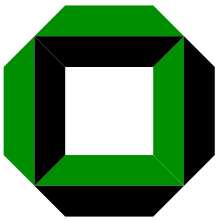
Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

Virtual Environments

Seminar - Sommersemester 2003

Betreuung:
Matthias Baas, Jan Bender, Dieter Finkenzeller,
Bertrand Klimmek, Stefan Preuß, Sven Thüring

Interner Bericht 2003-23



Universität Karlsruhe

Fakultät für Informatik

ISSN 1432 - 7864

Zusammenfassung

Dieser Bericht stellt die Ergebnisse des Seminars *Virtual Environments* (VE) zusammen. Ein wichtiges Ziel von VE ist die Immersion, die Einbindung des Benutzers als aktiven Teilnehmer in eine computergenerierte Welt. Vorausetzung dafür sind Techniken zur Simulation von „Lebendigen virtuellen Welten“, also zur Simulation von 3D-Szenen mit realistischem Verhalten. Es geht dabei um Kollisionserkennungsalgorithmen, haptisches Rendering, Navigations- und Interaktionstechniken, programmierbare Grafik-Hardware, verteilte virtuelle Welten bis hin zur Modellierung und Simulation von virtuellen Menschen.
Die virtuelle Realität hat sich inzwischen in verschiedenen Anwendungsbereichen durchgesetzt und wird auch im Rahmen des SFB 588 „Humanoide Roboter - Lernende und kooperierende multimodale Roboter“ für die Simulation des humanoiden Roboters und die Evaluierung der Mensch-Roboter-Schnittstelle eingesetzt.

Das Seminar *Virtual Environments* wurde im Sommersemester 2003 am Institut für Betriebs- und Dialogsysteme, Fakultät für Informatik der Universität Karlsruhe durchgeführt.

Teilnehmer

Stephan Bergmann
Marc Diensberg
Erdem Heper
Özdem Heper
Jörn Herwig
Thomas Hettler
Marco Lepper
Michael Niedermaier
Markus Schwarz
Bert Völker
Christian Wallenta
Zhixing Xue

Betreuung

Matthias Baas
Jan Bender
Dieter Finkenzeller
Bertrand Klimmek
Stefan Preuß
Sven Thüring

Prof. Dr. Alfred Schmitt
Institut für Betriebs- und Dialogsysteme
Fakultät für Informatik
Universität Karlsruhe

Inhaltsverzeichnis

1 Augmented Reality

Erdem Heper

1.1 Einführung

Mit zunehmender Technisierung sieht sich der Mensch einer immer größeren Informationsflut gegenüber, die er alleine nichtmehr bewältigen kann. Augmented Reality ist ein unterstützendes System, welches versucht Menschen zu helfen, diese Informationen sinnvoll einzusetzen.
In meiner Ausarbeitung möchte ich einen Überblick zu Augmented Reality (im Folgenden AR) bieten. AR geniesst derzeit keinen so großen Bekanntheitsgrad wie Vitual Environments (VE), was nicht zuletzt an dessen Alter liegt. VE hat bereits zu diversen Filmen inspiriert, wie Tron, Der Rasenmähermann und die Matrix Trilogie, welche die populärsten darstellen. Eine solche Entwicklung ist bei AR eher nicht möglich, da die Technik einfach zu nützlich und zu günstig ist. Vielmehr ist eine stillschweigende Verbreitung zu erwarten, ähnlich der von Handys, PCs und vielem mehr.
Meine Ausarbeitung richtet sich grösstenteils nach
[Azuma '97], einem ausführlichen Überblick zum Gebiet der AR.

1.1.1 Überblick

Abschnitt eins widmet sich der Frage was AR ist. Dabei wird das System im Allgemeinen betrachtet und auch auf verschiedene Eigenschaften von AR eingegangen. Im folgenden werden ein paar Anwendungsmöglichkeiten, die bisher mit AR in Verbindung gebracht wurden, genannt. Der dritte Abschnitt beschäftigt sich mit der Frage, wie AR Systeme aufgebaut sind und geht auf die Probleme gegenwärtiger Umsetzungen ein. Im vierten Abschnitt wird ein allgemeines Interface zu AR-Systemen vorgestellt, was am Beispiel einer speziellen Umsetzung geschieht. Im letzten Abschnitt biete ich einen Ausblick und eine Zusammenfassung zum gegenwärtigen Entwicklungsstand.

1.2 Was ist Augmented Reality?

1.2.1 Definition

Allgemein betrachtet ist Augmented Reality eine Spielart von Virtual Environments, die den Benutzer nicht vollständig in eine künstliche Umgebung hüllt. Stattdessen erweitert sie die Realität um vereinzelte virtuelle Objekte oder Informationen. Bei genauerer Betrachtung sind drei Bedingungen zu erkennen, die AR-Systeme erfüllen müssen.

1. Kombination von Realität und Virtualität

2. Interaktion in Echtzeit

3. Registrierung virtueller und realer Objekte

Bis auf Punkt 2 sind das für VE-Systeme vollkommen unbekannte Kriterien und selbst dabei haben es VE-Systeme nicht so schwer wie AR-Systeme, da sich Verzögerungen oder andere Abweichungen von der Echtzeit auf das ganze System auswirken und somit den Benutzer nicht so stark irritieren. An der allgemeinen Formulierung der Bedingungen ist aber auch zu erkennen, dass dem Entwickler alle Freiheiten gegeben sind, was die Technik zur Umsetzung von AR-Systemen betrifft.
Wie in VE streben die Entwickler auch in AR danach nicht nur den Sehsinn anzusprechen. Es sollen auch Haptik und Akustik augmentiert werden, um beispielsweise Oberflächenstrukturen zu simulieren oder Geräusche ein- oder auszublenden.

1.2.2 Eigenschaften

Charakteristische Eigenschaften von AR-Systemen sind Mobilität, Beherrschung verschiedener Interaktionsformen, aufwendige Algorithmen und Echtzeitfähigkeit. Diese Eigenschaften sind nur ein Teil der Charakteristik, stellen aber die Hauptprobleme von AR-Systemen dar. Mobilität ist ein wichtiger Aspekt, da der Benutzer sich in der Realität bewegt und am besten nicht (oder möglichst wenig) eingeschränkt wird in seiner Bewegungsfreiheit. Deshalb gestalten sich AR-Systeme als aufwendige mobile Systeