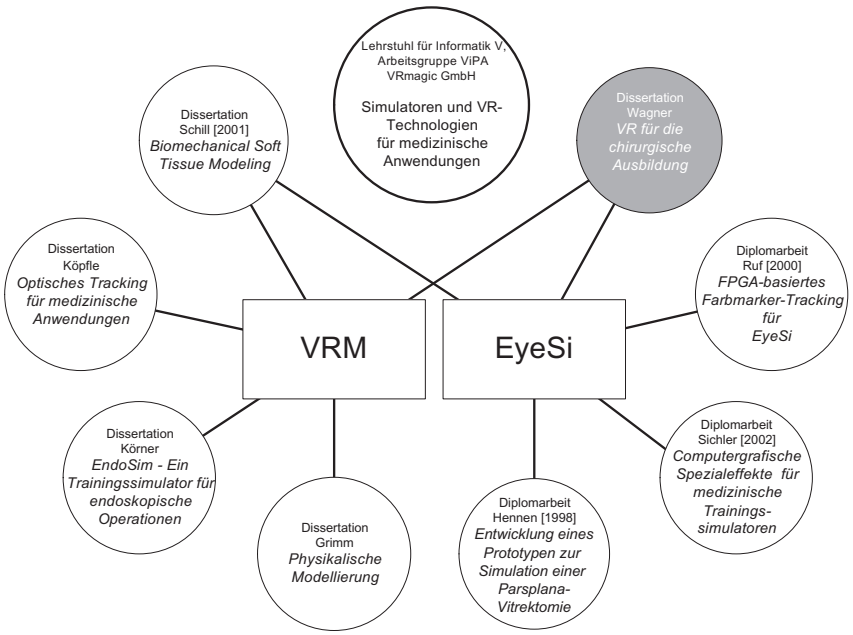


Abbildung 1.2: Im Zusammenhang mit der Softwarebibliothek VRM und dem Augenoperationssimulator EyeSi stehende Arbeiten.

Weibliche und männliche Formen – der Chirurg, die Programmiererin – sind der Einfachheit halber regellos verteilt.

Algorithmen werden in einem allgemein verständlichen Pseudocode angegeben; um Platz zu sparen, sind zusammenhängende Blöcke nicht mit begin/end-Klammern, sondern mit Einrückungen kenntlich gemacht.

Verweise auf Internet-Seiten werden, soweit möglich, nur für ergänzende Informationen angegeben, da sie schnell an Aktualität verlieren. Sie wurden bei Fertigstellung der Arbeit im April 2003 überprüft.

1.3 Virtuelle Realität

Eine der ersten dem Autor bekannten Beschreibungen virtueller Realitäten gibt Lem [1964][1] in seiner *Summa technologiae*. Dabei sieht er wesentliche

[1]Im gleichen Jahr veröffentlichte Galouye seinen Roman *Simulacron-3* über virtuelle Welten, dessen Idee von einigen erfolgreichen Kinofilmen wieder aufgegriffen wurde. Neben Fassbinders Galouye-Verfilmung *Welt am Draht* (1974) gehören dazu *Dark City* (1998), *Matrix* (1999), *The Thirteenth Floor* (1999, eine Galouye-Neuverfilmung), *eXistenZ* (1999), auf gewisse Weise auch die *Truman-Show* (1998) und *Pleasantville* (1998). In den cineastischen Visionen der späten 90er Jahre sind virtuelle Welten bemerkenswert

Aspekte von VR voraus; einige Zitate werden daher zur gedanklichen Gliederung dieses Abschnitts verwendet. Lem nennt die Technik zur Erzeugung virtueller Realitäten „Phantomatik“. In seinen Worten beschäftigt sie sich mit folgendem Problem:

> Wie lassen sich Realitäten erzeugen, die für die in ihnen verweilenden vernünftigen Wesen in keiner Weise von der normalen Realität unterscheidbar sind, doch anderen Gesetzen unterliegen als diese?

Bemerkenswert an diesem Zitat ist der scheinbare Widerspruch zwischen der Forderung nach Ununterscheidbarkeit der simulierten und der „normalen“ Realität und der Feststellung, dass die simulierte Realität anderen Gesetzen folgt. Kapitel 5 wird sich mit diesem Punkt bei der Unterscheidung von *deskriptiver* und *physikalischer Modellierung* beschäftigen.

Um virtuelle Realität zu definieren, wird zunächst Lems „Ununterscheidbarkeit“ durch den schwächeren Begriff der *Präsenz* ersetzt:

Definition 1.1 (Präsenz) Präsenz *ist die subjektive Empfindung, sich in einer bestimmten Umgebung zu befinden.*

Aus diesem Begriff lässt sich direkt eine Definition für virtuelle Realitäten ableiten:[2]

Definition 1.2 (Virtuelle Realität) *Eine* virtuelle Realität *ist eine Einrichtung, die auf künstliche Weise das Gefühl der Präsenz in einer bestimmten Umgebung erzeugt.*

Im Vergleich zu den üblichen Ansätzen[3] lässt die Allgemeinheit dieser Definitionen offen, auf welche Weise die Illusion der Realität vermittelt wird. Eine HiFi-Anlage, der man mit geschlossenen Augen zuhört, kann im Sinne von Def. 1.2 die virtuelle Realität eines Konzertsaals erzeugen.

Ähnlich wie der Turing-Test stützt sich die Definition der virtuellen Realität auf einen subjektiven Eindruck. Eine objektiv klassifizierbare Eigenschaft vieler virtueller Realitäten bezeichnet der Begriff der *Immersion* („Verschmelzung“).

präsent. Ihre Darstellung hat häufig einen beinahe religiösen Charakter.

[2] Die Definition stammt im wesentlichen von Deutsch [1997], der schreibt: „[Virtual Reality] refers to any situation in which a person is artificially given the experience of being in a specified environment“.

[3] Z.B. Haller [2000, Abschnitt 2.1] zum Begriff der virtuellen Umgebung: „Virtuelle Umgebungen sind demnach dreidimensionale, vom Computer generierte, simulierte Umgebungen, die in Echtzeit ‚gerendert‘ werden und schließlich dem Benutzer die Möglichkeiten bieten, sich in ihnen frei zu bewegen“.

Definition 1.3 (Immersion) *Ein bestimmtes Medium erzeugt* Immersion*, wenn die angesprochenen Wahrnehmungskanäle keine ausserhalb des Mediums ausgesandten Signale mehr empfangen.*

Immersive Medien sind hilfreich, um Präsenz zu erzeugen, besonders, wenn mehrere Wahrnehmungskanäle auf diese Weise angesprochen werden.[4] Immersion ist aber weder hinreichend noch notwendig für Präsenz: Zum einen wird die Qualität der virtuellen Realität durch den Begriff nicht spezifiziert – wenn beispielsweise ein simuliertes Objekt nicht auf Interaktionsversuche des Benutzers reagiert, geht der Eindruck von Präsenz auch bei einer vollständig immersiven Darstellung sofort verloren. Zum anderen können mentale Prozesse die Signale eines nicht-immersiven Mediums so ergänzen, uminterpretieren oder ignorieren, dass die Präsenz erhalten bleibt.[5]

1.3.1 Wahrnehmung

> Die Welt ist in meinem Kopf. Mein Körper ist in der Welt.
>
> —Paul Auster

Ein scheinbar wesentliches Charakteristikum der Realität R, ihre unmittelbare Erfahrbarkeit, ist in Wahrheit eine Illusion. Unsere Wirklichkeit[6] ist das Ergebnis eines Wahrnehmungsprozesses $W(R)$, bei dem die Sinnesorgane elektromagnetische und akustische Wellen detektieren, chemische Analysen durchführen und mechanische Kräfte messen und auf nervliche Reize abbilden. Umgekehrt werden Entscheidungen unseres Selbst[7] S in Handlungen $H(S)$ umgesetzt, indem über nervliche Reize die entsprechenden Muskeln aktiviert werden.

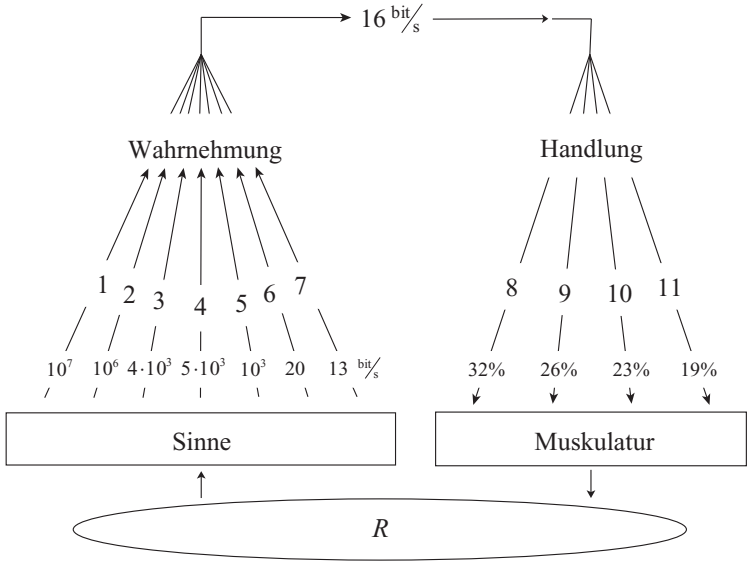
Auf dem Weg zum Bewusstsein werden die Daten um viele Grössenordnungen reduziert. Die Sinnesorgane können etwa 10 Millionen bit/s aufnehmen,

[4] Im technischen Sinne nennt man das *Multimodalität* [Blach et al. 1998].

[5] In der Literatur herrscht Uneinigkeit über die Abgrenzung der Begriffe Präsenz und Immersion, vgl. etwa den Streit zwischen Witmer und Singer [1998] und Slater [1999]. Häufig werden die beiden Begriffe auch gleichgesetzt, so zum Beispiel in dem anregenden Artikel von Smith et al. [1998], der unter anderem die „immersive Kraft“ von Tagträumen und Büchern diskutiert, obwohl es mit den oben angegebenen Definitionen um *Präsenz* gehen würde.

[6] *Wirklichkeit* („wirken“, einen Effekt haben) ist der Bereich der menschlichen Erfahrung, während *Realität* (lateinisch „res“, das Ding) die Welt an sich bezeichnet [Peschl und Riegler 1999]. Ob „die Welt an sich“ für den Menschen ein relevanter Begriff ist, soll hier nicht diskutiert werden.

[7] Hier wird unter „Selbst“ das verstanden, was objektiv die Identität einer bestimmten Persönlichkeit ausmacht.



1. Visuelle Wahrnehmung 2. Gehör 3. Tastsinn 4. Wärmesinn 5. Muskelsinn 6. Geruchssinn 7. Geschmackssinn 8. Skelett 9. Hand 10. Sprachmuskeln 11. Gesicht

Abbildung 1.3: Der Informationsfluss im Menschen (nach Nørretranders [1994, Kapitel 6]). Die Sinne können etwa 10 Millionen bit/s aufnehmen, bewusst verarbeitet werden aber nur einige bit/s. Die Genauigkeit der einzelnen Werte dient nur der relativen Einordnung.

bewusst verarbeit werden aber bloß einige bit/s (Abb. 1.3). Während der Datenreduktion werden Informationen ergänzt, interpretiert, gefiltert oder miteinander verknüpft, ohne dass wir uns dessen bewusst werden. Bestimmte Wahrnehmungen werden dabei ignoriert oder uminterpretiert.

Bemerkungen zur technischen Realisierung

Durch den großen Anteil, den die unbewusste Verarbeitung am Wahrnehmungsprozess hat, ist die Empfindung der Präsenz in bestimmten Grenzen fehlertolerant. Dies gilt besonders dann, wenn mehrere Wahrnehmungskanäle angesprochen werden.

Körner et al. [1999] beschreiben beispielsweise einen Algorithmus, der bei Engpässen in der haptischen Berechnung die Szene mit dem Interaktionspunkt des Benutzers mitbewegt. Da die gleichzeitige visuelle Darstellung diese Bewegung nicht anzeigt, wird sie dem Benutzer nicht bewusst.

Von Razzaque et al. [2002] wird ein ähnlicher Trick dazu verwendet, die räumlichen Beschränkungen einer CAVE zu verbergen. Wenn die Benutzerin ihre Position verändert, wird die Szene unmerklich so bewegt, dass sie wieder die Ausgangsposition einnehmen muss, um ihre Perspektive zu behalten. Wenn die Geschwindigkeit der Bewegung hinreichend gering ist, wird sie nicht bewusst wahrgenommen.

1.3.2 Schnittstellen

Wie bringt man einen Menschen in eine virtuelle Realität? Man verändert Eingabewerte oder die Berechnung von $W(R)$ selbst, so dass ihn glaubwürdige Signale erreichen, die er als Realität begreift.

$W(R)$ kann angepasst werden, indem man die Sinnesorgane durch einen Generator elektrischer Signale ersetzt, der die Sinnesnerven so reizt, wie es den realen Eindrücke entspricht: visuellen und auditiven Reizen, Empfindungen von Druck und Druckänderungen, Vibrationen, Feuchtigkeit, Temperatur, Schmerz, Beschleunigung, räumlicher Orientierung, Geruch und Geschmack. Die elektrischen Signale $H(S)$, die die Muskeln aktivieren, kann man mit einem ähnlichen Mechanismus messen (und deren Weiterleitung blockieren), um Interaktionen zu ermöglichen.

Probleme bei diesem Ansatz gibt es auf verschiedenen Ebenen. Eine technische Frage ist, wie man die Reizleitung von Nerven beeinflusst, ohne ihre ursprüngliche Funktion zu zerstören. Kurzweil [2001] schlägt zur Vermeidung chirurgischer Eingriffe vor, Nanoroboter einzusetzen, die an geeigneten Stellen Neuroimplantate anbringen. Ethisch bedenklich bleibt die Veränderung des menschlichen Nervensystems zur Erzeugung von Illusionen trotzdem.

Hinzu kommt, dass die Abbildung $W(R)$ nicht vollständig bekannt ist, vor allem nicht bei den Sinnesorganen mit hoher Eingangsbandbreite, die einen großen Teil der Verarbeitung selbst durchführen. So stehen den etwa 130 Millionen lichtempfindlichen Zellen der Retina nur eine Million Ganglienzellen gegenüber, die die Signale über den Sehnerv an das Gehirn übertragen.[8]

Einfacher ist es, nur die Eingabewerte von $W(R)$ zu verändern und die Sinnesorgane selbst als Schnittstelle zu verwenden. Für den visuellen Kanal beschreibt Lem [1964] diese Vorgehensweise wie folgt:

> Wenn nun der Mensch (unter normalen Bedingungen) nicht direkt mit den eigenen Augen, sondern durch das „Gegenauge“ blickt, dann sieht er alles in ganz normaler Weise, nur dass er auf der Nase so etwas wie eine (ein wenig komplizierte) Brille trägt, und diese „Brille“ ist nicht bloß ein lichtdurchlässiges „Einschiebsel“ zwischen seinem Auge und der Welt, sondern zugleich ein „Punktier“apparat, der das gesehene Bild in so viele Punktelemente zerlegt, wie die Netzhaut Zapfen und Stäbchen zählt.

Bemerkungen zur technischen Realisierung

Lems „Gegenauge“ entspricht weitgehend heutigen *Head-mounted Displays* (HMD), die allerdings mit etwa einer Million Pixeln eine weit geringere Auflösung haben als die Retina.

Für visuelle und akustische Reize stehen mit CAVEs, HMDs und Hifi-Surround-Anlagen brauchbare Näherungen für universelle Renderer zur Verfügung. Für die anderen Sinne des Menschen gibt es so etwas noch nicht: ein Phantom-Force-Feedback-Gerät beschränkt sich auf die Simulation punktueller haptischer Interaktionen [Massie und Salisbury 1994], so, als würde man die Welt nur mit einem Stift abtasten. Ein Exoskelett, das mit einem gewaltigen mechanischen Aufwand immerhin am ganzen Körper Kräfte erzeugen kann, spricht den Tastsinn nicht an. Der mechanische Aufbau zur Simulation eines Festmahls sorgt voraussichtlich dafür, dass die Versuchsperson den Appetit verliert.

Für die Simulation einer neuen chirurgischen Operation kann man zwar ein CAD-Modell eines Instruments in das Visualisierungsmodul eines bestehenden VR-Systems laden – um einen realistischen taktilen Eindruck

[8] Die Art der Verschaltung variiert über die Netzhaut. Am Rand des Gesichtsfelds werden mehr Zellen zu einer Einheit zusammengefasst, so dass dort die Auflösung geringer, die Lichtempfindlichkeit dagegen größer ist [Riedel 1997]. Deswegen nimmt man z.B. Monitorflimmern stärker wahr, wenn man den Bildschirm aus den Augenwinkeln beobachtet.

zu erzeugen, muss aber eine physikalische Nachbildung des Instruments (in Verbindung mit geeigneten Tracking- und Force-Feedback-Einrichtungen) gebaut werden.

Besondere Anforderungen an die Architektur eines VR-Systems ergeben sich daraus, dass verschiedene Ein- und Ausgabegeräte mit unterschiedlichen Zeitskalen arbeiten. Wegen der hohen Empfindlichkeit des Tastsinns auf Vibrationen benötigt ein Force-Feedback-Gerät eine Wiederholrate von etwa 1000Hz [Srinivasan und Basdogan 1997]. Ein optisches Tracking-System arbeitet häufig mit der PAL- oder NTSC-Bildwiederholrate (50 bzw. 60Hz), ein Stereo-Display benötigt manchmal eine feste Bildwiederholrate (z.B. 85Hz, frame-interlaced beim Sony LDI 100, mit dem der erste EyeSi-Prototyp aufgebaut wurde), eine grafische Benutzeroberfläche hat, je nach ausgewählter Funktion einen variablen Zeitbedarf. Um die Steuerung dieser Abläufe zu ermöglichen, muss ein VR-System nebenläufige Prozesse verwalten, synchronisieren und auf mehrere Prozessoren verteilen können (Kapitel 2).

Nebenbei bemerkt gibt es eine Fülle von Arbeiten, die sich mit neuen Eingabeparadigmen zur Steuerung von VR-Anwendungen beschäftigen – die normalen Eingabegeräte Tastatur und Maus lassen sich in eine VR-Umgebung häufig nicht integrieren, ohne die Präsenz zu zerstören. Neben der Erkennung von Sprache oder Gestik werden Metaphern wie z.B. ein virtueller Laserpointer [Goebbels et al. 1999], virtuelle [Wloka und Greenfield 1995] oder reale [Watsen et al. 1999] PDAs untersucht.[9] Bei Operationssimulationen wird gewöhnlich nicht die gesamte Operationsumgebung simuliert, sondern nur der Einblick in das Operationsgebiet, den der Chirurg beispielsweise über ein Stereomikroskop oder den Bildschirm eines Endoskops erhält. In diesem Fall wirkt es sehr unnatürlich, die Metafunktionen in der virtuellen Welt zu implementieren. Im EyeSi-System wird daher ein (realer) berührungsempfindlicher Bildschirm verwendet.[10]

1.3.3 Interaktivität

Was wir denken, welche Entscheidungen wir treffen, was wir tun, kurz: die zeitliche Entwicklung $\dot{S}$ unseres Selbst hängt von unserem augenblicklichen Zustand S und der Wahrnehmung $W(R)$ ab.

[9] Eine poetische Beschreibung eines Steuerungsparadigmas gibt Michael Crichton in seinem Roman *Disclosure*: ein Engel begleitet die Besucher der virtuellen Welt als schwebender Avatar und verarbeitet ihre gesprochenen Steuerkommandos.

[10] Eine Variante wäre, mit Hilfe der Operationsinstrumente, Fußpedale oder Mikroskop-Regler einen Schalter oder ein Menü der GUI zu betätigen. Trotz des höheren Immersionsgrades würde durch den fehlenden Realitätsbezug das Gefühl der Präsenz verringert werden.

Es existiert daher eine Funktion f mit:

$$\dot{S} = f(S, W, O_f) \tag{1.1}$$

Ob f berechenbar ist, ist Gegenstand vieler philosophischer Diskussionen [Hofstadter 1979, Searle 1980]. Odifreddi [1992] schreibt hierzu in Abschnitt I.8: „For the *biological computer*, we do not have yet a theory, and discussions of human computability are mostly rambling talk. We pursure both the synthetical (bottom-up) and the analytical (top-down) approaches, by analyzing the brain structure and theories of constructive reasoning, but we reach a dead end soon in both cases."

Der Parameter O_f hat die Funktion eines statistischen Orakels[11] und beschreibt den Einfluss nichtdeterministischer Einflüsse, etwa quantenmechanischer Effekte oder des „freien Willens".

Zeit und Realität kommen in Gleichung 1.1 nicht als explizite Variablen vor. Die Abwesenheit von R entspricht dem Konzept der *kognitiven Selbstreferenz*, bei dem die Stimulationen der Wahrnehmungsorgane $W(R)$ nur als „mere peripheral energetic conditions [...] for a semantically closed and self-organizing cognitive system" betrachtet werden [Peschl und Riegler 1999].

Die Zeit wird nur als Veränderung von W bzw. S wahrgenommen. Deutsch [1997] schreibt dazu in Kapitel 11: „We do not experience time flowing, or passing. What we experience are differences between our present perceptions and our present memories of past perceptions." Diese Feststellung kann als Anlass dazu genommen werden, auch ein VR-System nicht mit einem absoluten Zeitgeber, sondern im Takt der I/O-Geräte und Algorithmen arbeiten zu lassen. Probleme können dabei entstehen, wenn die einzelnen Prozesse nicht synchron arbeiten (Kapitel 2).

Der Begriff *Echtzeit* wird im Kontext eines VR-Systems daher nicht über ein explizites zeitliches Kriterium definiert, sondern in Bezug zur Wahrnehmung gesetzt. Die von Stankovic [1996] geforderte zeitliche Korrektheit („Real-Time systems are those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.") entspricht in VR-Systemen der Aufrechterhaltung der Präsenz.

Definition 1.4 (Echtzeit in virtuellen Realitäten) *Ein System arbeitet in* Echtzeit*, wenn sein zeitliches Verhalten (Latenz und Wiederholrate) das Gefühl der Präsenz nicht zerstört.*

[11] Der Begriff „Orakel" ist der Berechenbarkeitstheorie entnommen und hilft dort, nicht berechenbare Funktionen zu klassifizieren. Eine Funktion, die dann berechenbar ist, wenn sie an bestimmten Stellen der Rechnung ein Orakel O konsultiert, heißt *relativ berechenbar zu* O.

Schwankungen des zeitlichen Verhaltens sind bei einer hinreichend hohen Wiederholrate[12] des Systems in bestimmten Grenzen tolerierbar. Watson et al. [1997] geben an, dass Schwankungen der Wiederholrate um bis zu 40% um den Mittelwert keinen Einfluss auf die Leistungen der Benutzer in einer virtuellen Testumgebung haben, solange die Wiederholrate selbst hoch genug ist (20Hz). Mit der üblichen Unterscheidung zwischen *harten* und *weichen Echzeitsystemen*[13] entspricht Def. 1.4 daher einer weichen Echtzeitbedingung.

Mit Hilfe von $H(S)$ beeinflusst das Selbst die Realität R. Es gibt also eine Funktion g, die auf der Grundlage der physikalischen Gesetze die Auswirkungen von H auf die Realität beschreibt:

$$\dot{R} = g(R, H, O_g) \tag{1.2}$$

Im allgemeinen Fall wird auch hier ein statistisches Orakel O_g benötigt. Über den Stand der Diskussion, ob g berechenbar ist, geben Cooper und Odifreddi [2003] einen Überblick. Im Kontext von Gewebeinteraktionen existieren gute Näherungen für g, deren Qualität eher durch die Laufzeitanforderungen als durch prinzipielle Beschränkungen beeinflusst wird. Bei der Gewebesimulation spielt O_g z.B. dann eine Rolle, wenn verschiedene Ausprägungen eines Krankheitsbildes durch eine zufällige Wahl bestimmter Parameter modelliert werden.

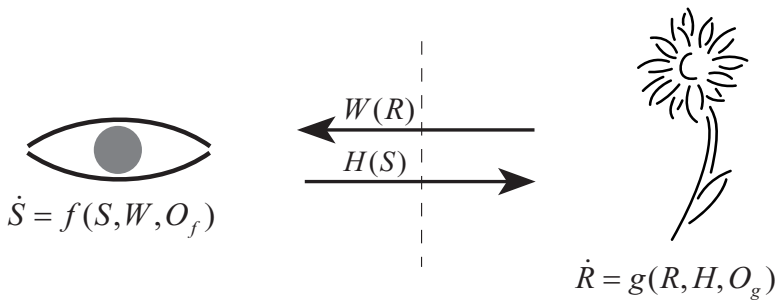


Abbildung 1.4: Selbst und Realität wirken durch Wahrnehmungen und Handlungen aufeinander ein.

[12] Der deutsche Ausdruck „Wiederholrate“ für „Framerate“ bzw. „Updaterate“ ist etwas unglücklich, da zwei aufeinanderfolgende Frames üblicherweise nicht die gleiche Information „wiederholen“.

[13] Das Zeitverhalten eines harten Echtzeitsystems ist deterministisch und unabhängig von der aktuellen Konfiguration (Zahl der laufenden Prozesse, Systemlast, etc. . .); das Zeitverhalten eines weichen Echtzeitsystems kann in definierten Grenzen schwanken.

Die Gleichungen 1.1 und 1.2 sind durch die Abhängigkeiten $W = W(R)$ und $H = H(S)$ gekoppelt (siehe auch Abb. 1.4). Diese Rückkoppelung – *Interaktivität* – ist wesentlicher Bestandteil der Realität. Wenn wir prüfen wollen, ob etwas „real“ ist, dann fassen wir es an. Nicht, weil wir dem Tastsinn an sich mehr vertrauen als dem visuellen Eindruck, sondern weil dies eine einfache Methode ist, in dem Wechselspiel von Muskelkraft und Gegenkraft des Objekts die Rückkoppelung zwischen W und H zu *begreifen.* Lem schreibt dazu:

> [...] Die Wirkung der Phantomatik [ist eine] „Kunst der Rückkoppelung“, die den ehemaligen Rezipienten zum aktiven Teilnehmer, zum Helden, zum Mittelpunkt programmierter Ereignisse macht.

In Card et al. [1983, Kapitel 2] werden eine ganze Reihe von Messungen zur Bestimmung der Eigenschaften dieser Rückkoppelung beschrieben. Abb. 1.5 zeigt als Beispiel ein Experiment zur Messung der Dauer von *recognize-act-cycles.*

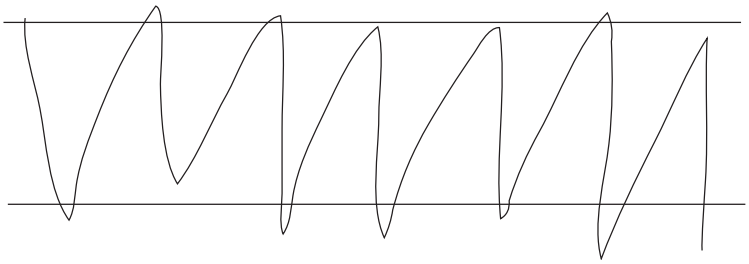


Abbildung 1.5: Skizze eines Experiments zur Messung der Dauer von *recognize-act-cycles* (nach Card et al. [1983, Abb. 2.5]): ein Stift musste möglichst schnell zwischen zwei Linien bewegt werden. Dabei wurden maximal 68 Umkehrpunkte in 5 Sekunden erreicht (74ms pro Umkehrpunkt); in dieser Zeit konnten aber mit Hilfe der visuellen Wahrnehmung nur 20 Korrekturen von Überschwingungen (240ms pro Korrektur) durchgeführt werden.

Für das Training manueller Fähigkeiten spielt das *Gesetz von Fitt* eine große Rolle. Es beschreibt die Zeit, die bei einer eindimensionalen zielgerichteten Bewegung benötigt wird, um eine Struktur der Größe S in einer Entfernung D zu erreichen. Gleichung 1.3 gibt das Gesetz in der Version von Card et al. wieder. Der Bereich der Rückkoppelungskonstante k (*index of performance*) wird dort mit 70...120ms angegeben.

$$T = k \log_2(\frac{D}{S} + \frac{1}{2}) \tag{1.3}$$

Bemerkenswert an dem Gesetz von Fitt ist, dass nur das Verhältnis von Entfernung D und Objektgröße S eingeht. Das bedeutet, dass der zeitliche Ablauf der Bewegung unabhängig vom Maßstab ist, in der sie stattfindet. Die Abtastfrequenz für zielgerichtete Bewegungen, wie sie bei Operationen stattfinden, ist daher unabhängig von der Größe des Operationsgebiets und der beteiligten Strukturen.

Der Eindruck eines zeitlichen Ablaufs entsteht durch die Bewegung auf einer Trajektorie durch den von S und W aufgespannten Raum (Abb. 1.6). Präsenz bleibt erhalten, solange die Trajektorien von Realität und Simulation nicht zu weit voneinander entfernt sind, die virtuelle Realität also in einem „Präsenzkorridor“ bleibt (äußere Kurven in Abb. 1.6).

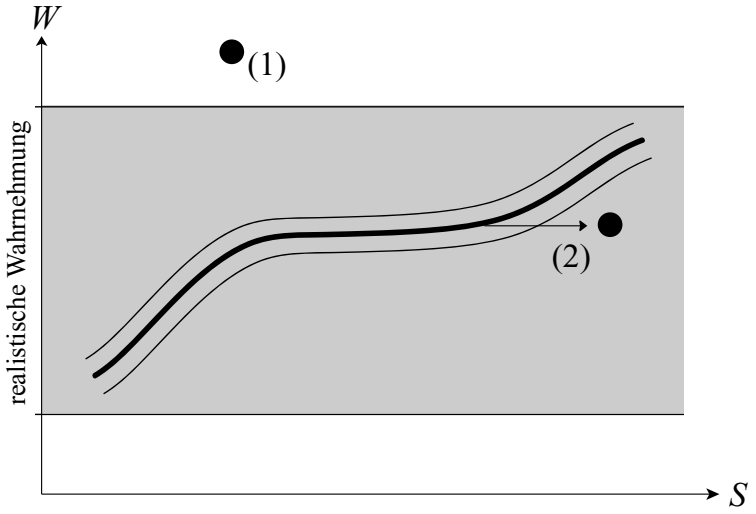


Abbildung 1.6: Die Bewegung auf einer Trajektorie (mittlere Kurve) durch den von S und W aufgespannten Raum wird als Zeit wahrgenommen. Unrealistische Sinnesreize (1) und eine zu hohe Latenz (2) führen gleichermaßen dazu, dass der Präsenzkorridor (äußere Kurven) verlassen wird.

Wenn die erzeugte Wahrnehmung einen Punkt oder eine Richtung so definiert, dass sich die Versuchsperson zu weit von der Trajektorie entfernt, wird der Präsenzkorridor verlassen: in Abb. 1.6 durch unrealistische Sinnesreize (1) oder eine zu hohe Latenz (2).

1.3.4 Latenz

Wir haben ein sehr feines Gespür für das zeitliche Antwortverhalten von W auf eine bestimmte Handlung H: Ein früher (1998) Prototyp des EyeSi-Systems wies sowohl eine zu große Latenz des Trackingsystems als auch

einen offensichtlichen Fehler im Beleuchtungsmodell auf (die Operationsinstrumente warfen einen Schatten, dessen Form unabhängig vom gerade verwendeten Instrument war und der über den Lichtkegel der Spotlichtquelle hinausging). Die zu große Latenz (End-to-End-Latenz: etwa 150ms) wurde von allen Versuchspersonen kritisiert, der Beleuchtungsfehler in den meisten Fällen nicht einmal bemerkt.

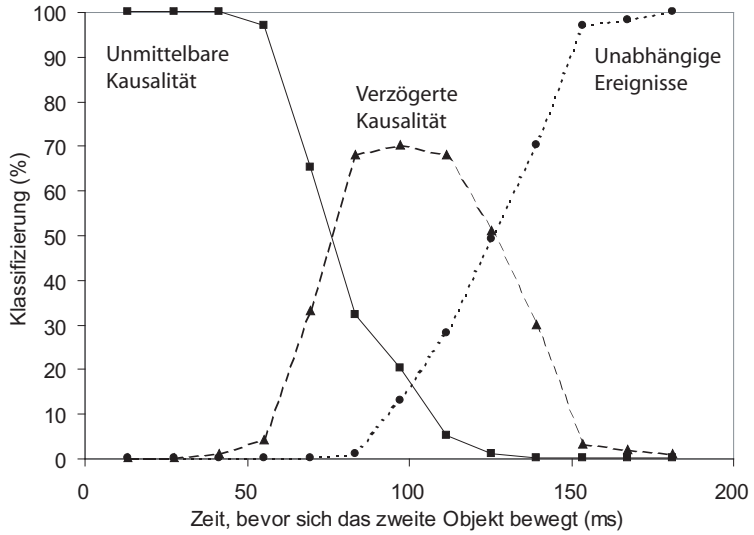


Abbildung 1.7: Wahrnehmung von Kausalität. Nach Michotte [1963], zitiert in Card et al. [1983, Abb. 2.9].

Abb. 1.7 zeigt das Ergebnis eines Versuchs, bei dem der Zusammenhang zwischen dem zeitlichen Abstand zweier Ereignisse und der Wahrnehmung ihres kausalen Zusammenhangs untersucht wurde. Die Versuchspersonen klassifizierten simulierte Stöße zwischen Billardkugeln, die ihnen visuell präsentiert wurden. Die Reaktion einer Kugel auf eine Kollision erfolgte mit variabler Latenz. In der Abbildung ist der zeitliche Abstand zwischen zwei Ereignissen gegen ihre Klassifizierung als „unmittelbar kausal“, „verzögert kausal“ und „unabhängig“ aufgetragen. „Verzögert kausal“ bedeutet, dass die Ereignisse als zusammenhängend klassifiziert wurden, ihr zeitlicher Abstand aber wahrnehmbar war.

Aus Abb. 1.7 geht hervor, dass Latenzen unterhalb etwa 50ms über den visuellen Kanal nicht wahrgenommen werden. Dieser Wert wird unterschritten, wenn mehrere Kanäle an der Wahrnehmung beteiligt sind, z.B. bei einem HMD, dessen Anzeige an die Position und Orientierung des Kopfes angepasst wird. Die Fusion der (künstlichen) visuellen Eindrücke mit den (realen) Signalen des Bewegungssinns ist sehr empfindlich auf zeitliche Verschiebungen, die zu Übelkeit (*cyber-sickness*) und Drehschwindel (*Oscillop-*

sia) [Allison et al. 2001] führen – selbst wenn die zeitlichen Differenzen bewusst gar nicht wahrgenommen werden. Von Mazuryk und Gervautz [1995] wird beschrieben, dass bei einer solche Anordnung eine Latenz von 74ms die Präsenz bereits zerstört; Held und Durlach [1991] (zitiert von Wloka [1995]) geben für den Fall eines Durchsicht-HMDs für Augmented-Reality-Anwendungen eine Latenz-Obergrenze von 30ms an. Der in Abb. 1.5 beschriebene Versuch ergibt für einen *recognize-act-cycle* einen Wert von etwa 200ms; unter der Annahme, dass die Latenz eines VR-Systems unerkannt bleibt, wenn sie eine Größenordnung unter diesem Wert liegt, ergibt sich ebenfalls eine Beschränkung der Latenz auf wenige 10ms.

Innerhalb dieser Zeit muss ein VR-System die Sensoren auslesen, die Veränderung g der virtuellen Welt und eine passende Ausgabe berechnen. Problematisch dabei ist, dass reale Vorgänge massiv parallel ablaufen, während deren Nachbildung im Computer sequentiell oder in vergleichsweise bescheidenem Umfang parallel berechnet wird. Dies stellt hohe Anforderungen an die verwendeten Algorithmen.

— Falls, wie oben geschildert, Nervensignale unter Umgehung der Sinnesorgane erzeugt werden, ist es übrigens denkbar, in einer virtuellen Realität den gewohnten zeitlichen Abstand zwischen einer bestimmten Handlung und der korrespondierenden Wahrnehmung nicht zu über-, sondern zu unterschreiten. Dies geschieht dann, wenn der Generator der Signale schneller auf eine bestimmte Handlung reagiert als die Sinnesorgane. Ein solches Experiment wurde noch nicht durchgeführt, so dass wir nur vermuten können, welcher Eindruck dadurch hervorgerufen werden könnte: Zu dem Zeitpunkt, an dem die Signale erfahrungsgemäß erwartet werden, sind sie schon angekommen. Dieses Phänomen könnte als Erinnerung fehlinterpretiert werden; man würde die virtuelle Realität dann als anhaltendes *déja vu* erleben.

2

Die VRM-Architektur

Zum Bau von medizinischen VR-Applikationen wurde in den vergangenen Jahren die VRM-Softwarebibliothek (*Virtual Reality in Medicine*) entwickelt. Dieses Kapitel beschreibt die Komponenten von VRM und die Architektur einer typischen VRM-Applikation.

Um die in Kapitel 1 beschriebene Echtzeitbedingung (Def. 1.4) zu erfüllen, müssen Wiederholrate und Latenz einer VRM-Applikation über bzw. unter einer bestimmten Grenze bleiben. In den Abschnitten 2.5.1 und 2.5.2 werden mögliche Ursachen für Latenz angegeben und verschiedene Ausführungsmuster diskutiert, mit deren Hilfe das Zeitverhalten einer Applikation optimiert werden kann.

Zur zeitlichen Steuerung stellt die VRM-Bibliothek Klassen zur Verfügung, die in Abschnitt 2.5.3 beschrieben werden. In Abschnitt 2.5.4 wird mit Hilfe einiger Messungen gezeigt, dass das Zeitverhalten von Windows XP und Linux für Echtzeitsysteme im Sinne von Def. 1.4 geeignet ist.

2.1 Stand der Forschung

Es gibt eine beinahe unüberschaubare Vielfalt von VR-Bibliotheken. Die Beschreibung der wichtigsten Ansätze wurde daher in Anhang A ausgelagert.

Das Ziel einer VR-Bibliothek ist die Zusammenfassung und Abstraktion derjenigen Funktionen, die für eine virtuelle Realität benötigt werden – typische Beispiele sind Szenegraphen zur Formulierung bestimmter Beziehungen zwischen den Objekten, die vereinheitliche Ansteuerung von verschiedenen Sensoren und Ausgabegeräten sowie die Unterstützung von Netzwerkfunktionen für verteilte VR. Häufig wird eine Skriptsprache angeboten,

mit deren Hilfe Objekte und Eigenschaften der virtuellen Welt spezifiziert werden können. Zur Unterstützung von Interaktionen sind meist einfache Algorithmen zur Kollisionserkennung implementiert.

Deformierbare Objekte werden dagegen nur von wenigen Bibliotheken unterstützt. Beispiele sind KISMET und Spring, die in Anhang A beschrieben sind; diese beiden Bibliotheken werden explizit für medizinische Anwendungen entwickelt und stellen FEM- und Feder-Masse-Algorithmen zur Berechnung von Deformationen zur Verfügung.

Während etwa mit einer GUI-Bibliothek für beinahe jede denkbare Applikation eine grafische Benutzeroberfläche erstellt werden kann, besitzt keine existierende VR-Bibliothek eine vergleichbare Universalität. Was die Ansteuerung von VR-Schnittstellen betrifft, liegt dies an dem in Abschnitt 1.3.2 beschriebenen Problem, dass nicht für alle Sinne ein universeller Renderer existiert. Für Interaktion und Visualisierung gibt es mit geometrischen Verfahren zur Kollisionserkennung, FEM-Lösern für die Gewebesimulation und Raytracing- und Radiositymodellen zwar leistungsfähige Algorithmen, die erforderliche Rechenzeit übersteigt jedoch in den meisten Fällen die VR-Echtzeit-Bedingung um einige Größenordnungen.

2.2 Architektur der VRM-Bibliothek

Auch die VRM-Bibliothek erhebt nicht den Anspruch, den gerade beschriebenen Widerspruch zwischen Echtzeit und Universalität aufzulösen. Die Echtzeitfähigkeit der mit ihr entwickelten Applikationen wird durch den Einsatz hardwareunterstützter, hardwarenaher und deskriptiver Algorithmen erreicht; da Universalität auf diese Weise nicht zu erreichen ist, wurden Strukturen entwickelt, die kurze Entwicklungszyklen ermöglichen:

Flexibles Datenmodell Das MGE-Datenmodell kann sowohl die inneren Eigenschaften von Objekten (Geometrie, Topologie, Gewebeeigenschaften) als auch deren Beziehungen zueinander repräsentieren. Sie ist weitgehend unabhängig von der Art der Algorithmen, die die Interaktionen berechnen. Daher können schnell Algorithmen ergänzt, verändert oder kombiniert werden, ohne die Struktur der Daten selbst zu verändern.

Mehrschichtige API Die von VRM unterstützten Funktionen können auf einem hohen Abstraktionsniveau verwendet werden, um schnelle Ergebnisse zu erhalten. Es ist aber auch möglich, auf die unteren Ebenen von VRM zuzugreifen, um z.B. Treiber und Algorithmen zum Ansteuern von Spezialhardware einzubinden.

Unabhängige Komponentenentwicklung Funktionell unterschiedliche Bereiche sind auch auf Entwicklungsebene voneinander getrennt. Auf VRM-Ebene ist es daher möglich, unabhängige C++-Bibliotheken für die einzelnen Komponenten zu erstellen. Auf Applikationsebene existiert eine scharfe Trennung zwischen GUI-, IO- und VR-Entwicklung: GUI-Programmierer benötigen nur die Schnittstelle zur eigentlichen Applikation, nicht aber deren Implementierung; die Ein- und Ausgabe eines Simulators kann zur Entwicklung und zum Test durch beliebige andere Ein- und Ausgabegeräte (Tastatur, Maus, Monitor oder Standard-VR-Interfaces wie Spacemouse, Phantom oder Shutterdisplay) ersetzt werden.

Techniken für große Mehrbenutzer-VR – etwa die Konsistenzerhaltung von verteilten Szenegraphen oder der Umgang mit Netzwerklatenzen – sind nicht Teil der aktuellen VRM-Entwicklung.

2.3 Komponenten der VRM-Bibliothek

Abb. 2.1 gibt einen Überblick über die Komponenten der VRM-Bibliothek und der darauf aufbauenden Applikationen. Alle Komponenten bilden eigene C++-Bibliotheken, so dass sie separat entwickelt, getestet und verwendet werden können

Die VRM-Bibliothek kann auf Windows- und Linux-Systemen eingesetzt werden. Als Entwicklungsumgebung unter Windows (aktuell 2000, XP, XP home) wird das Microsoft Visual Studio .NET, unter Linux der GCC-Compiler (aktuell: Version 3.2) verwendet. Die Bibliothek besteht zurzeit (Ende 2002) ohne Applikationen aus etwa 135.000 Zeilen Quellcode.

Ablaufsteuerung Mit den VRM-Ablaufsteuerungs-Klassen kann eine Applikation auf hoher Ebene ihre einzelnen Prozesse koordinieren. Dies schließt Mechanismen wie Speicherschutz, Synchronisation und Zeitmessung ein. Die freie ACE-Bibliothek [Schmidt 1993] wird zur Abstraktion von betriebssystemabhängigen Aufrufen vor allem bei der Threadsteuerung verwendet.

MGE-Datenrepräsentation Der Aufbau der virtuellen Welt wird mit Hilfe einer multigraph-basierten Struktur repräsentiert. Durch eine kompakte Speicherung wird eine hohe Zugriffs- und Übertragungsgeschwindigkeit erreicht.

Hardware-Ansteuerung Auf dieser Ebene werden Treiber zur Kommunikation mit VR-Interfaces entwickelt – etwa A/D-Wandlern zum

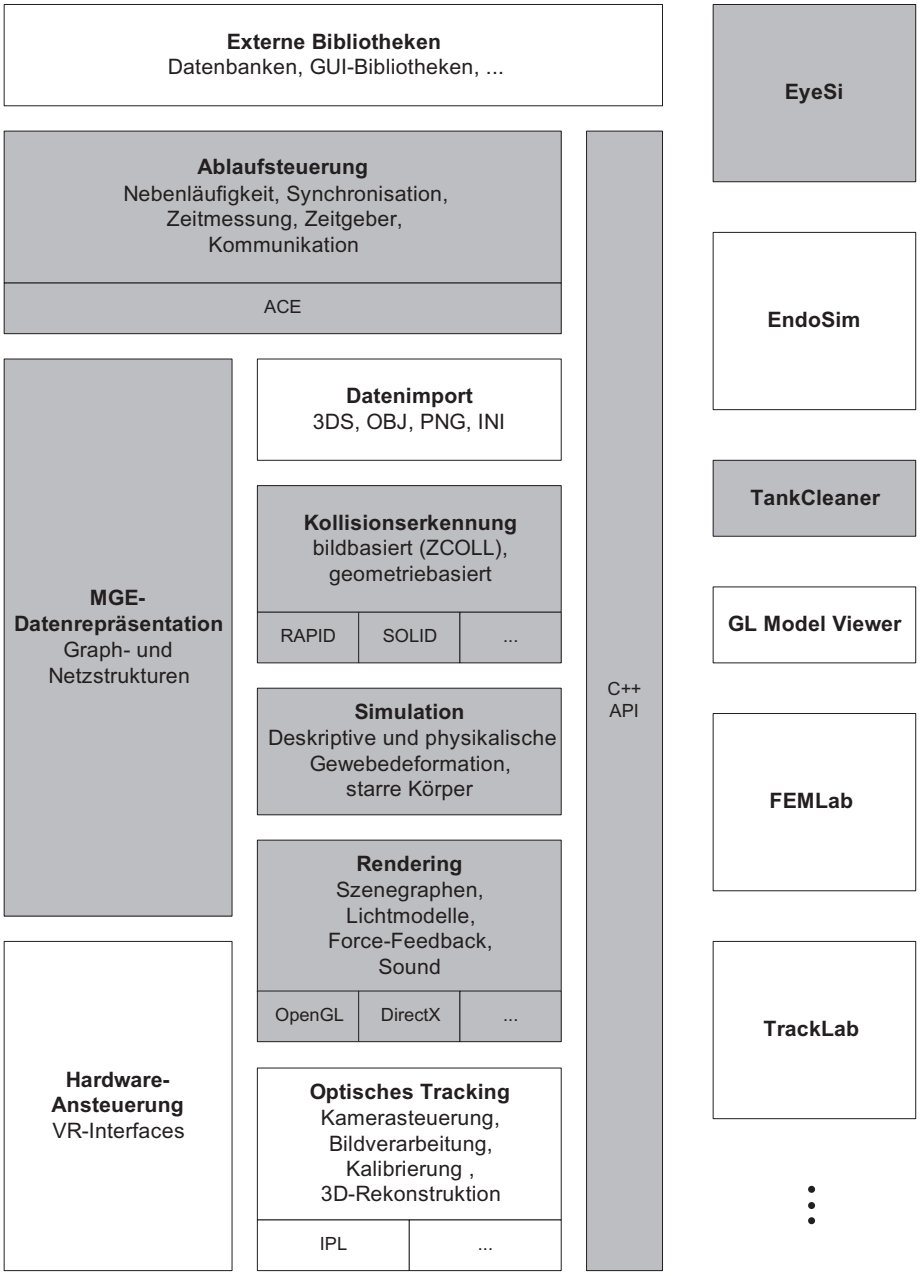


Abbildung 2.1: Die Komponenten der VRM-Bibliothek und darauf aufbauende Applikationen. Abhängigkeiten der einzelnen Komponenten sind durch die horizontale Anordnung angedeutet. Komponenten in der gleichen Spalte können unabhängig verwendet werden, bauen aber auf ihre „linken Nachbarn“ auf. Die wichtigsten externen Bibliotheken sind ebenfalls aufgeführt. Der Schwerpunkt der Arbeit lag bei der Konzeption und Entwicklung der eingefärbten Komponenten.

Auslesen von Operationspedalen oder Mikroskopreglern. Durch die hardwarenahe Entwicklung kann Betriebssystemunabhängigkeit nur durch Parallelentwicklung auf allen Zielplattformen erreicht werden.

Datenimport Geometrie, Topologie und Texturen eines virtuellen Objekts werden im allgemeinen mit 3D-Modellern wie dem 3D-Studio Max erzeugt. Für die gebräuchlichen 3D-Datenformate existieren Importfilter, die die Dateien in eine MGE-Struktur konvertieren.

Kollisionserkennung und Simulation Diese beiden Komponenten stellen die Algorithmen zur Verfügung, mit denen die Interaktionsfunktion g aus Gleichung 1.2 berechnet wird. Zur Kollisionserkennung wird neben geometriebasierten Ansätzen (unter anderem die beiden Bibliotheken RAPID und SOLID) das in dieser Arbeit entwickelte bildbasierte Verfahren ZCOLL verwendet. Zur Simulation von Gewebereaktionen stehen deskriptive und physikalische Modelle zur Verfügung.

Rendering Diese Komponente umfasst alle Algorithmen, die die interne Repräsentation der virtuellen Objekte in eine Form umwandeln, die der menschlichen Wahrnehmung zugänglich ist – Methoden zur Visualisierung, zum haptischen Rendering und zur Erzeugung von Audiosignalen. Die Visualisierungskomponente baut auf OpenGL auf, stellt aber eigene Licht- und Abbildungsmodelle zur Verfügung. Für die haptische Darstellung mit einem Phantom-Force-Feedback-Gerät wurden neue Algorithmen entwickelt [Körner et al. 1999, Körner 1999]. Die Audio-Ausgabe kapselt DirectX-Funktionen (und ist damit als einzige VRM-Komponente zurzeit noch nicht betriebssystemunabhängig).

Optisches Tracking Optisches Tracking ist ein vielseitiges Eingabemedium für virtuelle Realitäten: es arbeitet berührungsfrei und kann hohe Wiederholraten und Genauigkeiten erreichen. In VRM sind Algorithmen zur geometrischen und Farbkalibrierung von Kameras implementiert sowie zur Rekonstruktion und zum Filtern von 3D-Positionen [Ruf 2000, Nock 2000].

API Das *Application Programming Interface* setzt sich aus den APIs der einzelnen Komponenten zusammen und wird ergänzt durch eine dünne Schicht von Abstraktionen, die eine bestimmte Funktionalität zusammenfassen oder vereinfachen.

Externe Bibliotheken Die VRM-Architektur erzwingt keine bestimmten strukturellen Eigenschaften einer Applikation, die den Einsatz externer Bibliotheken verhindern könnten.[1]

[1] Im Gegensatz dazu ist beispielsweise die Ausführungsschleife des OpenGL-Toolkits GLUT [Kilgard 1996] blockierend, so dass in dem ausführenden Thread keine anderen

Applikationen Zurzeit werden folgende VRM-Applikationen entwickelt:

- Der Simulator EyeSi zur Simulation vitreoretinaler Operationen.
- Der Endoskopiesimulator EndoSim [Körner und Männer 2002].
- Der TankCleaner, eine (kleine) Applikation zur Entwicklung von Tankreinigungsanlagen, bei der die ZCOLL- Kollisionserkennung und die VRM-Visualisierungskomponente zur Simulation und Darstellung des Reinigungsvorgangs verwendet werden.
- Das FEMLab, eine Umgebung zur Konfiguration und Durchführung von FEM-Simulationen.
- Das TrackLab für Konfiguration und Test von optischen Trackingsystemen.

2.4 Architektur einer VRM-Applikation

Es gibt zwei verschiedene Arten, mit einem VR-System zu interagieren – im Kontext der virtuellen Welt und beim Aufruf von „Metafunktionen“ wie den folgenden:

- Benutzerverwaltung zur Speicherung von Trainings- und Prüfungsergebnissen
- Steuerung von Multimedia-Präsentationen zur Vermittlung von Wissen im Umfeld einer bestimmten Operation
- Aufzeichnung und Wiedergabe von Trainingsläufen
- Fernüberwachung und -steuerung von Trainingssitzungen

Während die ersten beiden Punkte weitgehend unabhängig von der Ablaufsteuerung und dem Datenfluss eines VR-Systems realisiert werden können, müssen die letzten beiden Punkte bei der Architektur des Systems berücksichtigt werden.

2.4.1 Trennung von I/O und VR

VR-Systeme haben zwei charakteristische Komponenten: das VR-Interface zum Ankoppeln an Wahrnehmung $W(R)$ und Handlung $H(S)$ („I/O-Teil“) und die Simulation der physikalischen Vorgänge („VR-Teil“). In einer VRM-Applikation wird eine Trennung dieser Komponenten unter zwei Aspekten durchgeführt:

Aufrufe mehr möglich sind; die GUI-Bibliothek Qt muss im Hauptthread der Applikation ausgeführt werden; die ACE-Bibliothek erzwingt eine main-Funktion der Form `int main (argc, argv)`.

- als *logische Trennung*: im I/O-Teil werden reale Objekte und Reize mit den symbolischen Bezeichnern der Software verknüpft, während im VR-Teil virtuelle Objekte verarbeitet werden (Abb. 2.2).

  Ein Beispiel: Position und Orientierung eines realen Modellauges werden im I/O-Teil mit einem optischen Trackingsystem festgestellt und in einer Tabelle unter einem Bezeichner („Transformationsmatrix_Auge“) abgelegt.

  Im VR-Teil existiert ein virtuelles Objekt „Auge“, das neben seiner Transformationsmatrix auch Gewebeeigenschaften, die Einbindung in den Szenegraphen der virtuellen Welt, seine geometrischen, topologischen und visuellen Eigenschaften und Informationen für Simulationsalgorithmen speichert.

- als *Projekttrennung*: I/O- und VR-Teil sind vollständig voneinander getrennte Softwareprojekte. Es gibt keine gemeinsam zu entwickelnde Klassen, globale Strukturen oder Projektdaten (Pfade, Compilereinstellungen, eingebundene Dateien).

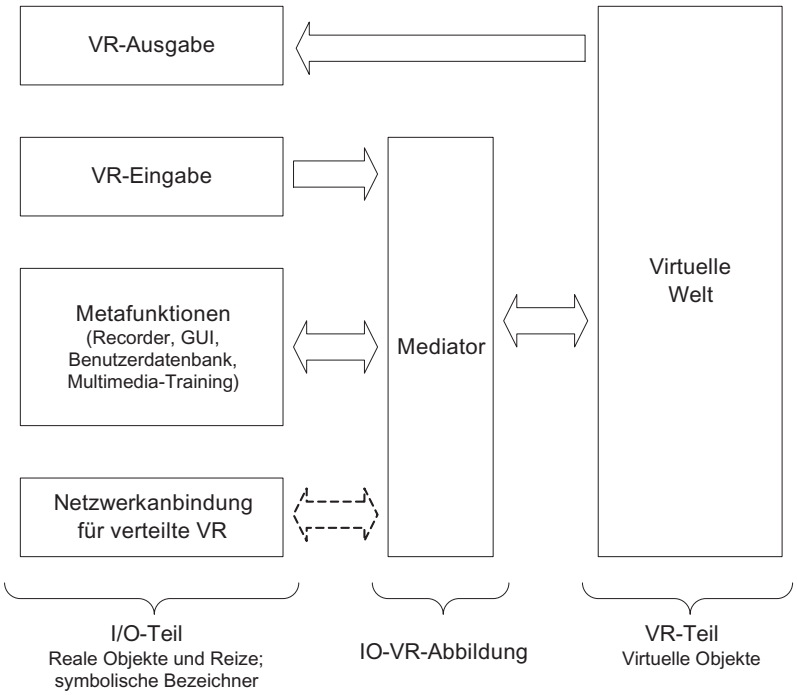


Abbildung 2.2: Trennung des IO- und VR-Teils einer Applikation. Funktionen für verteilte VR sind vorgesehen, aber noch nicht implementiert.

Die Projekttrennung beschleunigt Einarbeitung, Entwicklung und Test. Sie wird über das Design-Muster *Mediator* erreicht, das von Gamma et al. [1995] so beschrieben wird:

> [Mediator] defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Die unabhängigen Komponenten der Applikation greifen auf Mediatorbasisklassen zu, deren Methoden zur Kommunikation mit den anderen Komponenten leer sind oder Default-Werte zurückgeben. Wenn die Teile zusammengefügt werden, werden die Basisklassen durch abgeleitete Klassen ersetzt, bei denen alle Kommunikationsrichtungen implementiert sind. Zur Entwicklung und zum Test kann ein anderer Mediator instanziiert werden. Beispielsweise werden dann anstelle der Daten eines aufwändigen VR-Interfaces die Tastatur- und Mauseingaben eines Entwickler-PCs auf die Objektbewegungen in der virtuellen Welt abgebildet.

2.4.2 Datenfluss

Zur Darstellung des Datenflusses in einer VRM-Applikation werden Gane-Sarson-Diagramme verwendet, deren Symbole in Abb. 2.3 erläutert sind.

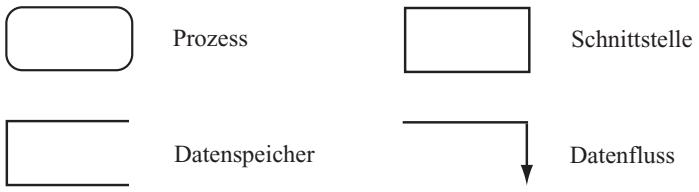


Abbildung 2.3: Legende zu Gane-Sarson-Datenflussdiagrammen

I/O-Teil

Abb. 2.4 stellt den Datenfluss im I/O-Teil dar. Es wird davon ausgegangen, dass die Metafunktionen über eine nicht-immersive GUI gesteuert werden. Wahrnehmung $W(R)$ und Handlung $H(S)$ werden nur für den schattierten Teil des VR-Interfaces diskutiert.

Eine Handlung des Benutzers S wird über geeignete Sensoren aufgenommen. Die rohen Sensordaten werden im Sensorauswertungs-Modul gefiltert

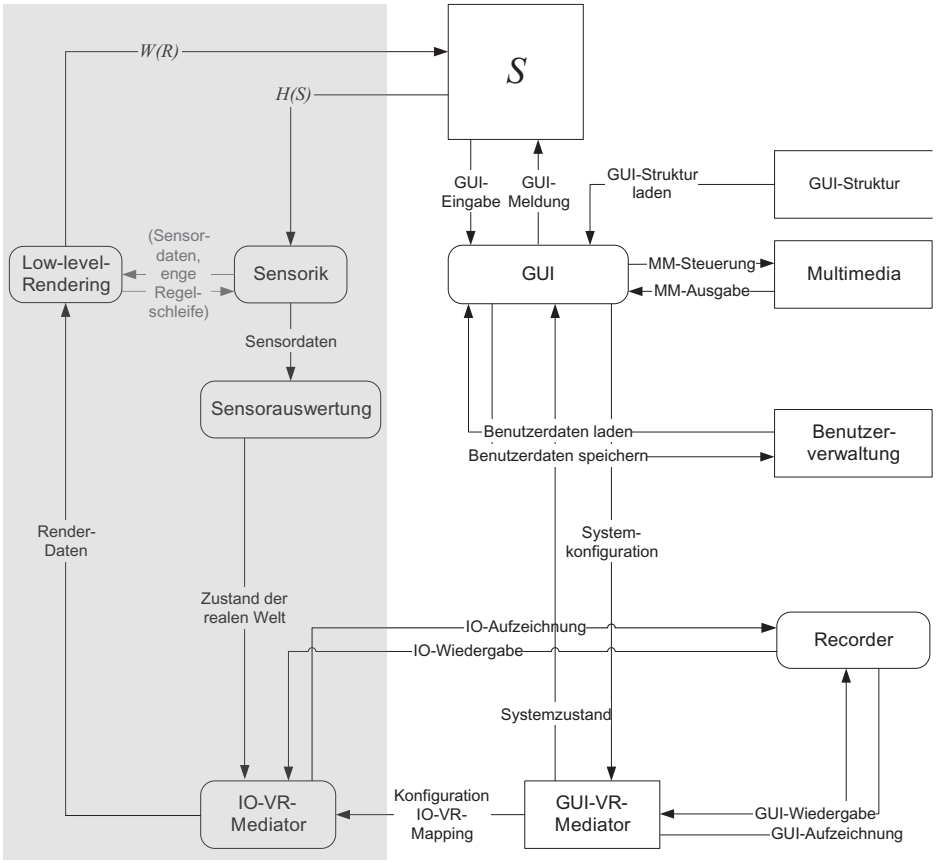


Abbildung 2.4: Datenfluss im I/O-Teil. Der Benutzer (das Selbst S) interagiert mit der Applikation über ein VR-Interface (schattierter Bereich) und eine klassische GUI-Steuerung. Mediatorklassen koppeln den IO-Teil von der VR-Berechnung (folgende Abbildung) ab.

und fusioniert. Die aufbereiteten Daten (Beispiele: Transformationsmatrizen, normierte Reglerwerte) werden an den IO-VR-Mediator übergeben und dort auf Objekte der virtuellen Welt abgebildet. Diese Abbildung kann modifiziert werden – etwa, wenn über die GUI ein neues Operationsszenario angewählt wird, bei dem die realen Instrumentmodelle anderen virtuellen Instrumenten entsprechen („Konfiguration IO-VR-Mapping“).

Die Render-Daten werden vom VR-Teil in einer Form geliefert, die den Ausgabegeräten bzw. deren Treibern angemessen ist. Bei dem Beispiel einer OpenGL-Visualisierung bedeutet dies einen Satz von Vertex-Arrays, Transformationsmatrizen und geeigneten Parametereinstellungen für das Lichtmodell (an dieser Stelle spielt die Fähigkeit des in Kapitel 3 beschriebenen MGE-Datenmodells eine große Rolle, zwei Gesichter annehmen zu können: eine vernetzte Struktur für die Berechnungen der VR-Seite und eine kompakte, blockweise Datenspeicherung für den Rendering-Vorgang).

Bei einem haptischen Renderer tritt ein Problem auf, das durch die direkte Verbindung von Sensorik und Low-level-Rendering gelöst wird („Sensordaten, enge Regelschleife“ in Abb. 2.4): Force-Feedback-Systeme erfordern durch die grosse Vibrationsempfindlichkeit des Tastsinns eine Update-Rate von 500-1000Hz [Srinivasan und Basdogan 1997]. In den meisten Fällen arbeitet die Berechnung im VR-Teil hierfür zu langsam. Force-Feedback-Treiber arbeiten daher häufig asynchron auf einer *intermediären Repräsentation* [Körner et al. 1999], die so lange als statisch angenommen wird, bis neue Simulationsdaten vorliegen. Zur Regelung der Kraft werden die Positionsdaten des Force-Feedback-Geräts ausgelesen.

Der GUI-VR-Mediator berechnet im Gegensatz zum IO-VR-Mediator keine Abbildung von Bezeichnern auf Objekte, sondern koppelt lediglich die GUI-Entwicklung softwaretechnisch ab. Der Preis für diese Vorgehensweise ist, dass alle Methodenaufrufe doppelt implementiert werden müssen: einmal im Mediator und ein zweites Mal in derjenigen Klasse, in der die eigentliche Funktionalität implementiert ist.

Als Beispiele sind in Abb. 2.4 die Kommunikation der GUI mit einer Multimedia-Komponente und einer Benutzerverwaltung dargestellt. Die Entwicklung dieser Komponenten ist aber unabhängig von der VRM-Bibliothek und der eigentlichen VR-Applikation.

Aufzeichnung und Übertragung

Aufzeichnung, Wiedergabe und Netzwerk-Übertragung von Sitzungen in der virtuellen Realität werden im I/O-Teil durchgeführt. Bei der Wiedergabe werden die Eingabedaten nicht von den Eingabegeräten, sondern vom Recorder bzw. über ein Netzwerk an die Mediatoren geliefert, die an den

VR-Teil die gleichen Steuersignale wie in der aufgezeichneten „echten“ Sitzung weitergeben. Es gibt zwei Alternativen zu diesem Ansatz: (1) die Übertragung oder Aufzeichnung der Ausgabesignale (vor allem des Video- und Audiostreams), wie sie z.B. in der Spring-Bibliothek [Montgomery et al. 2002] durchgeführt wird und (2) die Speicherung des internen Zustands der simulierten Welt. Problematisch ist in beiden Fällen die große Datenmenge, die bei (2) vor allem durch die Beschreibung von deformierbaren Objekten entsteht.

Im Gegensatz zu Alternative (1) bietet unsere Herangehensweise überdies die Möglichkeit, die Signale des Recorders bzw. einer zweiten VR-Applikation in die virtuelle Realität zu integrieren. Im Falle einer medizinischen Ausbildungssituation können die Instrumente der Ausbilderin dann beispielsweise als durchscheinende „Geisterobjekte“ dargestellt werden, deren Bewegungen der Auszubildende folgen muss.

Wenn über die GUI Systemeinstellungen vorgenommen werden können, die die virtuelle Welt beeinflussen, müssen auch diese aufgezeichnet werden (Beispiel: Wahl und Steuerung der Instrumente). Dagegen werden die Daten der engen Regelschleife des Force-Feedback-Treibers nicht aufgezeichnet. Die Kraftregelung eines Force-Feedback-Geräts hängt von der Reaktion des Benutzers ab, so dass es nicht möglich ist, den Verlauf der Positionen und Kräfte beim Abspielen ohne Veränderung wiederzugeben.

Die Antworten eines statistischen Orakels und der zeitliche Verlauf der Simulation werden ebenfalls vom Recorder erfasst.

VR-Teil

Funktionalität und Daten werden im VR-Teil auf *VR-Objekte*, *Simulationen* und *Szenarien* verteilt (Abb. 2.5).

Ein VR-Objekt speichert die Daten über den inneren Zustand eines Objekts (Beispiel: Gewebeeigenschaften) und seine Beziehungen zu anderen Objekten der virtuellen Welt (Beispiel: Verwandtschaftsbeziehung im Szenegraphen). Es ist aber nicht in der Lage, diese Daten selbstständig zu verändern, sondern überlässt diese Funktionalität den Simulationsklassen, die in Form von Assoziationen an die VR-Objekte gekoppelt sind.

Dies widerspricht zwar dem objektorientierten Paradigma in der Form „Objekt = Daten + zugehörige Algorithmen“, man erreicht durch Assoziationen anstelle von Vererbungen aber eine höhere Flexibilität: verschiedene Algorithmen können auf das gleiche Objekt zugreifen und kombiniert oder ausgetauscht werden. Falls mehrere Objekte an einer Simulation beteiligt sind (z.B. bei der Kollisionserkennung), ist die Formulierung natürlicher, wenn die Simulation als eigenständige Klasse existiert – analog zu physikalischen Gesetzen, die auf Objekte angewendet werden, etwa die mechanischen

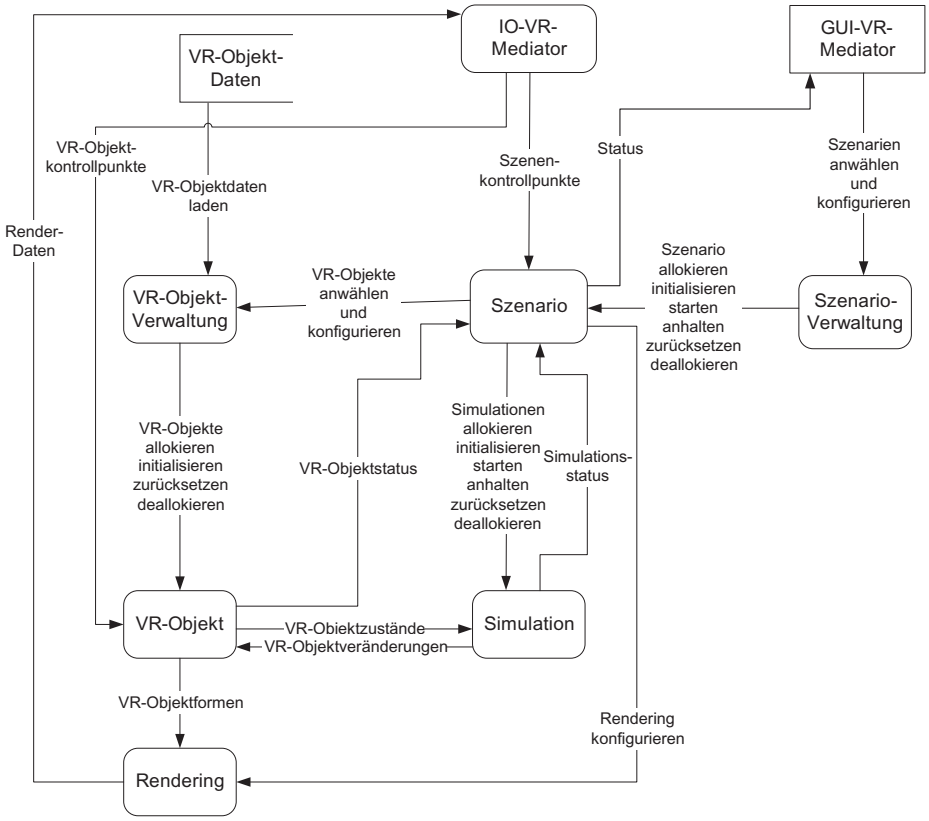


Abbildung 2.5: Datenfluss im VR-Teil. Die Funktionalität wird auf *VR-Objekte*, *Simulationen* und *Szenarien* verteilt. Der Zustand der simulierten Welt wird in den Objekten gespeichert, ihre Gesetze werden durch die Simulationen bestimmt. Ein Szenario konfiguriert und steuert eine bestimmte Situation.

Bewegungsgleichungen auf starre Körper mit Eigenschaften wie Trägheitsmoment, Masse oder Impuls.

Ein Szenario ist ein Punkt im Raum aller mit der Applikation simulierbaren virtuellen Welten, im Kontext von Operationssimulationen also z.B. eine bestimmte Pathologie. Aus programmtechnischer Sicht kapselt eine Szenario-Klasse alle Implementierungen, die für eine bestimmte Situation notwendig sind. Bestehende Objekte (Beispiel: eine deformierbare Netzhaut) können von der Objektverwaltung der Applikation angefordert und mit Simulationsalgorithmen aus der VRM-Bibliothek kombiniert werden.

Die beiden Verwaltungsklassen (VR-Objektverwaltung und Szenarioverwaltung) sind dafür zuständig, VR-Objekte und Szenarien anzulegen und an die Stellen weiterzugeben, an denen sie benötigt werden. Die VR-Objektverwaltung kann dazu die Objektdaten aus Dateien lesen (z.B. Geometrie, Gewebeparameter, Texturen) und zur internen Repräsentation zusammenstellen; um Wartezeiten zu vermeiden, können wahlweise alle verwalteten VR-Objekte beim Applikationsstart oder beim ersten Anfordern allokiert und initialisiert werden.

Die GUI-Befehle zur Auswahl und zur Steuerung einer bestimmten Operationssituation werden vom GUI-VR-Mediator an die Szenario-Verwaltung weitergegeben, die das entsprechende Szenario konfiguriert.

Der IO-VR-Mediator verteilt die von den Eingabegeräten ankommenden Informationen auf die Objekte (Beispiele: Transformationsmatrizen, interne Zustände wie den Öffnungswinkel der Scheren einer Pinzette) und Szenarien (globale Zustände, etwa Fokus und Brennweite eines über ein Fußpedal gesteuerten Stereomikroskops).

2.5 Zeitverhalten einer VRM-Applikation

2.5.1 Latenzquellen

Latenz in einem VR-System ist der zeitliche Abstand zwischen einem realen Ereignis und seiner wahrnehmbaren Wirkung in der virtuellen Realität. Wloka [1995] und Jacobs et al. [1997] teilen die Berechnung, die für diese Latenz verantwortlich ist wie folgt auf:

Sensorlatenz Die Zeit, die zwischen einem physikalischen Ereignis, der Reaktion des Sensors und der Ankunft der Daten beim Hostsystem verstreicht.

Applikationsabhängige Berechnungslatenz Der zeitliche Aufwand zur Berechnung der Funktion g (Gleichung 1.2).

Render-Latenz Die Zeit für die Aufbereitung der Render-Daten und deren Anzeige. Diese beiden Schritte werden unter Umständen unabhängig voneinander ausgeführt, nämlich dann, wenn die Anzeige der Render-Daten ein eigenständiger Prozess ist (Beispiele: die innere Schleife eines Force-Feedback-Renderers oder ein Abtastlücken-synchronisierter Grafiktreiber).

Im Allgemeinen ist die Gesamtlatenz eines Systems (*end-to-end-Latenz*) nicht die Summe dieser Latenzen. Sie kann geringer sein (durch parallele Verarbeitung), sie kann aber auch größer sein, wenn zusätzliche Latenzen durch die zeitliche Abstimmung der realen Handlungen und Wahrnehmungen und der einzelnen Prozesse des Systems entstehen. Der folgende Abschnitt beschäftigt sich mit Abtastphänomenen, die *Synchronisationslatenz* erzeugen; Abschnitt 2.5.4 diskutiert den Einfluss der Nicht-Echtzeit-Betriebssysteme Linux und Windows XP auf das Zeitverhalten einer VR-Applikation.

2.5.2 Synchronisationslatenz

Eine hohe Wiederholrate bedingt nicht zwingend eine geringe Latenz des Gesamtsystems: im – zugegebenermaßen etwas pathologischen – Beispiel eines Audio-Datenträgers liegt die Samplingrate bei einigen 10kHz, die Latenz zwischen der realen Handlung und ihrer Wahrnehmung kann aber je nach Aufnahmedatum mehrere Jahre betragen.

Umgekehrt ist eine geringe Latenz auch bei einer geringen Wiederholrate möglich, etwa, wenn Ereignisse in einem diskreten Zeitraster (z.B. alle 10s) stattfinden und synchron (ebenfalls alle 10s, aber jeweils kurz nach dem Ereignis, beispielsweise 1ms später) ausgewertet werden. Im Beispiel erreichte man bei einer Wiederholrate von 0,1Hz eine Latenz von 1ms.

Das zweite Beispiel zeigt, dass Synchronizität Latenz entscheidend verringern kann. Der Grund dafür ist, dass der Zeitpunkt, zu dem das Ereignis stattfindet, nicht gemessen werden muss. Es muss nur noch sein „Charakter“ (Stärke eines Signals, räumliche Position eines Objekts, usw. . .) festgestellt werden.

Synchronizität kann nur zur Verringerung der Latenz bei der Zusammenarbeit verschiedener Prozesse im VR-System ausgenutzt werden, nicht aber bei der Verbindung des Systems mit Handlungen oder Wahrnehmungen des Benutzers, von denen man nicht weiß, wann sie stattfinden.

Synchronisations-Latenz entsteht dadurch, dass Daten in einer Lücke zwischen zwei Messungen anfallen. Bei einer Abtastfrequenz ν und zwei aufeinanderfolgenden Messungen zu den Zeiten t_1 und t_2 (mit $t_2 = t_1 + \nu^{-1}$) beträgt die Synchronisations-Latenz T_{sync} im schlimmsten Fall

$$T_{sync,worst} = \frac{1}{\nu} \tag{2.1}$$

(wenn ein Ereignis wenig später als t_1 stattfindet), verschwindet im besten Fall (wenn ein Ereignis wenig früher als t_2 stattfindet) und ist im durchschnittlichen Fall

$$T_{sync} = \frac{1}{2\nu} \tag{2.2}$$

(da das Ereignis „irgendwann“ zwischen den Zeitpunkten t_1 und t_2 stattfindet).

Abb. 2.6 zeigt, wie die Synchronisationslatenz durch verschiedene zeitliche Ausführungsmuster beeinflusst werden kann. Dabei wird von einem System ausgegangen, in dem die Verarbeitung in drei unabhängig ausführbare Module aufgeteilt ist: Auslesen von Sensordaten (S_i), Berechnung der Systemreaktion (B_j) und Rendering (R_k).

Das Berechnungsmodul benötigt Sensordaten, das Render-Modul Berechnungsdaten; der Austausch dieser Daten erfolgt latenzfrei (bei paralleler Ausführung kann dies z.B. durch Double-Buffering der Daten realisiert werden). Der zeitliche Aufwand für die einzelnen Module sei T_S, T_B, T_R. Die Zeitachse geht in der Abbildung nach rechts, so dass aus der Länge der Kästchen die Ausführungszeit des Moduls hervorgeht. Untereinanderstehende Module werden parallel ausgeführt; die Indizes deuten an, wie oft das Modul schon ausgeführt wurde.

Wenn man von einer idealen Parallelisierung ausgeht (jedem parallel ausgeführten Modul steht ein eigener Prozessor zur Verfügung; es entstehen keine Wartezeiten durch Taskwechsel und Speicherzugriffskonflikte), dann ergibt sich der Datendurchsatz der einzelnen Alternativen aus der Fläche, die in Abb. 2.6 von den Kästchen bedeckt wird.

Die grau markierten Kästchen dienen dazu, den Weg eines gemessenen Ereignisses nachzuverfolgen. Die Latenz wird vom Beginn der Messung (Start der Ausführung von S_0) bis zum Ende desjenigen Rendermoduls R_k angegeben, das auf den Daten der Messung S_0 (bzw. in Fall (4) der Messungen S_0 und S_1) aufbaut.

Sequentiell Die Module werden hintereinander ausgeführt. Diese Konfiguration tritt zum Beispiel dann auf, wenn B und R auf den vorgegebenen Takt eines Sensors synchronisiert sind und von einer CPU innerhalb eines Threads oder Prozesses ausgeführt werden. Die Latenz ergibt sich aus der Summe der Berechnungszeiten für die einzelnen Module:

$$T_{seq} = T_S + T_B + T_R \tag{2.3}$$

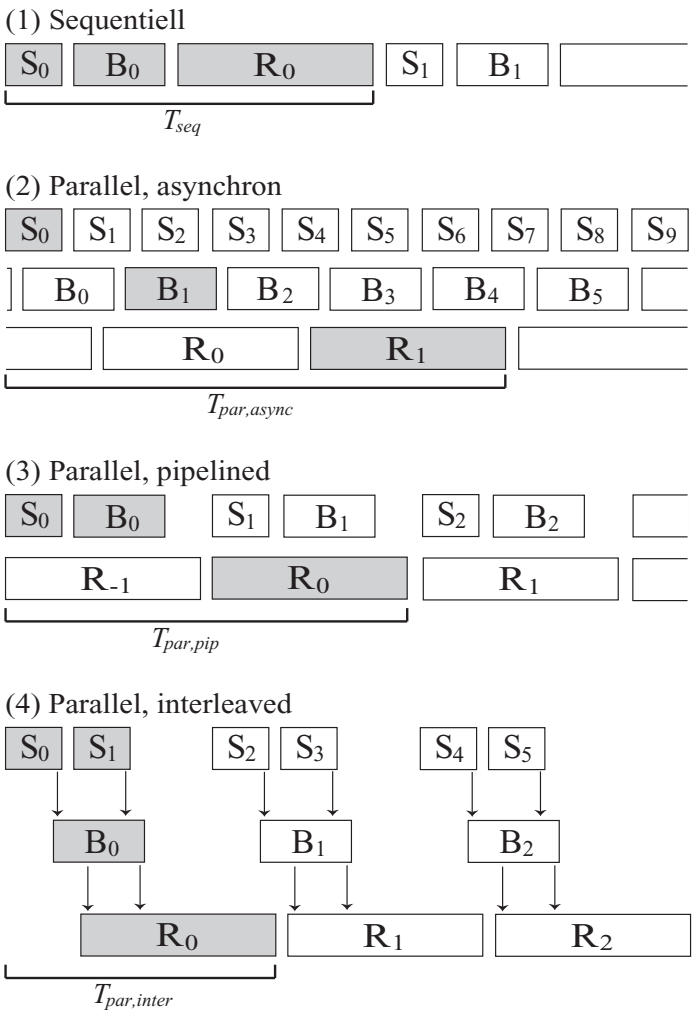


Abbildung 2.6: Verschiedene Synchronisationsmuster und deren Latenz. Die Module S_i lesen Sensordaten, die B_j führen Berechnungen durch und die R_k rendern das Ergebnis. T bezeichnet die Latenz zwischen einer Messung und dem Rendern ihrer Wirkung. Die Zeitachse geht nach rechts, die Indices geben die Iteration des entsprechenden Moduls an.

Parallel, asynchron Jedes Modul wird in einem eigenen, nicht synchronisierten Thread oder Prozess ausgeführt. Parallel-asynchrone Verarbeitung ist nicht zu vermeiden, wenn die beteiligten Subsysteme auf eine feste Frequenz angewiesen sind: ein optisches Trackingsystem mit PAL-Kameras, eine Visualisierungskomponente, die ihre Anzeige nur in der Austastlücke des Monitors aktualisieren kann (*VSync*) oder ein Force-Feedback-Gerät, dessen Treiber eine eigene Regelschleife mit fester Updaterate besitzt.

Berechnungs- und Render-Module greifen auf die Daten des letzten, vollständig ausgeführten Sensor- bzw. Berechnungsmoduls zu. Im Gegensatz zu den synchronen Beispielen variiert die Latenz während der Ausführung. Mit Gleichung 2.2 erhält man als durchschnittliche Latenz:

$$T_{par,async} = T_S + T_B + T_R + \underbrace{\frac{1}{2}(T_S + T_B)}_{T_{sync}} \tag{2.4}$$

Obwohl der Datendurchsatz höher ist als im sequentiellen Fall, hat auch die Latenz zugenommen. Dies liegt daran, dass die Daten meist nicht sofort verarbeitet werden, sondern im Schnitt erst nach T_{sync}. Wenn man diese Zeit minimieren möchte, muss man $T_S \ll T_B \ll T_R$ wählen; in diesem Fall werden aber die Ergebnisse einiger Berechnungssschritte von S und B nicht an die nächste Stufe weitergereicht (in Abb. 2.6 (2) z.B. S_1, B_2), so dass Rechenzeit umsonst aufgewendet wurde.

Die Wahrscheinlichkeit $P_{S,B}$, dass ein Ergebnis einer Messung S von einer Berechnung B weiterverarbeitet wird, ist[2]

$$P_{S,B} = \begin{cases} \dfrac{T_S}{T_B} & : \quad T_S < T_B \\ 1 & : \quad T_S \geq T_B \end{cases} \tag{2.5}$$

Analog für $P_{B,R}$. Minimale Latenz und maximaler nutzbarer Datendurchsatz ($T_R \leq T_B \leq T_S$) schließen sich daher aus.

Parallel, pipelined Um die Synchronisationslatenz des vorhergehenden Beispiels zu vermeiden, findet die Berechnung hier in einzelnen (syn-

[2]Erklärung für Gleichung 2.5: Sei $T_S < T_B$. Eine Berechnung gehe von t_1 bis $t_2 = t_1 + T_B$. Dann werden die Daten einer Messung S_i genau dann abgearbeitet, wenn S_i in einem Zeitintervall $(t_2 - T_S, t_2)$ endet. Dessen Länge verhält sich zur Dauer einer Berechnung wie T_S/T_B. Sei $T_S \geq T_B$. Der Fall, dass mehr als eine Messung in demselben Berechnungsintervall endet, kann nicht eintreten. Daher wird jedes Messergebnis verwendet.

chronen) Stufen einer Pipeline statt. Bei dem in Abb. 2.6-(3) dargestellten Beispiel ist die Sensor-Berechnungs-Stufe der Pipeline kürzer als die Render-Stufe, so dass Rechenzeit verschenkt wird. In diesem Fall ist daher die Latenz (bei beinahe doppelter Rendering-Frequenz) etwas größer als im sequentiellen Fall:

$$T_{par,pip} = 2 \cdot T_R \tag{2.6}$$

Im allgemeinen Fall einer Pipeline mit n Berechnungsstufen und Zeiten $T_1, \ldots, T_n$ beträgt die Latenz

$$T_{par,pip,n} = n \max_{i=1,\ldots,n} T_i \tag{2.7}$$

Parallel, interleaved Im günstigsten Fall (Abb. 2.6-(4)) können die Prozesse so angeordnet werden, dass sie ihre Daten genau zum benötigten Zeitpunkt erhalten: Die Berechnung B_0 startet, sobald die Sensordaten S_0 zur Verfügung stehen. Die Zeit für B_0 wird dazu genutzt, die Sensordaten S_1 einzulesen, die an passender Stelle ebenfalls für die Berechnung verwendet werden (Beispiel: S_0 wird zur Berechnung einer Gewebereaktion verwendet, S_1 für die Darstellung einer beweglichen Lichtquelle). Der Render-Vorgang R_0 wird gestartet, sobald die ersten berechneten Daten vorliegen (Beispiel: das simulierte Gewebe wirft keinen Schatten, so dass der Render-Pass zur Schattenberechnung schon stattfinden kann, während die Gewebedeformation berechnet wird).

Latenz und Durchsatz hängen von dem Grad an Parallelität ab, der erreicht werden kann. Ein hoher Durchsatz muss bei diesem Synchronisationsmuster nicht zwingend mit einer großen Latenz erkauft werden.

Die beiden ersten Fälle – serielle und parallel-asynchrone Verarbeitung – sind unter den betrachteten Aspekten Latenz und (effektivem) Datendurchsatz am ungünstigsten. Trotzdem haben sie eine gewisse Bedeutung: serielle Verarbeitung ist softwaretechnisch häufig einfacher zu handhaben. Parallel-asynchrone Verarbeitung ist nicht zu umgehen, wenn einzelne Prozesse nicht synchronisiert werden können.

Die Fälle Abb.2.6, (3) und Abb.2.6, (4) werden für Systeme mit festen, Austastlücken-synchronen Frameraten von Rohlf und Helman [1994] diskutiert. Wegen der zusätzlich entstehenden Latenz wird bei einer VRM-Applikation meist auf die Synchronisation mit dem VSync-Signal des Monitors verzichtet. Bei schnellen Bewegungen können dadurch „Nähte“ entstehen, wenn gleichzeitig Teile von unterschiedlichen Frames sichtbar sind.

2.5.3 VRM-Klassen zur Ablaufsteuerung

Um dabei zu helfen, nebenläufige oder funktional unterschiedliche Prozesse auf übersichtliche Weise zu implementieren und in den oben beschriebenen Synchronisationsmustern anzuordnen, stellt die VRM-Bibliothek Basisklassen für die Ablaufsteuerung zur Verfügung. Eine bestimmte Berechnungseinheit ist ein *Modul*. Sequentiell ausgeführte Module werden als verkettete Liste in einer *Sequenz* angeordnet. Wenn alle Module einer Sequenz in einen Stop-Zustand übergegangen sind, wird die Sequenz angehalten. Wenn mehrere Sequenzen instanziiert werden, wird jede Sequenz in einem eigenen Thread ausgeführt. Ein bestimmtes Modul kann in verschiedenen Sequenzen oder mehrfach in einer Sequenz referenziert werden.

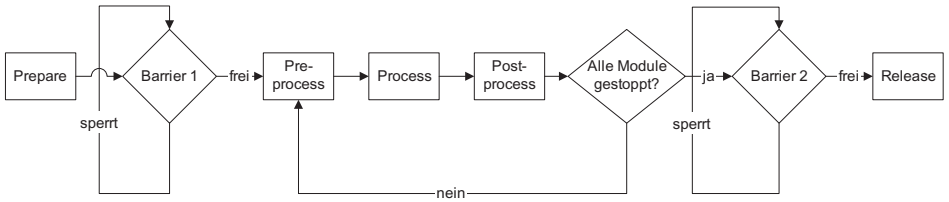


Abbildung 2.7: Ausführungsstufen einer Sequenz. Die Barrieren sperren so lange, bis alle Sequenzen die jeweils vorangegangene Stufe abgeschlossen haben.

Nach dem Start der Applikation werden die einzelnen Sequenzen in verschiedenen Stufen abgearbeitet (Abb. 2.7). In jeder Stufe werden entsprechende Methoden der Module (*Prepare(), Preprocess(), Process(), Postprocess()*) in der durch die Modulliste definierten Reihenfolge abgearbeitet. Die Trennung in drei verschiedene Process-Stufen wurde durchgeführt, um dem Programmierer dabei zu helfen, innerhalb eines Moduls die Ausführung zu strukturieren. Die Dreiteilung hat sich in der Praxis als sinnvoll herausgestellt, da sich viele Prozesse auf natürliche Weise in eine Vorbereitungs-, Verarbeitungs- und Nachbereitungsstufe aufteilen lassen.

Um verschiedene Sequenzen zu synchronisieren, stehen spezielle Module zur Verfügung, z.B. Barrieren und Semaphoren. Eine Barriere blockiert solange, bis der Ausführungspfad aller Sequenzen an der Barriere angekommen ist. Mit Semaphoren können Bereiche geschützt werden, die nicht von verschiedenen Sequenzen zugleich ausgeführt werden sollen: Referenzen auf eine Semaphore werden dazu vor und hinter die zu schützenden Teilsequenzen von Modulen gesetzt. Nur der Ausführungspfad je einer Sequenz kann zur gleichen Zeit den Bereich zwischen den beiden Referenzen betreten.

Beispiele: Abb. 2.6, (1) besteht aus einer Sequenz

$$SEQ_0 = (S, B, R)$$

Abb. 2.6, (2) kann mit den drei Sequenzen

$$SEQ_0 = (S), SEQ_1 = (B), SEQ_2 = (R)$$

formuliert werden, der synchronisierte Ablauf in Abb. 2.6, (3) wird mit einer Barriere BAR erreicht:

$$SEQ_0 = (S, B, BAR), SEQ_1 = (R, BAR)$$

Abb. 2.6, (4) ist ebenfalls mit Hilfe von Barrieren realisierbar, allerdings müssen dazu die Berechnungs- und Render-Schritte in mehrere einzelne Module unterteilt werden.

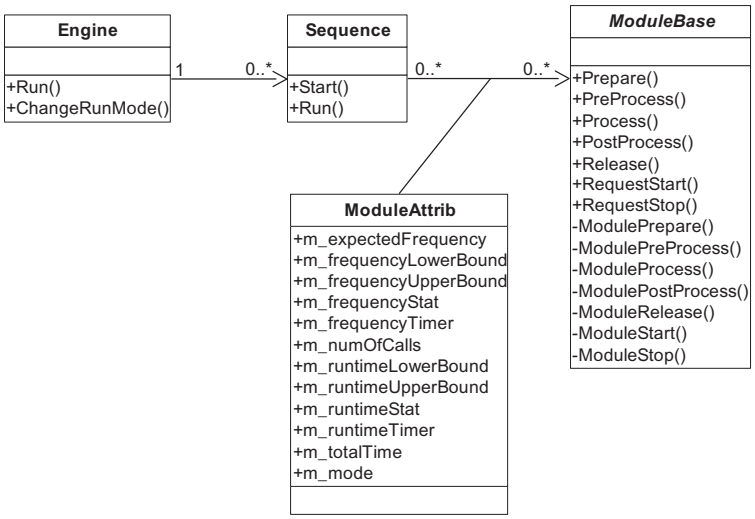


Abbildung 2.8: Ablaufsteuerungs-Klassen (vereinfachte Darstellung)

Abb. 2.8 stellt die statischen Beziehungen zwischen den Ablaufsteuerungs-Klassen dar. Mit der Singleton-Klasse *Engine* werden die Module und Sequenzen verwaltet und die Thread-Steuerung initialisiert. Nach der Definitions- und Initialisierungsstufe startet die Engine die Threads für die einzelnen Sequenzen, die dann die in Abb. 2.7 dargestellten Stufen durchlaufen. Der Hauptthread kann dabei entweder einer der Sequenzen zugeordnet werden oder wieder an die Stelle des Engine-Aufrufs zurückkehren. Dadurch wird die Zusammenarbeit mit externen Bibliotheken vereinfacht,

die im Hauptthread ausgeführt werden müssen oder eine eigene, blockierende Ausführungsschleife besitzen.

Während die Engine-Klasse und die Sequenzen auf Applikationsebene nicht explizit spezialisiert oder instanziiert werden, wird die Funktionalität der Module einer VRM-Applikation über Spezialisierungen der abstrakten Klasse *moduleBase* definiert.

Zur Frequenz- und Laufzeitkontrolle von Modulen können Mittelwerte und erlaubte Abweichungen angegeben werden. Werden diese von einem Modul verletzt, werden die virtuellen Methoden *RuntimeGuard()* und *FrequencyGuard()* des Moduls aufgerufen, die Ablaufsteuerung selbst greift aber nicht in die Ausführung des Moduls ein.

Der hier gewählte Ansatz zur Fomulierung von nebenläufigen Prozessen dient vor allem dazu, schnell und einfach verschiedene Synchronisationsmuster wie in Abb. 2.6 definieren zu können. Für die feingranulierte Parallelisierung eines bestimmten Verfahrens (z.B. einer Feder-Masse-Simulation) wird man dagegen die Nebenläufigkeit auf einer niedrigeren Ebene definieren.

2.5.4 Zeitverhalten von Linux und Windows XP

Der Verwaltungsaufwand der VRM-Ablaufsteuerung sowie das Verhalten des Betriebssystems erhöhen ebenfalls die Latenz einer VR-Applikation. Die Messung des Ablaufsteuerungs-Verhaltens wird in Anhang B beschrieben. Der zeitliche Aufwand für die Ablaufsteuerung liegt pro Iteration im Bereich von 0,1ms und ist daher vernachlässigbar.

Im folgenden wird untersucht, ob und unter welchen Voraussetzungen die beiden Nicht-Echtzeitbetriebssysteme Linux und Windows XP für den Einsatz in einem VR-System geeignet sind.[3] Dazu werden zwei Kriterien herangezogen: die zeitliche Regelmässigkeit, mit der eine Applikation ausgeführt wird und das Systemverhalten bei der nebenläufigen Ausführung.

Als Beispiel wird eine einfache Berechnung verwendet, die in der Process()-Methode eines Moduls M von der VRM-Ablaufsteuerung ausgeführt wird (Abb. 2.9). Um verfälschende Effekte durch Speicherzugriffe zu minimieren, werden rechenintensive Operationen ($\cos(x)$ und $\sqrt{x}$) durchgeführt und nur wenige Variablen verwendet.

Die Tests wurden auf einem Doppel-Pentium-III-PC mit 450MHz Prozessortakt unter Windows XP (Compiler: Microsoft Visual Studio 6.0) und Linux (Kernel-Version 2.4, Compiler: GCC Version 3.2) mit den Standard-Compilereinstellungen durchgeführt.

[3] Es geht in diesem Abschnitt aber *nicht* um einen direkten Vergleich der beiden Betriebssysteme.

```
  void M::Process () {
     double l_sum;
     for (int i=m_iterations; i>0; --i )
         l_sum += cos(sqrt(i*i+(i+1)*(i+1)+(i+2)*(i+2)));
}
```

Abbildung 2.9: Process-Methode des Beispielmoduls M

Zeitliche Schwankungen

Für Abb. 2.10 wurden fünf Instanzen des Moduls M während einer Zeitspanne von 10s sequentiell ausgeführt. Horizontal ist die Zeit, vertikal die Periodendauer (die Zeit zwischen zwei Process()-Aufrufen) aufgetragen. Unter Windows XP wurde die Testapplikation mit zwei unterschiedlichen Prozessprioritäten gestartet (normal und Realtime; letztere ist mit RT gekennzeichnet). Die Diagramme der rechte Spalte stellen den gleichen Datensatz wie das jeweilige linke Diagramm dar, allerdings in einem 1ms großen Ausschnitt der Periodendauer-Achse. Die Anzahl der Iterationen (`m_iterations`) in der Process()-Schleife von M betrug 50.000.

Die Werte $P_{\Delta t}$ geben die Wahrscheinlichkeiten an, dass die Periodendauer in einem geeignet gewählten Fenster der Größe $\Delta t = 1$ms (linke Spalte) bzw. $\Delta t = 0,2$ms liegt. Zwischen 95% (Linux) und 100% (XP RT) der Schwankungen bleiben im für eine VR-Anwendung eher relevanten Ein-Millisekunden-Fenster. Das Zeitverhalten unter Linux ließe sich noch verbessern: es ist eine periodische Störung erkennbar, die einmal pro Sekunde die Periodendauer um etwa 2ms erhöht und durch eine sorgfältige Konfiguration des Betriebssystems sicherlich vermieden werden könnte.

Einfluss von Kontextwechseln

Wenn mehr Threads oder Prozesse als Prozessoren zur Verfügung stehen, wird das Zeitverhalten der Applikation vom *scheduler* des Betriebssystems abhängig. Bei einem Thread-Kontextwechsel geht vor allem dadurch Zeit verloren, dass Cache-Inhalte unbrauchbar werden.

Um den Aufwand für Kontextwechsel zu beschränken, wird von XP und Linux jedem Thread ein Zeitfenster zur Verfügung gestellt, das erheblich größer als die Taktzeit der Scheduler ist, die bei beiden Betriebssystemen bei 10ms liegt.[4] Bei dem Versuch von Abb. 2.11 wurde die Größe des Zeitfensters für drei Konfigurationen (XP, XP mit Realtime-Prozess-Priorität,

[4]Wie bereits erwähnt, konnte Linux nur in der Kernel-Version 2.4 getestet werden. Im experimentellen Kernel 2.5 wurde die Scheduler-Taktzeit auf 1ms gesenkt.

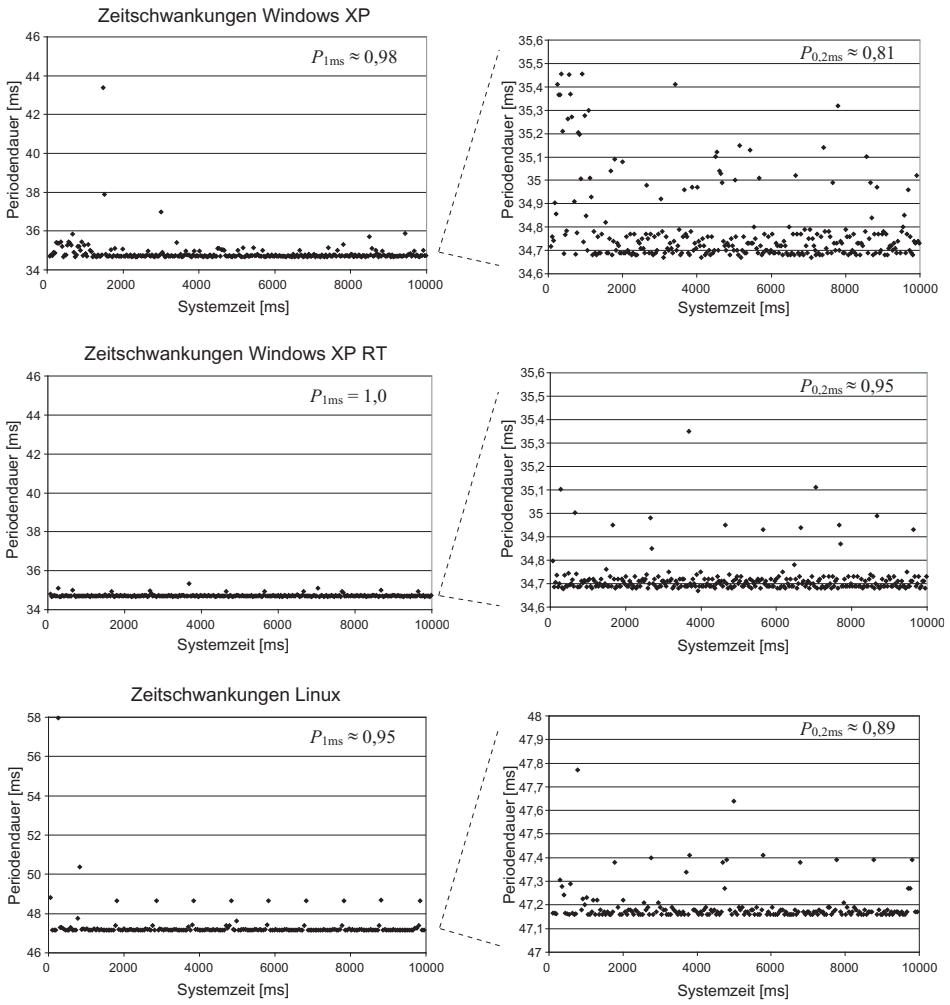


Abbildung 2.10: Messung der zeitlichen Schwankungen bei der Ausführung. Aufgetragen ist die Systemzeit gegen die Zeit zwischen zwei Process()-Aufrufen des Beispielmoduls. In der rechten Spalte ist ein vergrößerter Ausschnitt der Zeitachse dargestellt. Ebenfalls angegeben sind die Wahrscheinlichkeiten dafür, dass die Periodendauer in einem geeignet gewählten Fenster der Größe 1,0ms bzw. 0,2ms liegt.

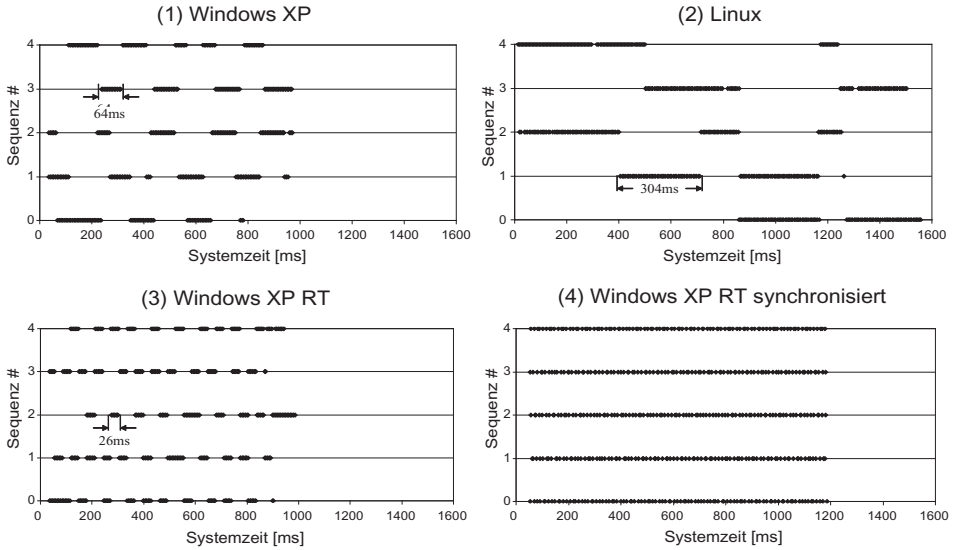


Abbildung 2.11: Zeitverhalten bei Kontextwechseln. Fünf parallele Sequenzen wurden in verschiedenen Konfigurationen auf einem Prozessor ausgeführt. Aufgetragen ist die Systemzeit gegen die Nummer der gerade ausgeführten Sequenz. Wenn der Kontextwechsel dem Scheduler überlassen wird (1-3), dann vergehen zwischen 26ms (3) und 304ms (2), bevor eine Sequenz die Ausführung abgeben muss. Bei synchronisierter Ausführung (4) wird mehr Zeit benötigt.

Linux) gemessen. Dazu wurde je eine Instanz von M in 5 Sequenzen ausgeführt:

$$S_i = (M_i); i = 1, \ldots, 5$$

Jede Instanz wurde über 100 Perioden mit jeweils 10.000 Schleifendurchläufen ausgeführt. Der Zeitpunkt jedes Aufrufs der Process()-Methode einer M-Instanz wurde mitprotokolliert und ist in der Abbildung als Punkt auf der Zeitachse der entsprechenden Sequenz aufgeführt.

Die typischen Größen eines Zeitfensters sind in Abb. 2.11 eingezeichnet. Es besteht ein deutlicher Unterschied zwischen den einzelnen Konfigurationen. Mit etwa 300ms gesteht Linux jedem Thread das größte Zeitfenster zu (Abb. 2.11-(2)); bei Windows XP (2.11-(1)) muss ein Thread eines Prozesses normaler Priorität typischerweise nach 60ms seine Ausführung abgeben, während bei einen XP-Realtime-Prozess (2.11-(3)) schon nach etwa 20ms ein Kontextwechsel veranlasst wird. Vor allem in den ersten beiden Konfigurationen schwankt die Größe des Zeitfensters sehr stark.

In allen Konfigurationen wird durch die parallele, asynchrone Ausführung auf einem Prozessor die Latenz der einzelnen Module erheblich vergrößert. Aufgrund dieses Ergebnisses werden in einer VRM-Applikation höchstens soviele parallele, asynchrone Sequenzen gestartet wie CPUs zur Verfügung stehen.

Durch Synchronisierung der Prozesse lässt sich das Problem lösen, wie Abb. 2.11-(4) zeigt. Hier wurde jede Sequenz S_i um die Barriere BAR ergänzt:

$$S'_i = (M_i, BAR); i = 1, \ldots, 5$$

Dadurch wird der Scheduler dazu gezwungen, nach jeder Periode zu den noch wartenden Modulen umzuschalten. Damit wird die Kontextwechsel-Zeit auf die Periodendauer reduziert. Der Zeitbedarf der Testapplikation erhöht sich allerdings gegenüber dem unsynchronisierten Fall um etwa 20%, da Prozessorkapazität beim Warten an der Barriere verloren geht.

2.6 Zusammenfassung und Diskussion

In diesem Kapitel wurde die VRM-Bibliothek für den Bau virtueller Realitäten in der Medizin vorgestellt. Zugunsten von Flexibilität und Geschwindigkeit bleibt die Bibliothek auf einem vergleichsweise niedrigen Abstraktionsniveau und verzichtet beispielsweise auf eine Skriptsprache zur Beschreibung von VR-Szenen. Sie stellt lose gekoppelte Module zur Datenrepräsentation, zur Kollisionserkennung und Simulation, zur Ansteuerung von Sensoren und Ausgabegeräten und zur Ablaufsteuerung zur Verfügung.

Für den Bau einer Applikation, die auf der VRM-Bibliothek aufbaut, wird eine Architektur beschrieben, mit deren Hilfe die Entwicklung von VR-Interface, Simulationsalgorithmen und GUI-Steuerung logisch und softwaretechnisch entkoppelt werden kann. Dadurch wird eine schnelle und wenig fehleranfällige Softwareentwicklung ermöglicht. Die Separierung des VR-Interfaces ermöglicht eine einfache Integration zusätzlicher oder alternativer Datenquellen wie beispielsweise einem Aufzeichnungs- und Wiedergabemechanismus oder der Netzwerkanbindung für verteilte VR.

Um Präsenz zu erzeugen, muss die Latenz einer VRM-Applikation unterhalb der menschlichen Wahrnehmungsgrenze von etwa 50ms liegen. Es wird gezeigt, dass die zeitlichen Unregelmässigkeiten bei der Ausführen eines Programms unter Windows XP und Linux deutlich unter diesem Wert liegen. Unter diesem Aspekt sind die beiden Betriebssysteme daher als Umgebung von VRM-Applikationen verwendbar.

Probleme entstehen aber bei beiden Systemen, sobald Kontextwechsel notwendig werden. Ihre Latenz liegt in der Größenordnung von 50ms oder darüber, so dass die VR-Echtzeitbedingung nicht mehr erfüllt wird. Daraus ergibt sich die typische Struktur einer VRM-Applikation: auf dem PC werden höchstens so viele asynchrone, nebenläufige Prozesse gestartet wie Prozessoren zur Verfügung stehen. Aufwändige Operationen werden auf spezialisierte Subsysteme ausgelagert, die autark arbeiten können – etwa eine Grafikkarte, die über DMA Daten aus dem Hauptspeicher überträgt und verarbeitet; ein optisches Trackingsystem, das die Kameradaten selbstständig auswertet oder ein Hardwarebaustein, der die schnelle Regelungsschleife eines Force-Feedback-Geräts ausführt.

3

Die MGE-Datenstruktur

Die Veränderung der Realität wird von der Funktion g in Gleichung 1.2 beschrieben. Einer ihrer Eingabewerte ist der aktuelle Zustand der Realität. Ein VR-System muss daher über angemessene Repräsentationen der simulierten Realität verfügen.

Übliche VR-Systeme verwenden *Szenegraphen* für die Darstellung von geometrischen Beziehungen zwischen Objekten, ihrer inneren Struktur liegt dagegen kein abstraktes Repräsentationsformat zugrunde. Das ist besonders dann unbefriedigend, wenn mit Simulationsalgorithmen auf die Objekte zugegriffen werden soll, um sie zu deformieren.

Die graphbasierte MGE-Datenstruktur wurde entwickelt, um eine universell verwendbare Repräsentation für die virtuelle Welt und ihre Objekte zur Verfügung zu haben. Sie wird in den VRM-Applikationen sowohl als Szenegraph als auch als Datenmodell für die Gewebesimulation eingesetzt.

Nach einer Diskussion über graphbasierte Modelle im Allgemeinen (Abschnitt 3.1) wird in Abschnitt 3.2 die MGE-Datenstruktur definiert. Abschnitt 3.3 beschreibt ihre Implementierung in der VRM-Bibliothek.

3.1 Stand der Forschung

Eine graphähnliche Struktur wird immer dann verwendet, wenn der Zusammenhang zwischen diskreten Objekten beschrieben werden soll (Beispiel: die Verbindungen zwischen den einzelnen Punktmassen eines Feder-Masse-Modells, nicht aber das Interferenzfeld zweier quantenmechanischer Wellenfunktionen). Definition 3.1 zeigt, dass Graphen und Relationen eng verwandt sind.

Definition 3.1 (Graphen und gerichtete Graphen) *Ein* gerichteter Graph *ist ein Paar* (V, E) *(von* Knoten *und* Kanten*), wobei* V *eine Menge und* E *eine binäre Relation auf* V *ist (d.h.* $E \subseteq V \times V$*).*

Man schreibt für ein Paar (v_1, v_2) *aus* E *auch* $v_1 \rightarrow v_2$*.*

Bei einem ungerichteten Graphen *ist* E *symmetrisch (d.h.* $\forall v_1, v_2 \in V : (v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$*).*

Aus der Perspektive der Graphentheorie spielt die Art der Elemente in V keine Rolle. Die Bedeutung eines Graphen entsteht durch die Bildung von E aus $V \times V$.[1] Für die Repräsentation von Informationen ist dies im Allgemeinen ungenügend. Um verschiedene Arten von Beziehungen darstellen zu können, *beschriftet* man daher die Kanten. Mit einer Menge L von *Kantenbezeichnern* gilt nun $E \subseteq V \times V \times L$. Den Elementen von V und L werden Eigenschaften zugeordnet, die den atomaren Eigenschaften der zu repräsentierenden Objekte entsprechen. Auf diese Weise erhält man ein *semantisches Netz*, das zuerst von Quillian [1968] zur Wissensrepräsentation eingesetzt wurde.

Gute Darstellungen des großen Gebiets der Wissensrepräsentation von Datenbanken und KI-Systemen geben z.B. Bürsner [1997] und Richter [1989]; ein klassisches wissensbasiertes System, das auf semantischen Netzen aufbaut, ist KL-ONE [Brachmann und Schmolze 1985]. Graphbasierte Formate in diesem Bereich ergänzen semantische Netze um leistungsfähige Abfragesprachen und die Möglichkeit zur Strukturierung der Knoten- und Kantenmenge, etwa durch Typisierung, Vererbung und einer in das Repräsentationsformat integrierten Beschreibung der inneren Struktur von Knoten oder Kanten (*nested-graph models*).

Im Gegensatz dazu verwenden VR-Systeme (s. Anhang A) meist gerichtete azyklische Graphen (ohne Kantenbezeichner), die *Szenegraphen*.[2] Die Knoten eines Szenegraphen repräsentieren geometrische Primitive (z.B. Dreiecksnetze) und Zustandsänderungen des Rendering-Systems, seine Kanten die Reihenfolge, in der die Knoten während des Rendering-Prozesses durchlaufen werden (meist depth-first; beim Backtracking werden die Zustandsänderungen wieder rückgängig gemacht). Die Knoten sind meist

[1] Trotz der einfachen Konstruktion hat man mit Graphen ein mächtiges Werkzeug zur Hand, um strukturelle Zusammenhänge zu formulieren. Beispielsweise konstruiert Aczel [1988] mit Hilfe von Graphen ein Axiomensystem für nicht-wohlfundierte Mengen. „Enthalten sein“ wird dabei als Relation zwischen Graphknoten aufgefasst. „Die Menge, die sich selbst als einziges Element enthält“ entspricht dann dem gerichteten Graphen $(\{A\}, \{A \rightarrow A\})$.

[2] In der AFML-Szenenbeschreibungssprache [Beier 1997; 1998] werden Szenegraphen sogar auf Bäume einschränkt, so dass das gleiche Objekt nicht an verschiedenen Stellen der Szene verwendet werden kann. Dadurch bleibt das *Lokalitätsprinzip* [Wißkirchen 1990] erhalten; lokale Veränderungen im Szenegraphen wirken sich nur lokal in der Szene aus.

in unterschiedliche Knotentypen klassifiziert, die für die Art ihrer Zustandsänderung stehen, etwa Material-, Transformierungs- und Gruppierungsknoten, Knoten zur Spezifizierung von Kameraperspektiven und der Auswertung von Sensordaten.

Häufig korrespondiert die hierarchische Beziehung zwischen den Objekten in einem Szenegraphen mit ihrer räumlichen Anordnung. In diesem Fall kann der bestehende Graph zusätzlich eine Bounding-Box-Hierarchie für Sichtbarkeits- und Kollisionstests (Kap. 4) repräsentieren. Als Datenmodell für darüber hinausgehende Simulationen physikalischer Vorgänge sind Szenegraphen aufgrund ihrer Einschränkungen (azyklisch, Kanten repräsentieren Zustandsänderungen) aber nicht geeignet.

3.2 Multigraphelemente

Das Datenformat muss folgenden Anforderungen genügen:

Einheitliche Struktur für Rendering und Simulation Die Entwicklung, der Austausch, die Kombination und die Erweiterung von Algorithmen zum Rendering und zur Simulation werden vereinfacht, wenn die Algorithmen auf dem gleichen Datenformat arbeiten. Das Datenformat muss daher allgemein genug sein, um verschiedenartige Strukturen auf natürliche (d.h. dem jeweiligen Algorithmus angemessene) Weise zu repräsentieren.

Schneller sequentieller Datenzugriff Sequentieller Zugriff ist dann notwendig, wenn eine Struktur unter einer bestimmten Sicht vollständig durchlaufen wird (Beispiele: Berechnung der Kräfte in allen Federn eines Feder-Masse-Modells; Übertragen der aktualisierten Geometriedaten auf die Grafikkarte).

Schnelle Navigation, schneller wahlfreier Zugriff Nicht jeder Simulationsalgorithmus kann sequentiell abgearbeitet werden (Beispiel: Kollisionserkennung zwischen Bounding-Box-Hierarchien, Kapitel 4), so dass auch wahlfreier Zugriff auf einzelne Elemente der Datenstruktur ohne Suche möglich sein muss. Um die Zusammenarbeit zwischen verschiedenen Algorithmen zu ermöglichen, muss auf lokalen Zusammenhängen navigiert werden können (Beispiel: Eine Kollisionserkennung liefert ein bestimmtes Dreieck zurück. Dessen Ecken sind gleichzeitig Punktmassen einer Feder-Masse-Struktur. Auf die benachbarten Federn muss eine bestimmte zusätzliche Kraft ausgeübt werden, um eine angemessene Reaktion auf die Kollision zu erzeugen).

Veränderbarkeit Verändern, Löschen und Hinzufügen von strukturellen Eigenschaften und Attributen muss mit geringem Zeitaufwand möglich sein.

Ausgehend von diesen Forderungen wurde die MGE-Datenstruktur entwickelt. Um die Forderung nach Allgemeinheit zu erfüllen, baut sie auf semantischen Netzen auf.

Definition 3.2 (MGE-Datenstruktur) *Eine* MGE-Datenstruktur *ist ein Tripel* (V, E, L)*, bestehend aus einer Menge* V *von* Multigraphelementen (MGEs)*, einer Menge* L *von* Kantenbezeichnern *und einer Menge* E *von* Kanten *mit:*

1. $E \subseteq V \times V \times L$.
2. *Die Elemente der Menge* $\Gamma(v, l) := \{v' \in V | (v, v', l) \in E\}$ *heißen die* l-Nachbarn von v*. Auf jeder solchen Menge ist eine Totalordnung definiert. Die* Nachbarn von v *sind die Elemente von* $\Gamma(v) := \bigcup_{l \in L} \Gamma(v, l)$.
3. *Auf* L *ist eine Funktion* $r : L \to L$ *definiert, die symmetrisch*[3] *(* $\forall x : r(x) = y \Rightarrow r(y) = x$ *) und antireflexiv (* $\forall x : r(x) \neq x$ *) ist.* $r(x)$ *heißt* Referenzbezeichner von x*. Für jede Kante* $e = (v_1, v_2, l) \in E$ *existiert eine* Referenzkante $\bar{e} = (v_2, v_1, r(l)) \in E$.

Im Gegensatz zu semantischen Netzen wird in 3.2-(2) eine Ordnung auf den Knotennachbarn definiert. Bei der Repräsentation geometrischer Objekte kann dies sehr nützlich sein, beispielsweise ergibt sich aus der Reihenfolge der drei Eckpunkte eines Dreiecks, in welche Richtung die Normale seiner „Vorderseite“ weist. Beim Durchlaufen eines Szenegraphen hängt der Zustand des Renderes an einem bestimmten Knoten meist ebenfalls von den vorangegangenen Knoten ab.

Übliche Implementierungen eines gerichteten Graphen verwenden Listen von Zeigern auf Nachbarelemente (*Adjazenzlisten*). Jeder Zeiger repräsentiert dabei eine Kante. Bei dieser Herangehensweise kann die Umgebung eines Knotens in Kantenrichtung schnell erreicht werden, während die Gegenrichtung eine Suche auf der Knotenmenge des Graphen erfordert. Um dies zu vermeiden, wird in 3.2-(3) zu jeder Kante eine *Referenzkante* erzwungen, deren *Referenzbezeichner* eineindeutig aus dem Bezeichner der Kante hervorgeht.

Man wählt zum Beispiel $L = \mathbb{Z}\backslash\{0\}$ und $r(x) = -x$. Da von der MGE-Datenstruktur immer endliche Datenmengen repräsentiert werden, ist die

[3] also auch bijektiv

Abzählbarkeit der so definierten Menge L keine Einschränkung. In einer frühen Version des MGE-Datenmodells wurde die hier ausgeschlossene Null verwendet, um eine ungerichtete Standardstruktur bereitzustellen. Durch den Zwang, eine bestimmte Beziehung auszeichnen zu müssen, wird aber die Forderung nach Kombinierbarkeit und Austauschbarkeit verschiedener Algorithmen (mit wahrscheinlich unterschiedlichen Standardstrukturen) verletzt.

Zum schnellen sequentiellen Datenzugriff werden in der Implementierung Listeniteratoren bereitgestellt (Abschnitt 3.3.1; zusätzlich kann ein Ausschnitt der Struktur in einem definierten, zusammenhängenden Speicherblock gespeichert werden, mit dem Blocktransfers und eine optimale Cache-Ausnutzung ermöglicht werden (Abschnitt 3.3.2).

Typisierung und Attributierung von MGEs wird durch Spezialisierung mit abgeleiteten Klassen erreicht.

3.2.1 Beispiel für eine MGE-Szene

Als Beispiel für eine MGE-Struktur zeigt Abb. 3.1 einen Ausschnitt aus einer EyeSi-Szene. Über die Retina ist eine pathologische Membran gespannt, die mit Hilfe einer Pinzette deformiert werden kann.

Der linke Teil der Struktur entspricht einem konventionellen Szenegraphen: die `GR`-Kanten werden während des grafischen Rendering-Prozesses abgelaufen. Die durch sie verbundenen MGEs enthalten (unter anderem) Informationen über Zustandsveränderungen der Grafik-Engine, Transformationsmatrizen und der Geometrie der repräsentierten Objekte. Die Pinzette besteht aus einem zylinderförmigen Hals g_3 und zwei Scheren g_4 und g_5, die aus dem gleichen Geometrie-Datensatz g_6 aufgebaut sind.

Für Gewebe und Pinzette ist ein eigenes Transformations-MGE g_1 vorgesehen, so dass weniger teure Umrechnungen der Koordinatensysteme bei der Berechnung von Interaktionen benötigt werden.

Der rechte Teil von Abb. 3.1 enthält Informationen für eine Feder-Masse-Simulation und die Visualisierung der Struktur. Vier Gitterknoten $n_0, \ldots, n_3$ sind über Federn $s_0, \ldots, s_4$ miteinander verbunden. Die Visualisierung verwendet dieselben Knoten als Eckpunkte der Dreiecke t_0 und t_1.

Die Verbindung zwischen dem Szenegraphen und der inneren Struktur des Gewebestücks wird über das MGE g_2 geschaffen, von dem je eine Kante `NL,SL,TL` zu jedem Gitterknoten, jeder Feder und jedem Dreieck ausgeht (der Übersicht halber ist dies in der Abbildung nur angedeutet).

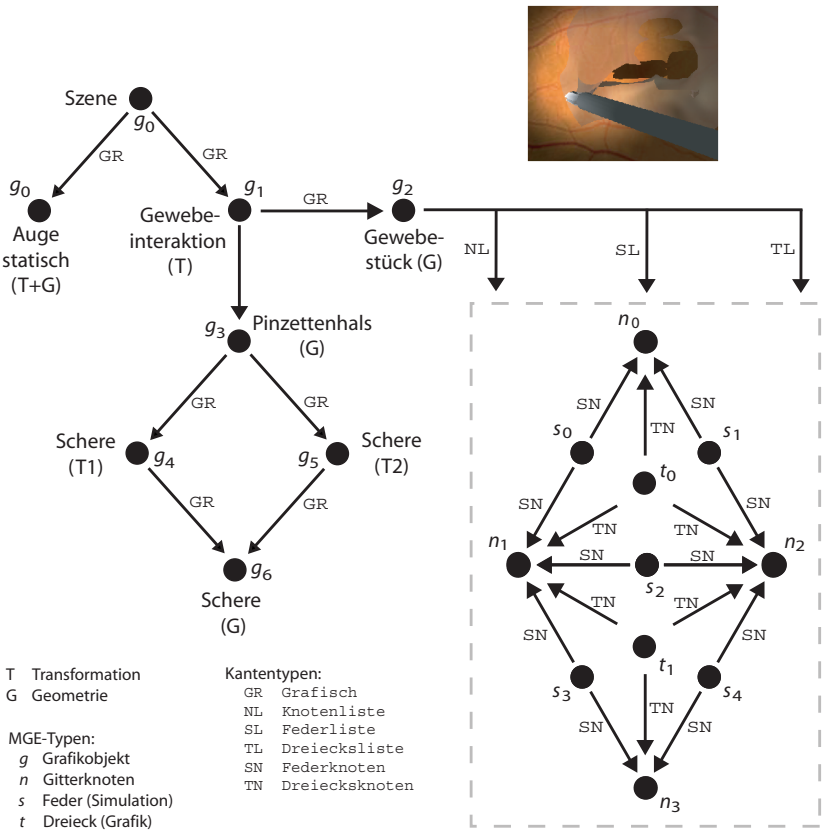


Abbildung 3.1: EyeSi-Szene und ihre MGE-Modellierung. Ein typischer Szenegraph (linker Subgraph) wird auf die gleiche Weise repräsentiert wie die innere Struktur einer pathologischen, deformierbaren Membran (rechter, gestrichelt markierter Subgraph). Vom Knoten g_2 weisen Kanten NL, SL, TL zu allen Punktmassen, Federn und Dreiecken der Membran.

3.3 Implementierung

Alle MGE-Objekte sind Instanzen einer MGE-Basisklasse oder einer davon abgeleiteten Klasse. Die Basisklasse stellt die Funktionalität zum Verwalten der Kanten zu den benachbarten MGEs bereit. In einem MGE v wird für jedes l, für das $\Gamma(v, l)$ nichtleer ist, ein STL-Vektor[4] angelegt, der Zeiger auf die entsprechenden l-Nachbarn enthält (obwohl es sich um ein Feld und nicht um eine Liste handelt, wird ein solcher Vektor auch im Folgenden als Adjazenzliste bezeichnet).

Um Bezeichner und Adjazenzlisten einander zuordnen zu können, besitzt jedes MGE eine STL-`map`, die die Paare (Bezeichner, Adjazenzliste) so anordnet, dass mit dem Bezeichner als Schlüssel ein $O(\log n)$-Zugriff garantiert wird, wobei n die Anzahl der nichtleeren $\Gamma(v, l)$ des MGEs ist. Für die Bezeichner wird wie oben beschrieben $L = \mathbb{Z}\backslash\{0\}$ und $r(x) = -x$ gewählt, so dass man eine leicht handhabbare Beziehung zwischen Bezeichnern und Referenzbezeichnern erhält.

Bei dieser Implementierung treten einige Probleme auf:

Speicherfragmentierung Die einzelnen MGEs und ihre Adjazenzlisten werden im Allgemeinen nicht fortlaufend und in der später benötigten Reihenfolge angelegt. Dadurch wird eine MGE-Struktur unter Umständen weiträumig im Speicher verteilt, so dass Cache-Mechanismen nicht effektiv genutzt werden können. Darüberhinaus ist der schnelle blockweise Datentransfer nicht möglich, der bei der Übertragung von Objektbeschreibungen auf die Grafikkarte eine Rolle spielt (Kapitel 6).

Logarithmisches Zeitverhalten Wenn auf eine bestimmte Adjazenzliste mehrfach zugegriffen wird, muss für jeden Zugriff mit einer logarithmischen Suche der Bezeichner der Liste auf die Liste selbst abgebildet werden, obwohl das Ergebnis der Suche jedes Mal identisch ist. Eine Lösung des Problems wird von der Schnittstelle des MGEs selbst bereitgestellt: es ist möglich, die Iteratoren[5] auf Listenanfang und -ende anzufordern und dann mit diesen auf die Liste zuzugreifen. Ein Nachteil dieser Methode ist, dass die Programmiererin die Iteratoren speichern muss, wenn sie an verschiedenen Stellen auf die

[4] Die *standard template library* ist eine C++-Bibliothek, die Datenstrukturen und Zugriffsalgorithmen zur Verfügung stellt [Josuttis 1999]. Ein STL-Vektor ist ein eindimensionales Feld mit Zugriffsmethoden. Es werden keine Listen, sondern Felder verwendet, da davon ausgegangen wird, dass wahlfreier Zugriff häufiger stattfindet als Löschen oder Hinzufügen von Nachbarn.

[5] eine spezielle STL-Variante von Zeigern

Liste zugreifen möchte. Zur Motivation für diese umständliche Vorgehensweise benötigt sie Informationen über den internen Aufbaus des MGEs.

Fehlende Typisierung Aus der Perspektive der MGE-Basisklasse besteht die gesamte repräsentierte Information aus den Beziehungen zwischen den einzelnen MGEs, nicht aber in der „inneren“ Information eines einzelnen Elements. In der Implementierung äußert sich das darin, dass beim Zugriff über MGE-Zugrifffunktionen Zeiger auf die Basisklasse zurückgegeben werden, die von der Programmiererin auf die jeweilige abgeleitete Klasse ge*casted* werden müssen. Wenn die *casts* falsch angewendet werden, können schwer zu lokalisierende Fehler entstehen.

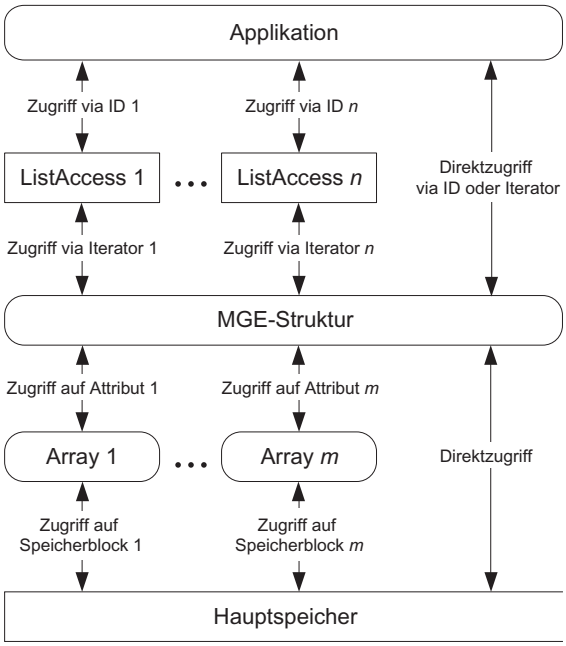


Abbildung 3.2: MGE-Zugriff mit ListAccess- und Array-Klassen. Über eine ListAccess-Klasse sieht eine Applikation nur denjenigen Subgraphen, dessen Kanten den entsprechenden Bezeichner (ID) besitzen. Speicherplatz für MGE-Attribute wird nicht vom Betriebssystem, sondern von einer zwischengelagerten Speicherverwaltung angefordert, die gleichartigen Attributen zusammenhängenden Speicherplatz zuweist.

Diese Probleme werden durch je eine Abstraktionsschicht ober- und unterhalb der MGE-Struktur gelöst (Abb. 3.2). Zur Applikation hin vermitteln *ListAccess-Objekte*, die die Sicht auf einen bestimmten Unterausschnitt der Struktur einschränken und für Typsicherheit sorgen. Der Speicher für die

Attribute eines MGEs wird über eine eigene Speicherverwaltung angefordert, die sicherstellt, dass gleichartige Attribute in einem zusammenhängenden Array abgespeichert sind. Die folgenden beiden Abschnitte gehen auf Details ein.

3.3.1 MGE-Zugriff über ListAccess-Objekte

Die ListAccess-Klasse und ihre Beziehung zum MGE ist in Abb. 3.3 dargestellt. Es kann nur auf die Adjazenzliste `mp_list` mit Bezeichner `m_listID` des MGEs `mp_mge` zugegriffen werden, so dass die Sicht der Programmiererin auf den entsprechenden Subgraphen eingeschränkt ist.[6] Typsicherheit wird dadurch erreicht, dass nur Objekte vom Typ `T_ElementType` akzeptiert werden. Auf das MGE selbst hat die Programmiererin keinen Zugriff mehr.

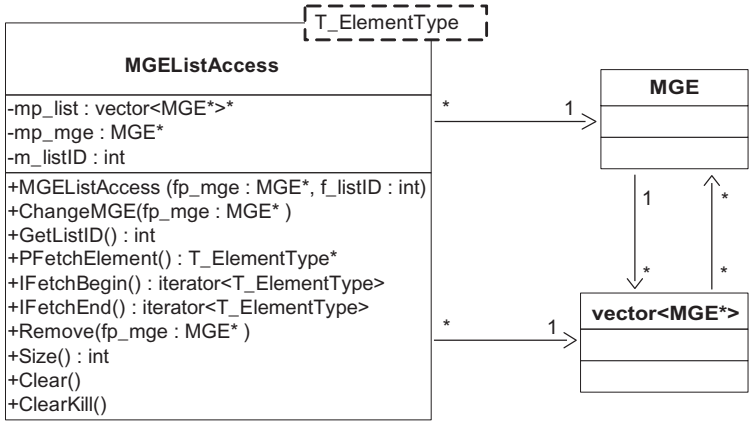


Abbildung 3.3: Statische Ansicht der ListAccess-Klasse, die die MGE-Funktionalität kapselt. Mit einem ListAccess-Objekt kann nur auf eine bestimmte Adjazenzliste (in der Abbildung: `vector<MGE*>`) eines bestimmten MGEs zugegriffen werden. Typsicherheit wird mit Hilfe des Template-Parameters `T_ElementType` gewährleistet.

Ein `MGEListAccess`-Objekt nimmt bereits im Konstruktor den Kantenbezeichner und das MGE entgegen, auf die es sich bezieht. Da keine Suche in den Kantenbezeichnern des MGEs notwendig ist, wird das Zeitverhalten von $O(\log n)$ auf $O(1)$ verbessert.

Auf ein MGE wird häufig über mehrere ListAccess-Instanzen zugegriffen, da es üblicherweise verschiedene Sichten auf eine MGE-Datenstruktur gibt.

[6] Idee und Implementierung der List-Access-Klasse stammen von Johannes Grimm (ViPA).

3.3.2 Kompakte Speicherung von Attributen in Arrays

Der sequentielle Zugriff auf die Attribute einer MGE-Struktur kann erheblich beschleunigt werden, wenn die Daten nicht objektzentriert, sondern in zusammenhängenden Speicherbereichen abgelegt werden. Dazu wird zwischen der MGE-Struktur und der Betriebssystem-Speicherverwaltung eine dünne Verwaltungsschicht geschoben, mit deren Hilfe einzelne Attribute Speicher anfordern, belegen und wieder freigeben können.

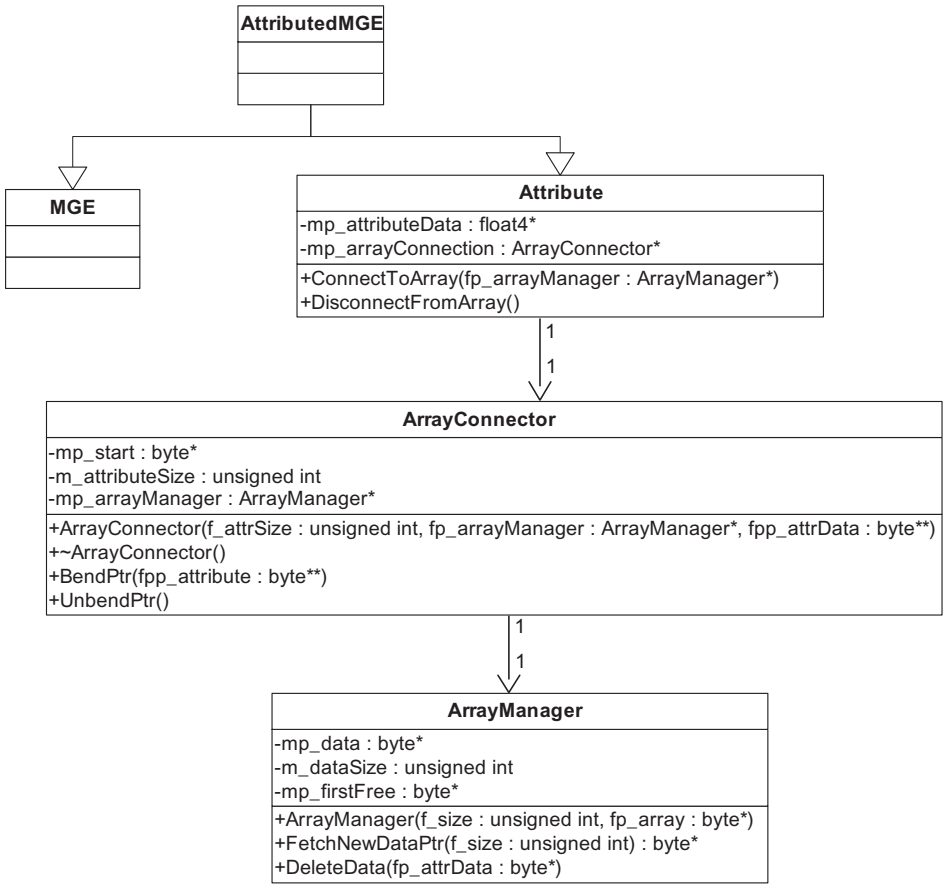


Abbildung 3.4: Verbindung zwischen einem attributierten MGE und einem Array (statische Ansicht). In diesem Beispiel wird die MGE-Basisklasse um einen Vierervektor („`float4`“) ergänzt. Falls mehrere Instanzen des attributierten MGEs angelegt werden, stellt der ArrayManager sicher, dass sich alle Vierervektoren in einem zusammenhängenden Speicherbereich befinden.

Abb. 3.4 stellt am Beispiel eines um einen Vierervektor des benutzerdefinierten Typs `float4` ergänzten MGEs die Klassen und Methoden dar, die

dazu notwendig sind. Der Zugriff auf den Vektor findet über den Zeiger `mp_attributeData` statt, der zur Verbindung mit dem Array umgebogen werden kann.

Die Attribut-Klasse wird über einen `ArrayConnector` mit dem `ArrayManager` verbunden, der den zusammenhängenden Speicherbereich verwaltet. Der Vorgang des Verbindens und Trennens ist in Abb. 3.5 dargestellt. Der `ArrayConnector` wird benötigt, um die elementspezifischen Daten der Verbindung abzuspeichern (Größe des Attributs `mp_attributeSize`, Lokalisierung im Speicher `mp_start`).

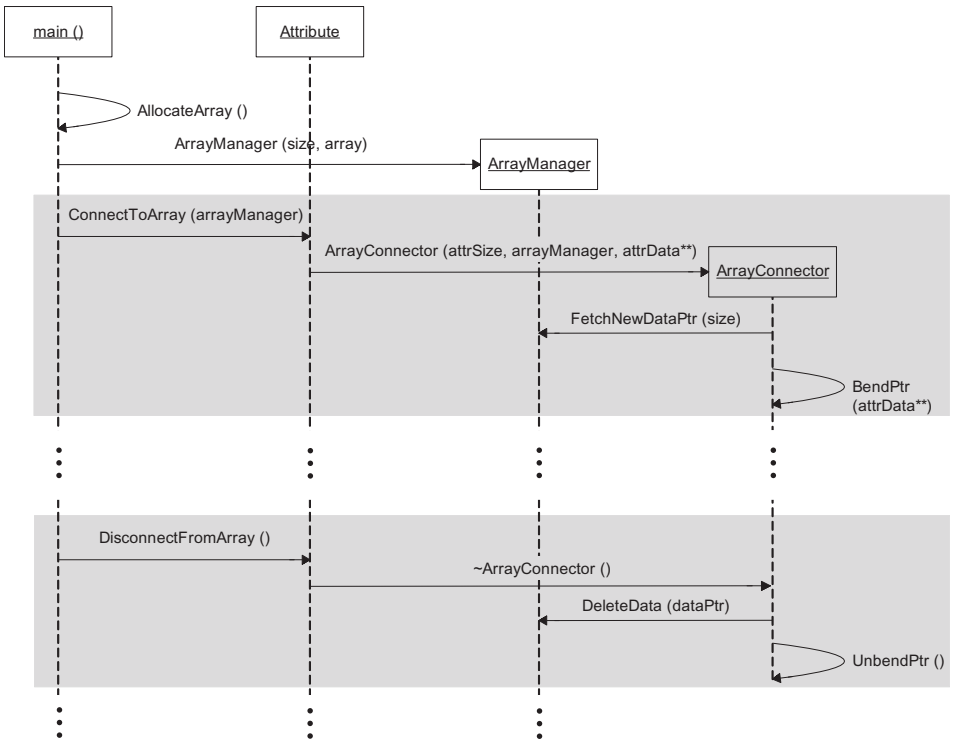


Abbildung 3.5: Ein attributiertes MGE wird mit einem Array verbunden und wieder getrennt (Sequenzdarstellung). In einer MGE-Struktur werden die grau markierten Bereiche für jedes gleichartige Attribut ausgeführt.

Der aufrufende Programmteil wird im Beispiel als `main()` bezeichnet, muss aber nicht unbedingt von der Applikationsprogrammiererin realisiert werden. Für Standard-Anwendungen (etwa ein grafisches Mesh) stellt die VRM-Bibliothek attributierte MGEs und dazugehörige Verwaltungsklassen zur Verfügung, die die Zusammenfassung der einzelnen Attribute zu Arrays selbstständig durchführen können.

Zunächst wird im `main()` Speicher in der erforderlichen Größe allokiert,

der dann im Konstruktor an den `ArrayManager` übergeben wird. Der `ArrayManager` übernimmt die Allokierung nicht selbst, da je nach Anwendung unterschiedliche Speicherbereiche angesprochen werden müssen. Beispielsweise wird zum Übertragen von grafischen Daten häufig AGP-Speicher verwendet, auf den die Grafikkarte per DMA direkt zugreifen kann, ohne Prozessorlast zu erzeugen.

Der `ConnectToArray`-Methodenaufruf teilt dem Attribut mit, dass es seine Daten nun mit Hilfe der `ArrayManager`-Klasse speichern soll. Dazu legt es zunächst einen `ArrayConnector` an, der das Attribut bei der Verwaltung seiner Daten entlastet. Im Konstruktor der `ArrayConnector`-Klasse wird die Größe der Attributdaten, der zuständige `ArrayManager` und ein Doppelzeiger auf die Attributdaten übergeben (Letzterer ist notwendig, um den `mp_attributeData`-Zeiger der Attributinstanz umbiegen zu können).

Der `ArrayConnector` fordert mit `FetchNewDataPtr` beim `ArrayManager` einen Speicherblock der erforderlichen Größe an und erhält einen Zeiger auf diesen Speicherblock zurückgeliefert. In der Methode `BendPtr` kopiert er nun die Daten des Attributes in diesen Speicherblock und biegt den `mp_attributeData`-Zeiger auf den Speicherblock um. Alle Speicherzugriffe des Attributs arbeiten von nun an auf dem Array.

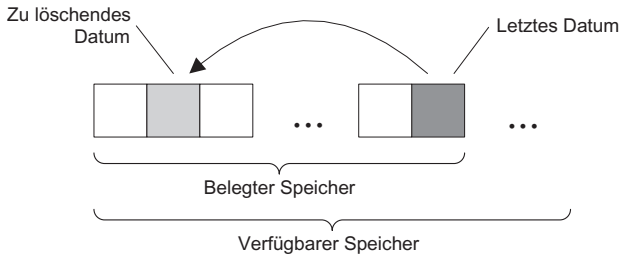


Abbildung 3.6: Löschen eines Datums im Array

Wenn das Attribut wieder vom Array getrennt wird (zum Beispiel beim Löschen des MGEs), wird im Destruktor `~ArrayConnector` zunächst dem `ArrayManager` mitgeteilt, dass der entsprechende Speicherblock freigegeben werden kann. Um eine Fragmentierung des Speichers zu vermeiden, verschiebt der Array-Manager das letzte gespeicherte Datum an diese Stelle (Abb. 3.6). Voraussetzungen sind, dass die Größe aller gespeicherten Attribute gleich ist und dass die Reihenfolge der Attribute im Array keine Bedeutung hat. Eine Methode, die ohne diese Voraussetzungen auskommt, aber erheblich langsamer ist (vor allem im ungepufferten AGP-Speicher), wäre, alle Daten „rechts“ des zu löschenden Elements um seine Größe nach „links“ zu verschieben.

Attribute, deren Daten in Arrays organisiert werden, müssen über Zeiger auf die Daten zugreifen. Im Vergleich zur Speicherung in einer Instanzva-

riablen wird der Zugriff über die Instanz verlangsamt, da zur Laufzeit aus dem Zeiger zunächst die Speicheradresse der Daten gelesen werden muss. Für Attribute, bei denen Lese-Schreib-Zugriffe über die Instanz häufiger sind als Zugriffe über den Array bietet die Array-Verwaltung daher die Alternative, nicht den Zeiger auf die Daten umzubiegen, sondern die Daten doppelt zu speichern – sowohl als Instanzvariable als auch als Element in einem Array. Bei Bedarf müssen die Daten der einzelnen Instanzen dann explizit in den Array umkopiert werden. Geschwindigkeitsvorteile bietet diese Herangehensweise zum Beispiel bei einem Simulationsalgorithmus, der häufigen wahlfreien Zugriff benötigt (Zugriff über die die Instanzen) und dessen Ergebnis anschliessend visualisiert werden soll (Umkopieren in Array und Übertragen auf die Grafikkarte).

3.4 Zusammenfassung und Diskussion

In diesem Kapitel wurde das MGE-Datenmodell vorgestellt, das in der VRM-Bibliothek zur Repräsentation von Szenegraphen und der inneren Struktur von deformierbaren Objekten dient. Sein Vorbild sind semantische Netze, die eine flexible Formulierung von Zusammenhängen ermöglichen.

Zentrales Element des Modells ist die Klasse MGE (*Multigraphelement*), die den Knoten des semantischen Netzes entspricht; verschiedene Knotentypen werden durch Vererbung von Attributen differenziert. Kanten werden durch knotenzentrierte Adjazenzlisten formuliert; sie sind mit Hilfe von Bezeichnern typisiert.

Kanten sind bei dieser Herangehensweise keine eigenständigen Objekte, was einen scheinbaren Gegensatz zur typischen Struktur eines deformierbaren Objekts darstellt – Beispiele wie Federn und Punktmassen oder Chains und ChainMail-Elemente legen eigentlich nahe, die punktförmigen Elemente als Knoten und ihre Verbindungen als attributierte Kanten aufzufassen. Eine derartige Differenzierung ist auf der Ebene des Datenmodells aber unhandlich und formal unbefriedigend: die Knoten- und Kantenobjekte müssten über Adjazenzlisten miteinander verbunden werden, wodurch ein zweiter, wieder nicht attributierbarer Kantentyp entstehen würde. Die Implementierung der Verwaltungsmethoden dieser Listen wäre für Knoten- und Kantenlisten identisch, so dass die Semantik der beiden Klassen extensional nicht unterscheidbar wäre.[7]

Der Zugriff auf bestimmte Teile einer MGE-Datenstruktur, die durch feste Knoten- und Kantentypen definiert sind, kann mit Hilfe einer dünnen

[7] Eine frühe Version der VRM-Bibliothek unterschied tatsächlich zwischen Knoten- und Kantenklassen. Olaf Körner (ViPA) vollzog den gedanklichen Schritt, die beiden Klassen zum MGE zu identifizieren.

Abstraktionsschicht gekapselt werden, die Typsicherheit und einen schnelleren Zugriff durch Speicherung der entsprechenden Adjazenzliste ermöglicht. Schneller blockweiser Zugriff auf die Daten einer Menge gleichartig attributierter Knoten wird dadurch erreicht, dass Teile der MGE-Struktur mit Hilfe einer Speicherverwaltungsschicht in einem zusammenhängenden Bereich abgelegt werden können.

Soweit dem Autor bekannt, ist die MGE-Datenstruktur im Bereich der grafischen und VR-Systeme der einzige Ansatz, der Beziehungen zwischen Objekten und deren innere Strukturen auf einheitliche Weise repräsentiert.[8] Die übliche Kluft zwischen Flexibilität und Geschwindigkeit wird durch die hybride Struktur des MGE-Formats mit einer objektzentrierten Schicht und einer darunterliegenden Array-Schicht überwunden. Der Preis dafür ist ein erhöhter Speicherplatzbedarf. Er ist vertretbar, da die Größe der repräsentierbaren Strukturen durch die Echtzeitbedingung viel stärker eingeschränkt wird als durch die Grenzen des Hauptspeichers.

[8] Im Bereich der Wissensverarbeitung ist es dagegen üblich, verschiedene Ebenen in einem einheitlichen Repräsentationsformat zu mischen – etwa im Hyperknoten-Datenmodell von Poulovassilis und Levene [1994], das Knoten und Graphen identifiziert. Der Autor zeigt in Wagner [1996, Kapitel 2] die Äquivalenz dieses Datenformats mit einem semantischen Netz in Bezug auf Komplexität und Ausdrucksmächtigkeit.

4

Kollisionserkennung

Die Erkennung von Kollisionen zwischen Objekten ist der erste Schritt einer Interaktionsberechnung.[1] Algorithmen zur Kollisionserkennung beantworten z.B. die folgenden Fragen: Hat eine Kollision stattgefunden? Wo berühren sich die kollidierenden Objekte? Wie weit sind sie ineinander eingedrungen? Welche Teile eines Objekts sind an der Kollision beteiligt? Wie weit sind nicht kollidierende Objekte voneinander entfernt?

In vielen Anwendungen beansprucht die Kollisionserkennung einen großen Teil der verfügbaren Rechenkapazität. Zur Verringerung des Rechenaufwands existiert eine Vielzahl von verschiedenen Ansätzen. Abschnitt 4.1 gibt einen ausführlichen Überblick. Dabei wird besonders auf bildbasierte Ansätze eingegangen (Abschnitt 4.1.4), die die Grundlage des ZCOLL-Verfahrens bilden. ZCOLL wurde in dieser Arbeit für die Interaktion zwischen chirurgischen Instrumenten und deformierbarem Gewebe entwickelt und in der VRM-Bibliothek implementiert. Die algorithmischen Details und sein Zeitverhalten werden in Abschnitt 4.2 diskutiert.

Bemerkung: Im Folgenden wird häufig zwischen *volumenartigen* und *flächenartigen* Objekten (*Membranen*) unterschieden.[2] Bei dem für diese

[1] Es sei denn, man behandelt Kollisionen gar nicht als diskrete Ereignisse – sie können einfacher und physikalisch angemessener auch als Wirkungen kurzreichweitiger Kräfte beschrieben werden. Allerdings werden zur Simulation solcher Kräfte hoch aufgelöste Zeit- und Raumskalen benötigt, die für makroskopische Objekte im Allgemeinen nicht in Echtzeit realisierbar sind.

[2] Im mathematischen Sinne findet man eine Definition der Fläche z.B. bei Heuser [1991, Abschnitt 208, „Flächen und Oberflächenintegrale im Raum“]. Der Charakter eines volumenartigen Objekts wird dadurch bestimmt, dass es ein definiertes *Inneres* besitzt. Am treffendsten ist dafür der topologische Begriff des *Gebiets*, der ebenfalls bei Heuser definiert wird (Abschnitt 161, „Bogenzusammenhängende Mengen“).

Arbeit wichtigen Beispiel einer intraokularen Operation kann die Kollisionserkennung häufig auf Instrument-Membran-Interaktionen beschränkt werden.

4.1 Stand der Forschung

4.1.1 Überblick

Ein Grund für den algorithmischen Aufwand bei der Kollisionserkennung ist das *all-pairs problem*: wenn sich in der simulierten Welt n bewegliche und m raumfeste Objekte befinden, müssen bei einem Brute-Force-Ansatz $\binom{n}{2} + nm$ Objektpaare auf Kollisionen getestet werden – obwohl in den meisten Fällen nur sehr wenige dieser Kollisionen gleichzeitig stattfinden.[3]

Häufig wird die Kollisionserkennung daher in zwei Phasen durchgeführt: in der *broad phase* wird der Suchraum mit schnell durchzuführenden, konservativen Tests verkleinert – beispielsweise, indem die Objekte durch einfache geometrische Formen (*bounding objects* oder *Hüllkörper*) approximiert und in einer hierarchischen Struktur zusammengefasst werden.

In der zweiten Phase (*narrow phase*) wird für die verbleibenden Objektpaare eine genauere, aber aufwändigere Kollisionserkennung durchgeführt.[4]

Deformierbare Objekte, wie sie in chirurgischen Simulationen vorkommen, erschweren die Kollisionserkennung zusätzlich: ihre geometrischen und topologischen Änderungen können die Vorausberechnungen (etwa geometrische Approximationen der *broad phase*) ungültig machen.

Eine wichtige Rolle für die Klassifizierung von Kollisionserkennungsalgorithmen spielt die Art der Objektrepräsentation (Abb. 4.1).[5] Die *gerasterten Repräsentationen* (unterer Zweig von Abb. 4.1) beschreiben die Objekte

[3] Es ist sehr schwierig, n Objekte im dreidimensionalen Raum so anzuordnen, dass tatsächlich $\binom{n}{2}$ Kollisionen stattfinden. In dem natürlichen Beispiel eines Systems von n Molekülen im Gleichgewichtszustand ist die Anzahl der Kollisionen proportional n [Feynman et al. 1963].

[4] Die Unterscheidung in Broad Phase und Narrow phase wird zum Beispiel von Hubbard [1996] und Ganovelli et al. [2000] durchgeführt. Sie eignet sich nur bedingt zur Klassifikation, da bei einigen Algorithmen die Zuordnung nicht klar ist – z.B. können konvexe Körper sowohl in der Broad Phase (als konvexe Hüllen zur Approximation) als auch in der Narrow Phase verwendet werden (wenn die Objekte selbst aus konvexen Körpern zusammengesetzt sind).

[5] Der Charakter eines Verfahrens und der damit verbundenen Forschung wird dadurch entscheidend beeinflusst. Es ist bemerkenswert, dass diejenigen Forschungszweige, die sich mit geometrischen Verfahren befassen, die bildbasierten Ansätze kaum wahrzunehmen scheinen – keine (!) der hier angegebenen Veröffentlichungen über geometrische Kollisionserkennung enthält eine Referenz auf bildbasierte Methoden.

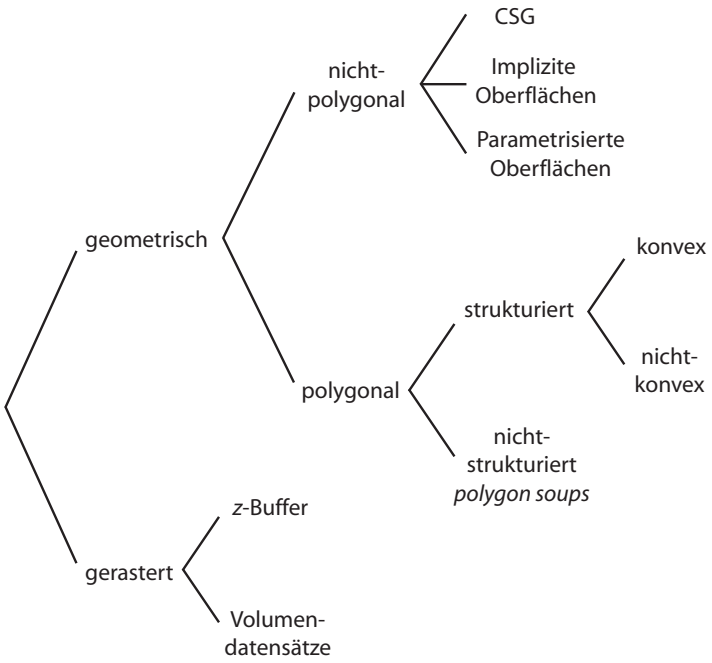


Abbildung 4.1: Verschiedene Objektrepräsentationen für die Kollisionserkennung. Der obere Zweig entspricht einem Diagramm von Lin und Gottschalk [1998].

mit Hilfe von Voxeln (Volumendatensätze) bzw. Pixeln mit Tiefeninformation (z-Buffer-basierte Ansätze).

Volumendatensätze können wegen ihrer großen Datenmenge problematisch sein. Häufig repräsentieren die einzelnen Voxel Messwerte einer kontinuierlichen Größe (z.B. der Dichte); das Objekt enthält dann keine definierte begrenzende Oberfläche, so dass geometrische Ausschlusskriterien schwer zu formulieren sind. He [1996, Kapitel 6] nennt Kollisionen zwischen volumetrischen Objekten daher *soft collisions*. Er beschreibt sie, indem er den einzelnen Voxeln Wahrscheinlichkeitswerte zuordnet; die Wahrscheinlichkeit einer Kollision zweier Objekte an einem bestimmten Raumpunkt ist das Produkt der Wahrscheinlichkeitswerte der beiden Objekte an dieser Stelle. Für die schnelle Entscheidung, ob eine Kollision überhaupt möglich ist, werden die Voxel-Datensätze in einer baumartigen Struktur von begrenzenden Kugeln (einem *sphere tree*) angeordnet. Jeder Kugel ist die minimale und maximale Wahrscheinlichkeit des Objektsegments zugeordnet, das von ihr eingeschlossen wird. Durch die Angabe von unteren und oberen Wahrscheinlichkeitsgrenzen kann der *sphere tree* verkleinert werden.

Gibson [1995; 2000] verwendet für die Kollisionserkennung von volumetrischen Objekten eine *occupancy map*, die auch für polygonale Objekte benutzt werden kann und auf die Abschnitt 4.1.3 eingeht.

Z-Buffer-basierte Methoden arbeiten auf gerasterten Repräsentationen, wie sie während der Visualisierung von Oberflächenmodellen entstehen. Durch die Rasterung wird das Kollisionserkennungsproblem auf den Vergleich von Tiefenwerten auf einem regulären 2D-Gitter reduziert. Der besondere Charme dieser Verfahren besteht darin, dass die notwendigen Operationen von der Grafikkarte in hoher Geschwindigkeit durchgeführt werden können. Vor allem sind keine Vorberechnungen notwendig, so dass sich diese Verfahren auch für deformierbare Objekte eignen. Eine genauere Beschreibung gibt Abschnitt 4.1.4.

Polygonale Repräsentationen von Objekten sind in der Echtzeit-Computergrafik vorherrschend (Kapitel 6). Für Computergrafik- und VR-Applikationen ist die Bedeutung von Kollisionserkennungsalgorithmen, die auf solchen Repräsentationen aufbauen, daher sehr groß. Im einfachsten Fall werden Objekte als *polygon soups*, d.h. als unstrukturierte Mengen von Polygonen beschrieben. Zusätzliche geometrische Information können zum Beispiel darin bestehen, dass die Nachbarschaftsrelationen zwischen den Polygonen bekannt sind oder dass die Polygone konvexe Polyeder bilden.

Das All-Pairs-Problem, das oben für den Fall von n kollidierenden Objekten beschrieben wurde, setzt sich für die einzelnen Elemente zweier kollidierender Polygonmengen fort. Die Lösung ist häufig dieselbe wie auf Objektebene: die Anzahl der zu vergleichenden Paare wird durch eine hierarchische Struktur von Hüllkörpern (Abschnitt 4.1.2) oder durch die Strukturierung des Raums selbst (Abschnitt 4.1.3) eingeschränkt.

Geometrische, nicht-polygonale Methoden verwenden geschlossene Beschreibungen. Die *constructive solid geometry* (CSG) geht im Gegensatz zu polygonalen Repräsentationen nicht von Oberflächen-Repräsentationen, sondern von soliden Objekten aus. Mit Hilfe von Mengenoperationen (Vereinigung, Schnitt, Differenz) werden daraus neue Objekte erzeugt.

Eine geschlossene Beschreibung kann auch als *parametrisierte* oder *implizite Oberfläche* gegeben sein. Erstere ist eine Abbildung $f_p : \mathbb{R}^2 \to \mathbb{R}^3$, letztere besteht in der Lösungsmenge der Gleichung $f_i = 0$ für eine Funktion $f_i : \mathbb{R}^3 \to \mathbb{R}$.

Solche Beschreibungen zeichnen sich durch eine kompakte Darstellung, die Möglichkeit zur Verwendung algebraischer Methoden und eine im Prinzip unbegrenzte Genauigkeit aus. Häufig ist der algorithmische Aufwand aber zu hoch, um die entsprechenden Verfahren in Echtzeitsystemen einzusetzen. Vor allem bei physikalischen Simulationen ist es außerdem nicht immer möglich, eine geschlossene Beschreibung anzugeben. In der VRM-Bibliothek werden mit Hilfe geschlossener Beschreibungen daher nur Kollisionen von sehr einfachen Objekten[6] berechnet. Im Übrigen verweisen wir

[6] z.B. Quader, Kegel, Kugeln, Zylinder. In der EyeSi-Software werden solche Objekte

auf den Übersichtsartikel von Lin und Gottschalk [1998], der eine Vielzahl von Ansätzen beschreibt.

Das zeitliche Verhalten einer Applikation spielt für die Kollisionserkennung ebenfalls eine Rolle. Wenn sich zwei Objekte innerhalb eines Zeitschritts nur geringfügig bewegen, dann ändert sich ihre geometrische Beziehung zueinander kaum. Diese Eigenschaft heißt *zeitliche Kohärenz* und wird von einigen Ansätzen zur Einschränkung des Suchraums genutzt (Abschnitt 4.1.2). *Räumliche Kohärenz* beschreibt, dass die Objekte sich zu einer bestimmten Zeit nur in einem räumlich begrenzten Bereich der Welt befinden, so dass der Einsatz von Hüllkörpern möglich ist (Abschnitt 4.1.2).

Bei einer zu geringen Abtastfrequenz geht die zeitliche Kohärenz verloren. Im schlimmsten Fall findet eine Kollision dann nur zwischen zwei Iterationsschritten statt, so dass sie nicht festgestellt werden kann. Der allgemeinste Ansatz, um dies zu verhindern, ist die Extrusion der Objekte entlang ihrer Trajektorie in der vierdimensionalen Raumzeit. Zwei Objekte kollidieren genau dann, wenn ihre Extrusionen sich schneiden. Da deren Berechnung aber sehr aufwändig ist, betrachtet man meist nur ihre Projektionen in den 3D-Raum. Anstelle der ursprünglichen Objekte wird die Kollisionserkennung auf den *swept volumes* durchgeführt, die sich aus der Position der Objekte im letzten und im aktuellen Iterationsschritt ergeben [Jiménez et al. 2001, Lombardo et al. 1999].

Für weiterführende Informationen über Algorithmen zur geometrischen Kollisionserkennung siehe Lin und Gottschalk [1998] und Jiménez et al. [2001].

4.1.2 Polygonbasierte Ansätze

Konvexe Polyeder

Konvexe Objekte besitzen geometrische Eigenschaften, die zur Verkleinerung des Suchraums bei der Kollisionserkennung genutzt werden können:

Lokale Suche Für den Fall zeitlicher Kohärenz besteht bei konvexen Polyedern (*Polytopen*) auch eine „topologische Kohärenz“ bei der Suche nach minimalen Abständen: Gegeben sei ein Punkt P minimalen Abstands zu einer Ecke, Kante oder Fläche eines Polytops. Wenn die Bewegung des Polytops hinreichend klein ist, befindet sich der neue Punkt minimalen Abstands auf einer benachbarten Kante, Fläche oder Ecke bzw. es kann in der Richtung derjenigen Nachbarn gesucht werden, bei denen sich der Abstand am meisten verringert (dass dies

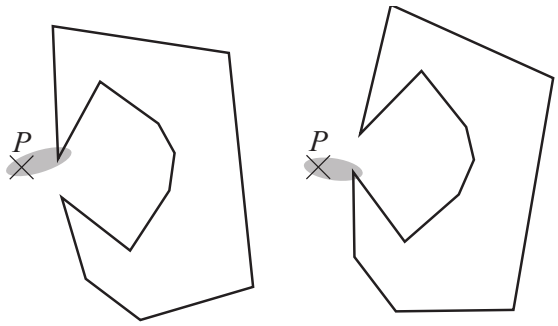


Abbildung 4.2: Bei nicht-konvexen Objekten kann die Suche nach nächsten Punkten in aufeinanderfolgenden Iterationen nicht auf eine Nachbarschaft beschränkt werden – trotz zeitlicher Kohärenz.

für nicht-konvexe Objekte nicht zutrifft, illustriert Abb. 4.2).

Viele Algorithmen machen sich diesen Umstand zunutze, um „beinahe“ ein $O(1)$ Zeitverhalten zu erreichen. Die Einschränkung muss gemacht werden, da bei zunehmender Knotendichte des Objekts und gleichbleibendem zeitlichem Raster der Suchraum auf eine iterierte Nachbarschaft erweitert werden muss. In der Literatur wird dieses Zeitverhalten daher als *almost constant time* oder *expected constant time* bezeichnet (im Folgenden: $O(1+\varepsilon)$-Verhalten).

Trennende Ebene Zwei konvexe Körper A und B kollidieren genau dann nicht, wenn es eine trennende Ebene zwischen ihnen gibt. Lin und Canny [1991] nutzen diese Beobachtung, um eine obere Grenze für die Komplexität der Kollisionserkennung für Polytope anzugeben. Sei $E : \vec{n}\vec{x} = d$ die Gleichung der gesuchten Ebene. Falls E existiert, gilt dann für Ecken $\vec{a} \in A$ und $\vec{b} \in B$: $\vec{a}\vec{n} < d$ und $\vec{b}\vec{n} > d$. Diese Ungleichungen können als lineares Programm (LP) aufgefasst werden (mit den vier Unbekannten $\vec{n}$ und d), das in Linearzeit in der Anzahl der Ungleichungen gelöst werden kann [Megiddo 1984, Seidel 1990]. Daher kann in höchstens $O(n+m)$ Schritten festgestellt werden, ob zwei konvexe Polyeder mit n und m Ecken kollidieren.

LP-Löser sind sehr aufwändig, so dass ihr günstiges Zeitverhalten bei der Suche nach trennenden Ebenen nur bei großen Problemen zur Geltung kommt. Die meisten Algorithmen zur Kollisionserkennung vermeiden daher den Umweg über ein lineares Programm. Baraff [1990] nutzt dazu den folgenden Satz:

bei abstrakten Trainingsaufgaben sowie zur Approximation von bestimmten Instrumentformen eingesetzt (Kapitel 7).

Satz 4.1 *Zwischen zwei nicht kollidierenden Polytopen existiert eine trennende Ebene, die entweder eine Fläche eines der Polytope oder eine Kante jedes der Polytope enthält.*

Die Fläche oder die beiden Kanten werden als *Zeugen* gespeichert, dass keine Kollision stattgefunden hat. In der darauffolgenden Iteration wird mit ihnen eine neue Ebene bestimmt. Falls sie keine trennende Ebene ist, wird die Suche bei den angrenzenden Kanten und Flächen fortgesetzt. Zeitliche Kohärenz vorausgesetzt, kann so in den meisten Fällen schnell eine trennende Ebene gefunden werden.

Der nach den Anfangsbuchstaben der Autoren benannte GJK-Algorithmus [Gilbert et al. 1988] kann nicht nur Kollisionen zwischen Polytopen erkennen, sondern auch ihren Abstand berechnen, falls sie nicht kollidieren. Er arbeitet auf einer alternativen Repräsentation der beiden Polytope A und B, der *Minkowski-Differenz* M[7]:

$$M := A - B := \{\vec{x} - \vec{y} : \vec{x} \in A, \vec{y} \in B\} \tag{4.1}$$

M beschreibt ein Polytop, das die folgenden Eigenschaften besitzt: Falls $0 \in M$, kollidieren A und B. Sonst ist der Abstand von M zum Nullpunkt gleich dem Abstand von A und B. Damit nicht die ganze Menge M erzeugt werden muss (in $O(nm)$ Schritten), konstruiert GJK eine Folge von *Simplices*[8], die in M enthalten sind. In jedem Iterationsschritt wird ein neuer Simplex bestimmt, der näher am Ursprung liegt. Die dazu verwendete Suchrichtung kann auf den Polytopen A und B konstruiert werden. Die Originalveröffentlichung testet dazu in $O(n + m)$ Schritten alle Ecken; durch das Ausnutzen der zeitlichen Kohärenz und der Konvexität von A und B wird dies z.B. von Cameron [1997a;b] und van den Bergen [1999] auf $O(1 + \varepsilon)$ Schritte reduziert. Mit dieser Erweiterung kann der Algorithmus (der dann *enhanced GJK* heißt) unter der Voraussetzung zeitlicher Kohärenz in $O(1+\varepsilon)$ Schritten den Abstand zweier Polytope bzw. ihre Eindringtiefe berechnen. Eine anschauliche Darstellung von GJK findet sich in der Veröffentlichung von Cameron [1997a].

Während die Simplices des GJK-Verfahrens sich im Innern des Polytops bewegen, läuft der Algorithmus von Lin und Canny [1991] auf der Oberfläche der beiden Polytope A und B entlang. In jedem Iterationsschritt wird

[7] Die Namensgebung in der Literatur ist uneinheitlich. M wird z.B. bei van den Bergen [1999] als *Minkowski-Summe* bezeichnet, bei Cameron [1997a;b] dagegen als *translational C-space obstacle*.

[8] Ein Simplex ist die konvexe Hülle einer affin unabhängigen Menge von Punkten. Im dreidimensionalen Raum ist ein Simplex ein Punkt, eine Gerade, ein Dreieck oder ein Tetraeder [van den Bergen 1999].

ein neues Paar von *features* (Ecken, Kanten, Flächen) ausgewählt, bei denen getestet wird, ob sie gesuchte Punkte geringsten Abstandes der beiden Polytope enthalten. Entscheidungskriterium ist der folgende Satz, der z.B. von Mirtich [1998] bewiesen wird:

Satz 4.2 *Seien F_A und F_B Features von disjunkten Polytopen, $VR(F_A)$ bzw. $VR(F_B)$ die dazugehörigen Voronoi-Regionen und $P_A \in F_A, P_B \in F_B$ zwei Punkte minimalen Abstands von F_A und F_B. Wenn $P_A \in VR(F_B)$ und $P_B \in VR(F_A)$, dann sind P_A und P_B zwei Punkte minimalen Abstands der beiden Polytope.*

Wenn diese Bedingung nicht erfüllt ist, wird in der nächsten Iteration das jeweilige benachbarte Feature gewählt. Auch hier kann unter der Voraussetzung zeitlicher Kohärenz ein $O(1 + \varepsilon)$-Zeitverhalten erreicht werden.

Wie in Cohen et al. [1995] und Mirtich [1998] beschrieben, hat der Lin-Canny-Algorithmus zwei Nachteile: (1) die Behandlung von Sonderfällen ist aufwändig, insbesondere, wenn es keinen eindeutig bestimmten Punkt kleinsten Abstandes gibt – beispielsweise wenn das aktuelle Feature-Paar aus parallelen Seitenflächen besteht. (2) Im Inneren der Polytope sind keine Voronoi-Zellen definiert. Bei einer Kollision kann es daher vorkommen, dass der Algorithmus nicht terminiert. Cohen et al. umgehen das zweite Problem, indem sie das Innere der Polytope ebenfalls in Zellen aufteilen, die aus den Verbindungen zwischen dem Schwerpunkt eines Polytops und seinen Kanten bestehen. Beide Nachteile werden im Algorithmus *V-clip* von Mirtich dadurch aufgehoben, dass die Bedingung von Satz 4.2 mit Hilfe von einfachen Clipping-Operationen überprüft wird. Da dabei keine Punkte kürzesten Abstands auf den Features berechnet werden, müssen keine Sonderfälle behandelt werden.

Hüllkörper-Hierarchien

Zur Verringerung des Suchraums bei der Lösung des All-Pairs-Problem benötigt man Ausschlusskriterien, die schnell überprüft werden können. Eine Möglichkeit besteht darin, die einzelnen Objekte mit *Hüllkörpern* zu umgeben. Nur wenn die Hüllkörper zweier Objekte kollidieren, muss eine genauere Kollisionserkennung durchgeführt werden. Um eine bestimmte Form genauer zu approximieren, werden die Hüllkörper meist hierarchisch angeordnet.

Kriterien für die Wahl einer bestimmten Hüllkörper-Form sind:

- Der Aufwand für die Kollisionstests zwischen den Hüllkörpern.

- Der Aufwand, um die Hüllkörper für ein bestimmtes Objekt zu erzeugen. Dieser Aspekt spielt besonders dann eine Rolle, wenn das Objekt deformierbar ist und der Hüllkörper in jeder Iteration angepasst werden muss.
- Die Genauigkeit, mit der sich die Objektform approximieren lässt.

Zu den vorgeschlagenen Hüllkörpern gehören Kugeln [Hubbard 1996], orientierte [Gottschalk et al. 1996] oder achsenparallele Quader [van den Bergen 1999] und k-DOPs (*k-discrete oriented polytopes*, Polytope, deren Flächen in k Raumrichtungen orientiert sind [Held et al. 1995, Mezger et al. 2002]).

Bei Kugeln muss für einen Kollisionstest lediglich der Abstand ihrer Mittelpunkte berechnet werden. Im Gegensatz dazu ist die Erzeugung einer kugelbasierten Hüllkörper-Hierarchie aufwändig: Von Hubbard [1996] wird das Objekt zunächst durch eine Menge P von Punkten auf seiner Oberfläche approximiert. Für die Elemente von P wird anschliessend ein Voronoi-Diagramm erstellt; die Ecken aneinandergrenzender Voronoi-Zellen haben die Eigenschaft, Mittelpunkte von Kugeln zu sein, auf deren Oberflächen sich jeweils vier Punkte aus P befinden. Die einzelnen Kugeln bilden die feinste Approximation des Objekts. Die nächstgröbere Hierarchiestufe wird dadurch erreicht, dass Paare von Kugeln zusammengefasst werden. Hubbard gibt für die Vorverarbeitung eines Objekts aus 10.171 Dreiecken (eine typische Zahl in einer VRM-Applikation) eine Berechnungsdauer von 2,7 Stunden auf einer HP 9000/735-Workstation an.

Einen orientierten Quader (OBB – *oriented bounding box*) kann man in $O(n^3)$ Schritten an ein polygonales Modell anpassen [O'Rourke 1985]. Für eine schnellere Berechnung verwenden Gottschalk et al. [1996] die Kovarianzmatrix des Objekts, deren Eigenvektoren zur Orientierung der OBB verwendet werden. Damit innere Knoten und eine ungleichmäßige Knotenverteilung das Ergebnis nicht verzerren, wird dazu die konvexe Hülle des Objekts berechnet und für die statistische Analyse abgetastet.

Zur Erkennung von Kollisionen zwischen zwei OBBs wird eine Folgerung aus Satz 4.1 verwendet, das *separating axis theorem* [Gottschalk 1996]. Eine *trennende Gerade* (separating axis) ist eine Gerade, auf der die Projektionen zweier Polytope nicht überlappen (Abb. 4.3).

Satz 4.3 (Separating axis theorem) *Zwei Polytope sind genau dann disjunkt, wenn es eine trennende Gerade gibt, die senkrecht auf der Seitenfläche eines oder einer Kante je eines Polytops steht.*

Gottschalk et al. geben an, dass sie für die Suche einer trennenden Geraden zwischen zwei Quadern nicht mehr als 200 Operationen benötigen

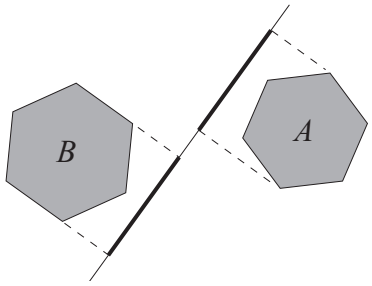


Abbildung 4.3: Trennende Gerade zwischen Polytopen

und um den Faktor zehn schneller sind als der GJK-Algorithmus (für allgemeine Polytope ist dieser Ansatz weniger geeignet, da die zu testenden Konfigurationen mit der Zahl der Flächen und Kanten schnell zunimmt).

Der Aufwand zur Berechnung der Hüllkörper-Hierarchie kann verringert werden, wenn man achsenparallele Quader (AABBs – axis-aligned bounding boxes) verwendet. Um sie nicht bei jeder Objekttransformation anpassen zu müssen, berechnet van den Bergen [1997] eine AABB-Hierarchie im jeweiligen objekteigenen Koordinatensystem. Zum Kollisionstest verwendet er ebenfalls Satz 4.3, testet aber nur die am häufigsten vorkommenden Konfigurationen. Dadurch wird der AABB-Baum zwar tiefer durchlaufen als geometrisch notwendig wäre, es werden aber trotzdem weniger Rechenoperationen durchgeführt. Von van den Bergen wird beschrieben, dass seine AABB-Implementierung in der Bibliothek SOLID um einen Faktor 1,4 bis 2,6 langsamer ist als die OBB-Implementierung von Gottschalk et al. (RAPID). Der große Vorteil von AABBs ist aber, dass sie sich sehr viel leichter an deformierbare Objekte anpassen lassen. Allerdings nimmt bei einer starken Deformation des Objekts der Überlapp der Quader in den einzelnen Ebenen der AABB-Hierarchie zu. Es werden dann mehr Tests notwendig als bei einem für eine bekannte Objektform aufgebauten eng angepassten AABB-Baum.

Cohen et al. [1995] gehen noch einen Schritt weiter und ordnen jedem Objekt eine AABB zu, die nicht im Objekt-, sondern im Welt-Koordinatensystem ausgerichtet wird. Bei Objektrotationen muss die AABB an die neuen Extrema angepasst werden. Je nach Problemstellung kann auch ein Würfel als AABB gewählt werden, der gerade groß genug ist, dass seine Grenzen bei Objektrotationen nicht überschritten werden können.

Die AABBs der einzelnen Objekte werden auf die x-, y- und z-Achse projiziert. Eine Kollision zwischen zwei AABBs tritt genau dann ein, wenn die erhaltenen Intervalle auf allen drei Achsen überlappen. Bei zeitlicher Kohärenz kann die Zeit für den Überlapp-Test von $O(n \log n)$ für den all-

gemeinen Fall auf $O(n + \varepsilon)$ reduziert werden.

4.1.3 Raumaufteilungen

Die bisher vorgestellten Ansätze nutzten bestimmte Eigenschaften der Objekte aus (Konvexität) oder erzeugten zusätzliche Informationen über ihre Struktur (Hüllkörper-Hierarchien). Man kann auch eine andere Perspektive wählen und dem Raum, in dem sich die Objekte bewegen, eine Struktur geben.

Gibson [1995] schlägt im Zusammenhang mit der Simulation und Darstellung voxel-basierter Objekte vor, den Raum mit einem regelmässigen *Belegungsgitter* (*occupancy map*) zu versehen. Jede einzelne Zelle enthält einen Nullzeiger oder einen Zeiger auf das Voxel des entsprechenden Objekts, das sich an dieser Stelle befindet. Eine Kollision findet dann statt, wenn Voxel von mehreren Objekten an einer bestimmten Stelle des Belegungsgitters eingetragen werden müssten. Der Zugriff kann in $O(1)$ Schritten erfolgen; Füllen und Kollisionstest benötigt daher bei n Voxeln $O(n)$ Schritte. Problematisch an diesem Ansatz ist der hohe Speicherplatzbedarf des Gitters.

Gregory et al. [1999] beschreiben einen hybriden Ansatz, um den besonderen Anforderungen haptischer Interaktion (hohe Updaterate, aber punktweise Interaktion am *haptic interaction point*) gerecht zu werden. Der Raum wird dazu mit einem groben Belegungsgitter aufgeteilt. In jeder Zelle des Gitters befindet sich ein OBB-Baum für diejenigen Teile des Objekts, die mit der Zelle überlappen. Durch Variation der Gittergröße kann ein angemessener Kompromiss zwischen Speicherplatzbedarf und Zugriffsgeschwindigkeit erreicht werden.

Zugunsten eines geringeren Speicherplatzbedarfs wird häufig anstelle eines Belegungsgitters ein Octree verwendet, der den Raum mit variabler Auflösung aufteilt. Eine hinreichend gleichmäßige Verteilung der Objekte vorausgesetzt, ändert sich das Zugriffsverhalten im Vergleich zu einem Belegungsgitter von $O(1)$ auf $O(\log n)$ [Bandi und Thalmann 1995]. Ganovelli et al. [2000] ordnen jedem Objekt eine AABB zu, die intern als Octree organisiert ist. Unter der Voraussetzung zeitlicher Kohärenz kann diese Struktur für deformierbare Objekte in $O(n)$ Schritten aktualisiert werden.

4.1.4 Bildbasierte Ansätze

Der *Bildraum* (*image space*) ist die Repräsentation einer Szene, die man erhält, nachdem die Objekte der Szene die Rasterisierungsstufe einer Grafik-Pipeline durchlaufen haben. Er besteht aus einem rechteckigen Array von Pixeln, die Informationen wie Farbe und Alpha-Wert, Entfernung

vom Beobachter (z-Wert) oder Stencil-Daten tragen. Meist werden die einzelnen Werte in getrennten *Buffern* gespeichert (*Color-Buffer, Alpha-Buffer, z-Buffer, Stencil-Buffer, ...*).[9]

Die Lösung geometrischer Probleme im Bildraum bietet zwei Vorteile: zum einen wird der Lösungsraum diskretisiert, so dass schnelle Näherungsverfahren mit definierbarer Genauigkeit auf einfache Weise formuliert werden können. Zum anderen können Teile des Verfahrens auf der Grafikkarte durchgeführt werden, wenn diese die benötigten Operationen unterstützt.

Algorithmen im Bildraum gibt es nicht nur im Bereich der Kollisionserkennung. Theoharis et al. [2001] gibt einen Überblick über z-Buffer-basierte Algorithmen unter anderem für die Berechnung von Voronoi-Diagrammen, CSG-Objekten, Schattenwurf (s. auch Kapitel 6), Voxelisierung und der Objektrekonstruktion aus Kamerabildern.

Bei der Kollisionserkennung kann der Bildraum als eine besondere Art der Raumaufteilung aufgefasst werden. Die einzelnen Zellen dieser Aufteilung sind Quader (bei einer Parallelprojektion) bzw. Kegelstümpfe (bei einer Zentralprojektion), die durch die *near plane* und die *far plane* des dargestellten Volumens begrenzt werden. Alle Objektteile, die sich in einer solchen Zelle befinden, sind Kandidaten für die Rasterisierung auf das Pixel, das die Basisfläche der Zelle bildet. Dies führt zu einem einfachen Ausschlusskriterium für Kollisionen:

Kollisionskriterium 1 *Objekte kollidieren nicht miteinander, wenn ihre rasterisierte Darstellung nicht an mindestens einem Pixel überlappt.*

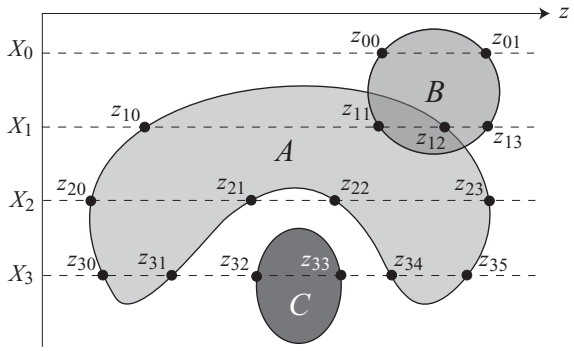


Abbildung 4.4: Die Oberflächen der Objekte A, B, C werden auf die Pixel $X_0, \ldots, X_3$ rasterisiert. Abbildung nach Shinya und Forgue [1991].

[9] Grundlagen der Echtzeit-Computergrafik kann man z.B. bei Foley et al. [1996] nachlesen.

Der Umkehrschluss trifft nicht zu. Im Beispiel von Abb. 4.4 werden Teile der Objekte A und C auf das Pixel X_3 rasterisiert, obwohl die beiden Objekte nicht kollidieren. In dem für bildbasierte Ansätze grundlegenden Artikel von Shinya und Forgue [1991] wird ein Algorithmus vorgeschlagen, der die nach Objekten gekennzeichneten z-Werte jedes Pixels in eine Liste abspeichert und nach Größe sortiert. Für Abb. 4.4 ergibt dies die folgenden Listen:

$$
\begin{aligned}
&X_0{:}(z_{00}, B), (z_{01}, B)\\
&X_1{:}(z_{10}, A), (z_{11}, B) | (z_{12}, A), (z_{13}, B)\\
&X_2{:}(z_{20}, A), (z_{21}, A) | (z_{22}, A), (z_{23}, A)\\
&X_3{:}(z_{30}, A), (z_{31}, A) | (z_{32}, C), (z_{33}, C) | (z_{34}, A), (z_{35}, A)
\end{aligned}
$$

Wie durch die Pipe-Symbole („|“) angedeutet, werden jeweils zwei Paare (z-Wert, Objekt-Bezeichner) zusammengefasst. Man erhält so das folgende Kollisionskriterium:

Kollisionskriterium 2 *Wenn ein Paar mit zwei verschiedenen Objekt-Bezeichnern existiert, kollidieren die entsprechenden Objekte.*

Im Beispiel bezeugt das Pixel X_1 die Kollision zwischen den Objekten A und B.[10]

Obwohl dieser Algorithmus bereits vor einiger Zeit (1991) vorgeschlagen wurde, unterstützen auch moderne Grafikkarten (2002) noch keinen z-Buffer, der mehrere Werte speichern oder gar sortieren kann. Für eine bestimmte Objektklasse, die *z-konvexen Objekte* können trotzdem hardwareunterstützte Grafikoperationen verwendet werden. Für die weitere Darstellung wird daher zunächst der Begriff der z-Konvexität nach Myszkowski et al. [1995] eingeführt.[11]

Definition 4.1 (z-Konvexität) *Ein Objekt A heißt* z-konvex*, wenn für eine Richtung z und jeden Punkt $P \in A$ gilt: der Schnitt von A mit der durch P in Richtung z verlaufenden Geraden ist zusammenhängend.*

[10] In einigen Fällen ist das Kriterium 2 nicht anwendbar, etwa bei Objekten, die kein definiertes „Inneres“ besitzen – Membranen oder Objekte, deren Oberflächenrepräsentation Löcher enthält. Ebenfalls nicht berücksichtigt wird der Spezialfall, dass die Objektoberfläche eine Projektionslinie nicht schneidet, sondern lediglich berührt, wie dies in Abb. 4.4 geschehen würde, wenn z.B. z_{21} und z_{22} zusammenfielen.

[11] z-Konvexität wird dort als π-Konvexität bezeichnet. Da die Richtung π mit der z-Achse des OpenGL-Standard-Koordinatensystems zusammenfällt, wird die Darstellung vereinfacht, wenn die Namensgebung diesen Umstand widerspiegelt.

Der Begriff ist auch für den bei intraokularen Operationen wichtigen Spezialfall von Membranen anwendbar: z-konvexe Membranen sind genau diejenigen, bei denen jeder Schnitt zwischen der Membran und einer Geraden in Richtung z, die durch einen Punkt auf der Membran geht, nur den Punkt selbst enthält. Abb. 4.5 erläutert die Definition am Beispiel eines volumenartigen Objekts und zweier Richtungen z und $\tilde{z}$.

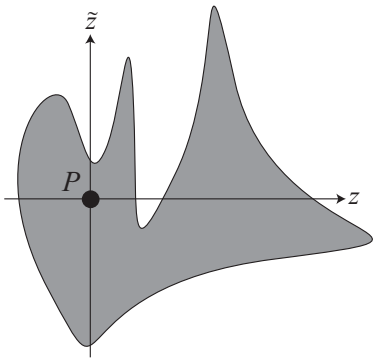


Abbildung 4.5: Zur Definition der z-Konvexität. Das Objekt ist $\tilde{z}$-konvex, aber nicht z-konvex.

Für die bildbasierte Kollisionserkennung sind solche Objekte relevant, wenn die Richtung z mit der Projektionsrichtung einer Parallelprojektion zusammenfällt. Die Rasterisierung eines z-konvexen volumenartigen Objekts ergibt dann an jedem Pixel ein Intervall. Zur Überprüfung des Kollisionskriteriums 2 für zwei Objekte A und B muss pixelweise die relative Lage der jeweiligen Intervalle $[a_{\min}, a_{\max}]$ und $[b_{\min}, b_{\max}]$ überprüft werden. Abb. 4.6-(1-7) stellt die möglichen Fälle dar.

Shinya und Forgue beschreiben eine Implementierung für z-konvexe Objekte[12], die lediglich voraussetzt, dass der z-Buffer-Test mit verschiedenen Vergleichen durchgeführt werden kann ($\leq$ und $\geq$. Konvention: wenn der Vergleich mit dem entsprechenden z-Buffer-Eintrag wahr ist, wird der Eintrag ersetzt). Die einzelnen Objekte werden separat mit je einem $\leq$- und einem $\geq$-Test gerendert; aus den resultierenden z-Buffern können direkt die Minimal- und Maximalwerte der von dem jeweiligen Objekt belegten z-Intervalle abgelesen werden.

Problematisch an diesem Verfahren ist neben der Einschränkung auf z-konvexe Objekte, dass zur Kollisionserkennung zwischen zwei Objekten insgesamt vier z-Buffer in den Hauptspeicher kopiert (*buffer readback*) und dort von der CPU miteinander verglichen werden müssen. Buffer readback

[12] In der Veröffentlichung von Shinya und Forgue wird die stärkere Eigenschaft der Konvexität verwendet. Tatsächlich ist die angegebene Vorgehensweise aber auch dann möglich, wenn nur z-Konvexität in Projektionsrichtung vorliegt.

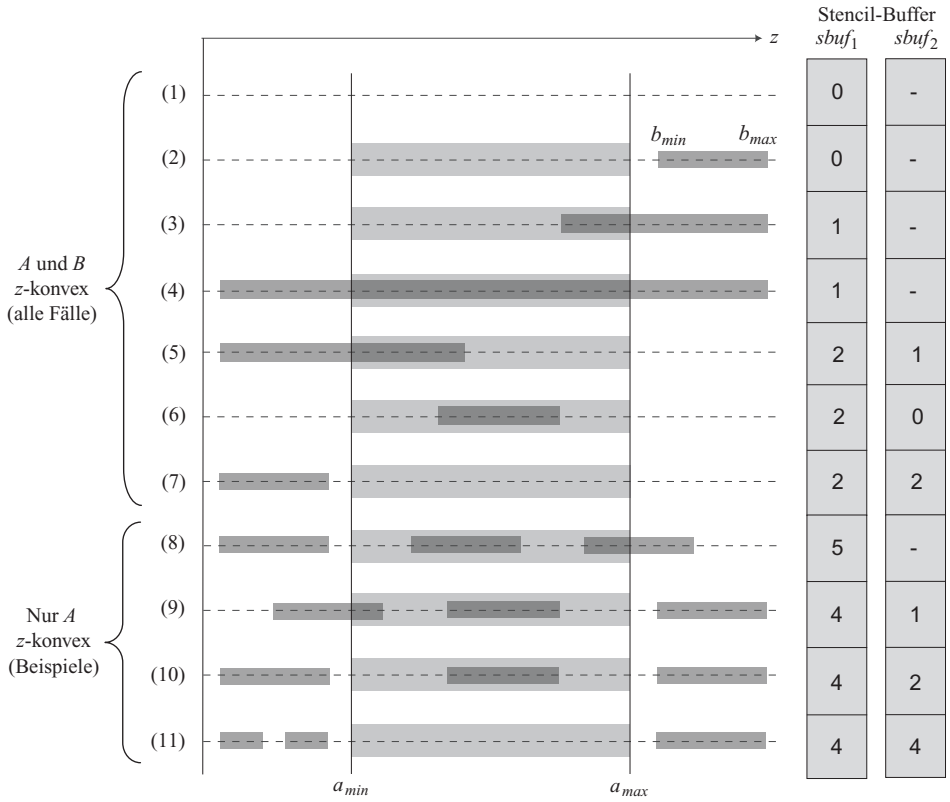


Abbildung 4.6: Überlapp-Konfigurationen der an einem Pixel rasterisierten Intervalle. Die Konfigurationen 1-7 entstehen bei der Kollision zweier z-konvexer Objekte, die Konfigurationen 8-11 sind Beispiele für Kollisionen, wenn das Objekt B nicht z-konvex ist. Die beiden rechten Spalten werden für die Erläuterung des Verfahrens von Myszkowski et al. [1995] benötigt. Der obere Teil der Abbildung stammt ebenfalls von Myszkowski et al.

ist eine sehr zeitaufwändige Operation; durch den Vergleich der einzelnen Buffer wird die CPU belastet.

Myszkowski et al. schlagen ein Verfahren vor, das auf der Idee von Shinya und Forgue aufbaut, aber weniger Render- und Kopiervorgänge benötigt. Um die von der CPU durchgeführten Vergleiche auf die Grafikkarte zu verlagern, nutzen sie zusätzlich zum z-Buffer Operationen auf einem Stencil-Buffer.

Das Verfahren geht von der Beobachtung aus, dass bei einer Kollision mit einem z-konvexen Objekt A mindestens an einem Projektionsstrahl in z-Richtung mindestens ein Intervall des anderen Objekts (B) von einer Oberfläche von A geschnitten wird. Mit anderen Worten: Von dieser Oberfläche aus betrachtet, liegt zu größerem wie zu kleinerem z je eine ungerade Zahl von Intervallgrenzen von B. Beispielsweise liegt in Abb. 4.6-(8) „rechts“ von a_{max} eine Intervallgrenze; „links “ von a_{max} befinden sich fünf Intervallgrenzen.

Mit Hilfe des Stencil-Buffers kann die Anzahl der Intervallgrenzen auf einer Seite von a_{max} bzw. a_{min} gezählt werden. In Abb. 4.6 führen für a_{max} im ersten Render-Schritt und für a_{min} im zweiten Render-Schritt diejenigen Intervallgrenzen (rasterisierte Oberflächen von B) zu einer Erhöhung des Stencil-Buffers ($sbuf_1$ und $sbuf_2$), die einen $<$-Vergleich mit der jeweiligen Oberfläche von A bestehen.

In Alg. 4.1 ist das Verfahren in Pseudocode angegeben. Im Vergleich zu Myszkowski et al. wurde eine Änderung vorgenommen: Dort wird eine eigene Bit-Ebene verwendet, um diejenigen Pixel zu markieren, an denen A rasterisiert wurde. In den **for**-Schleifen wird diese Ebene dazu verwendet, um zu verhindern, dass der Stencil-Buffer an Stellen überprüft wird, an denen die Rasterisierungen von A und B nicht überlappen (was zu falschen Klassifizierungen führen würde). Dieser Test ist aber nicht notwendig, da Pixel, die nicht zu A gehören, den z-Wert 0 behalten. Der Stencil-Buffer wird beim Rasterisieren von B an diesen Stellen nicht beschrieben, so dass auch auf $sbuf_1 > 0$ überprüft werden kann.

Im Einzelnen besteht der Algorithmus aus den folgenden Schritten:

- Zum Ermitteln der a_{max}-Werte wird das Objekt A mit einem $\geq$-Vergleich in den z-Buffer gerendert.
- Objekt B wird nun in den Stencil-Buffer gerendert, wobei der z-Test mit einem $<$-Vergleich durchgeführt wird. Für jedes rasterisierte Polygon wird der Wert des Stencil-Buffer-Eintrags ($sbuf_1$ in Abb. 4.6) um eins erhöht.
- Wenn alle Stencil-Einträge auf Null stehen, hat keine Kollision stattgefunden. Ist der Stencil-Eintrag ungerade, wurde der Schnitt eines

Algorithmus 4.1 Kollisionserkennung nach Myszkowski et al. [1995]

```
1: ClearZBuffer (0)
2: ClearStencilBuffer (0)
3: SetZBufferTest (≥)
4: RenderObjectA (Z) // must be z-convex
5: SetZBufferTest (<)
6: SetStencilOp (INCREMENT_ON_Z_SUCCESS)
7: RenderObjectB (STENCIL)
8: second_rendering ← false
9: sbuf_1 ← ReadStencilBuffer ()
10: for all pixels (x, y) do
11:    if sbuf_1(x, y) odd then
12:       return COLLISION
13:    if sbuf_1(x, y) ≠ 0 then
14:       second_rendering ← true
15: if ¬second_rendering then
16:    return NO_COLLISION
17: ClearZBuffer (∞)
18: ClearStencilBuffer (0)
19: RenderObjectA (Z)
20: RenderObjectB (STENCIL)
21: sbuf_2 ← ReadStencilBuffer ()
22: for all pixels (x, y) where sbuf_1(x, y) > 0 do
23:    if sbuf_2(x, y) odd ∨ sbuf_1(x, y) ≠ sbuf_2(x, y) then
24:       return COLLISION
25: return NO_COLLISION
```

B-Intervalls erkannt und damit eine Kollision festgestellt. Wenn der Eintrag gerade ist, muss ein zweites Mal gerendert werden.

- Im zweiten Render-Schritt wird das Objekt A mit einem $<$-Vergleich in den z-Buffer gerendert, so dass an jedem Pixel die a_{min}-Werte stehen.
- Objekt B wird ebenfalls mit einem $<$-Vergleich gerendert, so dass man die Einträge der $sbuf_2$-Spalte in Abb. 4.6 erhält.
- Eine Kollision hat dann stattgefunden, wenn an den mehrdeutigen Stellen ($sbuf_1(x, y)$ gerade und ungleich Null) in $sbuf_2$ ein ungerader Wert steht (Schnitt zwischen a_{min} und einem Intervall von B) oder $sbuf_2(x, y)$ zwar gerade, aber ungleich $sbuf_1(x, y)$ ist (ein Intervall von B ist komplett in A enthalten).

Im Vergleich zum Ansatz von Shinya und Forgue sind weniger Readbacks notwendig. Wenn nur ein ja/nein-Kollisionstest durchgeführt werden soll, kann das Verfahren häufig schon nach Pass 1 abgebrochen werden. Ist schon vor der Kollisionserkennung bekannt, welches Objekt näher am Beobachter ist (kleineres z), sollte dieses als Objekt A gewählt werden, um die Ausführung von Pass 2 in möglichst vielen Fällen zu vermeiden.

Von Baciu und Wong [1997], Baciu et al. [1998] wird das Verfahren erweitert, indem mit Hilfe von Bounding-Boxen der kleinste rechteckige Bereich der Projektionsebene bestimmt wird, auf dem Teile von beiden Objekten rasterisiert werden (*minimum overlap region*). Nur dieser Bereich ist für die Kollisionstests relevant, so dass der teure Buffer-Readback darauf beschränkt werden kann.

4.1.5 Selbstkollisionen

Deformierbare Objekte können mit sich selbst kollidieren. Die in den vorangegangenen Abschnitten vorgestellten Ansätze kommen damit nicht alle gleichermaßen zurecht: so können sich die Oberflächen von Polytopen höchstens berühren (in dem entarteten Fall, dass das Polytop zu einer Fläche kollabiert ist), aber nicht schneiden.[13]

Eine Hüllkörper-Hierarchie kann zwar „gegen sich selbst“ verglichen werden – problematisch ist dabei aber, dass für räumlich eng zusammenliegende Elemente des Objekts auch dann fast die gesamte Hierarchie durchlaufen werden muss, wenn sie nicht kollidieren. Volino und Thalmann [1994]

[13] Kuffner et al. [2002] verwenden trotzdem den Begriff der Selbstkollision im Zusammenhang mit der Kollisionserkennung von Polytopen. Sie meinen damit aber ein zusammengesetztes Objekt, dessen einzelne Bestandteile konvex sind.

und Provot [1997] schlagen daher vor, mit Hilfe der lokalen Oberflächenkrümmung Bereiche zu finden, in denen keine Selbstkollision stattfinden kann, so dass die Suche auf der Hüllkörper-Hierarchie eingeschränkt werden kann.

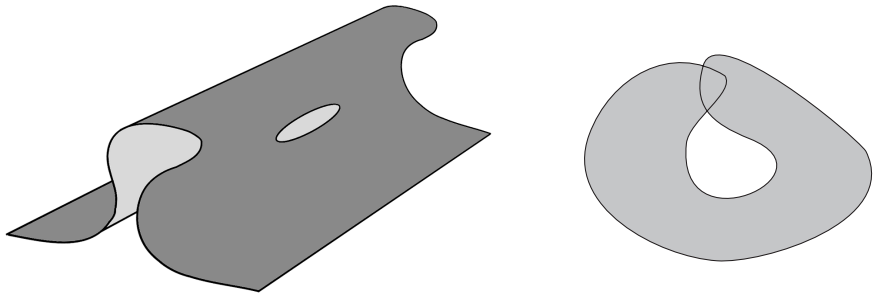


Abbildung 4.7: Selbstkollisionen aufgrund von Krümmung (1) und nichtkonvexer Kontur (2) (nach Volino und Thalmann [1994])

Zur Quantifizierung der Krümmung wird der Kegel mit minimalem Öffnungswinkel verwendet, der alle Normalen eines Oberflächenausschnitts enthält. Wenn der Öffnungswinkel kleiner als 90° ist, kann keine Selbstkollision wie in Abb. 4.7-(1) stattfinden, bei der sich die Fläche im Raum auffaltet. Bemerkung: Diese Bedingung ist äquivalent mit der Forderung, dass die Oberfläche z-konvex in der Richtung der Symmetrieachse des Kegels ist.

Volino und Thalmann schließt einen Fall wie in Abb. 4.7-(2) durch die zweite Forderung aus, dass sich die Projektion der Fläche in z-Richtung nicht schneidet. Provot vernachlässigt diese Bedingung mit der Begründung, dass sie bei der in der Veröffentlichung untersuchten Simulation von Textilien selten auftritt.

Die hier diskutierten bildbasierten Verfahren können z-Konvexität leicht feststellen (Kriterium: ein Objekt ist z-konvex, wenn nach dem additiven Rendern in den Stencil-Buffer kein Eintrag > 1 ist). Es bleibt zu untersuchen, ob auf diese Weise das Verfahren von Volino und Thalmann und Provot beschleunigt werden kann – für ein geometrisches Verfahren spricht aber, dass der aufwändigste Teil, die Berechnung der Normalen, für die Visualisierung ohnehin durchgeführt werden muss.

Mit den in Abschnitt 4.1.3 beschriebenen Raumaufteilungen kann ebenfalls festgestellt werden, ob eine Selbstkollision stattgefunden hat. Auch hier muss berücksichtigt werden, dass benachbarte Objektteile auch ohne Kollision in die gleiche Zelle einsortiert werden.

Für die bisher realisierten Gewebeinteraktionen wurde die Erkennung von

Selbstkollisionen nicht benötigt. In der VRM-Bibliothek sind daher noch keine entsprechenden Verfahren implementiert.

4.2 ZCOLL

ZCOLL ist eine Erweiterung des Verfahrens von Myszkowski et al. für Gewebe-Instrument-Interaktionen. Es wird gezeigt, dass dieses Verfahren in einer solchen Konfiguration auf einen Render-Durchgang verkürzt werden kann. Die Auswertung zusätzlicher Informationen im z- und Color-Buffer hilft bei der Kollisionsantwort. In den nächsten Abschnitten werden diese Änderungen im einzelnen beschrieben, um anschließend als Algorithmus formuliert und auf Genauigkeit und Zeitverhalten untersucht zu werden.

4.2.1 Beschränkung auf flächenartige Objekte

Bei Gewebe-Instrument-Interaktionen kann die Kollisionserkennung auf flächenartige Objekte beschränkt werden. Warum dies so ist, zeigt Abb. 4.8: ein Instrument B steht in verschiedenen geometrischen Beziehungen zu einem Gewebestück A. Die Silhouette von B entspricht einem *Vitrektom*, wie es in der intraokularen Chirurgie verwendet wird; ohne die Einsaugöffnung an der Spitze des Instruments wäre es z-konvex.

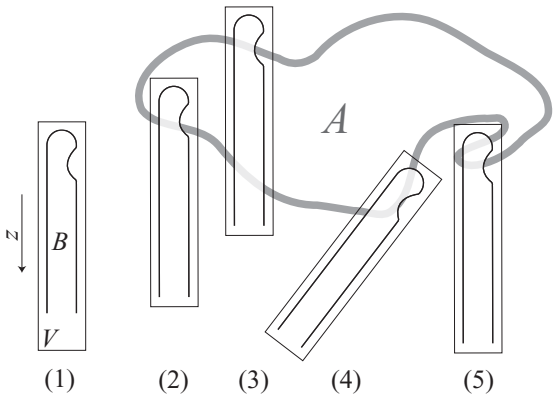


Abbildung 4.8: Verschiedene Konfigurationen bei der Interaktion zwischen einem Gewebestück A und einem Instrument B. Um die Buffer-Auflösung möglichst gut auszunutzen, wird das Rendering-Volumen V eng an das Instrument angepasst. V und z sind objektfest.

Die Konfigurationen (2) und (4) entstehen, wenn der Chirurg beginnt, mit dem Gewebe zu interagieren. Solange er zielgerichtete, vorsichtige Bewegungen durchführt, wird Objekt A schnell genug ausweichen (durch Deforma-

tion oder Bewegung), so dass die Konfigurationen (3) und (5) gewöhnlich nicht eintreten.[14]

Aus diesem Grund und wegen des häufigen Auftretens von Membranen in der intraokularen Chirurgie wird von einer Interaktion mit einem in V flächenartigen Objekt A ausgegangen. In Abb. 4.6 wird aus dem Intervall $[a_{min}, a_{max}]$ dann ein einzelner Punkt. Kollisionen werden genau dann erkannt, wenn $sbuf_1$ ungerade ist. Ein zweiter Rendering-Schritt ist dann nicht notwendig.

4.2.2 Deformationsvektoren

z-Buffer-Einträge können als Anhaltspunkt für die Deformation des Gewebes genutzt werden: An einem Pixel (x, y) enthalte der z-Buffer nach dem Rendern der beiden Objekte die Werte $a(x, y)$ und $b_{min}(x, y)$ (Abb. 4.9).

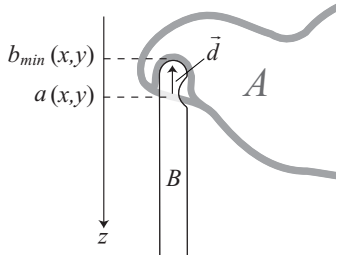


Abbildung 4.9: Bei der Kollision eines Instruments mit einem Gewebestück werden Deformationsvektoren $\vec{d}$ entlang der z-Richtung bestimmt.

Das kollidierende Polygon von A muss so verschoben werden, dass es an der Stelle (x, y) nicht mehr mit dem z-Wert $a(x, y)$, sondern mit $b_{min}(x, y)$ rasterisiert wird. Mit Hilfe der Projektionsmatrix M_P, die die A-eigenen Koordinaten in Bildschirmkoordinaten transformiert, erhält man den Deformationsvektor $\vec{d}$ in den Objektkoordinaten von A wie folgt:

$$\vec{d} = M_P^{-1} \begin{pmatrix} 0 \\ 0 \\ b_{min}(x,y) - a(x,y) \end{pmatrix} \tag{4.2}$$

[14] Dieser Argumentation liegt die Skaleninvarianz des Gesetzes von Fitt zugrunde, die in Abschnitt 1.3.3 beschrieben wurde: der Chirurg muss bei der Interaktion mit dem kleineren Gewebelappen in Konfiguration (5) langsamere Bewegungen durchführen, so dass die Abtastfrequenz nicht verändert werden muss. Dies gilt nur für kontrollierte Bewegungen, insbesondere also dann nicht, wenn die Größe der Strukturen ähnlich der Amplitude des menschlichen Zitterns ist.

Die Deformationsvektoren weisen immer parallel zur z-Richtung. Je nach Situation kann es daher angemessen sein, z in Richtung der Instrumentbewegung zu wählen. In diesem Fall muss in jedem Rendering-Schritt ein neues z-Profil des Instruments aufgenommen werden (im unten angegebenen Algorithmus 4.2.4 wird dazu vor jedem Aufruf $zbuf_B$ auf Null gesetzt).

Damit ein Polygonnetz nicht auseinanderfällt, müssen bei der Deformation die Nachbarschaftsbeziehungen (gemeinsame Knoten, gemeinsame Kanten) erhalten bleiben. Jedem Knoten wird dazu die Menge aller Polygone zugeordnet, die an ihn grenzen. Jedes einzelne Polygon wiederum wird üblicherweise auf mehrere Pixel rasterisiert. Jedes Pixel eines jeden Polygons trägt einen möglichen Deformationsvektor zu einem Knoten v bei, so dass man eine Menge $\mathbb{D}_v$ von Vektoren erhält. Der auf einen Knoten angewendete Deformationsvektor kann z.B. als Maximum oder Durchschnitt der Elemente von $\mathbb{D}_v$ gewählt werden. Abb. 4.10 zeigt das Ergebnis einer solchen Deformation.

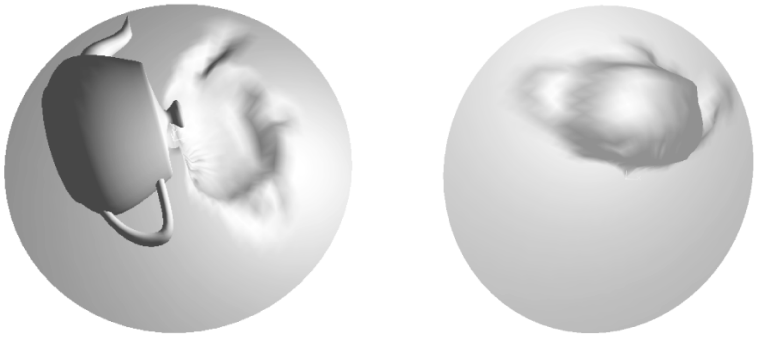


Abbildung 4.10: Beispiel für die Anwendung von Deformationsvektoren. Links die Vorder-, rechts die Rückseite der deformierten Membran.

4.2.3 Identifizierung kollidierender Polygone

Rendering ist eine Abbildung $R : G \rightarrow B$ von der geometrischen Beschreibung G eines Objekts auf die einzelnen Pixel des Bildraums B. Die Umkehrabbildung R^{-1} wird benötigt, um auf ein an einem bestimmten Pixel rasterisiertes Polygon zuzugreifen (z.B. zur Zuordnung von Deformationsvektoren).

Zur schnellen Berechnung von R^{-1} muss den einzelnen Pixeln die Information über das geometrische Urbild mitgegeben werden. ZCOLL nutzt dafür den Color-Buffer: Jedem Polygon wird eine eindeutige Farbe zugewiesen. Üblicherweise hat der Color-Buffer eine Farbtiefe von 24 Bit, so dass 16,7 Millionen Polygone unterschieden werden können. Wenn die Farben fortlaufend vergeben werden, erhält man eine Tabelle der Größe $O(n)$

(n: Anzahl der Polygone); R^{-1} kann dann in $O(1)$ Schritten berechnet werden.

Bemerkung: OpenGL stellt mit dem *select buffer* und dem *feedback buffer* ebenfalls Mechanismen zur Verfügung, um das Urbild von R zu ermitteln. Statt die Rasterisierungsstufe auszuführen, werden Arrays mit Informationen über die gerenderten Objekte gefüllt: symbolische Bezeichner (im Select-Modus) bzw. geometrische Daten wie Positionen, Normalen oder Farben (im Feedback-Modus). Ohne Rasterisierung kann aber nur festgestellt werden, welche Objekte die Clipping-Stage passiert haben, nicht jedoch, ob sie einen z- oder Stencil-Test bestanden haben.

Lombardo et al. [1999] schlagen daher vor, ein Operationsinstrument mit Schnittebenen zu beschreiben, um schon in der Clipping-Stufe kollidierende Polygone zu finden. Da dieses Verfahren vor der Rasterisierung ansetzt, werden Geschwindigkeit und Genauigkeit nicht durch Buffer-Readbacks und die beschränkte Auflösung beim Rendering beeinflusst. Es hat aber zwei Nachteile: (1) OpenGL unterstützt lediglich sechs feste Schnittebenen (zur Beschreibung des Rendering Volumens) sowie sechs frei definierbare Schnittebenen. Die Geometrie eines Operationsinstruments kann daher nur sehr grob angenähert werden. (2) Obwohl die Transformationsstufe beim normalen Rendering auf der Grafikkarte durchgeführt wird, gibt es unseres Wissens keine Karte, mit deren Hilfe eine Hardwarebeschleunigung für das Schreiben in die Select- und Feedback-Buffer möglich ist. Es muss dann im Softwaremodus gerendert werden, was zu Geschwindigkeitseinbußen und höherer CPU-Last führt.

4.2.4 Pseudocode-Formulierung

Im Unterschied zum Verfahren von Myszkowski et al. muss ZCOLL (Alg. 4.2) zur Erzeugung von Deformationsvektoren zunächst die Tiefeninformation aus Objekt B (dem Instrument) erzeugen. Da B nicht deformierbar ist, wird dies nur durchgeführt, falls noch kein aktuelles z-Profil verfügbar ist – wie etwa beim ersten Aufruf, falls ein Instrument seine Form geändert hat (z.B. eine Pinzette, deren Scheren geöffnet und geschlossen werden) oder falls eine neue z-Richtung festgelegt wurde. In diesen Fällen wird $zbuf_B$ vor dem Aufruf von ZCOLL auf Null gesetzt.

Objekt A wird nicht nur in den z-Buffer, sondern auch in den Farb-Buffer gerendert, um später über die Farbinformation das an einer Stelle rasterisierte Dreieck wiederfinden zu können. Da von einem oberflächenartigen, z-konvexen Objekt ausgegangen wird, könnte an dieser Stelle auch ohne z-Test gerendert werden.

Das Stencil-Rendering von Objekt B wird analog zu Algorithmus 4.1 durchgeführt. Bei einem ungeraden *sbuf*-Eintrag an einem Pixel (x, y) wird mit

Algorithmus 4.2 ZCOLL

```
1: ClearZBuffer (∞)
2: ClearStencilBuffer (0)
3: SetZBufferTest (<)
4: if zbuf_B = 0 then // first call or z-direction/B's shape have changed
5:    RenderObjectB (Z) // B: Instrument
6:    zbuf_B ← ReadZBuffer ()
7:    ClearZBuffer (∞)
8: RenderObjectA (Z|COLOR) // A: Tissue
9: zbuf_A ← ReadZBuffer ()
10: cbuf_A ← ReadColorBuffer ()
11: SetStencilOp (INCREMENT_ON_Z_SUCCESS)
12: RenderObjectB (STENCIL)
13: sbuf ← ReadStencilBuffer ()
14: for all pixels (x, y) do
15:    if sbuf(x, y) odd then
16:       T ← HashTriangle (cbuf_A(x, y))
17:       AddDeformationVector (T, (x, y, zbuf_B(x, y) − zbuf_A(x, y)))
```

Hilfe des Eintrags im Color-Buffer das kollidierende Dreieck T von A ermittelt. Es wird davon ausgegangen, dass T eine Menge von Deformationsvektoren verwalten kann, die um die z-Buffer-Differenz am Pixel (x, y) ergänzt wird.

Abb. 4.11 zeigt die einzelnen Phasen von ZCOLL an einem Beispiel.[15]

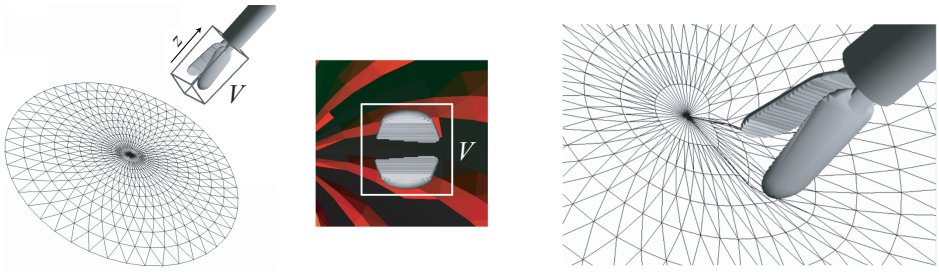
4.2.5 Variationen

Verringerung des Readback-Aufwands

In vielen Fällen ist Buffer-Readback die teuerste Operation bei der Ausführung eines bildbasierten Verfahrens (s. 4.2.7). Je nach Eigenschaften und Anforderungen der Situation kann ZCOLL modifiziert werden, um den Aufwand für die Buffer-Readbacks zu verringern:

Beschränkung auf Deformationsvektoren In diesem Fall muss der Color-Buffer nicht ausgelesen werden; es wird dann ein Feld von Deformationsvektoren zurückgegeben, das für die Simulation verwendet wird.

[15] Die in der Abbildung dargestellte Triangulierung ist für eine Simulation ungeeignet, da die Größe der Dreiecke stark variiert. Überdies besitzt der mittlere Knoten eine sehr große Anzahl von Nachbarn, was zu numerischen Instabilitäten führen kann.



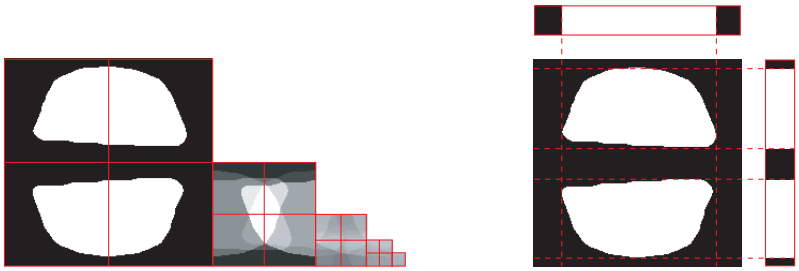
(1) (2) (3)

Abbildung 4.11: Instrument-Gewebe-Interaktion mit ZCOLL: (1) Das Rendering-Volumen V wird möglichst eng an den relevanten Bereich des Instruments angepasst. (2) Die einzelnen Dreiecke des Gewebestücks sind durch eindeutige Farben identifizierbar (gerendert wird nur der markierte Bereich). (3) Anwendung der Deformationsvektoren.

Beschränkung auf kollidierende Polygone Der z-Buffer wird vor allem zur Berechnung der Deformationsvektoren ausgelesen. Wenn diese nicht benötigt werden, müssen zwei Color-Buffer miteinander verglichen werden. Mit der Hintergrundfarbe 0 wird Bedingung (*) in Algorithmus 4.3 dann zu $cbuf_B(x, y) = cbuf_A(x, y) \wedge cbuf_B(x, y) \neq 0$.

Beschränkung auf ja/nein-Informationen Wenn nur getestet werden soll, ob überhaupt eine Kollision stattgefunden hat oder erst anschließend ein aufwändigerer Test durchgeführt wird, kann das Durchsuchen des entsprechenden Buffers auch auf die Grafikkarte verlagert werden. Dazu werden die Objekte in jeweils einer Farbe in den Color-Buffer gerendert. Die Instrumentfarbe erhält den Wahrheitswert *wahr*. Der Color-Buffer wird in vier Quadranten aufgeteilt, die OR-verknüpft übereinander gerendert werden (Abb. 4.12-(1); zur Veranschaulichung wurden die Quadranten nicht OR-verknüpft, sondern mit einem alpha-Wert übereinanderkopiert). Falls die Abmessungen des Buffers die Form 2^n haben, kann dieser Vorgang solange wiederholt werden, bis nur ein Pixel übrigbleibt, das dann zurückgelesen und auf *wahr* (Kollision) oder *falsch* (keine Kollision) überprüft werden kann.

Berechnung der ROI Wenn der Rendering-Bereich nicht in Quadranten, sondern in rechteckige Bereiche aufgeteilt wird, erhält man nach mehrfachem OR-Übereinanderkopieren je eine Zeile und Spalte (Abb. 4.12-(2)), die die *region of interest* (ROI) bestimmen, die zurückgelesen werden muss. Der Zusatzaufwand lohnt sich dann, wenn Kollisionen nur auf einem kleinen Ausschnitt des Rendering-



(1) (2)

Abbildung 4.12: (1) Wenn nur eine ja/nein-Information von der Kollisionserkennung erwartet wird, kann die Größe des zurückgelesenen Bereichs durch Übereinanderkopieren der Quadranten verringert werden. Zur Veranschaulichung wurde beim Kopieren ein Alpha-Wert <1 verwendet. (2) Zeilen- und spaltenweises Übereinanderkopieren kann zur Berechnung von ROIs verwendet werden.

Bereichs stattfinden.

Beschränkung auf Frontflächen

Für den Fall ausgeprägter Vorzugsrichtungen von Deformation oder Instrumentbewegung kann der Algorithmus weiter vereinfacht werden (Algorithmus 4.3, „ZCOLL/F“): Im Beispiel von Abb. 4.11 berührt nur die Spitze der Pinzette die kreisförmige Membran. Daher muss auch nur die Frontfläche der Pinzette für die Kollision berücksichtigt werden. Auf das Berechnen der Intervallgrenzen mit Hilfe des Stencil-Buffers und zwei Rendering-Pässen für Objekt B kann verzichtet werden, so dass der z-Buffer-Test als Kollisionskriterium ausreicht: eine Kollision findet statt, wenn an einer Stelle (x, y) das Instrument einen geringeren z-Buffer-Wert als die Membran hat (Bedingung (*) in Alg. 4.3). Die Deformationsvektoren werden auch hier über die z-Differenz ermittelt.

Auf diese Weise kann der Stencil-Buffer-Readback sowie ein Aufruf der Rendering-Methode von Objekt B eingespart werden.

4.2.6 Genauigkeit

Die Genauigkeit eines bildbasierten Verfahrens wird durch die Auflösung bei der Rasterisierung bestimmt. Dieser Zusammenhang wird im Folgenden erläutert.

Bei einer Parallelprojektion eines $[-1, 1]$-Einheitsvolumens auf einen Buffer der Größe $r \times r$ Pixel ist die laterale Pixelgröße $p_l = 2/r$. p_l bestimmt die

Algorithmus 4.3 ZCOLL/F (nur Frontfläche von Objekt B)

```
1: ClearZBuffer (∞)
2: SetZBufferTest (<)
3: if zbuf_B = 0 then // first call or z-direction/B's shape have changed
4:    RenderObjectB (Z) // B: Instrument
5:    zbuf_B ← ReadZBuffer ()
6: RenderObjectA (Z|COLOR) // A: Tissue
7: zbuf_A ← ReadZBuffer ()
8: cbuf_A ← ReadColorBuffer ()
9: for all pixels (x, y) do
10:   if zbuf_B(x, y) < zbuf_A(x, y) < ∞ then // Condition (*)
11:      T ← HashTriangle (cbuf_A(x, y))
12:      AddDeformationVector (T,(x, y, zbuf_B(x, y) − zbuf_A(x, y)))
```

Mindestausdehnung w_{min} einer erkennbaren Struktur, so dass gilt:

$$w_{min} > \frac{2}{r} \tag{4.3}$$

Abb. 4.13 stellt eine Testszene dar, bei der ein 4×4-Gitter B aus Kegeln in eine Kreisscheibe A sinkt.

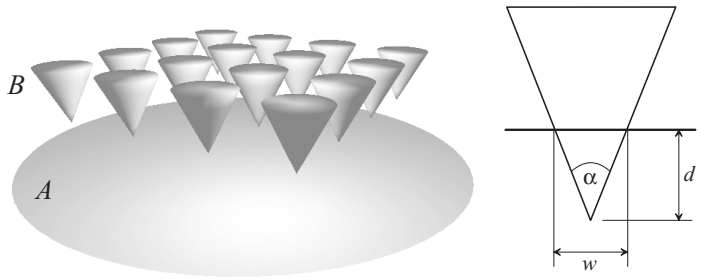


Abbildung 4.13: Testszene zur Genauigkeitsmessung: Erst bei einer bestimmten Eindringtiefe d wird die Kollision der Kegel mit der Membran erkannt.

Bei Kegeln ist die laterale Größe w der in A eingedrungenen Strukturen proportional zu ihrer Eindringtiefe d: $w = 2 \tan \frac{\alpha}{2} \cdot d$. Mit Ungleichung 4.3 gilt für die Tiefe, bei der alle eingedrungenen Kegel rasterisiert werden: $d \approx \left(r \tan \frac{\alpha}{2}\right)^{-1}$.

Abb. 4.14 bestätigt dieses Ergebnis. Bei variabler Auflösung wurde B solange abgesenkt, bis ZCOLL/F die Kollisionen mit allen Kegeln erkannt hat ($\alpha = 50°$).

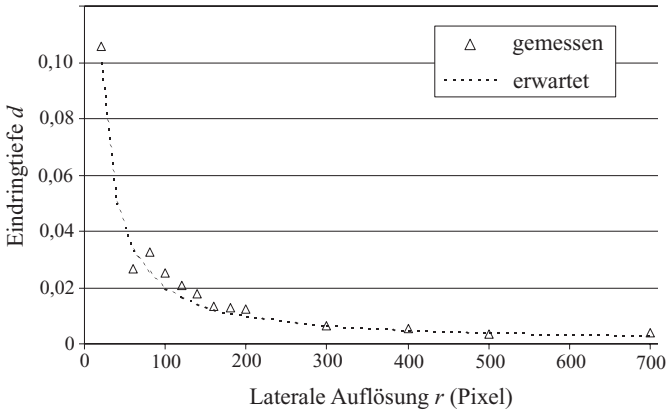


Abbildung 4.14: Abhängigkeit von Auflösung und Genauigkeit in der Szene von Abb. 4.13.

Neben der Mindestgröße der erkennbaren Strukturen wird durch die Rendering-Auflösung auch die maximale Zahl der erkennbaren Polygone bestimmt – wenn mehrere Polygone auf einem Pixel rasterisiert werden, kann nur noch eines von ihnen über seinen Farbwert erkannt werden. Abb. 4.15 stellt diesen Effekt dar.

4.2.7 Zeitverhalten

Das Zeitverhalten eines bildbasierten Verfahrens wird durch drei Arten von Operationen bestimmt:

Geometrieoperationen Dazu gehören der Geometrietransfer auf die Grafikkarte und die Anwendung der Transformations- und Projektionsmatrizen. Der Aufwand dafür ist proportional zur Anzahl n der Polygone.

Rasterisierung Für ein Rendering-Gebiet mit $r \times r$ Pixeln werden $O(n \cdot r^2)$ Schritte benötigt.

Analyse Zur Analyse müssen zunächst mit einem Aufwand von $O(r^2)$ einige Buffer-Readbacks durchgeführt werden. Um die Buffer durchzugehen und zu vergleichen, sind ebenfalls $O(r^2)$ Operationen notwendig. p sei die Anzahl der kollidierenden Polygone. Wegen $p \leq n$ kann der Zugriff auf die Farb-Tabelle und die Berechnung von Deformationsvektoren in $O(n)$ Schritten durchgeführt werden.

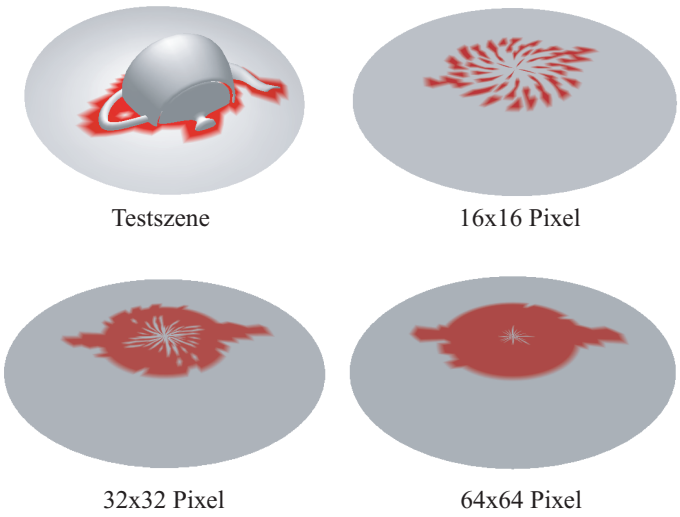


Abbildung 4.15: Mit zunehmender Rendering-Auflösung nimmt auch die Anzahl der erkannten Kollisionen (rot) zu. Die Kreisscheibe ist zur Mitte hin feiner trianguliert.

Insgesamt ergibt sich ein Zeitbedarf von $O(n \cdot r^2)$. Die folgenden absoluten Zeitmessungen wurden auf dem EyeSi-System[16] durchgeführt. Den Messungen liegt die Implementierung von ZCOLL/F in der VRM-Bibliothek zugrunde. In den folgenden drei Abschnitten werden die Schritte Geometrietransfer, Buffer-Readback und Buffer-Analyse diskutiert. Die Ergebnisse werden anschließend zusammengefasst und mit geometriebasierten Ansätzen verglichen.

Geometrietransfer

Abb. 4.16 zeigt eine Datenreihe, bei der die Zeit für einen `RenderObjectA(Z|COLOR)`-Aufruf (Zeile 7 von Algorithmus 4.3) gemessen wurde. Die horizontale Achse des Diagramms ist logarithmisch unterteilt. Dreieckstransfer und Rasterisierungsaufwand sind nicht getrennt aufgeschlüsselt, da bei den für die Kollisionserkennung verwendeten geringen Auflösungen (typischer Wert: 100 × 100 Pixel) der Dreieckstransfer der begrenzende Faktor in der Rendering-Pipeline ist.

Die Dreiecke sind in einer MGE-Struktur organisiert, ihre für die Kollisionserkennung relevanten Daten (Positionen und Farben) werden aber in

[16] Pentium-IV mit 1,8GHz; AGP 4×; NVidia GeForce 4 Ti 4400 Grafikkarte; Treiberversion: Detonator 41.09

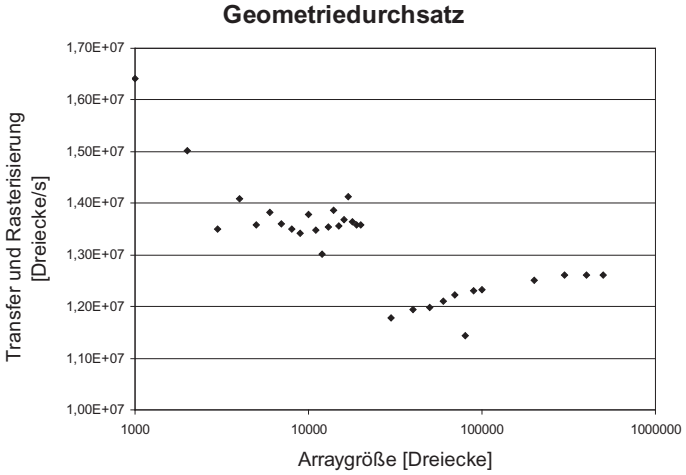


Abbildung 4.16: Geometriedurchsatz

Arrays gespeichert, die blockweise mit einem `glVertexArray()`-Aufruf gerendert werden (Kapitel 3 und 6).

Für jedes Dreieck werden neun `float`-Werte (36 Bytes) für die Koordinaten der Ecken und neun `unsigned byte`-Werte für die Farben übertragen. Da die Dreiecke zur Identifizierung von ZCOLL/F einfarbig dargestellt werden, würde ein Farbwert pro Dreieck ausreichen, in einer OpenGL-Beschreibung sind Farben aber stets den Eckpunkten, nicht den Flächen zugeordnet. Die untere Grenze von 12 Millionen Dreiecken/s in Abb. 4.16 entspricht einem Datendurchsatz von 515 MBytes/s. Der theoretische Höchstwert liegt bei einem AGP-4×-Bus bei 800 MBytes/s, wird aber mit den hier verwendeten GL-Vertex-Arrays nicht erreicht. Wie in Kapitel 6 gezeigt wird, ist es mit einer OpenGL-Erweiterung, den *NV-Vertex-Arrays* möglich, die AGP-4×-Transferrate voll auszunutzen.

Der Dreiecksdurchsatz lässt sich verdreifachen, wenn statt einzelner Dreiecke *triangle strips* verwendet werden, bei denen für das i-te Dreieck nur ein Eckpunkt x_i angegeben werden muss.[17] Problematisch an diesem Format sind topologische Änderungen, bei denen die Triangle-Strips wieder neu gebildet werden müssen. Für die bisherigen Anwendungen der VRM-Bibliothek war der Dreiecksdurchsatz von ZCOLL/F ausreichend, so dass der für Triangle-Strips notwendige algorithmische Aufwand nicht benötigt wurde.

[17] Das Dreieck wird aus den Punkten x_i, x_{i-1} und x_{i-2} zusammengesetzt. Nur für das erste Dreieck müssen alle drei Punkte angegeben werden.

Buffer-Readback

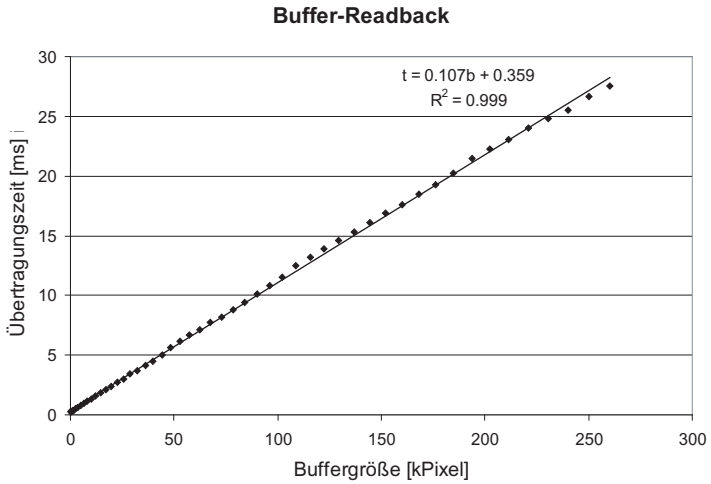


Abbildung 4.17: Zeitaufwand für den Buffer-Readback

In der Abb. 4.17 zugrundeliegenden Datenreihe wurde der Zeitaufwand für die Buffer-Readbacks von ZCOLL/F (Zeilen 5, 8 und 9 in Algorithmus 4.3) gemessen. Aufgetragen ist die Größe des Rendering-Fensters gegen den Zeitbedarf beim Readback. Dabei wurden pro Pixel zwei z-Werte und ein Farbwert übertragen ($4+2\cdot3$ Bytes/Pixel). Es besteht der folgende lineare Zusammenhang zwischen der Buffergröße b und der benötigten Zeit $t(b)$:

$$t(b) = t_0 + \frac{c}{d} \cdot b \tag{4.4}$$

Dabei ist t_0 die Zeit, die für die Etablierung des Transfers aufgewendet wird; d beschreibt den Datendurchsatz beim Readback. c dient der Anpassung der Einheiten. Aus den Parametern der Ausgleichsgeraden in Abb. 4.17 ergibt sich $t_0 = 0,36$ms und $d = 89$ MBytes/s. Der Datendurchsatz liegt damit eine Größenordnung unter dem oben gemessenen Durchsatz für den Geometrietransfer. Buffer-Readback war bisher kein Entwicklungsziel der Grafikkarten-Entwicklung – im Gegenteil: zur Entlastung von CPU und Bus werden möglichst viele Operationen vom Hostsystem auf die Grafikkarte verlagert. Überdies ist die Grafikkarten-Architektur dafür ausgelegt, einen kontinuierlichen Datenstrom bei der Erzeugung der Signale für die grafische Ausgabe zu liefern; dagegen steht nur wenig Speicherbandbreite zur Verfügung, um den Inhalt des Grafikspeichers zurückzulesen.

Buffer-Analyse

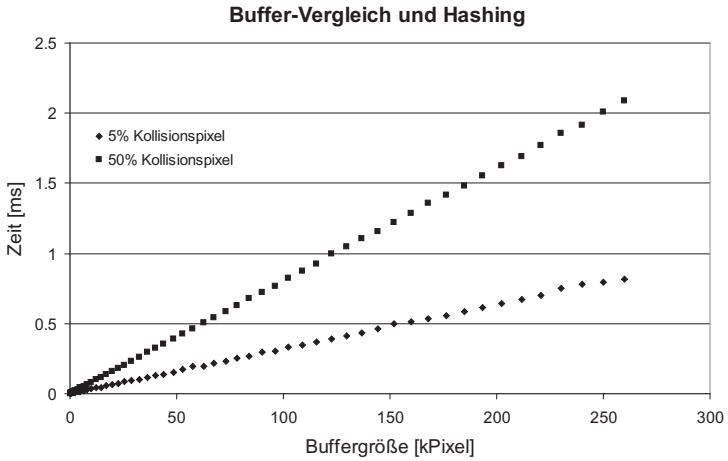
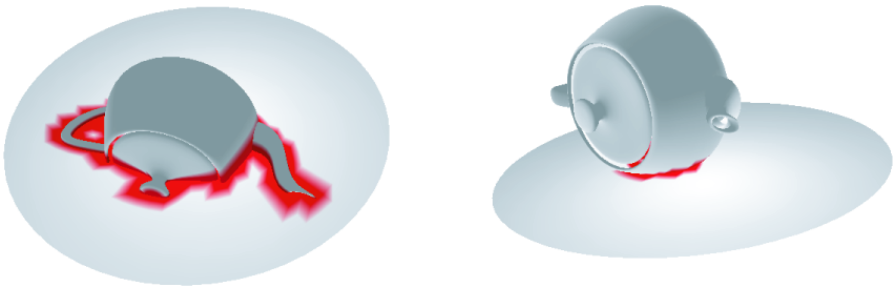


Abbildung 4.18: Buffervergleich und Hashing der kollidierenden Dreiecke

Im letzten Schritt werden die zurückgelesenen Buffer verglichen und aus der Farbinformation die kollidierenden Dreiecke ermittelt. Zur Messung des Zeitbedarfs wurde eine künstliche Situation erzeugt, bei der 5% bzw. 50% der Pixel zu kollidierenden Dreiecken gehören. Je zwei dieser Pixel hatten den gleichen Farbwert. In einer Hash-Tabelle wurde ein Zeiger auf das dazugehörige Dreieck ausgelesen und in einen vorher statisch angelegten Array abgelegt. Die Messung (Abb. 4.18) ergibt einen linearen Zusammenhang zwischen Buffergröße und aufgewendeter Rechenzeit.

Zusammenfassung des ZCOLL-Zeitverhaltens



(a) „Schwere“ Kollision (b) „Leichte“ Kollision

Abbildung 4.19: Testszene

Das zeitliche Verhalten von ZCOLL/F wird nun anhand einer Testszene (Abb. 4.19) im Ganzen dargestellt. Der Teapot wurde als typisches Benchmark-Objekt der Computergrafik gewählt. Der hier verwendete Datensatz wurde aus einem 3D-Studio-MAX-Objekt erzeugt. Die Auflösung von jeweils 2500 Dreiecken für den Teapot und die Membran entspricht einer typischen Auflösung bei der Gewebe-Instrument-Interaktion. Es werden zwei Konfigurationen betrachtet – eine „schwere“ Kollision (700 kollidierende Dreiecke der Membran) und eine „leichte“ Kollision (100 kollidierende Dreiecke).

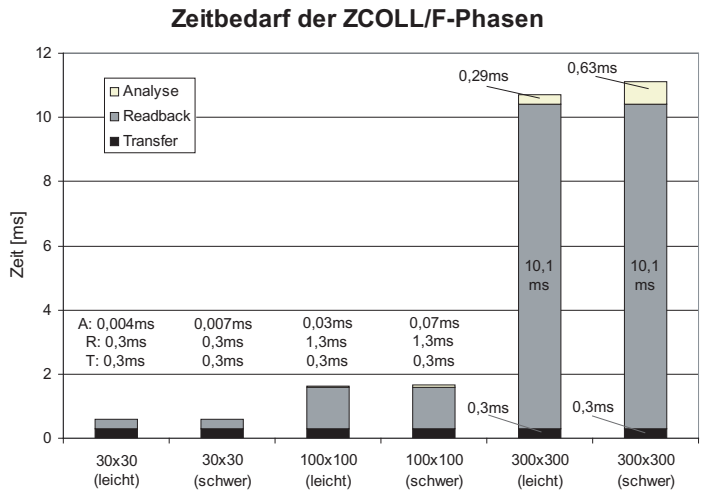


Abbildung 4.20: Zeitaufwand für die einzelnen Phasen von ZCOLL/F

Die oben einzeln betrachteten Phasen von ZCOLL/F (Geometrietransfer, Buffer-Readback und Buffer-Analyse) werden in Abb. 4.20 miteinander verglichen. Dabei wurde ZCOLL/F mit den Auflösungen 30×30, 100×100 und 300×300 Pixeln jeweils für die leichte und die schwere Kollision von Abb. 4.19 durchgeführt. Dabei zeigt sich, dass bei zunehmender Auflösung der Buffer-Readback zum begrenzenden Faktor des Verfahrens wird.

Bei gegebener Auflösung der Geometrie und des Renderings treten Laufzeitschwankungen nur in der Analyse-Phase auf. Wegen des geringen Anteils dieser Phase an dem gesamten Zeitverhalten können sie vernachlässigt werden.

Vergleich mit geometriebasierten Verfahren

In diesem Abschnitt wird das Zeitverhalten von ZCOLL/F mit den beiden geometriebasierten Bibliotheken RAPID und SOLID verglichen. RAPID[18]

[18] www.cs.unc.edu/ geom/OBB/OBBT.html

ist eine Implementierung der OBB-Kollisionserkennung von Gottschalk et al. [1996], während SOLID[19] den auf einer AABB-Hierarchie basierenden Vorschlag von van den Bergen [1997] implementiert. ZCOLL/F sowie Fassaden für RAPID und SOLID sind in der VRM-Bibliothek implementiert. Die Schnittstellen der drei Verfahren sind von einer gemeinsamen Basisklasse abgeleitet, so dass sie leicht gegeneinander ausgetauscht werden können.

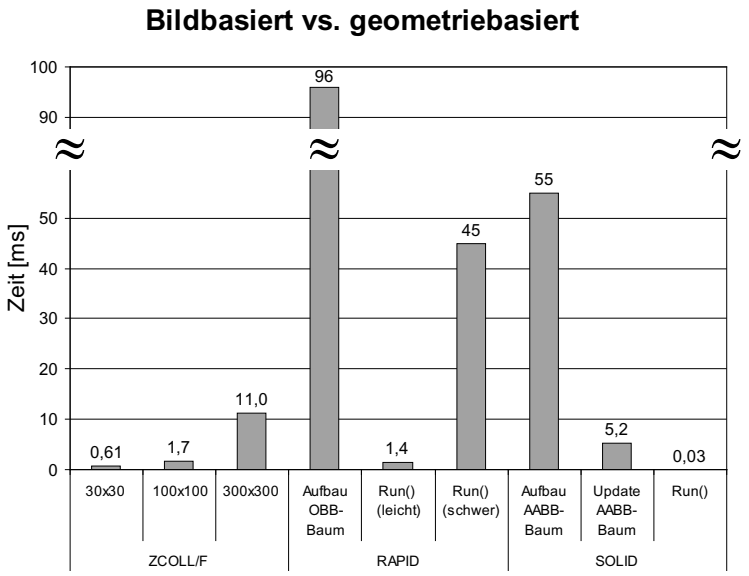


Abbildung 4.21: Vergleich des Zeitverhaltens geometrischer und bildbasierter Verfahren am Beispiel von ZCOLL/F, RAPID (OBB-Hüllkörper-Hierarchie) und SOLID (AABB-Hüllkörper-Hierarchie).

Zum Vergleich wurde wieder die Testszene des vorhergehenden Abschnitts (Abb. 4.19) verwendet. Die drei linken Balken von Abb. 4.21 entsprechen den „schweren“ Kollisionen aus Abb. 4.20. Da die Stärke der Kollision nur vernachlässigbare Auswirkungen auf das Zeitverhalten von ZCOLL/F hat, werden die „leichten“ Kollisionen nicht aufgeführt.

RAPID besteht aus einer Vorverarbeitungsphase („Aufbau OBB-Baum“ in Abb. 4.21), in der ein statischer OBB-Baum aufgebaut wird und einer Run()-Methode, bei der mit aktualisierten Transformationsmatrizen der Objekte eine neue Kollisionserkennung ausgelöst wird. Von der RAPID-Run()-Methode wird eine Menge mit allen kollidierenden Polygonpaaren zurückgegeben.

Da SOLID statt OBBs leichter anpassbare AABBs verwendet, kann die Bibliothek eine bestehende Hüllkörper-Hierarchie an ein deformiertes Objekt

[19] www.dtecta.com

anpassen („Update AABB-Baum“ in Abb. 4.21). Leider kann die SOLID-Run()-Methode keine vollständige Menge aller kollidierenden Polygonpaare aufbauen. Stattdessen gibt sie nächste Punkte, Berührpunkte und eine Kollisionsebene zurück – Eigenschaften der Kollision, die gut für eine Starre-Körper-Simulation, aber kaum für Deformationen verwendet werden können.

Wenn man von einem harmlosen Fall ausgeht – statische Objekte (Vorverarbeitung muss nur einmal ausgeführt werden), geringe erforderliche Genauigkeit (ZCOLL/F mit 30×30 Pixeln Auflösung), leichte Kollisionen –, dann benötigen alle drei Ansätze weniger als 1,5ms für die Kollisionserkennung und sind damit gut für den Einsatz in einer virtuellen Realität geeignet. Da SOLID die Suche abbricht, nachdem ein Zeuge für die Kollision gefunden wurde, ist die Ausführung der Run()-Methode deutlich schneller als bei den anderen beiden Verfahren.

Der Vergleich geht anders aus, wenn deformierbare Objekte berechnet werden müssen: RAPID müsste dann jedesmal eine neue OBB-Hierarchie aufbauen, wozu bereits 48ms benötigt werden (wenn nur eines der Objekte deformiert wurde) – zuviel für eine interaktive Simulation. Die Anpassung des AABB-Baums kann bei SOLID in 5,2ms durchgeführt werden, allerdings reicht die zurückgegebene Information nicht dazu aus, um eine Deformation durchzuführen. Das Zeitverhalten von ZCOLL/F ist dagegen unabhängig von dem Grad der Deformation und liegt deutlich unter dem der geometriebasierten Ansätze.

Bei RAPID fällt der große Unterschied im Zeitbedarf auf, der durch den Grad der Durchdringung der kollidierenden OBB-Bäume entsteht (Run() leicht vs. Run() schwer in Abb. 4.21). Ein solches Verhalten führt zu großen Schwankungen der Wiederholrate und ist daher für eine virtuelle Realität ungeeignet.

4.3 Zusammenfassung und Diskussion

Es existiert eine große Vielfalt von unterschiedlichen Ansätzen zur Kollisionserkennung. Viele Verfahren lösen das All-Pairs-Problem, indem sie bestimmte Zusatzannahmen über die Natur der betrachteten Szene treffen. Dazu gehört die zeitliche Kohärenz aufeinanderfolgender Iterationen, die Konvexität der betrachteten Objekte und die Konstanz ihrer Gestalt, die etwa zur Vorberechnung von Hüllkörper-Hierarchien benötigt wird.

Diese Voraussetzungen sind stark genug, um statt der $O(n^2)$ Schritte für einen paarweisen Vergleich einzelner Objektmerkmale nur noch $O(1 + \varepsilon)$ Schritte für eine lokale Suche zu benötigen.

Bei chirurgischen Simulationen mit ausgeprägten Objektdeformationen und topologischen Änderungen kann weder die Annahme der Konvexität noch der Konstanz der Gestalt aufrechterhalten werden. Die Standardverfahren, die auf Polytopen oder vorberechneten, unveränderbaren Hüllkörper-Hierarchien beruhen, sind auf diesen Fall nicht anwendbar.

In dieser Arbeit wurden daher die bildbasierten Algorithmen ZCOLL und ZCOLL/F als Anpassung der Verfahren von Shinya und Forgue [1991] und Myszkowski et al. [1995] an Instrument-Gewebe-Interaktionen entwickelt. Unter der Annahme langsamer Instrumentbewegungen und der lokalen Interaktion mit Oberflächen anstelle von Volumina benötigt ZCOLL im Gegensatz zum Vorschlag von Myszkowski et al. nur einen Rendering-Schritt. ZCOLL/F spart einen weiteren Rendering-Aufruf und verringert die Datenmenge beim Buffer-Readback, indem eine definierte Interaktions- und Deformationsrichtung zwischen Instrument und Gewebe angenommen wird. Für eine Kollisionsantwort berechnen ZCOLL und ZCOLL/F kollidierende Dreiecke und geben einen Vorschlag für Deformationsvektoren.

Im Vergleich zu geometriebasierten Standardverfahren besitzt ZCOLL(/F) die folgenden Vor- und Nachteile:

+ Es wird keine Vorverarbeitungsphase benötigt, so dass ZCOLL uneingeschränkt für deformierbare Objekte eingesetzt werden kann.

+ Das Laufzeitverhalten ist weitgehend unabhängig von der Schwere der Kollision. Es kann durch die Wahl der Auflösung an die gerade verfügbare Zeit angepasst werden. ZCOLL ist daher gut für den Einsatz in Echtzeitsystemen im Sinne von Def. 1.4 geeignet und kann bei Anwendungen verwendet werden, die Laufzeit und Genauigkeit ihrer Module dynamisch anpassen (*time-critical computing*).

+ Die von ZCOLL ermittelten Informationen (Deformationsvektoren und kollidierende Polygone) eignen sich gut als Ausgangsdaten für die Berechnung der Kollisionsantwort.

+ Das zentrale Charakteristikum der bildbasierten Verfahren ist die Auslagerung der wichtigsten Operationen (Geometrieoperationen, Rasterisierung, ein Teil der Buffer-Vergleiche) auf die Grafikkarte. Dies bietet einige Vorteile: Die Geschwindigkeit der Grafikhardware kann ausgenutzt werden; die Implementierung eines bildbasierten Verfahrens ist vergleichsweise einfach, da viele Operationen von der Grafikbibliothek zur Verfügung gestellt werden; es muss keine neue Datenstruktur aufgebaut werden, da die zur Visualisierung benötigte Struktur verwendet werden kann.[20].

[20] Dadurch kann die Eigenschaft einiger Grafikkarten ausgenutzt werden, die eine grobe geometrische Beschreibung auf der Karte verfeinern können. Zurzeit ist dies z.B. bei einigen ATI-Grafikkarten möglich; ATI nennt diese Technik TRUFORM™.

- ZCOLL ist im Gegensatz zu Hüllkörper-Hierarchien nicht für beliebige Objektformen geeignet. Stattdessen wird für eines der Objekte z-Konvexität gefordert. Allerdings ist das Verhalten des Verfahrens unkritisch, wenn diese Bedingung nicht erfüllt wird: Im Beispiel des Vitrektoms in den Abb. 4.8 und 4.9 kann sich ein Gewebestück in der Einsaugöffnung befinden, so dass es bei einer Projektion in z-Richtung verdeckt wird. ZCOLL erkennt dann fälschlicherweise eine Kollision und erzeugt Deformationsvektoren, die das Gewebe zur Spitze des Vitrektoms schieben.
- Die Genauigkeit ist wesentlich geringer als bei geometriebasierten Verfahren. Kollidierende Flächen werden besonders dann schlecht erkannt, wenn sie einen kleinen Winkel zur z-Richtung bilden.
- Selbstkollisionen können nicht erkannt werden.

Bildbasierte Verfahren sind für deformierbare Materialen eine gute Alternative. Ihre beschränkte Genauigkeit hat nach unserer Erfahrung weniger Einfluss auf die Realitätsnähe eines VR-Systems als der große Zeitbedarf geometriebasierter Ansätze.

5

Spezielle Aspekte der Gewebesimulation

Interaktivität ist von großer Bedeutung für die Realitätsnähe einer virtuellen Umgebung. Im Kontext chirurgischer Operationen wird sie vor allem durch den Umgang mit deformierbarem Gewebe charakterisiert. Dieses Kapitel geht auf diejenigen Aspekte ein, die während der Entwicklung der EyeSi-Gewebesimulationen von Bedeutung waren. Für eine allgemeinere Darstellung wird auf Schill [2001] verwiesen.

Abschnitt 5.1 diskutiert einige grundsätzliche Fragen der Modellierung. In Abschnitt 5.2 werden anschließend Probleme, Anpassungen und Erweiterungen von Feder-Masse-Modellen beschrieben, wie sie in der VRM-Bibliothek implementiert wurden.

5.1 Physikalische und deskriptive Modellierung

Das Ziel der *physikalischen Modellierung* ist es, für eine Menge von Beobachtungen eine physikalische Beschreibung zu finden. Diese Beschreibung hat meist die Form von Differentialgleichungen, deren Randbedingungen, Parameter und Variablen auf einfache Weise mit messbaren physikalischen Größen verknüpft sind.

Um einen bestimmten Prozess zu simulieren, werden die folgenden Schritte durchgeführt:

1. Zunächst wird ein physikalisches Modell gesucht, das den Prozess auf einer passenden Abstraktionsebene beschreibt. „Passend“ bedeutet

dabei, dass die Beschreibung in ihrer Komplexität handhabbar ist – es führt nicht weiter, zur Beschreibung eines Gewebestücks mit dem quantenmechanischen Verhalten von Eiweiß-Molekülen zu beginnen.

2. Die Parameter des Modells werden durch Messung oder theoretische Herleitung bestimmt.

3. Mit Hilfe eines geeigneten numerischen Verfahrens wird für die in (1.) aufgestellten und in (2.) parametrisierten Gleichungen eine Lösung gesucht.

Für die Anwendung bei der Echtzeit-Simulation einer chirurgischen Operation birgt jeder dieser Schritte ein Problem:

1. Häufig ist das physikalische Gesetz nicht bekannt, welches das Verhalten einer bestimmten Gewebeform bestimmt.

2. Die Messung der Parameter ist schwierig, wenn das Gewebe *in vivo* unzugänglich ist und *in vitro* ein anderes biomechanisches Verhalten aufweist.

3. Eine hinreichend exakte Simulation benötigt meist mehr Ressourcen, als zur Erfüllung der Echtzeitbedingung zur Verfügung stehen. „Hinreichend exakt“ im Sinne der physikalischen Modellierung meint, dass ein definierter Fehler nicht überschritten wird.

Physikalische Modelle werden häufig mit der Methode der finiten Elemente (FEM) berechnet [Bathe 1990], die ein kontinuierliches Problem in geometrisch einfache, diskrete Bereiche (*finite Elemente*) zerlegt. Innerhalb jedes finiten Elements wird eine einfache (z.B. lineare) Lösungsfunktion angenommen. Für aneinandergrenzende Elemente gelten Stetigkeitsbedingungen. Die Gesamtlösung wird aus den Teillösungen in den einzelnen Elementen zusammengesetzt. FEM-Simulationen sind unter bestimmten Bedingungen sogar für Echtzeit-Simulationen geeignet – etwa, wenn die Topologie des simulierten Objekts nicht verändert wird, so dass einzelne Schritte des Lösungsverfahrens vorberechnet werden können [Cotin et al. 1999]. In der VRM-Bibliothek wurden von Schill und Grimm einige FEM-Verfahren zur Gewebesimulation implementiert.

Das entscheidende Kriterium für die Qualität eines physikalischen Modells ist sein quantifizierbarer Zusammenhang mit dem realen Experiment. Bei bestimmten Anwendungen tritt dieses Kriterium in den Hintergrund: für eine Computeranimation ist die visuelle Glaubwürdigkeit entscheidend, für eine virtuelle Realität der Erhalt der Präsenz, für eine chirurgische Trainingssimulation ihr Trainingseffekt.

Wenn ein Modell aufgrund solcher nicht-physikalischer Kriterien entwickelt wurde, nennt man es *deskriptiv* [Schill et al. 1999b]. Ein deskriptives Modell besteht häufig aus einer Analogie zu einem realen Vorgang, die einfacher zu berechnen ist (um die Echtzeit-Bedingung zu erfüllen) und ein qualitativ ähnliches Verhalten aufweist – wobei „qualitativ ähnlich“ lediglich bedeutet: ähnlich genug, um Glaubwürdigkeit, Präsenz oder einen bestimmten Lerneffekt zu erzielen.

In der VRM-Bibliothek sind die folgenden deskriptiven Modelle implementiert:

- Feder-Masse-Simulationen, bei denen ein Objekt aus einzelnen Punktmassen zusammengesetzt ist, die mit Federn verbunden sind (Abschnitt 5.2).
- ChainMail-Objekte [Gibson 1997, Schill et al. 1998], die aus einzelnen starren Kettengliedern zusammengesetzt sind. Die Kettenglieder sind wie bei einem mittelalterlichen Kettenhemd miteinander verbunden. Jedes Kettenglied ist in einem bestimmten Bereich frei beweglich; anschließend stößt es an einen seiner Nachbarn, auf den die Bewegung übertragen wird. Im Gegensatz zu Feder-Masse-Modellen werden keine Kräfte übertragen, sondern Deformationen. Bei einem ChainMail-Gewebe ist garantiert, dass in $O(n)$ Schritten ein Gleichgewichtszustand erreicht wird. Es ist besonders zur Simulation von inelastischen Strukturen geeignet.

Bei diesen Modellen fällt auf, dass sie eine physikalische Terminologie verwenden. Bei der Unterscheidung zwischen deskriptiven und physikalischen Modellen wird keine Aussage über den physikalischen Gehalt des Modells gemacht – deskriptive Modelle können durchaus reale physikalische Systeme beschreiben, werden aber in einem anderen Kontext eingesetzt, der nicht durch physikalische Überlegungen motiviert ist. Umgekehrt kann ein Modell auf deskriptive Weise entstanden sein, aber zur physikalischen Modellierung verwendet werden. Bekanntestes Beispiel für einen deskriptiven Analogieschluss sind die Maxwell-Gleichungen, die aus Beobachtungen in der Strömungslehre entstanden sind.

Deskriptive Modellierung führt häufig zu Modellen mit den folgenden Eigenschaften:

- Bei dem Modell besteht kein direkter Zusammenhang zwischen den Modellparametern und den Eigenschaften des realen Vorbilds.
- Die Parameter des Modells werden nicht durch Messung, sondern durch einen iterativen Prozess bestimmt, bei dem der „Modellierer“

solange Veränderungen am Modell vornimmt, bis die „Reale-Welt-Expertin“ eine ausreichende Ähnlichkeit feststellt.

- Häufig werden an den Modellen Korrekturen vorgenommen, die keine physikalische Grundlage besitzen, sondern der numerischen Stabilität dienen oder seine qualitativen Eigenschaften verbessern. Einige Beispiele dieser Vorgehensweise findet man im folgenden Abschnitt 5.2.

- Der Gültigkeitsbereich eines deskriptiven Modells liegt meist innerhalb desjenigen Parameterraums, der von den bei der Modellierung verwendeten realen Situationen aufgespannt wird. Extrapolationen sind nur sehr begrenzt möglich; für eine neue Situation muss mindestens eine Anpassung der Parameter, üblicherweise sogar eine Erweiterung des Verfahrens selbst vorgenommen werden.

Bei der deskriptiven Modellierung können daher viele Entwicklungszyklen notwendig sein, bis ein überzeugendes Verhalten des Modells erreicht wird.

5.2 Feder-Masse-Modelle

Feder-Masse-Modelle sind ein Spezialfall von *Partikelsystemen*. Ein Partikelsystem beschreibt die Bewegungen von n punktförmigen, massebehafteten Teilchen. Ein bestimmtes Teilchen i hat einen Positionsvektor $\vec{x}_i$ und einen Geschwindigkeitsvektor $\vec{v}_i$. Die Vektoren aller Teilchen werden der Bequemlichkeit halber zu den $3n$-dimensionalen Vektoren $\underline{x} = (\vec{x}_0, \ldots, \vec{x}_{n-1})^T$ und $\underline{v} = (\vec{v}_0, \ldots, \vec{v}_{n-1})^T$ zusammengefasst. Auf jedes i wirkt eine Kraft $\vec{F}_i$ der in Gleichung 5.1 angegebenen Form.

$$\vec{F}_i(\underline{x}, \underline{v}) = \sum_{j \neq i} \vec{H}_{ij}(\vec{x}_i, \vec{x}_j) + \vec{R}_i(\vec{v}_i) + \vec{E}_i \tag{5.1}$$

Im Folgenden werden auch die Kräfte $\vec{F}_i$ als $3n$-dimensionalen Vektor $\underline{F} = (\vec{F}_0, \ldots, \vec{F}_{n-1})^T$ geschrieben.

Neben den *inneren Kräften* $\vec{H}_{ij}$ können auf jedes Teilchen i eine geschwindigkeitsabhängige Reibung (im einfachsten Fall wählt man $\vec{R}_i \sim \dot{\vec{x}}_i$) sowie externe Kräfte $\vec{E}_i$ wirken, die z.B. den Einfluss der Gravitation, einer laminaren Strömung oder geometrischer Zwangsbedingungen ausdrücken.

Die inneren Kräfte $\vec{H}_{ij}$ beschreiben den physikalischen Charakter des Partikelsystems, das durch die Kräfte der Gleichung 5.1 beschrieben wird. Setzt man beispielsweise $\vec{H}_{ij} \sim 1/||\vec{x}_i - \vec{x}_j||^2$, erhält man eine klassische (physikalische, nicht deskriptive!) Näherung für das Verhalten eines Planetensystems oder einer Ansammlung elektrischer Ladungen. Ein einfaches

Modell für Flüssigkeiten erhält man, wenn die $\vec{H}_{ij}$ keine feste Topologie beschreiben, sondern in jedem Iterationsschritt für die geometrisch nächsten Punktmassen neu bestimmt werden.

Ein Feder-Masse-Modell ist ein Partikelsystem, bei dem die einzelnen Punktmassen in einem Gitter angeordnet sind. Die Funktion $\vec{H}_{ij}$ beschreibt dann die entlang der Gitterkanten auftretenden Federkräfte, die vom Abstandsvektor $\vec{d}_{ij} = \vec{x}_i - \vec{x}_j$ abhängen und parallel zu ihm verlaufen. Häufig sind die Federkräfte proportional zur Abweichung der Länge dieses Vektors von einer gegebenen Nulllänge l_{ij} (Hooke'sches Gesetz). Mit der Federkonstanten k_{ij} wird dies durch den folgenden Zusammenhang für $\vec{H}_{ij}$ beschrieben:

$$\vec{H}_{ij}(\vec{d}_{ij}) = -\frac{\vec{d}_{ij}}{||\vec{d}_{ij}||} k_{ij}(||\vec{d}_{ij}|| - l_{ij}) \tag{5.2}$$

Bemerkung: Plastische Verformungen können modelliert werden, indem man die Nulllängen l_{ij} an die bisher erreichten Maximallängen der d_{ij} anpasst: $l_{ij}(t) = f(\max_{t'<t} d_{ij}(t'))$.

5.2.1 Integrationsmethoden

Es ist im Allgemeinen nicht möglich, für ein Feder-Masse-System eine geschlossene Lösung anzugeben. Daher müssen die durch die Kräfte aus Gleichung 5.1 gegebenen Bewegungsgleichungen numerisch gelöst werden. Für die dafür verwendeten Integrationsmethoden existieren die folgenden Kriterien [Volino und Thalmann 2001]:

- Größe des Zeitschritts
- Genauigkeit
- Stabilität
- Berechnungsaufwand pro Iterationsschritt

Häufig wird ein bestimmtes Integrationsverfahren nur solange ausgeführt, bis die für die Simulation vorgesehene Rechenzeit aufgebraucht ist. Wenn nur wenig Zeit zur Verfügung steht und die Genauigkeitsanforderungen gering sind, kann daher ein ungenaues Verfahren mit einem geringen Rechenaufwand pro Iterationsschritt besser geeignet sein als ein aufwändigeres Verfahren, obwohl es mehr Iterationen, kleinere Zeitschritte und insgesamt mehr Rechenzeit benötigt, um eine bestimmte Genauigkeit zu erreichen.

Bei der Berechnung von Gewebeinteraktionen muss zusätzlich berücksichtigt werden, ob ein bestimmtes Verfahren eine Vorverarbeitungsphase benötigt, die bei topologischen Änderungen wiederholt werden muss.

Für die folgende Darstellung werden die Gleichungen 5.1 zunächst in die kanonische Form von $6n$ Differentialgleichungen erster Ordnung gebracht. Mit der Massenmatrix $M \in \mathbb{R}^{3n \times 3n}$ und $\text{diag}(M) = (m_0, m_0, m_0, m_1, m_1, m_1, \ldots, m_{n-1}, m_{n-1}, m_{n-1})$ erhält man:

$$\begin{aligned} \underline{\dot{v}} &= M^{-1}\underline{F} \\ \underline{\dot{x}} &= \underline{v} \end{aligned} \tag{5.3}$$

Explizite Integration

Die einfachste Integrationsmethode ist das *explizite Euler-Verfahren*. Es geht davon aus, dass während eines Iterationsschritts s der zeitlichen Länge $\triangle t$ die Kräfte auf jedes Teilchen konstant sind. Wegen der Gleichungen 5.3 und mit $\underline{F}^s = \underline{F}(\underline{x}^s, \underline{v}^s)$ erhält man die Geschwindigkeits- und Positionsänderungen $\triangle \underline{v} = \underline{v}^{s+1} - \underline{v}^s$ und $\triangle \underline{x} = \underline{x}^{s+1} - \underline{x}^s$ vom s-ten zum $(s+1)$-ten Iterationsschritt wie folgt:

$$\begin{aligned} \triangle \underline{v} &= \triangle t \cdot M^{-1}\underline{F}^s \\ \triangle \underline{x} &= \triangle t \cdot \underline{v}^s \end{aligned} \tag{5.4}$$

Das günstige Zeitverhalten eines expliziten Euler-Schritts entsteht dadurch, dass jede Komponente der Funktion $\underline{F}$ nur einmal ausgewertet werden muss. Durch die Annahme einer konstanten Steigung werden aber im Allgemeinen große Integrationsfehler verursacht. An Umkehrpunkten tendiert das explizite Euler-Verfahren zum Überschwingen und dadurch zur Akkumulation von Energie, so dass das simulierte System schnell instabil wird.

Man kann Energie aus einem Feder-Masse-System abführen und dadurch seine Stabilität erhöhen, indem man die Reibungskraft $\vec{R}_i$ in Gleichung 5.1 erhöht. Gleichzeitig wird dadurch aber die Geschwindigkeit verringert, mit der das Gewebe auf äußere Einwirkungen reagiert. Eine Alternative ist das *quasi-statische Euler-Verfahren*. Es beruht auf der Annahme, dass das System nach jeder Iteration einen Gleichgewichtszustand mit $\vec{v}_i = 0$ erreicht, so dass die Teilchengeschwindigkeiten nicht aufsummiert werden müssen:

$$\triangle \underline{x} = (\triangle t)^2 \cdot M^{-1}\underline{F}^s \tag{5.5}$$

Durch die Annahme eines konstanten Gradientens innerhalb einer Iteration findet das Euler-Verfahren nur solche korrekten Lösungen, deren Taylor-Entwicklung nach dem linearen (ersten) Glied abbricht. Man nennt das Euler-Verfahren daher ein *Verfahren erster Ordnung*. Verfahren höherer Ordnung erhält man, indem man $\underline{F}$ an mehreren Stellen auswertet. Beispielsweise wird bei einem *Runge-Kutta-Verfahren zweiter Ordnung* zunächst ein Euler-Schritt mit der halben Schrittweite durchgeführt. An dieser Stelle wird erneut die Richtung des Gradienten bestimmt, die dann für den eigentlichen Iterationsschritt verwendet wird. Einen Beweis dafür, dass dieses Verfahren zweiter Ordnung ist, findet man z.B. in dem einführenden Artikel von Witkin und Baraff [1997].

Der Fehler eines Verfahrens n-ter Ordnung mit der Schrittweite $\triangle t$ ist von der Ordnung $O((\triangle t)^{n+1})$. Das bedeutet nicht notwendigerweise, dass ein Verfahren höherer Ordnung bei einem gegebenen $\triangle t$ genauer ist – der Fehler nimmt bei Verringerung der Schrittweite aber schneller ab.

Bei expliziten Verfahren wird der aktuelle Zustand des Systems als Grundlage der Berechnung für den nächsten Iterationsschritt verwendet. Dadurch entsteht ein Problem, das im Folgenden am Beispiel des expliziten Euler-Verfahrens und einer linearen Kette von $n+1$ Teilchen der Masse m und n Federn der Federkonstante k erläutert wird (eine ähnliche Darstellung findet man z.B. bei Kass [1995] und Desbrun et al. [2000]): Für große n ist dieses System eine Näherung für eine einzelne Feder mit der kontinuierlich verteilten Masse $m_0 = (n+1)m$ und der Federkonstanten $k_0 = k/n$. Die Ausbreitung einer Störung wird durch die Wellengleichung beschrieben [Gerthsen et al. 1989, Abschnitt 4.2.2]; ihre Ausbreitungsgeschwindigkeit ist $\sqrt{k_0/m_0}$.

Die Kette sei im nullten Iterationsschritt in einem Gleichgewichtszustand, so dass gilt: $\underline{v}^0 = 0$ und $\underline{F}^0 = 0$. Im nächsten Iterationsschritt wirke auf die nullte Punktmasse eine Störung $\vec{D}$ des Gleichgewichtszustands: $\vec{F}_0^1 = \vec{D}$. Gemäß Gleichung 5.4 führt diese Störung zunächst nur dazu, dass die Geschwindigkeit $\vec{v}_0^2$ einen endlichen Wert hat. Alle anderen Positionen und Geschwindigkeiten werden in dieser Iteration nicht verändert. Erst in der zweiten Iteration ändert sich die Position des nullten Teilchens, so dass die Störung erst in der dritten Iteration beim ersten Teilchen angekommen ist.

Insgesamt benötigt die Störung $2S$ Euler-Schritte, um durch das System zu propagieren (wenn ein Verfahren mehrere Funktionsauswertungen pro Iterationsschritt durchführt, verringert sich der Proportionalitätsfaktor entsprechend). Mit der oben erwähnten Ausbreitungsgeschwindigkeit und der Länge l der Kette ergibt sich für die Größe des Zeitschritts die folgende Abschätzung:

$$\triangle t \leq \frac{l}{2S}\sqrt{\frac{m_0}{k_0}} \tag{5.6}$$

Wegen Ungleichung 5.6 muss ein Euler-Zeitschritt umgekehrt proportional zur Feinheit der Diskretisierung sein. Da der Aufwand eines expliziten Euler-Schritts linear mit der Zahl der Elemente steigt, führt das zu einem quadratischen Zeitverhalten.

Eine ähnliche Abschätzung wird häufig auch für komplexere Feder-Masse-Strukturen mit variierenden Federkonstanten k_{ij} verwendet. So geben Vassilev et al. [2001] als empirisch ermittelten Zusammenhang für ihre Feder-Masse-Modelle $\triangle t \leq 0{,}4\pi\sqrt{m_0/k_0}$ mit $k_0 = \max k_{ij}$ an. Bei inhomogenem Gewebe muss sich der Zeitschritt daher nach der Stelle der größten Federkonstante richten.

Implizite Integration

Die im letzten Abschnitt geschilderten Probleme – Energieakkumulation durch Überschwingen, Abhängigkeit der Schrittweite von dem Grad der Diskretisierung – lassen sich vermeiden, wenn die Änderung der $\underline{v}^s$ und $\underline{x}^s$ zu Beginn einer Iteration s so gewählt wird, dass die am Ende erhaltenen Werte $\underline{v}^{s+1}$ und $\underline{x}^{s+1}$ konsistent zu dieser Änderung sind. Die Werte für $\underline{x}$ und $\underline{v}$ ergeben sich dann nicht mehr direkt aus den Gleichungen, die einen Iterationsschritt bestimmen. Daher nennt man solche Verfahren *implizit*.

Das *implizite Euler-Verfahren* ist durch die folgenden Iterationsvorschriften gegeben (vgl. das explizite Eulerverfahren in Gleichung 5.4[1]):

$$\triangle \underline{v} = \triangle t \cdot M^{-1} \underline{F}^{s+1} \tag{5.7}$$

$$\triangle \underline{x} = \triangle t \cdot \underbrace{(\underline{v}^s + \triangle \underline{v})}_{\underline{v}^{s+1}} \tag{5.8}$$

$\underline{F}$ kann von den Positionen und Geschwindigkeiten aller n Teilchen abhängen. Zum Zeitpunkt der Iteration s ist noch nicht bekannt, an welcher Stelle $\underline{F}^{s+1} = \underline{F}(\underline{x}^{s+1}, \underline{v}^{s+1})$ ausgewertet werden muss. Man nähert daher $\underline{F}$ mit einer Taylor-Entwicklung erster Ordnung an:

$$\underline{F}^{s+1} \approx \underline{F}^s + \nabla \underline{F}|_{\underline{x}^s, \underline{v}^s} \begin{pmatrix} \triangle \underline{x} \\ \triangle \underline{v} \end{pmatrix} \tag{5.9}$$

5.9 und 5.8 eingesetzt in 5.7 ergibt ein lineares Gleichungssystem für $\triangle v$:

[1] Baraff und Witkin [1998] vergleichen explizites (*forward*) und implizites (*backward*) Euler-Verfahren so: „Forward Euler takes no notice of wildly changing derivatives and proceeds forward quite blindly. Backward Euler, however, forces one to find an output state whose derivative at least points back to where you came from, imparting, essentially an additional layer of consistency (or sanity-checking, if you will).“

$$\triangle \underline{v} = \triangle t \cdot M^{-1} \left[\underline{F}^s + \nabla \underline{F}|_{\underline{x}^s, \underline{v}^s} \begin{pmatrix} \triangle t \cdot (\underline{v}^s + \triangle \underline{v}) \\ \triangle \underline{v} \end{pmatrix} \right] \tag{5.10}$$

Ein impliziter Euler-Schritt berechnet $\underline{F}^s$ und $\nabla \underline{F}$, löst das lineare Gleichungssystem 5.10 mit den Komponenten von $\triangle \underline{v}$ als Unbekannten und berechnet mit Gleichung 5.8 die Positionsänderung $\triangle \underline{x}$.

Mit einem direkten Verfahren (z.B. dem Gauss'schen Eliminationsverfahren) benötigt man für die Lösung eines linearen Gleichungssystems $O(n^3)$ Schritte. Bei einem Feder-Masse-System, bei dem jede Punktmasse nur mit einer kleinen Anzahl von Nachbarn verbunden ist, ist $\triangle \underline{F}$ aber nur dünn besetzt, so dass z.B. die Methode der konjugierten Gradienten[2] eingesetzt werden kann, die in $O(n^{3/2})$ Schritten eine Näherungslösung mit definierbarer Genauigkeit findet.

Baraff und Witkin [1998] beschreiben eine Implementierung dieses Verfahrens für die Simulation von Textilien. Sie gehen dabei auch auf die Realisierung von Zwangsbedingungen und Kollisionen ein. Für ein System mit 2.600 Teilchen werden 2,23 Sekunden/Iteration benötigt. Leider ist nicht angegeben, auf welchem System diese Zeit gemessen wurde.

Implizite Verfahren sind auch bei großen Zeitschritten stabil, verlieren aber an Genauigkeit, da hochfrequente Deformationen nicht mehr aufgelöst werden können.

5.2.2 Dehnungskorrektur von Provot

Bei den meisten realen Materialen gilt das Hooke'sche Kraftgesetz von Gleichung 5.2 nur in sehr engen Grenzen. Wenn sie überschritten werden, steigt die für eine bestimmte Deformation aufzuwendende Kraft stark an (Beispiele: Textilien, Haut). Ein entsprechendes Kraftgesetz in Gleichung 5.1 kann bei einer expliziten Integration schnell zu numerischer Instabilität führen.

Provot [1995] schlägt daher eine geometrische Korrektur vor, die diejenigen Federn kürzt, deren Länge d_{ij} einen kritischen Wert $\tau_c \cdot l_{ij}$ oberhalb der Nulllänge l_{ij} übersteigt (Algorithmus 5.1).

Das Verfahren wird nach jedem Integrationsschritt ausgeführt. Wenn eine Feder gefunden wurde, die gekürzt werden muss, wird zunächst der Vektor $\vec{d_c}$ bestimmt, der in Richtung der Feder weist und die kritische Länge $\tau_c \cdot N_{ij}$ hat. Falls möglich, werden die Endpunkte i und j anschliessend so bewegt, dass die Feder Länge und Orientierung von $\vec{d_c}$ besitzt. Dabei müssen auf die Knoten wirkende Zwangsbedingungen berücksichtigt werden. Wenn keiner der Knoten frei ist, bleiben die Positionen unverändert.

[2] Eine gute Einführung gibt Shewchuk [1994].

Algorithmus 5.1 Korrektur überdehnter Federn [Provot 1995]

```
1: for all springs s_ij do
2:   if ||d_ij|| > τ_c l_ij then
3:     d_c ← d_ij/||d_ij|| · τ_c l_ij // d_c: distance vector with critical length
4:     if ¬(node i fixed) ∧ ¬(node j fixed) then
5:       x_i ← (x_i + x_j)/2 + d_c/2
6:       x_j ← (x_i + x_j)/2 − d_c/2
7:     if (node i fixed) ∧ ¬(node j fixed) then
8:       x_j ← x_i − d_c
9:     if ¬(node i fixed) ∧ (node j fixed) then
10:      x_i ← x_j + d_c
```

Durch die Ausführung dieses Korrekturschritts propagieren Deformationen schneller durch die Feder-Masse-Struktur; die numerische Stabilität expliziter Verfahren erhöht sich, da die durchschnittliche Elongation der Federn und die dadurch entstehenden Kräfte verringert werden.

Ein Nachteil des Verfahrens ist, dass die Knotenpositionen in einem Nachverarbeitungsschritt und nicht innerhalb des numerischen Modells der Simulation verändert werden. Die Stabilität von impliziten Integrationsverfahren wird dadurch negativ beeinflusst.[3]

5.2.3 Berechnung von Kollisionsantworten

Die Aufgabe der *collision response* ist es, die Positionen, Geschwindigkeiten oder Beschleunigungen von kollidierenden Objekte so anzupassen, dass sie sich nicht mehr gegenseitig durchdringen. Dabei müssen Reibung und neu entstehende Zwangsbedingungen (z.B. beim Greifen eines Gewebestücks mit einer Pinzette) berücksichtigt werden.

Es wird im Folgenden davon ausgegangen, dass die Kollisionserkennung Distanzinformationen über die beteiligten Objekte berechnen kann – zum Beispiel Deformationsvektoren, die bei einer Kollision mit einem Feder-Masse-Objekt den einzelnen Punktmassen zugeordnet werden.

Eine Kollisionsantwort besteht im einfachsten Fall darin, die Positionen der kollidierenden Punktmassen um die Deformationsvektoren zu korrigieren. Ein Problem dieser Vorgehensweise wurde in Abschnitt 5.2.2 bereits erwähnt: die Positionsänderung findet unabhängig vom Integrationsmodell

[3] Baraff und Witkin [1998] schreiben dazu: „Unfortunately, simply changing particle positions produced disastrous results. [...] Altering particle positions arbitrarily introduced excessively large deformation energies in an altered particle's neighborhood."

der Simulation statt und verringert die Stabilität impliziter Integrationsverfahren. Um dieses Problem zu umgehen, ergänzen Baraff und Witkin [1998] die Gleichungen ihrer impliziten Integration um einen Korrekturterm, der schon vor der Durchführung eines impliziten Euler-Schritts die Positionskorrektur enthält. Mehr über die Berechnung von Kollisionsantworten bei impliziter Integration findet man z.B. bei Volino und Thalmann [2000a] und Hauth und Etzmuss [2001].

Sortierung der Federn nach Interaktionspunkten

Stabilitätsprobleme durch das Versetzen von Knoten treten auch bei expliziter Integration auf: die Federn zwischen den versetzten Knoten und ihren Nachbarn werden unter Umständen so stark gespannt, dass die auftretenden Kräfte zu Instabilitäten führen.

Zur Lösung dieses Problems haben wir einen Vorschlag von Brown et al. [2001] mit der in Abschnitt 5.2.2 beschriebenen Dehnungskorrektur von Provot [1995] kombiniert. Brown et al. sortieren die Punktmassen in Abhängigkeit von ihrer (Graph-)Distanz zu einem Interaktionspunkt. Sie integrieren das Modell mit expliziten Euler-Schritten, die für jedes Teilchen in der Reihenfolge dieser Sortierung durchgeführt werden.

Entsprechend wird hier nach der Erkennung von Kollisionen an den Interaktionspunkten I zunächst eine Sortierung der Federn durchgeführt. Dazu baut Algorithmus 5.2 in einer einfachen Breitensuche Mengen S_d auf, die alle Federn mit Abstand d zu einem Interaktionspunkt enthalten.

Algorithmus 5.2 Sortierung der Federn nach Interaktionspunkten

1: I is the set of interaction elements.
2: Initialize S_0 with all springs adjacent to any $i \in I$.
3: Mark all springs $s \in S_0$.
4: $d \leftarrow 0$
5: **repeat**
6: $\quad d \leftarrow d + 1$
7: $\quad S_d \leftarrow \emptyset$ *// Will contain springs with distance d to I*
8: $\quad$ **for all** springs s adjacent to any spring in S_{d-1} **do**
9: $\quad\quad$ **if** s is unmarked **then**
10: $\quad\quad\quad S_d \leftarrow S_d \cup \{s\}$
11: $\quad\quad\quad$ Mark s.
12: **until** $S_d = \emptyset$

Komplexität von Algorithmus 5.2: Sei n die Gesamtzahl der Federn. Unter der Voraussetzung, dass jeder Knoten nur mit einer nach oben durch ein festes N beschränkten Zahl von Federn verbunden ist, wird die innere

Schleife in $O(|S_{d-1}| \cdot N)$ Schritten abgearbeitet; die äußere Schleife läuft von $d = 1$ bis $d = d_{max}$, so dass sich die lineare Gesamtlaufzeit $T_{sort} = O(n)$ aus der in 5.11 angegebenen Umformung ergibt.[4] Der zweite Schritt der Umformung ist möglich, da die Mengen S_d gemäß Konstruktion disjunkt sind und daher insgesamt nicht mehr als n Elemente enthalten können.

$$\begin{aligned} T_{sort} &= \sum_{d=1}^{d_{max}} O(|S_{d-1}| \cdot N) \\ &= N \cdot O\left(\sum_{d=1}^{d_{max}} |S_{d-1}|\right) \\ &= O(n) \end{aligned} \tag{5.11}$$

Wird die Dehnungskorrektur von Algorithmus 5.1 nacheinander für die Federn von $S_0, S_1, \ldots$ ausgeführt, dann propagiert die Deformation ausgehend von den Interaktionspunkten durch das Gewebe. Man erhält auf diese Weise ein ChainMail-ähnliches Verhalten.

Unterstützend kann, wie in Brown et al. [2001] beschrieben, die Integration in der Reihenfolge der Sortierung durchgeführt werden. Brown et al. schlagen zusätzlich vor, die Berechung abzubrechen, sobald die Knotenkräfte einen bestimmten Betrag unterschreiten. Auf diese Weise werden nur diejenigen Teile des Feder-Masse-Gewebes neu berechnet, die gerade stark deformiert werden. Bei vielen Operationen wechselt die Stelle der Interaktion häufig, bevor die Simulation einen Gleichgewichtszustand erreicht hat. Man kann dieses Problem lösen, indem man jeden ehemaligen Interaktionspunkt mit einem Zeitstempel versieht. Erst nach dem Ablauf einer bestimmten Anzahl von Iterationen darf er aus der Menge I der Interaktionspunkte entfernt werden.

Zwangsbedingungen

Wenn ein Instrument ein Gewebestück festhält, werden die kollidierenden Punktmassen an die Oberfläche des Instruments versetzt und dort mit ihm mitbewegt. Von der Positionsänderung durch den Simulationsalgorithmus werden diese Punktmassen dagegen ausgenommen. Bei den expliziten Verfahren in der VRM-Bibliothek geschieht dies mit Hilfe einer boole'schen

[4] Für die Netztopologien, die bei der Gewebesimulation üblich sind, ist die Annahme einer Obergrenze N für die Anzahl der Nachbarn vernünftig. Wenn man auf sie verzichtet, arbeitet Algorithmus 5.2 im Allgemeinen nicht mehr in Linearzeit. Beispiel: gegeben sei ein Knoten i, der über je eine Feder mit n weiteren Knoten verbunden ist. Für $I = \{i\}$ ist $|S_0| = n$, so dass die innere Schleife von Algorithmus 5.2 $O(n^2)$ Schritte benötigt.

Variablen; bei dem impliziten Verfahren von Baraff und Witkin [1998] wird der entsprechende Eintrag in der Massenmatrix auf ∞ gesetzt.

Ein Instrument, das in ein Gewebestück drückt, dehnt das Gewebe, wodurch eine rücktreibende Kraft in Richtung des Instruments erzeugt wird. Wenn dieser Vorgang mit einer Feder-Masse-Struktur simuliert wird, muss darauf geachtet werden, dass einzelne Punktmassen nicht zwischen den beiden möglichen Zuständen „ausserhalb des Instruments, rücktreibende Kraft wirkt“ und „im Instrument, Deformationsvektoren werden angewendet“ oszillieren und dabei Energie akkumulieren.

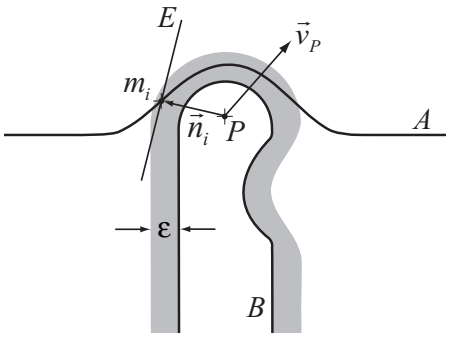


Abbildung 5.1: Interaktion zwischen einer Instrumentspitze und einem Gewebestück

Abb. 5.1 illustriert, wie diese Oszillationen verhindert werden können. Das Instrument (B) wird von einer *ε-Hülle* umgeben, die zur Oberfläche von B einen Abstand ε nach außen einhält. Die ε-Hülle wird für die Kollisionserkennung verwendet, während die Berechnung der Deformationsvektoren und die Visualisierung mit der ursprünglichen Geometrie arbeiten.

Ein deformiertes Gewebestück wird in die in Abb. 5.1 grau markierte Zone zwischen der ε-Hülle und dem Objekt bewegt, wo es nicht weiter deformiert, aber von der Kollisionserkennung trotzdem noch erkannt wird – obwohl es das Instrument *berührt* (sich zwischen dem Objekt und der ε-Hülle befindet), findet keine sichtbare Kollision mehr statt. [5]

Bemerkung: Bei einfachen Geometrien genügt es, die ε-Hülle durch ein skaliertes Instrument (typischer Skalierungsfaktor: 1,05) anzunähern.

Die Geschwindigkeitsvektoren der Punktmassen, die sich zwischen der

[5] Mezger et al. [2002] schlagen (im Zusammenhang mit k-DOP-Hüllkörpern) vor, den Hüllkörper in der Richtung der Instrumentbewegung stärker zu vergrößern, um Kollisionen zuverlässiger erkennen zu können, die zwischen zwei Zeitschritten auftreten. Bei den langsamen Bewegungen, die während einer chirurgischen Operation ausgeführt werden, bringt diese Herangehensweise aber kaum eine Verbesserung.

Oberfläche von B und der ε-Hülle befinden, werden mit Hilfe der folgenden Näherung angepasst:

Sei P ein fester Punkt innerhalb der Instrumentspitze und m_i eine Punktmasse, die B berührt und die Geschwindigkeit $\vec{v}_i$ hat ($\vec{v}_i$ ist nicht in der Abbildung dargestellt). Der Vektor $\vec{n}_i$ weist von P zu m_i. Er ist Normalenvektor der Ebene E, die m_i enthält. Seien $\vec{v}_{i,E}$ und $\vec{v}_{P,E}$ die Projektionen der Geschwindigkeitsvektoren auf E und $f_R \in [0,1]$ ein Reibungskoeffizient. Dann ergibt sich die Geschwindigkeit $\vec{v}_{i,corr}$ von m_i nach der Geschwindigkeitskorrektur durch die Kollision wie folgt:

$$\vec{v}_{i,corr} = \begin{cases} \vec{v}_i + f_R \cdot \vec{v}_{P,E} & : \quad \vec{v}_i \cdot \vec{n}_i > 0 \\ \vec{v}_{i,E} + f_R \cdot \vec{v}_{P,E} & : \quad \vec{v}_i \cdot \vec{n}_i < 0 \end{cases} \tag{5.12}$$

Bei diesem Modell wurden zugunsten einer schnellen Berechnung einige unphysikalische Annahmen gemacht: Zum einen ist die durch die Reibung übertragene Bewegungsenergie in der Realität abhängig von der Kraft, mit der das Gewebe auf das Instrument gedrückt wird, zum anderen ist die oben konstruierte Ebene E nur dann tangential zur Instrumentenspitze, wenn diese kugelförmig ist. Für einfache Geometrien wird trotzdem ein glaubwürdiges Verhalten erzielt; Interaktionen mit komplexen Instrumenten müssen dagegen angepasst werden. So werden z.B. bei der in EyeSi modellierten Pinzette (Kapitel 7) die kollidierenden Gewebestücke auf die Mittelebene der Scheren projiziert und dort mitbewegt, wenn der Öffnungswinkel der Scheren einen bestimmten Wert unterschreitet.

Mehrfachkollisionen

Wenn während eines Iterationsschritts mehrere Kollisionen innerhalb eines kleinen Volumens stattfinden (beispielsweise bei einem deformierbaren, stark gefalteten Objekt), wird häufig durch das Auflösen eines Ereignisses eine neue Kollision erzeugt. Provot [1997] (mit einer Korrektur bei Bridson et al. [2002]) gruppiert mehrfache Kollisionen in *Zones of impact* (IZ), die als zusammenhängendes Objekt betrachtet werden (unter der Voraussetzung eines vollständig inelastischen, schlupffreien Stoßes).

Um zu verhindern, dass das Gewebe in den IZs dauerhaft „zusammenklebt“, erweitern Huh et al. [2001] den Ansatz von Provot, indem sie die zusammenhängenden Gewebestücke innerhalb einer IZ durch Massenpunkte ersetzen, für die Bewegungsgleichungen aufgestellt und gelöst werden.

Volino und Thalmann [2000b] stellen ein lineares Gleichungssystem auf, das die durch die Kollisionen entstandenen Zwangsbedingungen auf alle Teilchen beschreibt. Die spezielle Struktur der Zwangsbedingungen hilft bei

einer effizienten Lösung des Gleichungssystems mit der Methode der konjugierten Gradienten. Man erhält bei diesem Verfahren eine global gültige Auflösung aller Kollisionen.

Man kann die Zahl der Mehrfachkollisionen bei deformierbaren Objekten verringern, indem man eine abstoßende, kurzreichweitige Kraft zwischen nicht benachbarten Bereichen einführt [Bridson et al. 2002].

Wie in Abschnitt 4.1.5 bereits erwähnt, wird die Erkennung von Selbstkollisionen in den VRM-Applikationen bisher nicht benötigt. Der andere Fall, bei dem Mehrfachkollisionen aufgelöst werden müssen – wenn sich mehrere (starre) Objekte auf engem Raum befinden – tritt bei den abstrakten Trainingsaufgaben von EyeSi auf (Kapitel 7). Wegen der geringen Anzahl der Objekte und ihrer einfachen geometrischen Gestalt konnte dieser Fall mit Hilfe eines iterativen Verfahrens bewältigt werden, das die Objekte solange verschiebt, bis alle Kollisionen aufgelöst sind.

5.2.4 Gitterverfeinerung

Das Stabilitätsverhalten expliziter Verfahren und Laufzeitbeschränkungen bei Echtzeitsystemen legen die Verwendung möglichst grob diskretisierter Feder-Masse-Strukturen nahe. Andererseits können mit höher aufgelösten Gittern feinere Strukturen approximiert werden.

Ein weiterer Effekt wurde in dem Beispiel der linearen Kette in Abschnitt 5.2.1 bereits angedeutet: Explizite Löser, die auf hoch aufgelösten Gittern arbeiten, können lokale Störungen schnell berücksichtigen, ihre räumliche Fortpflanzung benötigt dagegen viele Iterationsschritte.

Aufgrund dieser Beobachtung arbeiten *Mehrgitterverfahren* auf Gittern mit mehreren Auflösungen. Zunächst wird ein Berechnungsschritt auf einer hohen Auflösung durchgeführt. Das so geglättete Gitter wird sukzessive vergröbert (*restringiert*); auf der vergröberten Struktur wird jeweils wieder ein Berechnungsschritt durchgeführt. Das Ergebnis der Rechnung wird anschließend auf die feineren Strukturen übertragen (*Prolongation*), wo erneut Glättungsschritte durchgeführt werden. Die Startwerte für die hinzugekommenen Gitterpunkte werden dabei durch Interpolation ermittelt [Wittum 1990, Briggs et al. 2000].

Ein „halbes Mehrgitterverfahren“ wurde von Zhang und Yuen [2001] implementiert, deren Simulation mit einem groben Gitter startet. Sobald ein Gleichgewicht erreicht ist (wenn alle Knotengeschwindigkeiten eine bestimmte Grenze unterschritten haben), wird das Gitter global verfeinert. Zhang und Yuen geben eine 2,5-fach schnellere Konvergenz im Vergleich zu einer Simulation an, die durchgängig auf der feinsten Gitterauflösung arbeitet.

Problematisch an dem Verfahren von Zhang und Yuen ist, dass die für die Simulation benötigte Rechenzeit abhängig von der gerade aktuellen Verfeinerung stark schwankt. Der Einsatz von allgemeinen Mehrgitterverfahren für Echtzeit-Interaktionen ist durch die Laufzeitbedingung nur für sehr grobe Gitter möglich. Von verschieenen Autoren wird daher vorgeschlagen, ein Feder-Masse-Gitter nur an bestimmten Stellen zu verfeinern, um feine lokale Strukturen und geringen Rechenaufwand zu kombinieren.

Kriterien für die lokale Verfeinerung

Als Kriterium für die lokale Verfeinerung kann die Krümmung der Oberfläche verwendet werden, die über den Winkel zwischen benachbarten Federn [Hutchinson et al. 1996] oder zwischen Oberflächen- und Polygonnormalen [Villard und Borouchaki 2002] charakterisiert ist.

Bei diesen Ansätzen nimmt die Zahl der Elemente mit der Welligkeit der Oberfläche stark zu. Um eine obere Grenze der Rechenzeit definieren zu können, schlagen wir vor, die Anzahl der zur Verfeinerung verfügbaren Federn nach oben zu begrenzen. Wenn diese Grenze erreicht ist und weitere Federn benötigt werden, wird das Feder-Masse-Gitter an einer „wenig relevanten“ Stelle wieder vergröbert. Als Kriterium kann auch hierfür die Krümmung der Oberfläche herangezogen werden, für eine interaktive Simulation bietet sich aber ein anderer Gesichtspunkt an: eine Verfeinerung wird vor allem an denjenigen Stellen benötigt, an denen gerade eine Gewebeinteraktion stattfindet (Abb. 5.2).

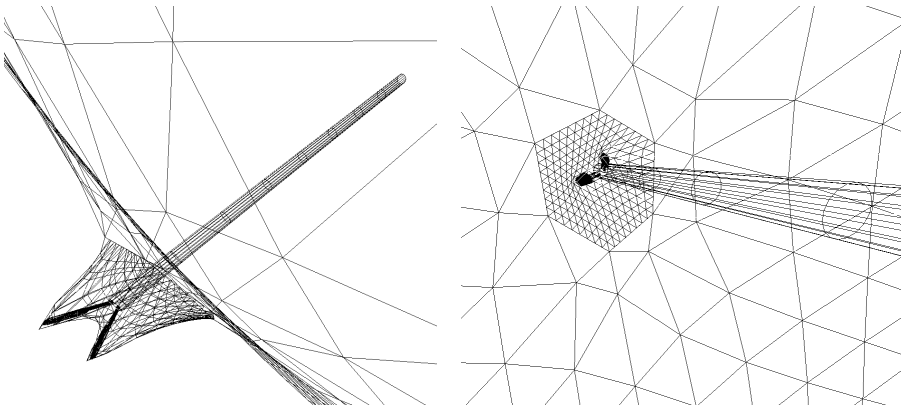


Abbildung 5.2: Lokale Gitterverfeinerung im Bereich der Interaktion. Die Deformationsvektoren wurden mit ZCOLL/F ermittelt.

Jeder verfeinerte Bereich wird mit einem Zeitstempel versehen. Wenn die Anzahl der für die Verfeinerung verfügbaren Elemente erschöpft ist, wird

die Stelle der am weitesten zurückliegenden Interaktion wieder vergröbert. Da die Aufmerksamkeit der Benutzerin auf dem aktuellen Bereich der Interaktion liegt, wird die Vergröberung auf diese Weise kaum wahrgenommen.[6] Überdies hat sich das Gewebe an älteren Interaktionsstellen häufig soweit entspannt, dass eine gröbere Diskretisierung wieder angemessen ist. Der Rechenaufwand für diesen Ansatz ist erheblich geringer als bei einem geometrischen Kriterium.

Wahl der neuen Parameter

Für die neu hinzugefügten Punktmassen und Federn müssen die geometrischen, topologischen und Simulationsparameter bestimmt werden. Wir gehen davon aus, dass die Feder-Masse-Struktur aus einer triangulierten Oberfläche extrahiert wurde, so dass jede Feder auf der Kante eines Dreiecks liegt. Ein solches Dreieck wird um eine Stufe verfeinert, indem die Mittelpunkte jeder Kante zu einem neuen Dreieck verbunden werden. Wenn man diesen Vorgang wiederholt, erhält man verschiedene Stufen (*Levels of Detail LOD*) der Gitterverfeinerung (Abb. 5.3). Für aufwändigere Ansätze wird auf die Arbeiten von Hoppe [1996; 1997] verwiesen.

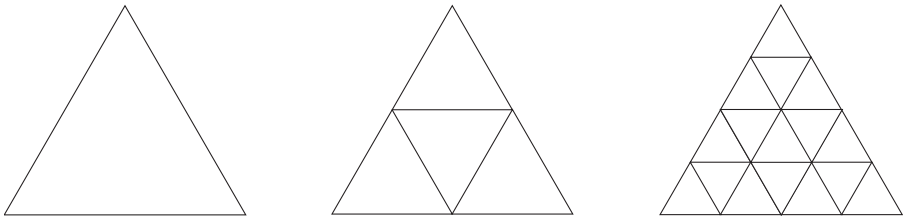


Abbildung 5.3: Verschiedene Stufen der Gitterverfeinerung.

In einem bestimmten Level-of-Detail n ist jede Seite eines Ausgangsdreiecks 2^n-fach unterteilt. Eine Feder der Länge l, der Federkonstante D und der Nulllänge l_0 wird durch Federn der Länge $l^{(n)}$, der Federkonstante $D^{(n)}$ und der Nulllänge $l_0^{(n)}$ ersetzt mit

$$l^{(n)} = \frac{l}{2^n} \tag{5.13}$$

$$D^{(n)} = 2^n \cdot D \tag{5.14}$$

$$l_0^{(n)} = \frac{l_0}{2^n} \tag{5.15}$$

[6] Mehr über die Anpassung des Rendering-Aufwands an die Wahrnehmung findet man z.B. bei Horvitz und Lengyel [1997].

Eine Seite des Ausgangsdreiecks enthält $2^n + 1$ Punktmassen. Durch Summation erhält man daher für die Zahl der Knoten, Federn und Dreiecke die folgenden Abhängigkeiten:

$$N^{(n)}_{\text{Knoten}} = (2^n + 1)(2^n + 2)/2 \tag{5.16}$$

$$N^{(n)}_{\text{Federn}} = 3 \cdot 2^{n-1}(2^n + 1) \tag{5.17}$$

$$N^{(n)}_{\text{Dreiecke}} = 2^{2n} \tag{5.18}$$

Um die Dichte des simulierten Materials konstant zu halten, ergibt sich die Masse $m^{(n)}$ jedes Knotens eines verfeinerten Dreiecks aus Gleichung 5.19 (bei mehreren zusammenhängenden Dreiecken muss dieser Zusammenhang modifiziert werden, um die gemeinsamen Knoten aneinandergrenzender Seiten zu berücksichtigen).

$$m^{(n)} = m^{(0)} \cdot \frac{N_{\text{Knoten}}(n)}{N_{\text{Knoten}}(0)} \tag{5.19}$$

Wie in Abschnitt 5.2.1 erläutert, hängt die Geschwindigkeit der Signalausbreitung bei expliziten Integrationsverfahren von der Feinheit der Diskretisierung ab. Für einen verfeinerten Bereich muss daher die Größe des Zeitschritts $\triangle t^{(n)}$ sowie die Anzahl der Iterationen $S^{(n)}$ angepasst werden. Für ein explizites Euler-Verfahren erhält man:

$$\triangle t^{(n)} = \frac{1}{2^n} \cdot \triangle t \tag{5.20}$$

$$S^{(n)} = 2^n \cdot S \tag{5.21}$$

An den Rändern eines verfeinerten Bereichs münden die neu hinzugefügten Federn nicht an den Randpunkten, sondern im Innern einer Feder (Abb. 5.4). Auf die an diesen Stellen erzeugten Knoten (*virtuelle Knoten*) wirken keine Feder-Masse-Kräfte, sondern Zwangsbedingungen, die die Knotenpositionen mit der jeweiligen Feder mitbewegen.

Bemerkung: Statt die Teilchenmassen im verfeinerten Bereich nach Gleichung 5.19 anzupassen, behalten Hutchinson et al. [1996] die Masse m der nicht verfeinerten Knoten bei. Dafür führen sie im verfeinerten Bereich nicht mehrere ($S^{(n)}$), sondern nur einen Zeitschritt der Größe $\triangle t^{(n)}$ durch („The response of the body due to its mass is governed by the time in which it is allowed to respond, and the magnitude of that mass. By employing smaller step sizes for those points which correspond to smaller regions of the sheet,

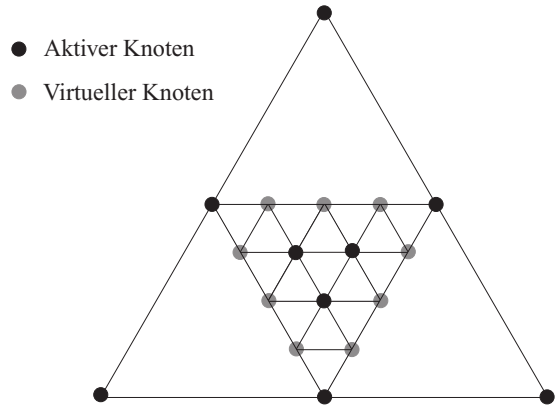


Abbildung 5.4: Virtuelle Knoten (grau) vermitteln zwischen verfeinerten und nicht verfeinerten Strukturen.

we achieve the same effect as a proportionally smaller mass would, as we employ Euler integration.“ [Hutchinson et al. 1996]).

Da die Dichte im verfeinerten Bereich durch die größere Anzahl von Teilchen mit unveränderter Masse zugenommen hat, wird für den Impulsübertrag bei einer Kollision ein entsprechender Korrekturfaktor verwendet. Der Grund für diese Vorgehensweise ist, dass so die einzelnen Punktmassen ihre mechanischen Eigenschaften behalten („The particles have uniform mass, so respond to forces in the normal way“ [Hutchinson et al. 1996]). Diese Argumentation trifft unseres Erachtens nur in dem seltenen Fall zu, dass die Anzahl der an einer Interaktion beteiligten Punktmassen unabhängig vom Verfeinerungsgrad ist.

Vergröberung

Bei der Vergröberung eines Bereichs werden die verfeinerten Strukturen durch die ursprünglichen Knoten und Federn ersetzt. Dabei tritt in den meisten Fällen eine Diskontinuität in der Bewegung der Struktur auf. Sie wird dadurch verborgen, dass der vergröberte Bereich durch die oben beschriebene zeitliche Sortierung aus dem Fokus der Wahrnehmung gerückt wird. Es bleibt zu untersuchen, ob es andere Möglichkeiten gibt, die Diskontinuität zu mildern (etwa durch schrittweise Vergröberung).

Schnittkanten im zu vergröbernden Bereich führen in der aktuellen Implementierung dazu, dass das entsprechende grobe Dreieck vollständig gelöscht wird. Auf diese Weise wird verhindert, dass die Komplexität des Gewebestücks durch fortlaufende Topologieänderungen zunimmt.

5.3 Zusammenfassung und Diskussion

Um auf eine bestimmte Interaktion zu reagieren, stehen für Kollisionserkennung und Simulation in einer virtuellen Realität typischerweise weniger als 10ms Rechenzeit zur Verfügung – in den meisten Fällen zu wenig, um ein physikalisches Modell zu rechnen. Zur Echtzeit-Simulation von Operationen werden daher deskriptive Modelle verwendet. Aufbauend auf der MGE-Datenstruktur sind in der VRM-Bibliothek Feder-Masse- und ChainMail-Modelle implementiert.

Zur Integration von Feder-Masse-Modellen werden bisher ausschließlich explizite Verfahren (explizites Euler-Verfahren, quasi-statisches Euler-Verfahren, Runge-Kutta-Verfahren zweiter Ordnung) verwendet, die pro Iteration nur einen geringen Rechenaufwand besitzen. Ihr großer Nachteil ist, dass die Anzahl der benötigten Iterationen bei feiner Diskretisierung und steifem Gewebe schnell ansteigt. Mit Hilfe eines einfachen Verfahrens zur adaptiven Gitterverfeinerung und der Längenkorrektur von Provot [1995] konnten diese Grenzen erweitert werden. Die Längenkorrektur wurde um eine Federsortierung ergänzt, die – ähnlich wie ein ChainMail-Algorithmus – eine gezielte Ausbreitung der Deformationsinformation innerhalb einer einzigen Iteration ermöglicht. Um eine bessere Approximation im Interaktionsgebiet zu erreichen, wurde ein einfaches Verfahren zur lokalen Gitterverfeinerung vorgestellt.

Wenn als Antwort auf eine Kollision ein Feder-Masse-Objekt deformiert wird, treten häufig Kräfte auf, die die nächste Kollision verursachen. Dadurch entstehen destabilisierende Oszillationen. Um sie zu vermeiden, wird um das interagierende starre Objekt (Operationsinstrument) eine „ϵ-Hülle gelegt. Wenn sich ein Gewebestück in dieser Hülle befindet, „berührt“ es das Instrument; seine Bewegung wird dann so eingeschränkt, dass es nicht mehr eindringen kann.

Nach unseren Erfahrungen müssen deskriptive Modelle an jede neue Operationssituation durch Veränderungen an den Parametern und den Algorithmen selbst angepasst werden. Es ist daher wichtig, eine möglichst große Bandbreite von verschiedenen Ansätzen zur Verfügung zu haben, die schnell miteinander kombiniert werden können. Die in Kapitel 3 beschriebene MGE-Struktur ist die Voraussetzung dafür.

Mit der steigenden Rechenkapazität der VR-Plattformen gewinnen aufwändigere Verfahren auch für interaktive Simulationen an Bedeutung. Echtzeit-FEM-Simulationen werden im Rahmen der Dissertation von Johannes Grimm untersucht; es ist geplant, implizite Integrationsverfahren für Feder-Masse-Modelle in die VRM-Bibliothek zu integrieren. Ebenfalls Gegenstand der zukünftigen Arbeit wird die Simulation von Blutungen sein.

6

Spezielle Aspekte der Visualisierung

In diesem Kapitel werden einige Besonderheiten von Visualisierung und Computergrafik angesprochen, die bei der Entwicklung der VRM-Applikationen von Bedeutung waren. Für weitere Informationen wird auf das Standardwerk zur Computergrafik [Foley et al. 1996] verwiesen, auf die aktuelle Entwicklung der 3D-Bibliotheken DirectX und OpenGL sowie auf die technischen Dokumentationen der Grafikkarten-Hersteller[1]. Bedingt durch die schnelle Entwicklung der 3D-Grafikhardware stellen letztere die beste Quelle für die Beschreibung hardwarenaher Implementierungen neuer Effekte dar.

6.1 Anzeigegeräte

Um visuelle Immersion zu erreichen, muss die simulierte Darstellung das menschliche Sichtfeld vollständig ausfüllen (sonst kann der *fishtank*-Effekt entstehen, bei dem man das Gefühl hat, die simulierte Welt wie in einem Aquarium zu betrachten). Ein Brute-Force-Ansatz zur Realisierung dieser Forderung ist die CAVE – ein geschlossener Raum, an dessen Wände stereoskopische Bilder projiziert werden. Die Bewegung der Benutzerin wird mit Hilfe eines Trackingsystems gemessen, so dass der visuelle Eindruck

[1] Die führenden Hersteller von Grafikkarten sind zurzeit ATI (www.ati.com) und NVidia (www.nvidia.com und developer.nvidia.com).

ihrer aktuellen Position angepasst werden kann.[2] Neben dem hohen technischen Aufwand steht dem Einsatz von CAVEs in Operationssimulationen entgegen, dass mechanische Zusatzgeräte (vor allem zur Realisierung des haptischen Renderings) wegen ihrer optischen Abschattungen nur schwer integriert werden können.

Bei *head-mounted displays* (HMDs) besteht das Problem der Abschattung nicht. Dafür ist es mit den heute existierenden Systemen noch nicht möglich, das Sichtfeld vollständig auszufüllen. Nach unserer Einschätzung wird die Bedeutung von *virtual retinal displays* [Viirre et al. 1998, McQuaide 2002] in naher Zukunft zunehmen, bei denen das Bild nicht über eine Linse auf einem Display betrachtet wird, sondern mit verschiedenfarbigen Lasern direkt auf die Netzhaut projiziert wird.

Gute Voraussetzungen für eine realistische Nachbildung des visuellen Eindrucks bieten Operationen, die mit Hilfe eines optischen Instruments durchgeführt werden, beispielsweise einem Stereo-Mikroskops oder dem Bildschirm eines Endoskopie-Systems. Durch den definierten Einblickwinkel ist in diesem Fall kein Pupillen- oder Kopftracking notwendig.

Stereoskopische Anzeige

Zur Erzeugung stereoskopischer Bilder wird die gleiche Szene zweimal aus verschiedenen Ansichten gerendert. Ein stereoskopisches Display sorgt dafür, dass jeder Kanal nur von dem entsprechenden Auge wahrgenommen wird. Die verschiedenen technischen Ansätze (z.B. aktive Shutter-Brillen, passive Rot-Grün- oder Polarisationsbrillen, separate bilderzeugende Linsensysteme in HMDs, über Augentracking gesteuerte Mikrolinsen [Schwerdtner und Heidrich 1998], Stereo-Beamer für CAVE-Projektionsflächen) haben ein gemeinsames Problem: ihre bildgebenden Elemente befinden sich auf einer Ebene, auf die beide Augen fokussieren. Um die Tiefeninformation zu kodieren, befinden sich korrespondierende Punkte der beiden Bilder nicht in dem Abstand, der dieser Fokuseinstellung entspricht (die Differenz nennt man *Disparität*). Die Blickrichtung der Augen (*Konvergenz*) weist auf ein virtuelles Objekt vor oder hinter der Bildebene, der Fokus dagegen ist auf die Bildebene selbst eingestellt.

Die ungewohnte Trennung von Konvergenz und Fokus kann bei manchen Menschen zu Übelkeit (*simulator sickness*) führen.[3] Siegel et al. [1998]

[2] Der Name CAVE steht übrigens nicht nur für *CAVE Automatic Virtual Environment* – Informatiker lieben rekursive Akronyme –, sondern ist auch eine Anspielung auf Platons Höhlengleichnis (Platon, *Der Staat*).

[3] Wir konnten diesen Effekt beim EyeSi-Stereomikroskop bisher nicht feststellen; das kann daran liegen, dass während des Operierens häufig Steuereinstellungen am Gerät vorgenommen werden müssen, bei denen die Mikroskopsicht verlassen wird.

schlagen daher vor, die Disparität im Vergleich zur Realität stark zu verringern. Die Autoren geben an, dass dadurch die Simulator-Sickness entscheidend verringert werden kann, obwohl sich der Tiefeneindruck kaum verschlechtert – vor allem, wenn die Szene über genügend andere Tiefenhinweise verfügt (z.B. Bewegungsparallaxe, Verdeckungen, Schattenwurf, Tiefenschärfe). Mehr über Stereo-Visualisierung findet man zum Beispiel bei Holliman [2002].

6.2 Computergrafische Modelle

Ähnlich wie bei der Gewebesimulation (Kapitel 5) können auch in der Computergrafik deskriptive und physikalische Modelle unterschieden werden. Für photorealistisches Rendering, etwa für digitale Effekte in Kinofilmen, werden physikalische Ansätze (Raytracing- und Radiosity-Modelle) verwendet, die noch nicht mit den für VR erforderlichen Wiederholraten ausgeführt werden können.

Der von 3D-Grafikkarten unterstützte Ansatz ist dagegen deskriptiv: entscheidendes Kriterium sind die visuelle Glaubwürdigkeit und schnelle Berechenbarkeit in Hardware. Im *Standardmodell* der Echtzeit-3D-Grafik werden Objekte durch polygonale Oberflächen beschrieben und von der Grafikkarte rasterisiert. Texturkoordinaten, Farben und Helligkeitswerte werden an den Eckpunkten jedes Polygons spezifiziert bzw. berechnet und in seinem Innern linear interpoliert. Sobald die Größe eines rasterisierten Polygons einige wenige Pixel überschreitet, führt vor allem die Interpolation der Helligkeitswerte (*Gouraud-Shading*) zu sichtbaren Artefakten.

Erweiterungen des Standardmodells

Viele optische Effekte sind im Standardmodell nicht enthalten. Dazu gehören:

- Spiegelungen
- Schattenwurf
- Linsenverzerrungen und Tiefenschärfe
- Lichtreflexe an feinen (nicht geometrisch modellierten) Oberflächenstrukturen: Glanzlichter, anisotropes Shading [Kautz et al. 2002])
- Volumenartige Objekte wie Rauch oder Flüssigkeiten (Beispiel: Einblutungen in das transparente, gallertartige Auginnere)

Für das Training von Operationen sind einige dieser Effekte sehr wichtig: Schattenwurf wird zur Abschätzung von Tiefe und Objektabständen benötigt; durch Linsenverzerrungen, begrenzte Tiefenschärfe, Einblutungen oder Schlierenbildung kann die Sicht behindert werden. Eine genaue Nachbildung von Oberflächenstrukturen und Reflexionseigenschaften wird nicht nur für die realistische Darstellung, sondern auch zur Charakterisierung bestimmter Pathologien und Operationssituationen benötigt.

Vertex-Shading und Pixel-Shading

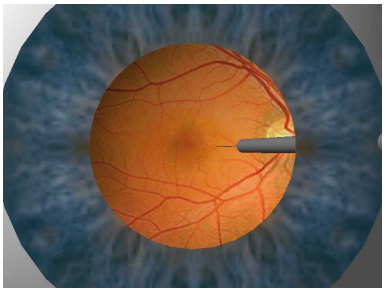
Um Spezialeffekte zu ermöglichen, wurde die Pipeline von 3D-Grafikkarten in den letzten Jahren fortlaufend um spezielle Funktionen ergänzt. Zunächst wurden diese Funktionen durch das Setzen von neuen Zuständen der grafischen *state machine* gesteuert. Seit kurzer Zeit besitzen die Grafikkarten programmierbare Einheiten mit eigenem Programmspeicher, Registern und einer arithmetisch-logischen Einheit [Lindholm et al. 2001, Doggett 2002], die sich im Vertex- bzw. Pixel-Strom der Grafikpipeline befinden (*vertex shader* bzw. *pixel shader*).

Mit Hilfe von Vertex-Shadern können Geometriedaten transformiert werden, um z.B. Linsenverzerrungen oder Objektverformungen zu erzeugen; sie führen die Beleuchtungsmodell-Rechnungen und die Projektion von Texturkoordinaten durch. Pixel-Shader bestimmen die Farbe eines Pixels mit Hilfe von Textur-Lookups, -filterung und -mischung. Sie können Per-Pixel-Lichtmodelle berechnen, was den Einsatz von *phong shading* und pixelbasierten Spotlights ermöglicht. Durch Uminterpretation von Texturwerten als Oberflächennormalen können mit Hilfe von Pixel-Shadern (*bump mapping*)-Verfahren realisiert werden, bei denen Oberflächenunebenheiten nicht mehr durch geometrische Modellierung erzeugt werden müssen.

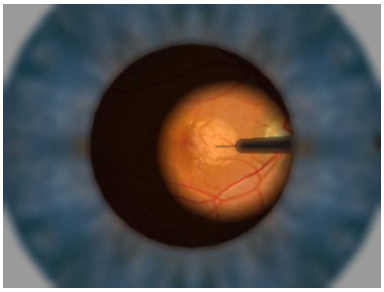
Die in der VRM-Bibliothek implementierten Vertex- und Pixel-Shading-Programme wurden vor allem für die Simulatoren EyeSi und Endosim entwickelt. Sie entstanden im Rahmen der Dissertation von Körner (ViPA), die noch in Arbeit ist und der Diplomarbeit von Sichler [2002]. In EyeSi wird mit Hilfe von Vertex- und Pixelshadern Tiefenschärfe, Bump-Mapping, pixelbasiertes Spotlight, Phong-Shading und hardwaregestütztes Shadow-Buffering (s. nächster Abschnitt) berechnet. Abb. 6.1 vermittelt einen Eindruck von der Wirkung dieser Spezialeffekte im Vergleich zum Standardmodell.

Schattenwurf

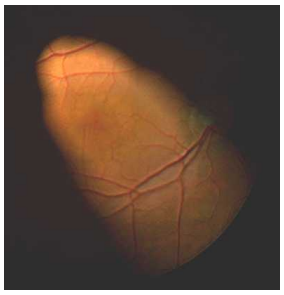
Da Schattenwurf bei intraokularen Operationen ein wichtiges Hilfsmittel für die Orientierung ist, wurden während der EyeSi-Entwicklung drei Verfahren implementiert.



(a) Standardmodell



(b) Per-Pixel-Spotlight, Tiefenschärfe, Bump-Mapping



(c) Detailaufnahme: Per-Pixel-Spotlight, Bump-Mapping

Abbildung 6.1: Steigerung des Realitätsgrads durch Vertex- und Pixel-Shader am Beispiel einer EyeSi-Szene.

Beim *Projektionsschatten*-Algorithmus [Tessman 1989] wird das schattenwerfende Objekt mit Hilfe geometrischer Berechnungen auf das beschattete Objekt projiziert. Wenn letzteres eine sehr einfache Form hat (Ebene, Kugel), ist dieses Verfahren schnell und benötigt keine über das Standardmodell hinausgehenden grafischen Operationen. Bei komplexeren Geometrien steigt der Rechenaufwand stark an – im allgemeinen Fall muss Projektion und Clipping für jedes Polygon einzeln durchgeführt werden, was die zur Verfügung stehende Rechenzeit schon bei geringen Polygonzahlen übersteigt.

Der *Volumenschatten*-Algorithmus [Heidmann 1991] konstruiert einen Pyramidenstumpf, dessen Deckfläche die Silhouette des schattenwerfenden Objekts ist. Mit Hilfe von Stencil-Buffer-Operationen wird bei der Rasterisierung festgestellt, welche Teile der Welt im Schattenvolumen liegen. Die Szene muss dazu zweimal gerendert werden. Das Verfahren ist allgemeiner als der Projektionsschatten, hat aber den Nachteil, dass in jeder Iteration die Objektsilhouette berechnet werden muss.

Beim *Shadow-Buffer*-Algorithmus wird die Szene in einem eigenen Durchgang aus der Sicht der Lichtquelle gerendert (die erhaltene Tiefenkarte nennt man *shadow map* oder *shadow buffer*). Diejenigen Pixel, die im Shadow-Buffer nicht sichtbar sind, liegen im Schatten und werden beim eigentlichen Rendering in der Schattenfarbe gezeichnet. Durch direktes Rendering in den Texturspeicher und Projektion der Textur auf die sichtbare Szene kann dieses Verfahren vollständig auf der Grafikkarte ausgeführt werden; die Grafikkarte muss dazu aber die Möglichkeit haben, benutzerdefinierte Per-Vertex- und Per-Pixel-Operationen auszuführen [Everitt et al. Erscheinungsdatum unbekannt]. Da dieses Verfahren unabhängig von der Objektform ist, ist es sehr flexibel. Anders als beim Volumenschatten-Ansatz benötigt man auch dann nur einen zusätzlichen Rendering-Durchgang, wenn zur stereoskopischen Darstellung auf zwei Kanälen gerendert wird.

Volumenvisualisierung

Wie bereits erwähnt, schliesst die Oberflächendarstellung des Standardmodells volumenartige Objekte aus. „Echte“ Volumengrafik hat einen hohen Speicher- und Rechenzeitbedarf. Obwohl bereits Bilder hoher Qualität in interaktiven Raten dargestellt werden können[4], ist der Rechenbedarf noch um etwa eine Größenordnung zu hoch, um Volumengrafik in PC-basierten VR-Systemen einsetzen zu können.

[4] Z.B. von der VolumeGraphics GmbH, einer Ausgründung des Lehrstuhls für Informatik V (www.volumegraphics.com)

Von Standard-3D-Grafikkarten werden trotzdem volumetrische Objekte wie Rauch oder Fell unterstützt. Sie werden meist durch Texturen auf semitransparenten, gestaffelten Ebenen dargestellt. Die Qualität einer solchen Darstellung hängt davon ab, wieviele solcher Ebenen verwendet werden; die zusätzliche Tiefeninformation bei stereoskopischer Darstellung verringert die Glaubwürdigkeit des Effekts.

6.3 Datentransfer auf die Grafikkarte

Von besonderer Bedeutung für die schnelle Verarbeitung komplexer Szenen ist die Möglichkeit moderner Grafikkarten, über den AGP-Bus Geometriedaten ohne Erzeugung von CPU-Last aus dem Hauptspeicher zu lesen. In diesem Abschnitt wird das Zeitverhalten auf dem aktuellen EyeSi-System[5] anhand eines Testprogramms beschrieben.

Das Testprogramm zeichnet ein Grafikobjekt aus 18.000 Dreiecken und führt auf den 1000 Knoten eines anderen Datensatzes eine Abstandsberechnung durch. Die Geometriedaten des Grafikobjekts werden in einem zusammenhängenden Speicherblock im RAM des Rechners gespeichert, einem *vertex array*. Für die Übertragung auf die Grafikkarte werden zum Vergleich Standard-OpenGL-Operationen (`GL_vertex_array`) und eine OpenGL-Erweiterung für NVidia-Grafikkarten (`NV_vertex_array`[6]) verwendet. Die NV-Vertex-Arrays liegen im *AGP-Speicher*, einem Teil des Hauptspeichers, auf den per DMA über den AGP-Bus zugegriffen werden kann.

Um das Verhalten bei zunehmender CPU-Last zu messen, wird die Abstandsberechnung bis zu zwanzig Mal pro Rendering-Schritt durchgeführt. Die für diese Berechnungen aufgeführte Zeit ist in der untersten Kurve in Abb. 6.2 aufgetragen. Pro Aufruf der Berechnung werden etwa 0,06ms benötigt. Die benötigte Zeit steigt linear mit der Anzahl der Aufrufe.

Für die oberen beiden Kurven wurde die für einen Iterationsschritt – n Berechnungen und ein Rendering-Schritt – benötigte Zeit gemessen. Für die Positionen der Eckpunkte jedes Dreiecks werden 9 float-Werte (jeweils 4 Bytes) übertragen. Aus den Messpunkten für $x = 0$ ergibt sich eine effektive Dreiecksleistung des Systems von 22 Millionen Dreiecken/s (NV-Vertex-Arrays) bzw. von 14 Millionen Dreiecken/s (GL-Vertex-Arrays). NV-Vertex-Arrays nutzen die Übertragungsrate des AGP-4×-Bus von 800 MBytes/s nahezu vollständig aus.[7]

[5] Pentium-IV mit 1,8GHz; AGP 4×; NVidia GeForce 4 Ti 4400 Grafikkarte; Treiberversion: Detonator 41.09

[6] siehe Spitzer und Everitt [Erscheinungsdatum unbekannt]

[7] $22 \cdot 10^6 \cdot 9 \cdot 4$ Bytes $= 792 \cdot 10^6$ Bytes

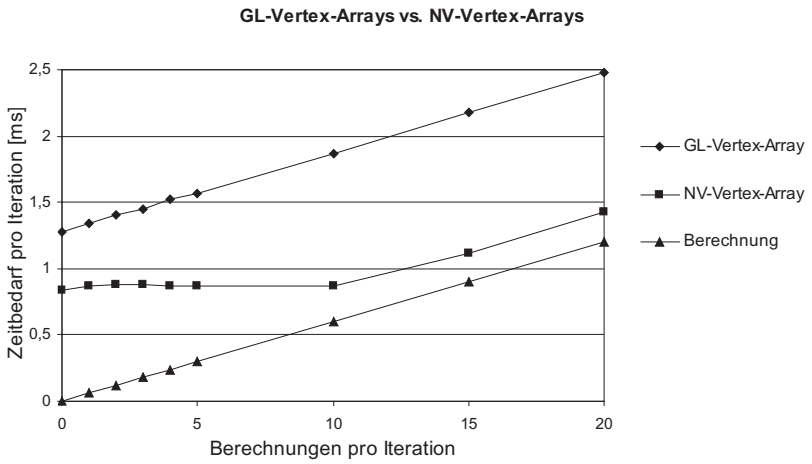


Abbildung 6.2: Datentransfer mit GL- und NV-Vertex-Arrays

Bei den GL-Vertex-Arrays (obere Kurve) addiert sich der zeitliche Aufwand für die Abstandsberechnung zur Rendering-Leistung: während der Übertragung der Geometriedaten auf die Grafikkarte kann die CPU keine anderen Operationen ausführen. Bei der Übertragung von NV-Vertex-Arrays dagegen kann die CPU weiter genutzt werden (mittlere Kurve), da der Datentransfer von der Grafikkarte durchgeführt wird: bei bis zu zehn Aufrufen der Berechnungsroutine bleibt die Gesamtlaufzeit eines Iterationsschritts konstant (bei etwa 0,9ms), um dann mit einem zeitlichen Abstand von etwa 0,2ms zur von der CPU ausgeführten Berechnung anzusteigen. In dieser Lücke werden blockierende Grafikoperationen (hier: clear buffers, swap buffers) ausgeführt.

Die größte Einschränkung bei der Verwendung von Vertex-Arrays im AGP-Speicher ist, dass der Zugriff auf die Daten ungepuffert (*uncached*) erfolgt. Dieser Umstand ist dann problematisch, wenn deformierbare Objekte repräsentiert werden. Abb. 6.3 zeigt, was geschieht, wenn die Daten für die oben beschriebene Abstandsmessung im AGP-Speicher gehalten werden: die Berechnung wird um etwa einen Faktor 10 langsamer.

6.4 Diskussion

Die Entwicklung der hardwareunterstützten 3D-Grafik für PC-Plattformen verlief in den letzten Jahren schneller als die Entwicklung neuer CPUs. Gemessen an der Anzahl der Transistoren, sind Grafikprozessoren inzwi-

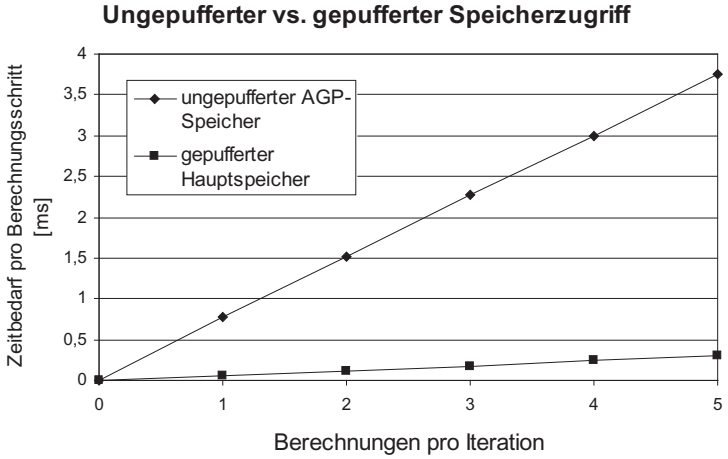


Abbildung 6.3: Vergleich der Zugriffsgeschwindigkeiten auf AGP-Speicher und Hauptspeicher

schen komplexer als CPUs.[8] Dank der Hardware-Unterstützung von T&L-Operationen (*transform and lighting*), der schnellen Übertragung von Geometriedaten über den AGP-Bus und der flexiblen Implementierung von Lichtmodellen und Spezialeffekten mit Vertex- und Pixelshadern können inzwischen überzeugende Visualisierungen oberflächenbasierter Modelle realisiert werden. Die Verlagerung komplexer Operationen auf die Grafikkarte ist noch im Gange – beispielsweise sind ATI-Grafikkarten inzwischen in der Lage, Geometriebeschreibungen selbstständig zu verfeinern und damit das über den AGP-Bus übertragene Datenvolumen zu reduzieren [ATI 2001].

Vor allem der deskriptive Charakter der Lichtmodelle führt bisher dazu, dass die Implementierung eines neuen Effekts aufwändig ist. Es ist davon auszugehen, dass innerhalb der nächsten Jahre physikalische Modelle für das 3D-Echtzeit-Rendering an Bedeutung gewinnen werden – etwa hardwaregestütztes Echtzeit-Raytracing auf PC-Plattformen [Schmittler et al. 2002].

Um die bei der Verlagerung von Operationen auf die Grafikkarte freiwerdende CPU-Zeit für andere Berechnungen nutzen zu können, muss das System mit Hilfe der in Abschnitt 2.5 diskutieren Synchronisationsmuster sorgfältig ausbalanciert werden.

[8] Ein moderner ATI Radeon 9700 Pro Grafikchip hat 100 Millionen Transistoren, während ein Pentium-IV-Prozessor mit 42 Millionen Transistoren auskommt.

7

EyeSi: VR für die augenchirurgische Ausbildung

> Any sufficiently advanced technology is indistinguishable from magic.
>
> —Arthur C. Clarke

EyeSi ist ein Trainingssimulator für ophthalmochirurgische Operationen, dessen Software auf der VRM-Bibliothek beruht. Nach der Entwicklung mehrerer Prototypen wurde im letzten Jahr eine marktreife Nullserie aufgebaut. Hard- und Software wird in den Abschnitten 7.3 und 7.4 beschrieben. Zuvor behandelt Abschnitt 7.1 den medizinische Hintergrund. Abschnitt 7.2 stellt vergleichbare Simulatoren vor.

Unterschiedliche Aspekte von EyeSi wurden bereits in Schill et al. [1999a], Schill [2001], Wagner et al. [2001; 2002a;b] beschrieben.

7.1 Medizinischer Hintergrund

Intraokulare Operationen gehören zu den schwierigsten Aufgaben der Mikrochirurgie: Die Strukturen im Auge sind extrem empfindlich; die Operationen werden unter einem Stereomikroskop durchgeführt, was die Hand-Auge-Koordination erschwert; Blut oder Eintrübungen behindern die Sicht

der Chirurgin. Die für die Instrumentbewegung notwendige Präzision liegt im Submillimeterbereich, nahe der Grenzen der menschlichen Feinmotorik. Kräfte sind bei der Bewegung der Instrumente kaum zu spüren, da das Gewebe im Auge sehr weich ist.

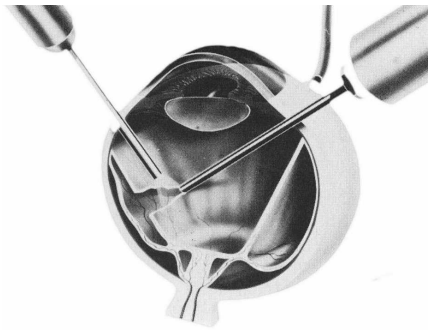


Abbildung 7.1: Schematische Darstellung einer vitreoretinalen Operation mit einer Lichtquelle (von links) und einem Vitrektom (von rechts). Aus Freyler [1985].

Abb. 7.1 illustriert eine Operation im hinteren Augenabschnitt (*intraokulare* oder *vitreoretinale Chirurgie*). Dabei werden zwei Instrumente in das Auge eingeführt. Das eine ist meist eine Lichtquelle, das andere ein Instrument zur Gewebeinteraktion – beispielsweise ein einfacher Haken, eine Pinzette, eine Hohlnadel oder ein *Vitrektom*. Ein Vitrektom ist ein Saug-Schneideinstrument, mit dem Gewebe aus dem Auge entfernt werden kann. Dazu ist an einer Hohlnadel eine seitliche Öffnung angebracht. In der Nadel herrscht ein variabler Unterdruck, mit dessen Hilfe Gewebe eingesaugt wird. Ein pneumatisch betriebenes, oszillierendes Messer in der Nadel schneidet einzelne Gewebestücke ab, die dann aus dem Auge transportiert werden.

Stereomikroskop-Einstellungen, Frequenz und Schneidrate des Vitrektoms sowie (bei einigen modernen Instrumenten) der Öffnungswinkel der Pinzettenscheren werden mit Fußpedalen gesteuert; zu ihrer Konfiguration und der Steuerung weiterer Gerätefunktionen steht meist ein Touchscreen zur Verfügung.

Zu den größten Gefahren bei der Manipulation der Instrumente im Auge gehört die Verletzung der Netzhaut und der *Linsenkapsel*, die die Augenlinse umgibt und an ihrem Platz hält. Wenn die Linsenkapsel einreißt, kommt es zum *Linsenverlust*. Die Linse sinkt auf die Retina herab und muss dort entfernt werden. Beim Einsatz des Vitrektoms muss darauf geachtet werden, dass die Saugöffnung nicht zur Netzhaut weist, da sie sonst abgehoben werden kann.

Die räumliche Position der Instrumente und ihr Abstand zur Netzhaut oder zu pathologischen Strukturen werden durch die Tiefeninformation des Stereomikroskops sowie durch den Schattenwurf abgeschätzt. Die richtige

Einschätzung dieser visuellen Informationen ist eine wichtige Fähigkeit, die beim Operationstraining erlernt werden muss.

Im Folgenden werden einige Prozeduren der vitreoretinalen Chirurgie beschrieben.

Entfernung des Glaskörpers Das Auge ist mit einer transparenten, gallertartigen Flüssigkeit gefüllt, dem *Glaskörper* (*corpus vitreum*). Bei einer *Vitrektomie* wird der Glaskörper aus dem Auge entfernt und durch eine Elektrolytlösung oder ein Silikonöl ersetzt. Vitrektomien müssen durchgeführt werden, wenn sich der Glaskörper trübt oder zusammenzieht. Sie werden ebenfalls notwendig, wenn sich die Netzhaut auf pathologische Weise verändert (siehe unten).

Entfernung von Gewebewucherungen Krankhafte Veränderungen der Netzhaut, wie sie zum Beispiel bei Diabetes entstehen (*diabetische Retinopathie*), führen zu Einblutungen und unkontrolliertem Gewebewachstum in den Glaskörperraum. Eine besondere Gefahr besteht darin, dass die Gewebestränge kontrahieren und die Netzhaut abziehen. Sie müssen daher aus dem Auge entfernt werden.

Entfernung der inneren Laminarmembran Die Netzhaut besteht aus vielen, funktional unterschiedlichen Schichten. Die innerste Schicht zum Glaskörper hin heisst ILM (*innere Laminarmembran*). Sie wird abgezogen, um darunterliegende Strukturen bei pathologischen Veränderungen freizulegen.

Retina-Relokation Die Stelle schärfsten Sehens, die *Makula* kann im Alter ihre Sehfähigkeit verlieren (*altersabhängige Makuladegeneration*). In einer neu entwickelten Therapie wird die Retina durch Unterspritzen mit einer Flüssigkeit abgehoben, rotiert und wieder an den Augenhintergrund angedrückt. Dadurch werden funktionell intakte Stellen der Retina in das optische Zentrum gerückt.

Zurzeit wird die Ausbildung von Ophthalmochirurgen an Tieraugen in *Wetlabs* sowie beim Assistieren bei realen Operationen durchgeführt. Die Ausbildung am Patienten ist risikoreich; bei Tieraugen können viele Pathologien nicht trainiert werden, die an Tieren in anderer Form oder überhaupt nicht auftreten. Überdies weist totes Gewebe ein anderes biomechanisches Verhalten auf. Die aufwändige, teure und riskante Ausbildung kann mit Hilfe einer realistischen VR-Simulation erheblich verbessert werden.

7.2 Stand der Forschung

Um die Steuerung eines Roboters für Augenoperationen zu trainieren, wurde an der Universität Auckland bereits 1994 eine virtuelle Realität für Augenoperationen entwickelt [Mallinson et al. 1994]. Die Operationsinstrumente wurden über eine GUI, den Roboter oder ein angeschlossenes Trackingsystem gesteuert, auf das Mallinson et al. nicht näher eingehen. Die computergrafische Darstellung des Auges war sehr detailreich[1] – beispielsweise wurde das Aderngeflecht, von dem das Auge umgeben ist, mit Hilfe eines fraktalen Modells erzeugt. Die Simulation war auf die Deformation der *Cornea* (Hornhaut) des Auges eingeschränkt, die mit Hilfe eines FEM-Modells durchgeführt wurde. Die Updaterate der Visualisierung lag (1994!) für 1.100-5.700 Polygone bei 40-420ms auf einer Grafik-Workstation. Die Simulation wurde auf einem zweiten Rechner durchgeführt. Zum Erzeugen des Stereoeindrucks wurde eine Shutter-Brille und ein Monitor verwendet. Soweit uns bekannt ist, wird das Projekt nicht weitergeführt.

Die Universität von Illinois in Chicago hat 1998 einen Simulator für Vitrektomien vorgestellt [Neumann et al. 1998]. Zur Instrumentbewegung wurden 3D-Mäuse verwendet. Ähnlich wie bei EyeSi wurden Retina und intraokulares Gewebe mit Hilfe von Feder-Masse-Membranen modelliert. Dabei wurde besonderer Wert auf die Gitterverfeinerung bei der Berechnung von Schnittlinien gelegt. Die Software lief auf einer Silicon Graphics MXE Octane Workstation. Das Projekt wurde eingestellt.

Die Firma SimEdge[2], die aus einem Projekt der Universität Lille [Meseure et al. 1995] hervorgegangen ist, hat einen Simulator zum Training von Spaltlampen-Untersuchungen und Laserkoagulation entwickelt. Ähnlich wie bei EyeSi wird eine Stereomikroskop-Mimik mit kleinen Displays verwendet; die für die Untersuchung benötigten Regler sind dem Original nachempfunden, so dass ein hoher Realitätsgrad erzeugt wird. Interaktion mit Instrumenten ist bei dieser Art der Behandlung nicht notwendig.

Vom Asahikawa Medical College (Japan) und der Mitsubishi Electric Corp. wird ein Simulator für intraokulare Chirurgie entwickelt, dessen Funktionsweise leider nur in einer recht oberflächlichen Veröffentlichung angedeutet wird [Hikichi et al. 2000]. Als VR-Interface für die Instrumentbewegung dienen zwei Force-Feedback-Geräte sowie zwei Fußpedale. Die Visualisierung erfolgt über eine Stereomikroskop-Nachbildung mit 1280×1024 Punkten Auflösung (pro Kanal?). Für die Berechnungen werden eine Grafik-Workstation und ein PC verwendet, die über eine LAN-Verbindung miteinander verbunden sind. Trainingssitzungen werden mit Hilfe eines Videorekorders aufgezeichnet, der ein herunterskaliertes Videosignal erhält. Der

[1] Für Beispiele s. www.esc.auckland.ac.nz/Groups/Bioengineering/Images/Eye/index.html
[2] www.simedge.com

Simulator kann Gewebeinteraktionen berechnen, die Autoren geben aber keinen Hinweis auf die verwendeten Algorithmen.

Verma et al. [2003] von der Universität Hull (England) haben eine Machbarkeitsstudie für einen Augenoperationssimulator durchgeführt; das Auge wird dabei mit Hilfe eines Tischtennisballs nachgebildet, in den magnetisch getrackte Instrumente eingeführt werden. Die Interaktion mit dem Glaskörper wird simuliert, auch hier werden aber leider keine Details zur Realisierung angegeben.

Die schwedische Firma Melerit[3] bietet einen Augenoperationssimulator zum Training von Katarakt-Operationen an.

7.3 Hardware

7.3.1 Aufbau und Interface

Abb. 7.2 zeigt den mechanischen Aufbau von EyeSi, Abb. 7.3 Detailansichten der Interface-Komponenten. Die Chirurgin blickt durch zwei Okulare (a) auf organische LEDs (OLEDs) mit einer Auflösung von je 800×600 Farbpixeln und einer Bildwiederholrate von 85Hz. Sie stellen das computergenerierte stereoskopische Bild des Operationsgebiets dar. Ansteuerung der OLEDs und Okularoptik wurden von der VRmagic GmbH in Zusammenarbeit mit ViPA entwickelt. Die Funktionen der drei Regler sind frei konfigurierbar, üblicherweise werden mit ihnen (simulierte) Mikroskopeinstellungen wie Fokus, Zoom und Umgebungslicht gesteuert.

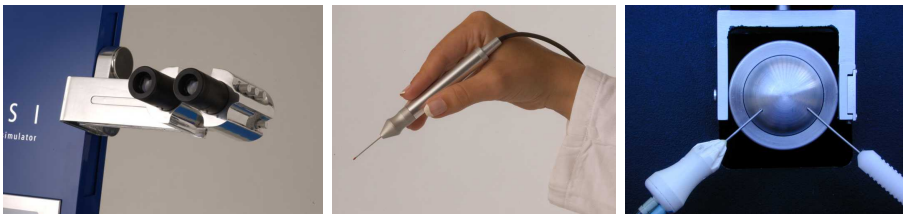
Originalgetreue Nachbildungen oder echte Operationsinstrumente (b) werden in ein mechanisches Model des Auges (c) eingeführt. Das Augenmodell ist eine kardanisch aufgehängte Halbkugel mit zwei festen Einstichlöchern; seine drei Rotationsfreiheitsgrade entsprechen denen des echten Auges. Die rücktreibenden Kräfte der Augenmuskeln werden durch Federn nachgebildet. Auge und Instrumente sind mit Farbpunkten markiert, um die optische Messung ihrer Position und Orientierung zu vereinfachen (Abschnitt 7.3.2).

Die Gesichtsmaske (d) wird für einen realistischen haptischen Eindruck benötigt: zum einen stützt sich die Chirurgin während der Operation häufig auf der Stirn des Patienten ab, zum anderen wird die Instrumentbewegung durch die Geometrie des Gesichts behindert.

Mit Hilfe der Fußpedale (e) werden Instrument- oder Mikroskopfunktionen gesteuert; der Dialog mit dem EyeSi-System wird – ähnlich wie die Bedienung der Geräte im Operationssaal – über einen Touchscreen (f) durchgeführt.

[3] www.melerit.se

Abbildung 7.2: EyeSi



(a) Stereomikroskop (b) Instrument (c) Auge und Instrumente

(d) Gesichtsmaske (e) Fußpedale (f) Touchscreen

Abbildung 7.3: EyeSi-Interface-Komponenten. Im Betrieb befindet sich das mechanische Auge unter der Gesichtsmaske.

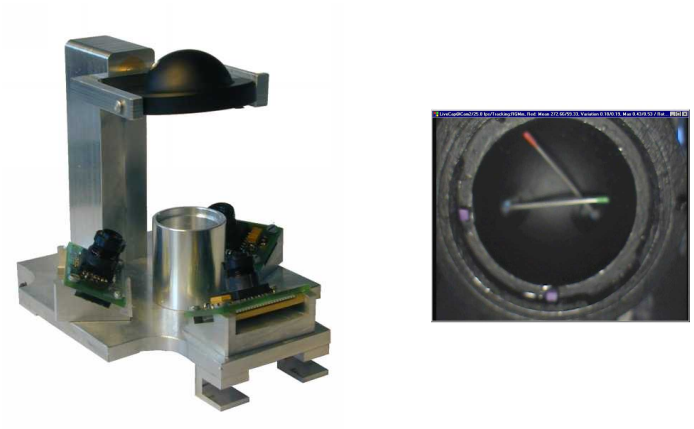
7.3.2 Optisches Tracking

Die Aufgabe des EyeSi-Trackingsystems ist es, die Rotation des mechanischen Auges und die Positionen der Instrumente zu messen. Um die Realitätsnähe des Simulators nicht zu verringern, darf das Trackingsystem keine zusätzlichen Kräfte auf die Instrumente erzeugen oder ihre Bewegung behindern. Aufgund der VR-Echtzeitbedingung muss es eine hohe zeitliche, wegen der feinen Strukturen im Auge eine hohe räumliche Auflösung besitzen.

Optische Trackingsysteme bieten sich an, da sie eine hohe Genauigkeit erreichen, berührungsfrei arbeiten und keine großen Veränderungen an den zu trackenden Objekten notwendig machen. Allerdings gibt es keine kommerziellen Systeme, die innerhalb der kleinen Abmessungen des Operationsgebiets (weniger als $3 \times 3 \times 3$ cm^3) die erforderliche Genauigkeit erreichen. Darüberhinaus erleichtert es die mechanische Konstruktion des Simulators, wenn das Tracking-System möglichst kompakte Abmessungen besitzt.[4]

Speziell für EyeSi wurde ein optisches Trackingsystem entwickelt (Abb. 7.4(a)). Drei CMOS-Farbkameras mit je 640×480 Pixeln Auflösung sind unter dem mechanischen Auge montiert und betrachten das Operati-

[4]Im ersten EyeSi-Prototypen (1997) konnte das Trackingsystem aufgrund der Kameraabmessungen nicht unter dem mechanischen Auge angebracht werden; stattdessen wurde der Strahlengang durch einen Spiegel auf eine separate Trackingeinheit umgelenkt.



(a) Mechanisches Auge und Kamerablock

(b) Livebild einer Kamera

Abbildung 7.4: Optisches Tracking im EyeSi-System

onsgebiet aus verschiedenen Perspektiven. Externe und interne Parameter der Kameras werden mit dem von Tsai [1987] beschriebenen Verfahren bestimmt.

Abb. 7.4(b) zeigt das Auge mit eingeführten Operationsinstrumenten aus der Sicht einer der Kameras. Erkennbar sind die Farbmarkierungen an den Instrumenten und dem Auge: die Rotation des Auges wird durch die beiden an seiner Peripherie angebrachten Markierungen sowie seinen rotationsinvarianten Mittelpunkt bestimmt. Position und Orientierung der Instrumente werden durch Farbmarkierungen an ihrer Spitze sowie die im Koordinatensystem des Auges festen Einstichlöcher definiert. Die so nicht erkennbare Rotation der Instrumente um ihre Längsachse wird mit Hilfe eines magnetischen Sensors gemessen.

Zur Erkennung der farbigen Markierungen würde ein typisches optisches Tracking-System die folgenden Schritte durchführen: (1) Bildaufnahme, (2) Übertragung über eine Framegrabber-Karte auf den PC, (3) Bildverarbeitung und (4) 3D-Rekonstruktion. Bei dieser Vorgehensweise wird die bei der Bildaufnahme verursachte Latenz vor allem durch die Schritte (2) und (3) erheblich vergrößert. Schritt (3) verringert darüberhinaus die für Simulation und Rendering verfügbare Rechenkapazität des PCs.

Um diese Nachteile zu vermeiden, wurde eine FPGA-basierte Bildaufnahme- und -verarbeitungseinheit entwickelt, die den von den Kameras erzeugten Pixelstrom in Hardware verarbeiten kann [Ruf 2000]. Nachdem für jede Markierung ein Bereich im Farbraum definiert wurde, der ihr zugeordnet ist, kann jedes Pixel klassifiziert werden; anschließend werden die Sensorkoordinaten der Farbschwerpunkte jeder Klasse

subpixelgenau bestimmt und via USB zum PC übertragen.

Diese Rechnungen werden vom FPGA in der Pixel-Pipeline durchgeführt, so dass keine zusätzliche Latenz durch Zwischenspeichern des Kamerabilds entsteht. Die Kameras arbeiten mit einer variablen Wiederholrate von bis zu 50Hz; bei der höchsten Wiederholrate ergibt sich daraus eine Latenz von 20ms.

Im PC wird mit jeweils einem Kamerapaar eine 3D-Rekonstruktion jedes Farbmarkers durchgeführt; die dritte Kamera dient zur Erhöhung der Genauigkeit und hilft bei Verdeckungen. Das EyeSi-Trackingsystem kann Bewegungen von weniger als 30μm auflösen [Schill 2001, Abschnitt 6.1.2]. Von Riviere und Khosla [1999] wird die Genauigkeit der Bewegung eines Mikrochirurgen mit 0,1mm angegeben; die Frequenz des menschlichen Zitterns liegt nach dieser Veröffentlichung bei 8-10Hz. Räumliche und zeitliche Auflösung des EyeSi-Trackings sind daher gut zur Messung dieser Parameter geeignet.

7.3.3 Rechnereinheit

Um die Kosten des Systems zu begrenzen, ist eine der Randbedingungen bei der EyeSi-Software-Entwicklung, dass sie auf Standard-PC-Plattformen einsetzbar ist. Zurzeit (2003) wird ein Pentium-IV-1,8GHz-System mit AGP-4×-Grafikbus, 256 MBytes RAM und einer GeForce 4 Ti 4400-Grafikkarte (Treiber: Detonator 41.09) unter Windows XP Home verwendet.

7.4 Software

Die EyeSi-Software beruht auf der Funktionalität, die von der VRM-Bibliothek zur Verfügung gestellt wird. Ihre Struktur richtet sich nach dem in Kapitel 2 beschriebenen Schema: die Entwicklungsstränge für Ein- und Ausgabe (optisches und magnetisches Tracking, Auslesen von Mikroskop-Reglern und Fußpedalen), GUI-Komponenten (Touchscreen-Steuerung, GUI-Software, Player/Recorder) und dem VR-Teil (Gewebesimulation, Rendering) sind voneinander getrennt, was eine schnelle und flexible Softwareentwicklung ermöglicht. Neben dieser „horizontalen“ Trennung besteht eine „vertikale“ Trennung (Abb. 7.5) bei der Entwicklung der VRM-Bibliothek, der EyeSi-Basissoftware und der einzelnen Trainingseinheiten (*EyeSi-Szenarien*).[5]

[5]Diese Unterscheidung wird ganz ähnlich auch bei Computerspielen durchgeführt: ein bestimmtes Spiel (vergleichbar mit der EyeSi-Basissoftware) basiert auf einer *Game engine* (der VRM-Bibliothek); innerhalb des Spiels müssen einzelne *Missionen* (EyeSi-Szenarien) erfüllt werden.

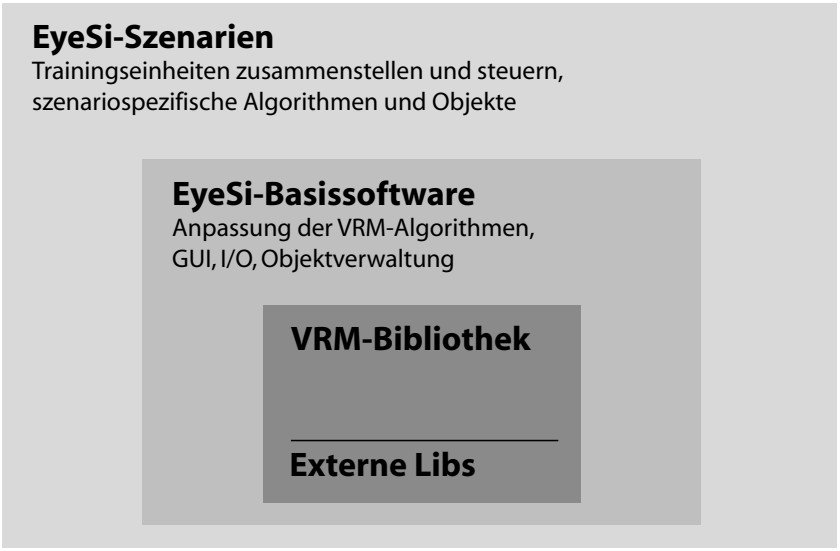
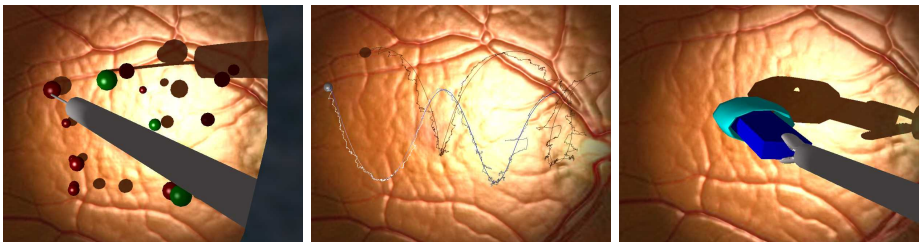


Abbildung 7.5: Spezialisierungsschichten der EyeSi-Software

7.4.1 Szenarien

EyeSi unterscheidet zwischen *abstrakten* und *Operationsszenarien*.[6] Abstrakte Szenarien sind Trainingseinheiten, mit denen eine bestimmte Fähigkeit anhand einer idealisierten Aufgabe trainiert werden kann – etwa die Navigation im Auge oder die Arbeit mit einem bestimmten Instrument.



(a) (b) (c)

Abbildung 7.6: Beispiele für abstrakte Szenarien

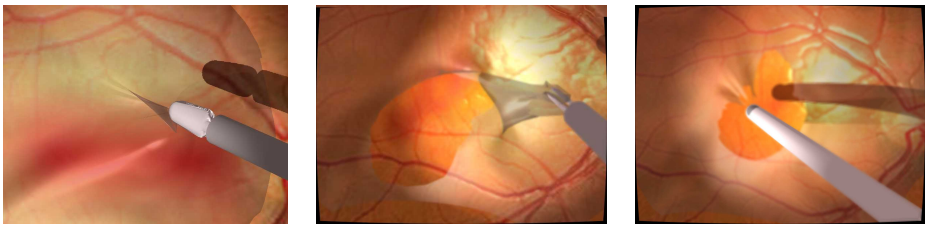
Ein abstraktes Szenario entspricht keiner realen Operationssituation; es verwendet einfache Objekte wie Kugeln, Quader oder Ringe, die im Auge manipuliert werden (Abb. 7.6, außen) oder Trajektorien, denen der Auszu-

[6] Diese Unterscheidung bezieht sich nur auf die Bedeutung der Szenarien in der realen Welt; die Art der Implementierung ist für abstrakte und Operationsszenarien dieselbe.

bildende folgen muss (Abb. 7.6, Mitte). Das System analysiert dabei Geschwindigkeit und Genauigkeit der Bewegung. Die bisher implementierten abstrakten Szenarien enthalten keine deformierbaren Objekte; die Kollisionserkennung kann mit einfachen geometrischen Ansätzen durchgeführt werden.

Aufwändiger sind Operationsszenarien. Bei ihnen wird ein Ausschnitt einer realen Operationssituation modelliert. Für die Kollisionserkennung werden geometrische Berechnungen und der bildbasierte Algorithmus ZCOLL/F (Kapitel 4), für die Simulation Feder-Masse-Modelle (Kapitel 5) verwendet. Die möglichen Gewebeinteraktionen beschränken sich zurzeit auf Instrument-Membran-Interaktionen, da die Visualisierung des volumetrischen Glaskörpers noch zuviel Rechenzeit benötigt (die Simulation des Glaskörpers mit ChainMail-Algorithmen beschreiben Schill et al. [1998] und Schill [2001]).

Die Gewebeparameter jedes Szenarios wurden in einem iterativen Prozess mit den klinischen Partnern des Projekts bestimmt. Weitere Szenarien werden zurzeit von der VRmagic GmbH entwickelt.



(a) (b) (c)

Abbildung 7.7: Operationsszenario: Membranentfernung mit Pinzette und Vitrektom.

Abb. 7.7(c) stellt verschiedene Schritte in einem Operationsszenario zur Entfernung einer pathologischen Membran dar, die an einigen Stellen mit der Retina verwachsen ist. Als Instrumente kommen eine Pinzette und ein Vitrektom zum Einsatz. Mit der Pinzette muss der Chirurg die Membran vorsichtig von der Netzhaut lösen (Abb. 7.7(b)), um sie anschließend mit dem Vitrektom aus dem Auge zu entfernen (Abb. 7.7). Dabei muss darauf geachtet werden, dass die Instrumente die Retina nicht verletzen (im unteren Bereich von Abb. 7.7(a) sind solche Verletzungen als Blutungen zu erkennen); geplant ist, das Szenario um eine Netzhautablösung zu erweitern, die bei zu starken Kräften an den Verwachsungsstellen eintreten kann.

In verschiedenen Schwierigkeitsstufen des Szenarios kann die geometrische Ausprägung der Membran (Nähe zur Retina; Stärke der Verwachsungen mit

der Retina) verändert werden; in einem ähnlichen Szenario mit einer eng an der Netzhaut anliegenden Membran und anderen Gewebeparametern kann das Abziehen der ILM trainiert werden.

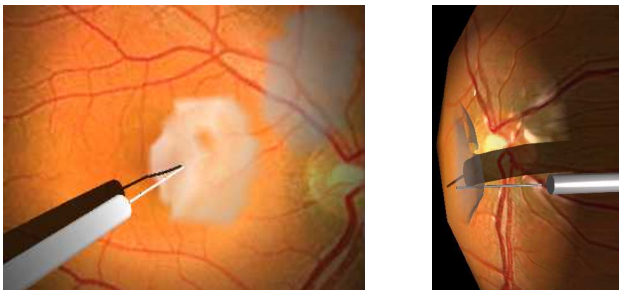


Abbildung 7.8: Operationsszenario: Netzhautabhebung als erster Schritt einer Retina-Relokation (rechts zur Veranschaulichung eine – in Realität nicht mögliche – Seitenansicht eines aufgeschnittenen Auges)

Ein weiteres Operationsszenario ist in Abb. 7.8 dargestellt. Ziel der Operation ist die Retina-Relokation bei altersabhängiger Makuladegeneration. Bisher wurde nur der erste Schritt der Operation modelliert – das Abheben der Retina durch Injektion einer Elektrolytlösung. Trainingsziel ist die vorsichtige Handhabung der Injektionsnadel. Wenn während des Injizierens das Instrument zu stark bewegt wird, reisst das Gewebe an dieser Stelle ein. In einem zweiten Schritt des Szenarios wird die Retina rotiert und wieder an den Augenhintergrund angedrückt werden.

Die Retina-Relokation ist noch Gegenstand der medizinischen Forschung und wird nur von wenigen Chirurgen praktiziert. VR-Ausbildungssysteme wie EyeSi können zur Verbreitung solcher neuer Operationstechniken einen entscheidenden Beitrag leisten.

7.4.2 Zeitverhalten

Die Konzeption der zeitlichen Steuerung von EyeSi richtet sich nach den in Abschnitt 2.5 erarbeiteten Erkenntnissen: um Synchronisationslatenz zu vermeiden, arbeitet die EyeSi-Software synchron zum Trackingsystem, dessen Wiederholfrequenz maximal bei 50Hz liegt; zu kleineren Werten hin kann die Frequenz stufenlos verändert werden. Auf Multithreading wurde wegen der in Abschnitt 2.5 diskutierten Probleme auf Einzelprozessor-Maschinen verzichtet. Die – von Natur aus eigentlich asynchrone – GUI-Steuerung wurde daher in die Simulations- und Visualisierungsschleife von EyeSi integriert.

Bei der Implementierung der Algorithmen für die einzelnen Szenarien wurde darauf geachtet, große zeitliche Schwankungen zu vermeiden, um

Wiederholrate und Latenz möglichst konstant zu halten und die Präsenz nicht zu zerstören. Bei der Kollisionserkennung werden keine hierarchischen Hüllkörper-Verfahren verwendet, deren Laufzeit stark vom Grad der Interaktion abhängt (Kapitel 4); die Gewebeverfeinerung (zurzeit nur in einer experimentellen Implementierung) wird nur bis zu einer vorher definierten Grenze durchgeführt (Kapitel 5).

Eine Alternative sind *time-critical computing*-Ansätze, bei denen sich die Komplexität eines Verfahrens dynamisch an die verfügbare Laufzeit anpasst. Nach unserer Erfahrung führt dies aber zu einem erheblich höheren Entwicklungsaufwand und einem weniger nachvollziehbaren Systemverhalten. Überdies tritt der größte Zeitbedarf gerade dann auf, wenn auch die größte Genauigkeit erfordert wird: bei starker Gewebe-Instrument-Interaktion. Die Szenarien sind daher auf diesen Fall optimiert. Wie in Kapitel 5 erwähnt, kann aber Zeit dadurch eingespart werden, dass sich die Berechnung auf die *region-of-interest* konzentriert und Randbereiche vernachlässigt, die vom Chirurgen ohnehin nicht wahrgenommen werden.

Eine typische EyeSi-Szene besteht aus etwa 15.000 Polygonen, wovon 10.000 zur Modellierung von starren Objekten dienen, während 5.000 deformiert oder eingefärbt (Retina-Blutungen) werden. Für die starren Objekte werden NV-Vertex-Arrays verwendet; die Daten der veränderbaren Objekte stehen in GL-Vertex-Arrays, um einen schnellen Zugriff durch die CPU zu ermöglichen (Kapitel 6).

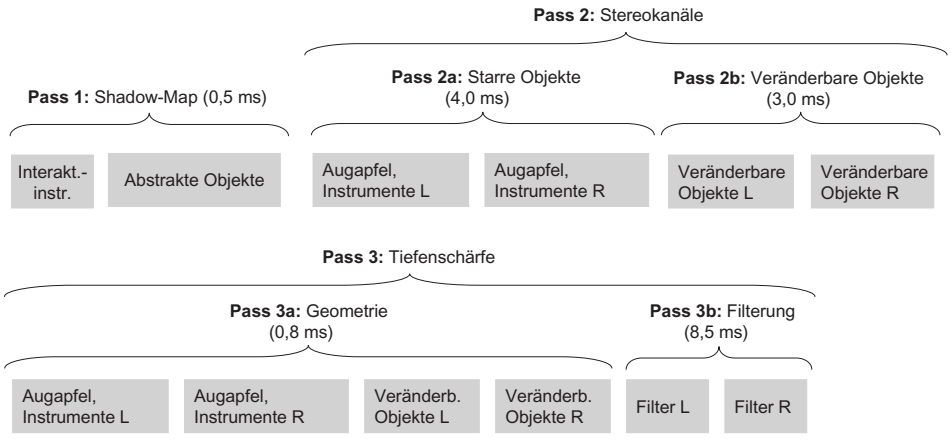


Abbildung 7.9: Renderingdurchgänge

EyeSi benötigt zum Rendern drei Durchgänge (Abb. 7.9): in Pass 1 wird die Shadow-Map für den Schattenwurf erzeugt; in Pass 2 wird die Szene in den (sichtbaren) linken und rechten Stereokanal gerendert, um dann in

Pass 3 mit dem Tiefenschärfe-Effekt versehen zu werden.

Um Zeit zu sparen und die unten beschriebene Parallelisierung zu erleichtern, werden in Pass 1 nur das Interaktionsinstrument sowie die Objekte eines abstrakten Szenarios gerendert. Auf die nicht berücksichtigten Objekte (veränderbare Objekte, Strukturen des Augapfels, Lichtquellen-Instrument) kann zwar ein Schatten fallen, sie werfen aber selbst keinen Schatten.

Für die Shadow-Map werden nur Vertex-Koordinaten übertragen und in den z-Buffer gerendert. Daher kann die theoretische Grenze für den Dreiecksdurchsatz (22 Millionen Dreiecke/s) voll ausgenutzt werden. Bei 10.000 Polygonen entsteht daraus eine Rendering-Zeit von weniger als 0,5ms.

Im zweiten Rendering-Durchgang wird in den linken und rechten Stereokanal gezeichnet. Dieser Durchgang ist aufgeteilt in die starren Objekte (Pass 2a) und die (zurzeit nur bei Operationsszenarios vorhandenen) deformierbaren Objekte (Pass 2b).

Neben den Vertex-Koordinaten (3 Floats/Vertex) werden Normalenvektoren (3 Floats/Vertex), Texturkoordinaten (2 Floats/Vertex) oder Farben (3 Bytes/Vertex), Bump-Map-Texturkoordinaten (2 Floats/Vertex) sowie Tangent-Space-Vektoren (ebenfalls für das Bump-Mapping; 3 Floats/Vertex) übertragen. Durch das erhöhte Datenaufkommen und die Ausführung von Vertex- und Pixelshader-Programmen sinkt der Dreiecksdurchsatz auf 5 Millionen Dreiecke/s (NV-Vertex-Arrays) bzw. 3,3 Millionen Dreiecke/s (GL-Vertex-Arrays). Für das Rendering der starren Objekte in Pass 2 werden 4ms benötigt, für die deformierbaren Objekte 3ms.

Zur Erzeugung der Unschärfe in Pass 3 benötigt man zunächst einen zusätzlichen Geometrie-Rendering-Schritt (Pass 3a), um die Tiefeninformation zu erzeugen (0,8ms); die tiefenabhängige Unschärfe-Filterung (Pass 3b) muss pixelweise z-Buffer- und Textur-Lookups durchführen. Dieser Schritt hat daher bei der Stereodisplay-Auflösung von $2 \times 800 \times 600$ Pixeln den höchsten Zeitbedarf aller Renderingschritte (8,5ms).

Die EyeSi-Sensorik (S) sowie Teile des Renderings (R) können von den jeweiligen Subsystemen (optisches Tracking, Grafikkarte) ausgeführt werden, ohne Systemlast zu erzeugen. Die freie CPU-Zeit wird für Berechnungen genutzt. Die Anordnung dieser parallelen Prozesse in EyeSi ist in Abb. 7.10 dargestellt. Sie basiert auf den in Kapitel 2 vorgestellten Synchronisationsmustern „parallel, pipelined“ und „parallel, interleaved“.

Eine bestimmte Handlung H der Benutzerin wird zunächst von den Kameras aufgenommen und an den PC übertragen (S_0 in Abb. 7.10). Mit diesen Werten wird die Berechnung B_0 der Gewebereaktion (die „Veränderung der Realität $\dot{R}$“ aus Kapitel 1) gestartet. Für die Berechnung eines expliziten

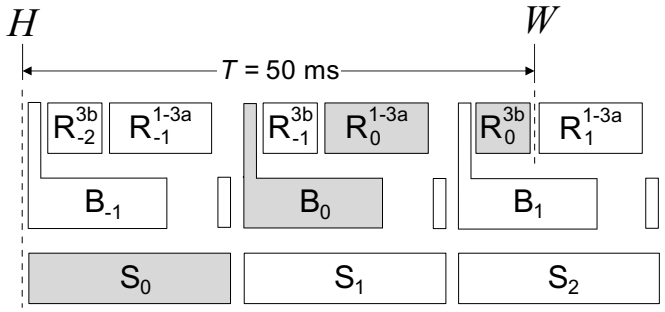


Abbildung 7.10: Aus Rechenzeit und Parallelisierung resultierende Latenz zwischen einer Handlung H und ihrer Wahrnehmung W. Die zu H gehörenden Verarbeitungsschritte von Sensorik, Berechnung und Rendering sind mit dem Index 0 versehen und grau eingefärbt.

oder statischen Euler-Schrittes mit Längenkorrektur (Kapitel 5) werden bei einem Gitter mit 1.500 Federn 2,0ms benötigt.

B_0 führt typischerweise zunächst eine bildbasierte Kollisionserkennung durch, für die Rechenkapazität des Rendering-Prozesses zur Verfügung gestellt werden muss (in Abb. 7.10 wird dies durch die nach oben weisende „Nase“ der B_i symbolisiert). Die anschließend ausgeführten Operationen (geometrische Kollisionserkennung, Feder-Masse-Schritte, Starre-Körper-Simulation) werden von der CPU durchgeführt, so dass parallel gerendert werden kann.

Der aufwändigste Rendering-Schritt ist die Unschärfefilterung des Tiefenschärfe-Effekts (Rendering-Pass 3b), die parallel zu einem Berechnungsschritt ausgeführt werden kann. Für das Unschärfe-Rendering muss die Szene aber schon vollständig berechnet und in die Stereokanäle gerendert sein. Daher wird Pipeline-parallelisiert: während der Ausführung von B_0 wird die Unschärfefilterung R_{-1}^{3b} auf den Ergebnissen der letzten Berechnung (B_{-1}) und des letzten Renderings (R_{-1}^{1-3a}) durchgeführt.

Im Schritt R_0^{1-3a} können die starren Objekte (Instrumente, Auge) schon während der Ausführung von B_0 gerendert werden, die dynamischen Objekte erst nach dem Ergebnis der Simulation.

In Abb. 7.11 sind die Laufzeiten der parallel und sequentiell ausgeführten Schritte dargestellt. Zur bildbasierten Kollisionserkennung wird ZCOLL/F z.B. für zwei verschiedene Ansichten des Operationsinstruments ausgeführt. „Misc“ enthält nicht im einzelnen aufgeführte Aufgaben der EyeSi-Software wie z.B. die Steuerung der GUI (einschließlich der Recorder/Player-Funktionalität) sowie die nicht-parallelisierten Grafikoperationen *buffer swap* und *buffer clear*.

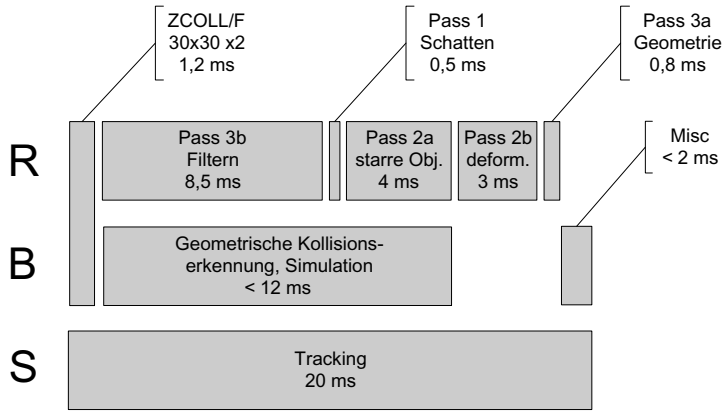


Abbildung 7.11: Zeitbedarf bei der parallelen Ausführung von (S)ensorauswertung, (B)erechnung und (R)endering (Ausschnitt aus Abb. 7.10).

Bisher noch nicht berücksichtigt wurde die Wiederholrate des Stereo-Displays von 85Hz. Da bei dem eingesetzten Display kein VSync notwendig ist, entsteht keine Synchronisationslatenz. Die Displaylatenz T_D entspricht im durchschnittlichen Fall der halben Zeit, die für den Bildaufbau benötigt wird: $T_D = 5,9\text{ms}$. Mit der Sensorlatenz T_S, der Zeit T_Z zur Ausführung von ZCOLL/F und der für die Unschärfefilterung im Rendering-Schritt R^{3b} benötigten Zeit T_{R3b} erhält man die EyeSi-Gesamtlatenz T wie folgt:

$$T = 2 \cdot T_S + T_Z + T_{R3b} + T_D \tag{7.1}$$

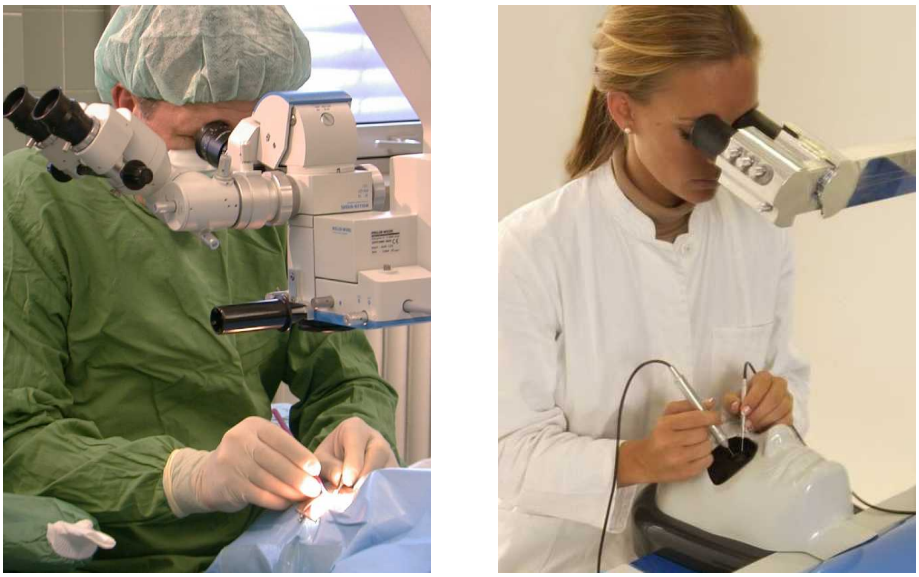
Für das hier diskutierte Beispielszenario (15.000 Dreiecke, von denen 5.000 veränderbar sind; eine Feder-Masse-Membran mit 1.500 Knoten) erhält man $T = 56,9\text{ms}$. Diese Werte entsprechen dem Operationsszenario „Membranentfernung“.

Da die abstrakten Szenarios nicht mit deformierbaren Objekten arbeiten, wird hier für B und R weniger Zeit benötigt. Bis auf das Schlüssel-Schloss-Szenario (Abb. 7.6(c)) ist für die Berechnung so wenig Zeit notwendig, dass die Unschärfefilterung R_0^{3b} noch vor Beenden von S_1 durchgeführt werden kann. Die Latenz verringt sich dadurch auf $T = 46,9ms$; der begrenzende Faktor ist hier nicht mehr Berechnung oder Rendering, sondern die Wiederholrate des Tracking-Systems.

Beim ILM-Peeling, dem zurzeit rechenintensivsten Operationsszenario sind für die Berechnung etwa 30ms notwendig, um das steifere Gewebe der ILM modellieren zu können. Das Tracking kann dann nur noch mit der halben Frequenz arbeiten ($T_S = 40\text{ms}$), so dass sich eine Gesamtlatenz von etwa

90ms ergibt. Durch die langsamen Bewegungen des Chirurgen ist dieser Wert noch tolerierbar, schränkt aber die Empfindung der Präsenz bereits ein. Er wird in Zukunft durch eine Optimierung der verwendeten Algorithmen sowie durch den Einsatz leistungsfähigerer PC-Hardware verringert werden.

7.5 Zusammenfassung und Diskussion



(a) (b)

Abbildung 7.12: OP-Szene und virtuelle Realität

Im EyeSi-System sind viele der in dieser Arbeit beschriebenen Ansätze realisiert. Dazu gehören die grundsätzlichen Überlegungen zu Präsenz und Immersion, die in Kapitel 1 angestellt wurden: wie in Abb. 7.12 dargestellt, wurden alle relevanten Aspekte der realen Situation im Simulator nachgebildet: die Auszubildende in Abb. 7.12(b) arbeitet mit Nachbildungen realer Instrumente; die mechanischen Parameter des Modellauges entsprechen denen des realen Vorbilds. Die Geometrie der Gesichtsmaske unterstützt oder behindert die Bewegungen des Chirurgen ebenso wie das Gesicht des Patienten.

Immersion ist gegeben, solange die Auszubildende durch die Okulare der Stereomikroskop-Nachbildung im Simulator blickt. Mit Hilfe dreidimensionaler Oberflächengrafik, deskriptiver Modellierung des Gewebeverhaltens und einer konsequent auf Echtzeit-Verarbeitung ausgerichteten Softwarear-

chitektur verhält sich das System hinreichend realistisch, um auch erfahrenen Chirurgen das Gefühl für Präsenz zu geben. In einer unabhängigen medizinischen Veröffentlichung [Koch und Lucke 2003] wird dieser Effekt wie folgt beschrieben:

> *Results:* Using the simulator we found it amazing how quickly the fact is forgotten that one is dealing only with a simulation. Injuring the retina results in truly felt anxiety. First practical tests by experienced surgeons and novices showed quite an astonishing difference in the level of performance in the various test modules. The performance of the students, moreover, improved within a few hours of training.

Die Metafunktionen des EyeSi-Systems werden über den Touchscreen bedient, mit dem auch die Instrumentsteuerung sowie die Konfiguration des Fußpedals und des Mikroskops erfolgt. Da in einer realen Operationssituation ebenfalls häufig Touchscreens für die Steuerung der Geräte verwendet werden, ist dieses Interface eine Mischung aus Realität und Virtualität – der Eindruck der Präsenz bleibt dabei erhalten, obwohl die Immersion nicht permanent gegeben ist.

Die Echtzeitforderung an eine VR-Applikation ist auch im EyeSi-System der bestimmende Faktor bei der Softwareentwicklung: Durch die Verwendung von Näherungsverfahren bei der Kollisionserkennung, der Beschränkung auf deskriptive Simulationsverfahren und Oberflächengrafik konnte die Latenz des Systems auf 50-90ms beschränkt werden. Bei einer Konfiguration an der unteren Grenze von 50ms haben optisches Tracking, Simulation und Visualisierung etwa den gleichen Anteil am Zeitverhalten; die obere Grenze wird erreicht, wenn für die Simulation von außergewöhnlich steifem Gewebe oder komplexen Geometrien mit aufwändiger Kollisionserkennung mehr Rechenzeit benötigt wird.

Die zukünftigen Arbeiten an EyeSi werden vor allem die Bandbreite der simulierten Operationen erweitern. Der Rechenzeitbedarf für Visualisierung und Tracking wird dabei voraussichtlich kaum steigen, dagegen wird die Komplexität der Interaktionen zunehmen. Der größte Entwicklungsaufwand wird daher darin bestehen, die für die Gewebesimulation eingesetzten Algorithmen zu erweitern und zu beschleunigen.

EyeSi zeigt, dass die in dieser Arbeit vorgestellten Techniken den Aufbau eines leistungsfähigen VR-Systems ermöglichen. In der virtuellen Realität von EyeSi können chirurgische Ausbildung, Operationsvorbereitung und die Entwicklung neuer Operationsmethoden und -instrumente durchgeführt werden (Abb. 7.13). Das System ist nicht nur ein *proof of concept*, sondern ein marktreifes Produkt.

Abbildung 7.13: EyeSi im Wetlab: Gegenwart und Zukunft der chirurgischen Ausbildung.

8

Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt Strukturen und Algorithmen, mit deren Hilfe virtuelle Realitäten für die chirurgische Ausbildung aufgebaut werden können. Anwendungsbeispiel und konkretes Ziel war die Entwicklung der Software des Augenoperationssimulators EyeSi; alle dabei entstandenen Verfahren wurden zu Modulen der Softwarebibliothek *VRM* für virtuelle Realitäten in der Medizin verallgemeinert. Die Darstellung in der Arbeit richtet sich in ihrer Allgemeinheit nach dieser Implementierung.

Neben neuen Ansätzen bei der Suche nach einer geeigneten Datenstruktur, der Kollisionserkennung, der graphischen Darstellung und der Simulation liegt ein Hauptbeitrag der Arbeit in der Kombination einer Fülle von Detaillösungen zu einer vielseitigen Softwarebibliothek und einer realistischen Trainingssimulation.

8.1 Strukturen

Die VRM-Softwarebibliothek integriert Verfahren für die unterschiedlichen Aufgaben einer virtuellen Realität – von der Hardware-Ansteuerung der VR-Schnittstellen bis zur Weichgewebesimulation. Sie stellt Daten- und Kommunikationsstrukturen zwischen den einzelnen Modulen zur Verfügung.

Eine wichtige Rolle für die Zusammenarbeit der Simulations- und Renderingmodule der VRM-Bibliothek spielt die Datenstruktur, die für deformierbare Objekte verwendet wird. Sie beeinflusst maßgeblich das Zeitver-

halten der Algorithmen sowie die Flexibilität, Geschwindigkeit und Fehleranfälligkeit der Softwareentwicklung.

Für die VRM-Bibliothek wurde ein graphbasiertes Repräsentationsformat entwickelt, die *MGE-Datenstruktur*. Die Graphstruktur wird in knotenzentrierten Adjazenzlisten gespeichert. Jeder Knoten kann eine beliebige Anzahl von Adjazenzlisten besitzen, die über symbolische Bezeichner referenziert werden.

Die MGE-Datenstruktur verwendet eine spezielle eigene Speicherverwaltung, die es ermöglicht, bestimmte Attribute der einzelnen Graphknoten in einem zusammenhängenden Speicherbereich abzulegen, was einen schnellen sequentiellen Zugriff ermöglicht (beispielsweise für die blockweise Übertragung von Geometriedaten auf die Grafikkarte).

Um dem Applikationsprogrammierer eine leichte und fehlerfreie Benutzung einer MGE-Struktur zu ermöglichen, kann ein bestimmter Ausschnitt eines Graphen mit Hilfe einer dünnen Schicht gekapselt werden, die die Typsicherheit garantiert und den Zugriff auf die entsprechende Substruktur einschränkt.

Ausdrucksmächtigkeit und Performance der MGE-Datenstruktur übertreffen die Möglichkeiten der sonst bei VR-Systemen üblichen Szenegraphen. Die Informationen über die gesamte virtuelle Welt einer VRM-Applikation werden daher in einer MGE-Datenstruktur abgelegt. Dazu gehören die inneren Strukturen von deformierbaren Objekten und die szenegraphähnlichen, geometrischen Beziehungen der Objekte untereinander. Durch diese Vereinheitlichung entfällt der Synchronisationsaufwand unterschiedlicher Datenformate. Die knotenzentrierte Speicherung der Information erlaubt darüberhinaus schnelle lokale Änderungen, wie sie beim Schneiden von Gewebe oder bei der lokalen Gitterverfeinerung auftreten.

Die Software einer VR-Applikation muss eine Reihe von verschiedenen Aufgaben erfüllen. Einige davon sind VR-spezifisch – Auslesen der Sensorik, Berechnung der Vorgänge in der virtuellen Welt, Ansteuerung der Anzeigegeräte –, wichtig sind aber auch „klassische“ Funktionen wie die Benutzerverwaltung, die Steuerung des Systems mit Hilfe einer grafischen Benutzeroberfläche oder die Aufzeichnung und Wiedergabe von Trainingssitzungen.

Es wird eine Architektur vorgeschlagen, bei der eine starke logische und softwaretechnische Trennung der Komponenten Sensorik, Verwaltung/Steuerung und VR-Berechnungen (Simulation/Rendering) durchgeführt wird. Die Softwareentwicklung einer *VRM-Applikation* kann dadurch in unabhängigen Teilprojekten erfolgen, was die Entwicklungsgeschwindigkeit erhöht, Komplexität und Fehleranfälligkeit dagegen reduziert.

Zur zeitlichen Steuerung einer VRM-Applikation wurden Klassen entwickelt, mit denen verschiedene Synchronisationsmuster nebenläufiger Pro-

zesse spezifiziert und ausgeführt werden können. Für die Grundmuster (sequentiell, parallel-asynchron, parallel-pipelined, parallel-interleaved) werden Datendurchsatz und Latenz diskutiert.

Messungen des zeitlichen Verhaltens der VRM-Ablaufsteuerung unter Linux und Windows XP zeigen, dass die beiden Betriebssysteme gut für den Einsatz einer VR-Applikation geeignet sind – allerdings nur dann, wenn das Betriebssystem keine Kontextwechsel zwischen verschiedenen Threads vornehmen muss. Anhand der Messungen wird demonstriert, dass der zeitliche Aufwand für solcher Kontextwechsel zu groß, ihre Frequenz dagegen zu gering ist.

Aufgrund dieser Ergebnisse wurden VRM-Applikationen so konzipiert, dass alle Aufgaben innerhalb eines Threads bearbeitet werden; Nebenläufigkeit tritt nur dann auf, wenn selbstständig arbeitende Hardware-Einheiten (Beispiele: FPGA-Bildverarbeitung eines optischen Trackingsystems, Grafikkarte) verwendet werden. Soweit möglich, werden diese Einheiten mit der Applikation synchronisiert. Die Latenz von EyeSi konnte durch diese Vorgehensweise erheblich verringert werden.

8.2 Algorithmen

Eine besondere Anforderung an chirurgische Simulationen ist die Berechnung von Gewebeinteraktionen. Es werden zwei Klassen von Algorithmen diskutiert – Verfahren zur Kollisionserkennung und zur Gewebedeformation. Eine vermittelnde Rolle spielt die *Kollisionsantwort*. Ausgehend von der Information eines Verfahrens zur Kollisionserkennung legt sie die Randbedingungen für eine bestimmte Deformation fest.

In der VRM-Bibliothek sind geometrische Verfahren für einfache Objekte (Ebene, Quader, Kugel, Zylinder) und Fassaden für zwei hüllkörperhierarchiebasierte Bibliotheken (RAPID und SOLID) implementiert. Da Hüllkörper-Hierarchien bei jeder Objektdeformation angepasst oder neu aufgebaut werden müssen, wurde das bildbasierte Verfahren ZCOLL entwickelt, das ohne Vorverarbeitung der Objektgeometrie auskommt.

ZCOLL basiert auf dem Vorschlag von Myszkowski et al. [1995], der mit Hilfe von z-und Stencilbuffer-Operationen von 3D-Grafikkarten feststellt, ob die rasterisierten Darstellungen zweier Objekte überlappen. Es ist speziell an die Interaktion zwischen einem starren Operationsinstrument und einem deformierbaren Gewebestück angepasst und in Bezug auf die Geschwindigkeit optimiert.

Es wird gezeigt, dass für den Fall langsamer Instrumentbewegung und schwach gefalteter Gewebestrukturen ein lokales Konvexitätskriterium

erfüllt ist, mit dessen Hilfe ein Rendering-Schritt des Verfahrens von Myszkowski et al. eingespart werden kann. Durch zusätzliche Bedingungen an die Interaktions- und Deformationsrichtungen entfällt ein weiterer Rendering-Schritt. ZCOLL nutzt die z- und Color-Buffer-Einträge für die Berechnung einer Kollisionsantwort. Deformationsvektoren werden über die z-Einträge ermittelt, kollidierende Polygone über ihre Farben identifiziert.

Im Gegensatz zu Hüllkörper-Hierarchien benötigt ZCOLL keine Vorverarbeitung der Objektgeometrie. Sein Zeitverhalten ist unabhängig von Objektdeformationen und der Schwere einer Kollision. Die Geschwindigkeit von ZCOLL ist vergleichbar mit dem besten Fall (nicht deformierbare Objekte, leichte Kollisionen) der hüllkörper-hierarchiebasierten Standardbibliothek RAPID.

Die Genauigkeit eines bildbasierten Verfahrens nimmt mit der Auflösung beim Rendern zu; hohe Auflösungen verlangsamen aber die Berechnung der Kollisionsantwort. Bei hohen Genauigkeitsanforderungen verliert man daher den Geschwindigkeitsvorteil gegenüber geometriebasierten Verfahren. ZCOLL ist für VRM-Applikationen einsetzbar, da solche Genauigkeiten in den üblichen Situationen einer Chirurgie-Simulation nicht erreicht werden müssen.

Zur Modellierung von Gewebe stellt die VRM-Bibliothek FEM-Verfahren, ChainMail- und Feder-Masse-Modelle zur Verfügung. In dieser Arbeit werden Feder-Masse-Modelle untersucht, mit denen bei einer hohen Rechengeschwindigkeit eine große Bandbreite von Gewebeverhalten modelliert werden können.

Es werden verschiedene Integrationsmethoden von Feder-Masse-Modellen diskutiert. Aufgrund des geringen Aufwands für einen Iterationsschritt wird das explizite Euler-Verfahren bevorzugt. Seine geringe numerische Stabilität bei steifem Gewebe wird durch die Dehnungskorrektur von Provot [1995] gemildert. Zur schnelleren Ausbreitung von Störungen wird die Dehnungskorrektor mit einer Feder-Sortierung verbunden.

Um unrealistische Oszillationen bei der Kollisionsantwort zu vermeiden, wird ein neues Modell für „Berühren“ beschrieben, bei dem Kollisionserkennung, -antwort und Gewebesimulation so angepasst sind, dass Gewebeteile nicht mehrfach in das Instrument eindringen.

Zur lokalen Gitterverfeinerung wird ein einfaches Verfahren vorgestellt, das kollidierende Dreiecke durch eine feinere Struktur fester Topologie ersetzt.

Es wird auf verschiedene Verfahren eingegangen, mit deren Hilfe Realismus und Geschwindigkeit der verwendeten Oberflächengrafik gesteigert werden kann. Besonders zu erwähnen sind zwei Eigenschaften moderner Grafikhardware: durch die Ausnutzung von DMA-Funktionen wird die grafische Objektbeschreibung ohne CPU-Last übertragen. Mit programmier-

baren Grafik-Subeinheiten (Vertex- und Pixel-Shader) werden Schatten- und Unschärfeeffekte erzeugt und feine Oberflächenstrukturen visualisiert.

8.3 EyeSi

EyeSi ist eine virtuelle Realität zum Training intraokularer Operationen. Die EyeSi-Hardware bildet alle wesentlichen Aspekte einer realen Operation nach. Anstelle eines Stereomikroskops werden kleine OLED-Displays eingesetzt, die über Okulare betrachtet werden. Der auszubildende Chirurg arbeitet in einem kardanisch gelagerten Metallauge mit originalgetreuen Instrumenten. Die Bewegung des Auges und der Instrumente wird mit einem optischen Trackingsystem gemessen. Die Bildverarbeitung erfolgt mit einer in der Arbeitsgruppe entwickelten FPGA-Hardware. Zur Gewebesimulation und Visualisierung wird ein handelsüblicher PC eingesetzt, der auch die Verwaltungsaufgaben des Systems übernimmt – Benutzerverwaltung, GUI-Steuerung über einen Touchscreen, Kontrolle, Auswertung und Aufzeichnung von Trainingsläufen.

Die gesamte Software des Systems wurde im Rahmen dieser Arbeit entwickelt. Algorithmik und Architektur basieren auf den hier vorgestellten und in der VRM-Bibliothek implementierten Ansätzen. Zurzeit können mit EyeSi verschiedene Operationen (Entfernung von intraokularen Gewebewucherungen; Entfernung der inneren Laminarmembran; Retina-Relokation) mit den Instrumenten Nadel, Pinzette, Vitrektom und Lichtquelle durchgeführt werden. Darüberhinaus stehen abstrakte Aufgaben (Beispiele: Nachfahren einer Trajektorie, Manipulation geometrischer Objekte) zur Verfügung, mit denen isolierte Fertigkeiten trainiert werden können – die Navigation im Auge, die ruhige und gezielte Bewegung der Instrumente oder die Arbeit in unmittelbarer Nähe zur Netzhaut.

Durch die Auslagerung von Renderingaufgaben (auf die Grafikkarte) und der Bildverarbeitung (auf die FPGA-Hardware) konnte die CPU soweit entlastet werden, dass das System auch Simulationen mit deformierbarem Gewebe mit 25-50Hz Updaterate und 54-74ms Latenz ausführen kann. Plattform des Systems ist dabei moderne PC-Standardhardware.

Durch die hohe Geschwindigkeit der Algorithmen, realistische Gewebesimulation und Visualisierung sowie das aufwändige VR-Interface von EyeSi ist eine überzeugende virtuelle Realität für die Ophthalmochirurgie entstanden. Das System wurde von der VRmagic GmbH zur Marktreife entwickelt und wird ständig erweitert.

8.4 Ausblick

Die größte Bedeutung für virtuelle Realitäten in der chirurgischen Ausbildung hat die Simulation der Vorgänge bei einer Operation und die Entwicklung realistischer VR-Interfaces. Auf beiden Gebieten wird die VRM-Bibliothek erweitert.

Gegenstand der aktuellen Arbeit ist die Simulation und Visualisierung von Flüssigkeiten, das Nähen von Gewebe sowie die Berechnung des Verhaltens besonders steifer Gewebearten. An Bedeutung gewinnen wird die Kombination von Simulationsalgorithmen und Animationstechniken, so dass an schwer zu simulierenden Stellen einer Interaktion vordefinierte Verhaltensmuster abgespielt werden können. Auf der anderen Seite – der physikalischen Modellierung – wird es darum gehen, Gewebeparameter tatsächlich zu messen, anstatt sie iterativ mit Hilfe eines „Reale-Welt-Experten“ zu bestimmen.

Geplant ist die Kopplung mehrerer VR-Simulatoren über ein Netzwerk, um Unterrichten aus der Distanz zu ermöglichen.

EyeSi wird um weitere Operationen im hinteren Augenabschnitt erweitert; zurzeit wird ein Modul für Operationen im vorderen Augenabschnitt entwickelt. Ebenfalls in Entwicklung ist eine Multimedia-Komponente, die das Operationstraining mit Video- und Textbeiträgen unterstützt.

Auf der Basis der VRM-Bibliothek werden weitere Traininigssimulatoren entwickelt, wie z.B. der Endoskopie-Simulator EndoSim, der über ein aufwändiges Force-Feedback-Interface verfügen wird.

— Solange die Eindrücke eines Menschen in einer virtuellen Realität mit Hilfe seiner Sinnesorgane und nicht durch direkte Ansteuerung seiner Nerven erzeugt werden, wird es keine universelle virtuelle Realität für Operationen, keinen „OP-in-der-CAVE“ geben – mechanische Randbedingungen und die Vielzahl der denkbaren Operationen verhindern so etwas. Spezialisierte Simulatoren wie EyeSi werden aber in naher Zukunft wesentlicher Bestandteil der chirurgischen Ausbildung sein.

Abbildung 8.1: Algorithmischer Aufwand schafft eine virtuelle Welt.

A

VR-Bibliotheken

In den folgenden Abschnitten werden einige Softwarebibliotheken zur Erstellung virtueller Realitäten vorgestellt. Weitere Übersichten geben auch Bierbaum und Just [1998] sowie Thalmann und Thalmann [1998]. Dieser Anhang wurde der Übersichtlichkeit halber aus dem Stand-der-Forschung-Abschnitt von Kapitel 2 ausgelagert.

Performer

Wie OpenInventor [Wernecke und Open Inventor Architecture Group 1994] ist (IRIS) Performer [Rohlf und Helman 1994] eine kommerzielle Bibliothek, die für Echtzeit-Grafiksysteme von Silicon Graphics (SGI) entwickelt wurde und mit Hilfe von Szenegraphen von low-level-Grafikfunktionen abstrahiert. Während OpenInventor im Prinzip auf jedem OpenGL-System implementiert werden kann, wurde Perfomer speziell an die SGI-Workstation-Architektur angepasst, um dort maximale Performance zu erzielen. Seit kurzem existiert auch eine Implementierung für PCs unter Linux.

Obwohl Perfomer keine VR-Bibliothek ist (dafür fehlt die Unterstützung von VR-Ein-und Ausgabegeräten), bietet er einige Funktionen, die über eine reine Grafikbibliothek hinausgehen und bei der Entwicklung von VR-Systemen nützlich sind. Neben der Erkennung von Kollisionen zwischen Liniensegmenten und Polygonen gehört dazu vor allem die Verwaltung von Mehrprozessorsystemen und die zeitliche Steuerung von Prozessen. Perfomer kann Prozesse kontrolliert auf verschiedene Prozessoren verteilen und somit Datendurchsatz und Latenz kontrollieren; es ist möglich, auf die Video-Ausgabe zu synchronisieren und feste Bildwiederholraten mit einem Level-of-detail-Mechanismus zu garantieren.

Es gibt einige VR-Bibliotheken (z.B. Avango [Tramberend 1999]), die auf Performer aufbauen.

MR Toolkit

Das MR (Minimal Reality) Toolkit [Shaw et al. 1993] stellt VR-Funktionalität auf niederer Ebene in Form von *Packages* zur Verfügung: Ein- und Ausgabegeräte werden durch ein client-server-Treibermodell unterstützt (Treiber für eine Vielzahl von Datenhandschuhen, Shutterbrillen und HMDs sind Teil der Bibliothek). Verteiltes Rechnen kann durch ein auf TCP/IP aufsetzendes data-sharing realisiert werden, wobei eine strikte Trennung zwischen genau einem Master-Prozess und den von ihm gestarteten Slave-Prozessen gemacht wird. Der Master-Prozess ist typischerweise für die Interaktion zuständig, während die Slave-Prozesse die Ein- und Ausgabe regeln – verteilte Umgebungen werden also nicht direkt unterstützt.

Die *Workspace Mapping Package* beschäftigt sich mit den geometrischen Berechnungen, die die Tracking-Daten und die Darstellungen in CAVEs, HMDs und dreidimensionalen Soundgeneratoren in einem gemeinsamen Koordinatensystem registrieren.

Ein wichtiges architektonisches Merkmal ist das *Decoupled Simulation Model*, das die Berechnung einer Simulation unabhängig von der Darstellung der virtuellen Umgebung durchführt. In einer engen Schleife zwischen Sensorik und Darstellung bewegt sich der Benutzer solange in einer quasi-statischen Welt, bis die übergeordnete Schleife Sensorik-Simulation-Modellupdate der Darstellung neue Daten zur Verfügung stellt.

Das MR Toolkit besitzt eine eigene Skriptsprache (*Object Modeling Language OML*); es existiert ein 3D-Modellierer (JDCAD+), der Szenenbeschreibungen in dieser Sprache erzeugt.

Die Entwicklung des MR Toolkits ist inzwischen eingestellt worden.

VRJuggler

VRJuggler [Bierbaum und Just 1998] wird an der Iowa State Universität entwickelt. Eines der Hauptziele ist eine möglichst flexible Hardware-Abstraktion. VRJuggler ist skalierbar von einer klassischen Monitor-Tastatur-Maus-Anwendung bis zur Mehrschirm-Projektion in einer CAVE. Mit dem *Environment Manager* soll es sogar möglich sein, zur Laufzeit eines Systems neue I/O-Geräte hinzuzufügen, zu entfernen oder neu zu starten. VRJuggler bietet Simulationen für jede Klasse von Eingabegeräten an, so dass Systeme getestet werden können, ohne dass die Zielkonfiguration verfügbar ist.

Dagegen stellt VR Juggler keine Funktionalität für Interaktionen zur Verfügung: „There is no event model and no built-in way to associate behaviors with graphic elements. These types of features could be supported in a higher level API running on top of VR Juggler.“ [Bierbaum und Just 1998]

VR Juggler setzt auf OpenGL oder Performer auf, ist aber so gebaut, dass die Grafik-API ausgetauscht werden kann.

Panda3D

Panda3D (*Platform Agnostic Networked Display Architecture*) wird von der Walt Disney Company vor allem für verteilte VR-Anwendungen entwickelt, ist aber ein Open-Source-Projekt.[1]

Mit Panda3D können allgemeine Graphstrukturen repräsentiert werden, so dass mit den Objekten einer Szene mehrere Szenegraphen (z.B. zur Repräsentation der sichtbaren Objekte) oder Datenflussgraphen definiert werden können.

Um eine möglichst große Portabilität zu gewährleisten, werden alle I/O-Funktionen gekapselt. Zum schnellen Entwickeln werden Skriptsprachen unterstützt. Panda3D kann aus seiner internen C++-Struktur aus Objekten, Methoden oder globalen Variablen eine Datenbank (*interrogate database*) erzeugen, auf die mit beliebigen Skriptsprachen zugegriffen werden kann. Bereits unterstützt wird die Open-Source-Smalltalk-Implementierung Squeak.[2]

Netzwerkkommunikation wird über einen Server für verteilte Objekte realisiert, der auf einer TCP/IP-Abstraktion beruht.

Alice

Der Schwerpunkt von Alice liegt auf der schnellen, einfachen Entwicklung von VR-Umgebungen. Alice-Programme werden in einer Skriptsprache (Python) geschrieben; ähnlich wie beim MR-Toolkit sind Simulation und Rendering entkoppelt, im Gegensatz zu diesem geschieht dies aber ohne Zutun des Programmierers.

Szenen werden mit einem Szenegraphen beschrieben, anders als bei Inventor- oder Performer-Szenegraphen ist die Semantik des Szenegraphen aber unabhängig von der Reihenfolge, in der er durchlaufen wird.

[1] www.etc.cmu.edu/projects/panda3d/

[2] www.squeak.org. Smalltalk [Goldberg und Robson 1985] ist eine frühe objektorientierte Sprache mit einem konsequenteren, aber nicht so flexiblen Aufbau wie C++ oder JAVA. Sie wird häufig interpretiert und eignet sich daher als Skriptsprache.

Avango

Avango [Tramberend 1999] erweitert den Szenegraphen von IRIS Performer um die Möglichkeit, Knoten über ein Netzwerk zu verteilen. Dazu werden sogenannte *Fields* definiert, die die Eigenschaften eines Objekts beschreiben (die Anzahl der Attribute, ihren Typ und ihren Wert). Fields besitzen ein generisches Streaming-Interface, mit dem ihre Daten per Netzwerk übertragen werden können. Durch Ableiten der Klassen der Performer-Klassenhierarchie werden diese mit der Field-Funktionalität ergänzt.

Die übertragenen Knoten bilden einen verteilten Shared-Memory-Bereich. Für die zugrundeliegende Netzwerkfunktionalität wird das Ensemble-System [Hayden 1998] verwendet, das vor allem die Konsistenz des verteilten Speichers sichert – z.B. bilden die verschickten Nachrichten eine Totalordnung, so dass sie jeden Empfänger in der gleichen Reihenfolge erreichen.[3]

Der Datenfluss im System wird durch *Field connections* realisiert, die ein *Source field* mit einem *Destination field* verbinden. Wenn sich das Source field ändert, wird eine Nachricht an das Destination field gesendet. Der daraus entstehende Datenflussgraph wird in jedem Zeitschritt abgearbeitet; Schleifen werden erkannt und aufgelöst. Sensoren werden durch Source fields modelliert.

WTK

WTK ist eine kommerzielle Bibliothek zur Entwicklung von VR-Applikationen, die von Engineering Animation, Inc. entwickelt und vertrieben wird.[4] Die Bibliothek stellt Funktionen zum Rendern, zum Geometrieimport, zum Ansteuern von I/O-Geräten und für Netzwerkfunktionen zur Verfügung. Zur Beschreibung von Szenen wird ein Szenegraph verwendet, der die Informationen über die Objektgeometrien und Visualisierungsparameter hält. Die Visualisierung selbst basiert auf OpenGL; direkte OpenGL-Befehle können durch spezielle Knoten im Szenegraphen („OpenGL Callback Nodes“) ausgeführt werden.

Eine Vielzahl von Sensoren sowie die Entwicklung eigener Sensortreiber wird unterstützt; die Sensorausgabe wird bestimten Knoten im Szenegraphen zugeordnet und verändert so die räumlichen Eigenschaften der Objekte.

Das WTK stellt Funktionen zur Berechnung von Kollisionen zwischen geometrischen Primitiven (Polygonen, Quadern und Geraden) zur Verfügung.

[3] Eine zentrale Veröffentlichung zu diesem Thema stammt von Leslie Lamport [1978], dem Entwickler von LaTeX.

[4] www.sense8.com

Simulationsalgorithmen sind im WTK nicht implementiert. Über Callback-Funktionen, die einzelnen Objekten oder der gesamten Szene zugeordnet sind, kann Funktionalität hinzugefügt werden.

Mit Hilfe einer Zusatzapplikation, dem WorldUp Modeller können virtuelle Objekte für das WTK erstellt werden.

Spring

Spring wird am National Biocomputation Center in Stanford als Bibliothek für Operationssimulationen entwickelt [Montgomery et al. 2002]. Spring stellt Algorithmen zur Simulation von Gewebe und starren Körpern zur Verfügung. Für die Gewebesimulation werden Feder-Masse-Modelle verwendet, während Nähte durch starre Segmente mit drehbaren Verbindungen modelliert werden [Brown et al. 2001]. Zur Visualisierung wird OpenGL-Oberflächengrafik verwendet; die Kollisionserkennung basiert auf geometrischen Ansätzen (vor allem mit sphärischen Bounding-Objekten).

Auf der Basis von abstrakten I/O-Klassen existieren Spezialisierungen für eine Vielzahl von I/O-Geräten, wie z.B. elektromagnetischen Tracking-Systemen oder dem Phantom. Um trotz PC-basierter I/O-Hardware systemunabhängig zu werden, wird ein Netzwerkmodul zur Verfügung gestellt, das mit einem externen Rechner kommuniziert, der ein bestimmtes I/O-Gerät steuert. Über das Netzwerkmodul kann ein *Voice server* angeschlossen werden, mit dessen Hilfe das VR-System durch Sprachbefehle gesteuert werden kann.

Eine Anzahl medizinischer Applikationen wurde mit Spring entwickelt oder ist in Entwicklung. Dazu gehört ein Hysteroskopie-Simulator, ein Simulator für die Sezierung von Ratten und ein Kolonoskopie-Simulator.

Spring unterstützt die Entwicklung auf Windows- und Unix-Plattformen und soll als freie Software zur Verfügung gestellt werden. Das ehrgeizige Ziel der Bibliothek ist, der Standard zur Entwicklung von Operationssimulationen zu werden.

KISMET

KISMET (Kinematic Simulation, Monitoring and Off-Line Programming Environment for Telerobotics) wurde am Forschungszentrum Karlsruhe ursprünglich für die kinematische Simulation und grafische Darstellung von Robotern entwickelt und um Funktionen für Operationssimulationen erweitert [Kühnapfel et al. 2000]. Gewebeverhalten kann mit Feder-Masse- und FEM-Modellen simuliert werden. Für die Modellierung deformierbarer Objekte steht eine eigene Applikation (KisMo) zur Verfügung, mit der die

Geometrie und die Simulationsparameter von Objekten spezifiziert werden können.

Mit KISMET wurde ein Endoskopie-Simulator[5] und ein Gynäkologie-Trainer entwickelt [Kühnapfel et al. 1995, Çakmak et al. 2002]. Die Entwicklung der Manipulatoren des Teleoperations-System ARTEMIS [Voges et al. 1997] wurde mit Hilfe von kinematischen KISMET-Simulationen durchgeführt.

Die Bibliothek wurde für SGI-Workstations unter IRIX und Windows NT entwickelt.

[5]Der *Karlsruhe Virtual Endoscopic Surgery Trainer* wurde inzwischen zu dem kommerziellen Produkt VEST (*Virtual Endoscopic Surgery Trainer*) weiterentwickelt (www.select-it.de).

B

Zeitbedarf der VRM-Ablaufsteuerung

Um den Zeitbedarf der VRM-Ablaufsteuerung zu messen, wird ein Ablaufsteuerungs-Modul mit einer leeren Process()-Methode in i Instanzen in einer Sequenz bzw. parallel in i Sequenzen ausgeführt.

Während einer Iteration führt die Ablaufsteuerung die PreProcess()-, Process()- und PostProcess()-Methoden aller Instanzen einmal aus, misst deren Zeiten und testet auf Abbruchbedingungen und Zeitüberschreitungen. Dieser Aufwand wird im folgenden ermittelt.

Es werden 1, 50, 200, 500 und 2000 Iterationen der Ablaufsteuerung durchgeführt und 1-5 Module instanziiert. Das Ergebnis der Zeitmessung ist in Abb. B.1, (1)-(3) dargestellt und in Abb. B.1, (4) zusammengefasst.

Bei den in Abb. B.1-(1) und (2) dargestellten Messungen wurden die Module in einer Sequenz ausgeführt, so dass kein Aufwand durch die Verwaltung mehrerer Threads entstand. In Abb. B.1-(3) wurde dagegen für jedes Modul eine eigene Sequenz instanziiert und asynchron ausgeführt. Um den Kontextwechsel-Aufwand messen zu können, wurde die Ausführung aller Sequenzen an einen Prozessor gebunden. Da dies unter Linux im Augenblick nur in der experimentellen Kernel-Version 2.5 möglich ist, die zum Zeitpunkt der Messung noch nicht zur Verfügung stand, konnte diese Messreihe nur unter Windows XP durchgeführt werden.

Wie zu erwarten, ergibt sich ein linearer Zusammenhang zwischen der Anzahl der Sequenziterationen und dem Zeitbedarf. Die Steigungen der Ausgleichsgeraden geben den Zeitbedarf pro Sequenziteration an, ihr y-Achsenabschnitt den zeitlichen Aufwand für Start und Stop der Ablaufsteuerung. Das Bestimmtheitsmaß (Quadrat des Korrelationskoeffizienten

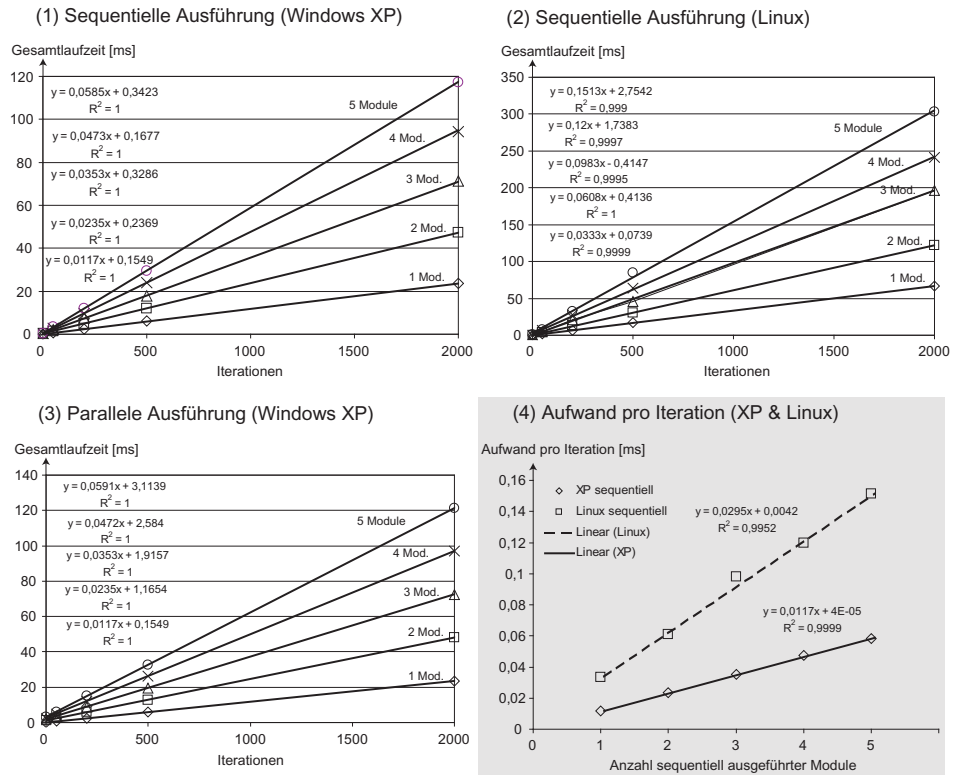


Abbildung B.1: Zeitbedarf der Ablaufsteuerung. Mehrere (1-5) Instanzen eines Ablaufsteuerungs-Moduls ohne Funktionalität wurden sequentiell (1,2) und parallel (3) ausgeführt. Der Aufwand pro Ablaufsteuerungs-Iteration in (4) liegt im 0,1ms-Bereich.

R) der Ausgleichsgeraden liegt in jeder Konfiguration nahe bei 1, wodurch lediglich ausgedrückt wird, dass das System bei den einzelnen Versuchen gleich konfiguriert war und nicht unter unterschiedlich hoher Last lief. R^2 ist *kein* Maß für die zeitliche Regelmäßigkeit, mit der die einzelnen Sequenziteration durchgeführt werden.

Abb. B.1-(4) trägt die Anzahl der Module gegen die Ausführungszeit pro Iteration (der Steigung der jeweiligen Ausgleichsgeraden) auf. Es ergibt sich auch hier ein linearer Zusammenhang, da die Ablaufsteuerung pro Iteration, Modul und Sequenz nur $O(1)$-Operationen durchführt. Für jedes Modul entsteht ein Zeitbedarf von 0,01ms (Windows XP) bzw. 0,03ms (Linux)[1], der im Vergleich zur typischen Laufzeit einer Iteration einer VR-Applikation von 1ms (Force-Feedback) bzw. 10-100ms (Simulation und Visualisierung) vernachlässigbar ist.

Bemerkenswert ist, dass die Steigungen der Ausgleichsgeraden im parallelen Fall in Abb. B.1-(3) nahezu identisch sind mit dem sequentiellen Fall in Abb. B.1-(3), d.h. der Aufwand für die Kontextwechsel war in den untersuchten Konfigurationen nicht messbar. Diese Beobachtung kann durch die in Abschnitt 2.5.4 ermittelte geringe Frequenz der Kontextwechsel erklärt werden.

Deutlich zu erkennen ist der zusätzliche Rechenaufwand im parallelen Fall dagegen in dem höheren Initialisierungs- und Deinitialisierungsaufwand, der durch das Starten der einzelnen Threads entsteht und sich in den hier im Millisekunden-Bereich liegenden y-Achsenabschnitten widerspiegelt.

[1] Die Gründe für den zeitlichen Unterschied zwischen Linux und Windows wurden nicht untersucht, da es hier nicht um einen Vergleich der beiden Systeme geht, sondern lediglich um die prinzipielle Aussage, ob die beiden Betriebssysteme für VR-Applikationen geeignet sind. Eine mögliche Erklärung sind die unterschiedlichen Compiler, die verwendet wurden.

Literaturverzeichnis

Peter Aczel. *Non-well-founded Sets*. Center for the Study of Language and Information - Lecture Notes, 1988. 46

Robert S. Allison, Laurence R. Harris, Michael Jenkin, Urszula Jasiobedzka, and James E. Zacher. Tolerance of temporal delay in virtual environments. In *IEEE Virtual Reality 2001 International Conference*, pages 247–254, 2001. 18

ATI Corporation. *ATI Technologies White Paper – Truform (TM)*, 2001. 125

George Baciu, Wingo S.-K. Wong, and Hanqiu Sun. RECODE: An image-based collision detection algorithm. In *Proceedings of Pacific Graphics*, 1998. 76

George Baciu and Wingo S.-K. Wong. Rendering in object interference detection on conventional graphics workstations. In *Proceedings of Pacific Graphics*, 1997. 76

Srikanth Bandi and Daniel Thalmann. An adaptive spatial subdivision of the object space for fast collision detection of animating rigid bodies. In *Proceedings of Eurographics*. Eurographics Association, Blackwell Publishers, 1995. 69

David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Computer Graphics (Proceedings SIGGRAPH)*, pages 43–54, 1998. 104, 105, 106, 107, 109

David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Proc. of SIGGRAPH 90*, Computer Graphics, pages 19–28. ACM, 1990. 64

Klaus-Jürgen Bathe. *Finite-Elemente-Methoden*. Springer, 1990. 98

Ekkehard Beier. AFML – an object-oriented 3D metafile format. In *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997. 46

Ekkehard Beier. Modellierung hierarchischer Strukturen in VR-Applikationen. In *4. Workshop 'Multimediale Informations- und Kommunikationssysteme', TU Ilmenau*, 1998. 46

Allen Bierbaum and Christopher Just. Software tools for virtual reality application development. *SIGGRAPH '98 Course 14: Applied Virtual Reality*, 1998. 155, 156, 157

Roland Blach, Jürgen Landauer, Angela Rösch, and Andreas Simon. A highly flexible virtual reality system. *Future Generation Computer Systems*, 14(3–4):167–178, 1998. 8

Ronald J. Brachmann and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985. 46

Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of SIGGRAPH*, 2002. 110, 111

William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial*. SIAM, 2nd edition, 2000. 111

Joel Brown, Stephen Sorkin, Jean-Claude Latombe, Kevin Montgomery, and Michael Stephanides. Algorithmic tools for real-time microsurgery simulation. *Lecture Notes in Computer Science*, 2208, 2001. 107, 108, 159

Simone Bürsner. *Situationsbezogene Wissensverarbeitung mit medizinischen Assistenzsystemen*. PhD thesis, Universität Kaiserslautern, 1997. 46

Stephen A. Cameron. A comparison of two fast algorithms for computing the distance between convex polyhedra. *IEEE Transactions on Robotics and Automation*, 13:915–920, December 1997. 65

Stephen A. Cameron. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3112–3117, April 1997. 65

Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Erlbaum, 1983. 15, 17

Hüseyin Çakmak, H. Maasz, G. Strauss, C. Trantakis, E. Nowatius, and U. Kühnapfel. Modellierung chirurgischer Simulationsszenarien für das Virtuelle Endoskopie Trainingssystem (VEST). In *Proceedings of CURAC, 1. Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e.V.*, 2002. 160

Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the ACM Interactive 3D Graphics Conference*, pages 189–196, 1995. 66, 68

S. Barry Cooper and Piergiorgio Odifreddi. Incomputability in nature. In S. Barry Cooper and S. S. Goncharov, editors, *Computability and Models: Perspectives East and West*. Kluwer Academic, 2003. Erscheinungsdatum des Buchs: 1. Februar 2003. In der Arbeit wurde eine elektronische Version des Artikels verwendet. 14

S. Cotin, H. Delingette, and N. Ayache. Real-time elastic deformations of soft tissuees for surgical simulation. *IEEE Transactions On Visualization and Computer Graphics*, 5(1):62–73, Jan–Mar 1999. 98

Michael Crichton. *Disclosure*. Ballantine Books, 1994.

Mathieu Desbrun, Mark Meyer, and Alan H. Barr. Interactive animation of cloth-like objects for virtual reality. *Journal of Visualisation and Computer Animation*, 2000. 103

David Deutsch. *The Fabric of Reality*. Penguin Books, 1997. 7, 13

Michael Doggett. Programmability features of graphics hardware. Technical report, ATI Technologies Inc., 2002. 120

Cass Everitt, Ashu Rege, and Cem Cebenoyan. Hardware shadow mapping. Technical report, NVidia, Erscheinungsdatum unbekannt. 122

Richard P. Feynman, Robert B. Leighton, and Matthew L. Sands. *The Feynman Lectures on Physics*, volume 1. Addison-Wesley, 1963. 60

James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice*. Addison Wesley, 2nd edition, 1996. 70, 117

Heinrich Freyler. *Augenheilkunde für Studium, Praktikum u. Praxis*. Springer, Wien, 1985. 128

Daniel F. Galouye. *Simulacron 3*. Heyne, 1983. (Erstausgabe in englischer Sprache unter dem Titel „Counterfeit World“, New York, 1964). 6

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 26

Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. Buckettree: Improving collision detection between deformable objects. In *Proceedings of the Spring Conference on Computer Graphics, Budmeria Castle, Bratislava*, May 2000. 60, 69

Christian Gerthsen, Hans O. Kneser, and Helmut Vogel. *Physik*. Springer, 1989. 103

Sarah F. F. Gibson. Beyond volume rendering: Visualization, haptic exploration, and physical modeling of voxel-based objects. In *6th Eurographics Workshop on Visualization in Scientific Computing*, May 1995. 61, 69

Sarah F.F. Gibson. 3d chainmail: a fast algorithm for deforming volumetric objects. In *Symposium on Interactive 3D Graphics*, pages 149–154, 1997. 99

Sarah F. F. Gibson. Using linked volumes to model object collision, deformation, cutting, carving and joining. Technical Report TR-2000-24, MERL – Mitsubishi Electric Research Laboratory, Cambridge, U.S.A., 2000. 61

Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, April 1988. 65

Gernot Goebbels, Nicolas Fournier, Martin Göbel, Herwig Zilken, Wolfgang Frings, Thomas Eickermann, and Stefan Posse. Remote visualization of radiological data on a responsive workbench. In *Proceedings of CARS*, 1999. 12

Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1985. 157

Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings SIGGRAPH '96*, pages 171–180, 1996. 67, 68, 92

Stefan Gottschalk. Separating axis theorem. Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996. 67

Arthur Gregory, Ming C. Lin, Stefan Gottschalk, and Russell Taylor. H-COLLIDE: A framework for fast and accurate collision detection for haptic interaction. In *VR*, pages 38–45, 1999. 69

Michael Haller. *Ein komponentenorientiertes Design für virtuelle Umgebungen*. PhD thesis, Universität Linz, 2000. 7

Michael Hauth and Olaf Etzmuss. A high performance solver for the animation of deformable objects using advanced numerical methods. In *Computer Graphics Forum (CGI)*, 2001. 107

Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998. 158

Taosong He. *Volumetric Virtual Environments*. PhD thesis, State University of New York at Stony Brook, May 1996. 61

Tim Heidmann. Real shadows, real time. *IRIS Universe*, 1991. 122

Martin Held, James T. Klosowski, and Joseph S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Proceedings of the Seventh Canadian Conference on Computer Graphics*, volume 3, pages 205–210, 1995. 67

R. Held and N. Durlach. Telepresence, time delay and adaption. In Stephen R. Ellis, editor, *Pictorial Communication in Virtual and Real Environments*. Taylor and Francis, 1991. 18

Marc Hennen. Entwicklung eines Prototypen zur Simulation einer Parsplana-Vitrektomie. Master's thesis, Universität Mannheim, August 1998.

Harro Heuser. *Lehrbuch der Analysis, Teil 2*. Teubner, 6th edition, 1991. 59

Taiichi Hikichi, Akitoshi Yoshida, Syo Igarashi, Nobuhiko Mukai, Masayuki Harade, Katsunobu Muroi, and Takafumi Terada. Vitreous surgery simulator. *Archives of Ophthalmology*, 118:1679–1681, 2000. 130

Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979. 13

Nick Holliman. 3D display systems. Technical report, University of Durham, Department of Computer Science, 2002. 119

Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996. 113

Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997. 113

Eric Horvitz and Jed Lengyel. Perception, attention, and resources: A decision-theoretic approach to graphics rendering. In *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence, Providence, RI, August 1997. Morgan Kaufmann: San Francisco, pp. 238-249*, pages 238–249, 1997. 113

Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996. 60, 67

Suejung Huh, Dimitris N. Metaxas, and Norman I. Badler. Collision resolutions in cloth simulation. In *Computer Animation IEEE 2001*, 2001. 110

Dave Hutchinson, Martin Preston, and Terry Hewitt. Adaptive refinement for mass/spring simulations. In *Proceedings of the 7th International Workshop on Computer Animation and Simulation*, 1996. 112, 114, 115

Marco C. Jacobs, Mark A. Livingston, and Andrei State. Managing latency in complex augmented reality systems. In *Proceedings of the Symposium on Interactive 3D Graphics*, 1997. 31

Pablo Jiménez, Frederico Thomas, and Carme Torras. 3D collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001. 63

Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison Wesley Longman, 1999. 51

Michael Kass. An introduction to physically based modeling: An introduction to continuum dynamics for computer graphics. SIGGRAPH '95 Course Notes, 1995. 103

Jan Kautz, Katja Daubert, and Hans-Peter Seidel. User-defined shading models for VR applications. Presented at the 1st OpenSG Symposium; electronically published at www.opensg.org., 2002. 119

Mark J. Kilgard. *OpenGL Programming for the X Window System.* Addison-Wesley Developers Press, 1996. 23

Frank H.J. Koch and Klaus Lucke. Training with a vitrectomy surgery simulator. In *Proceedings of the American Society of Retina Specialists Meeting*, page 76, August 2003. 144

Olaf Körner, Markus A. Schill, Clemens Wagner, Hans-Joachim Bender, and Reinhard Männer. Haptic volume rendering with an intermediate local representation. In R. Dillmanns and T. Salb, editors, *Proceedings of the 1st International Workshop on Haptic Devices in Medical Applications*, pages 79–84, 1999. 10, 23, 28

Olaf Körner and Reinhard Männer. Haptic display for a virtual reality simulator for flexible endoscopy. In S. Müller and W. Stürzlinger, editors, *Proceedings of the 8th Eurographics Workshop on Virtual Environments, Barcelona*, pages 13–18. ACM Press, May 2002. 24

Olaf Körner. Einsatz eines Force-Feedback-Gerätes in Virtuellen Realitäten. Master's thesis, University of Mannheim, 1999. 23

James Kuffner, Koichi Nishiwaki, Satoshi Kagami, Yasuo Kuniyoshi, Masayuki Inaba, and Hirochika Inoue. Self-collision detection and prevention for humanoid robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002. 76

Uwe Kühnapfel, H. Krumm, C. Kuhn, M. Hübner, and B. Neisius. Endosurgery simulations with kismet: A flexible tool for surgical instrument design, operation room planning and VR technology based abdominal surgery training. In *Proceedings VR'95 World Conference*, 1995. 160

U. Kühnapfel, H.K. Çakmak, and H. Maaß. Endoscopic surgery training using virtual reality and deformable tissue simulation. *Computers & Graphics 24*, pages 671–682, 2000. 159

Ray Kurzweil. Fine living in virtual reality. In Peter J. Denning, editor, *The Invisible Future: The Seamless Integration of Technology into Everyday Life*, pages 193–215. McGraw-Hill, Inc., 2001. 10

Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 158

Stanisław Lem. *Summa technologiae*. Kraków: Wydawnictwo Literackie, 1964. Dem Autor lag eine deutsche Übersetzung vor (Gleicher Titel, Suhrkamp, 2000). 6, 11, 15

Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 1008–1014, 1991. 64, 65

Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models: a survey. In *Proceedings of the IMA Conference on Mathematics of Surfaces*, 1998. 61, 63

Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proceedings SIGGRAPH)*, 2001. 120

Jean-Christophe Lombardo, Marie-Paule Cani, and Fabrice Neyret. Real-time collision detection for virtual surgery. In *Proceedings of Computer Animation*, pages 33–39, May 1999. 63, 81

Gordon D. Mallinson, Mark A. Sagar, David Bullivant, and Peter J. Hunter. A Virtual Environment and Model of the Eye for Surgical Simulation. In *Proc. of SIGGRAPH 94*, Annual conference series, pages 205–212, 1994. 130

Thomas H. Massie and J. K. Salisbury. The PHANToM haptic interface: A device for probing virtual objects. In *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 1994. 11

Tomasz Mazuryk and Michael Gervautz. Two-step prediction and image deflection for exact head tracking in virtual environments. In Frits Post

and Martin Göbel (Guest Editors), editors, *Proceedings of Eurographics*. Eurographics Association, Blackwell Publishers, 1995. 18

Sarah C. McQuaide. Three-dimensional virtual retinal display using a deformable membrane mirror. Master's thesis, University of Washington, 2002. 118

Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984. 64

Philippe Meseure, Jean-François Rouland, Patrick Dubois, Sylvain Karpf, and Christophe Chaillou. SOPHOCLE: A retinal laser photocoagulation simulator overview. In *Proceedings of CVRMed '95*, pages 105–114. Springer, 1995. 130

Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. Improved collision detection and response techniques for cloth animation. Technical report, WSI/GRIS, Universität Tübingen, 2002. 67, 109

A. Michotte. *The Perception of Causality*. Basic books, 1963. Wiederveröffentlichung, das französischsprachige Original stammt aus dem Jahre 1946 (La perception de la causalité, Publications Universitaires de Louvain). 17

Brian Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1998. 66

Kevin Montgomery, Cynthia Bruyns, Joel Brown, Stephen Sorkin, Frederic Mazzella, Guillaume Thonier, Arnaud Tellier, Benjamin Lerman, and Anil Menon. Spring: A general framework for collaborative, real-time surgical simulation. In *Medicine Meets Virtual Reality (MMVR02)*, 2002. 29, 159

Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11:497–511, 1995. 71, 73, 74, 75, 78, 81, 94, 149, 150

Paul F. Neumann, Lewis L. Sadler, and Jon Gieser M.D. Virtual Reality Vitrectomy Simulator. In Alan Colchester, William M. Wells, and Scott Delp, editors, *Medical Image Computing and Computer- Assisted Intervention – MICCAI '98*, pages 910–917, Cambridge, MA, USA, October 1998. Springer. 130

Nikolaj Nock. Aufbau eines für den Einsatz auf 'Robocup'-Robotern optimierten 360-Grad-Bildaufnahme- und -verarbeitungssystems. Master's thesis, Universität Mannheim, 2000. 23

Tor Nørretranders. *Spüre die Welt – die Wissenschaft des Bewußtseins*. Rowohlt, 1994. 9

Piergiorgio Odifreddi. *Classical recursion theory*. North-Holland, 1992. 13

Joseph O'Rourke. Finding minimal enclosing boxes. *International Journal of Computer and Information Sciences*, 14(3):183–199, 1985. 67

Markus F. Peschl and Alexander Riegler. Does representation need reality? In Alexander Riegler, Markus Peschl, and A. von Stein, editors, *Understanding Representation in Cognitive Science*. Kluwer Academic/Plenum Publishers, 1999. 8, 13

Alexandra Poulovassilis and Mark Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, 12(1):35–68, January 1994. 58

Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. *Graphics Interface*, pages 147–155, 1995. 105, 106, 107, 116, 150

Xavier Provot. Collision and self-collision handling in cloth model dedicated to design garments. *Graphics Interface '97*, pages 177–189, 1997. 77, 110

M. Ross Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968. 46

Sharif Razzaque, David Swapp, Mel Slater, Mary C. Whitton, and Anthony Steed. Redirected walking in place. In S. Müller and W. Stürzlinger, editors, *Proceedings of the 8th Eurographics Workshop on Virtual Environments, Barcelona*, pages 123–130. ACM, May 2002. 10

Michael M. Richter. *Prinzipien der Künstlichen Intelligenz. Wissensrepräsentation, Inferenz und Expertensysteme*. Reihe Leitfäden und Monographien der Informatik. Teubner, 1989. 46

Oliver H. Riedel. *3D-Echtzeit-Rendering unter Berücksichtigung der Anatomie und Physiologie des menschlichen Auges*. PhD thesis, Universität Stuttgart, 1997. 11

Cameron N. Riviere and Pradeep K. Khosla. Microscale tracking of surgical instrument motion. In Chris Taylor and Alan Colchester, editors, *Medical Image Computing and Computer- Assisted Intervention – MICCAI '99*, volume 1679 of *Lecture Notes in Computer Science*, pages 1080–1087, Cambridge; UK, September 1999. Springer. 135

John Rohlf and James Helman. Iris performer: A high-performance multiprocessing toolkit for real-time 3D graphics. In *Proceedings of SIGGRAPH 94*, pages 381–394. ACM Press, 1994. 36, 155

Thomas Ruf. Entwurf, Aufbau und Evaluierung eines FPGA-basierten Farbmarker-Trackings für den Augenoperations-Simulator EyeSi2. Master's thesis, Universität Mannheim, 2000. 23, 134

Markus A. Schill, Sarah F. F. Gibson, H.-J. Bender, and R. Männer. Biomechanical simulation of the vitreous humor in the eye using an enhanced chainmail algorithm. In Alan Colchester, William M. Wells, and Scott Delp, editors, *Medical Image Computing and Computer- Assisted Intervention – MICCAI '98*, pages 679–687, Cambribge, MA, USA, October 1998. Springer. 99, 137

Markus A. Schill, Clemens Wagner, Marc Hennen, Hans-Joachim Bender, and Reinhard Männer. Eyesi – a simulator for intra-ocular surgery. In Chris Taylor and Alan Colchester, editors, *Medical Image Computing and Computer- Assisted Intervention – MICCAI '99*, volume 1679 of *Lecture Notes in Computer Science*, pages 1166–1174, Cambridge; UK, September 1999. Springer. 127

Markus A. Schill, Clemens Wagner, Reinhard Männer, and Hans-Joachim Bender. Biomechanical modeling techniques and their application to the simulation of brain tissue. In Uwe Spetzger, H.Siegfried Stiehl, and Joachim M. Gilsbach, editors, *Navigated Brain Surgery – Interdisciplinary Views of Neuronavigation from Neurosurgeons and Computer Scientists*, pages 193–202. Mainz, 1999. 99

Markus A. Schill. *Biomechanical Soft Tissue Modeling – Techniques, Implementation and Applications.* PhD thesis, University of Mannheim, 2001. 97, 127, 135, 137

Douglas C. Schmidt. The adaptive communication environment. In *Proceedings of the 12th Sun User Group Conference*, 1993. 21

Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – a hardware architecture for ray tracing. In Thomas Ertl, Wolfang Heidrich, and Michael Doggett, editors, *Graphics Hardware*, pages 1–11, 2002. 125

Alexander Schwerdtner and Holger Heidrich. The dresden 3d display (D4D). In *Proceedings of SPIE*, volume 3295, 1998. 118

John Searle. Minds, brains, and programs. In *The Behavioral and Brain Sciences*, volume 3. Cambridge University Press, 1980. 13

Raimund Seidel. Linear programming and convex hulls made easy. In *Proceedings of the 6th Annual ACM Conference on Computational Geometry*, pages 211–215, 1990. 64

Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993. 156

Jonathan R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-TR-94-125, Carnegie Mellon University, 1994. 105

Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *Journal of Visualization and Computer Animation*, 2:132–134, 1991. 70, 71, 72, 74, 76, 94

Stefan Sichler. Computergrafische Spezialeffekte für medizinische Trainingssimulatoren. Master's thesis, Universität Mannheim, 2002. 120

Mel Siegel, Yoshikazu Tobinaga, and Takeo Akiya. Kinder gentler stereo. In *Proceedings of SPIE*, volume 3639, 1998. 118

Mel Slater. Measuring presence: A response to the witmer and singer presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 8(5):560–565, 1999. 8

Shamus Smith, Tim Marsh, David Duke, and Peter Wright. Drowning in immersion. In *Proceedings of UK Virtual Reality Special Interest Group*, 1998. 8

John Spitzer and Cass Everitt. Using GL_NV_vertex_array_range and GL_NV_fence on GeForce products and beyond. Technical report, NVidia Corporation, Erscheinungsdatum unbekannt. 123

Mandayam A. Srinivasan and Cagatay Basdogan. Haptics in virtual environments: Taxonomy, research status, and challenges. *Computers and Graphics*, 21(4):393–404, 1997. 12, 28

John A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys*, 28(1):205–208, 1996. 13

Thant Tessman. Casting shadows on flat surfaces. *IRIS Universe*, 1989. 122

Nadia Magnenat Thalmann and Daniel Thalmann. Virtual reality software and technology. *Encyclopedia of Computer Science and Technology, Marcel Dekker*, 41, 1998. 155

Theoharis Theoharis, Georgios Papaioannou, and Evaggelia-Aggeliki Karabassi. The magic of the z-buffer: A survey. In *Proceedings of the 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG*, 2001. 70

Henrik Tramberend. Avango: A distributed virtual reality framework. In *Proceedings of the IEEE Virtual Reality Conference*, 1999. 156, 158

Roger Y. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *IEEE Journal of Robotics and Automation*, RA-3(4):323–344, 1987. 134

Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphical Tools*, 1997. 68, 92

Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools*, 1999. 65, 67

Tzvetomir I. Vassilev, Bernhard Spanlang, and Yiorgos Chrysanthou. Efficient cloth model and collision detection for dressing virtual people. In *Proceedings of the ACM/EG Games Technology Conference*, 2001. 104

Dinesh Verma, Derek Wills, and M. Verma. Virtual reality simulator for vitreoretinal surgery. *Eye*, 17:71–73, 2003. 131

E. Viirre, H. Pryor, S. Nagata, and T.A. Furness. The virtual retinal display: A new technology for virtual reality and augmented vision in medicine. In *Proceedings of Medicine Meets Virtual Reality*, pages 252–257. IOS Press and Ohmsha, 1998. 118

Julien Villard and Houman Borouchaki. Adaptive meshing for cloth animation. In *Proceedings of the 11th International Meshing Roundtable*, pages 245–252, September 2002. 112

Udo Voges, Elmar Holler, Bernhard Neisius, Marc O. Schurr, and Thomas Vollmer. Evaluation of artemis, the advanced robotics and telemanipulator system for minimally invasive surgery. In *Proceedings IARP of the 2nd Workshop on Medical Robotics*, 1997. 160

Pascal Volino and Nadia Magnenat Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*, 13(3):155–166, 1994. 76, 77

Pascal Volino and Nadia Magnenat Thalmann. Implementing fast cloth simulation with collision response. In *Proceedings of Computer Graphics International (CGI)*, 2000. 107

Pascal Volino and Nadia Magnenath Thalmann. Accurate collision response on polygonal meshes. *Proceedings of Computer Animation*, 2000. 110

Pascal Volino and Nadia Magnenat Thalmann. Comparing efficiency of integration methods for cloth simulation. In *Proceedings of Computer Graphics International (CGI)*, 2001. 101

Clemens Wagner, Markus A. Schill, Marc Hennen, Reinhard Männer, Bettina Jendritza, Michael C. Knorz, and Hans-Joachim Bender. Virtuelle Realitäten für die augenchirurgische Ausbildung. *Der Ophthalmologe*, 98(4):409–413, 2001. 127

Clemens Wagner, Markus A. Schill, and Reinhard Männer. Collision detection and tissue modeling in a VR-simulator for eye surgery. In S. Müller and W. Stürzlinger, editors, *Proceedings of the 8th Eurographics Workshop on Virtual Environments, Barcelona*, pages 27–36. ACM Press, May 2002. 127

Clemens Wagner, Markus A. Schill, and Reinhard Männer. Intraocular surgery on a virtual eye. *Communications of the ACM*, 45(7):45–49, July 2002. 127

Clemens Wagner. Komplexitätsanalyse und effiziente Operationalisierung eines graphbasierten Wissensrepräsentationsformats und der dazugehörigen Abfragesprache. Master's thesis, Universität Heidelberg, 1996. 58

Kent Watsen, Rudolph P. Darken, and Michael V. Capps. A handheld computer as an interaction device to a virtual environment. In *Proceedings of the Third International Immersive Projection Technology Workshop (Stuttgart, Germany, 1999)*, 1999. 12

Benjamin Watson, Victoria Spaulding, Neff Walker, and William Ribarsky. Evaluation of the effects of frame time variation on VR task performance. In *IEEE Virtual Reality Annual Symposium, VRAIS '97*, pages 38–44, April 1997. 14

Josie Wernecke and Open Inventor Architecture Group. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor (TM)*. Addison-Wesley, 2nd edition, 1994. 155

Peter Wißkirchen. *Object-Oriented Graphics – From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, 1990. 46

Andrew Witkin and David Baraff. Physically based modeling: Principles and practice – differential equation basics. SIGGRAPH '97 Course Notes, 1997. 103

Bob G. Witmer and Michael J. Singer. Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998. 8

Gabriel Wittum. Mehrgitterverfahren. *Spektrum der Wissenschaft*, (4):78–90, April 1990. 111

Matthias M. Wloka and E. Greenfield. The virtual tricorder: A uniform interface for virtual reality. In *Proceedings of UIST*, pages 39–40, 1995. 12

Matthias M. Wloka. Lag in multiprocessor virtual reality. *Presence: Teleoperators and Virtual Environments*, 4(1):50–63, 1995. 18, 31

Dongliang Zhang and Matthew M.F. Yuen. Cloth simulation using multi-level meshes. *Computer & Graphics*, 25(3):383–389, 2001. 111, 112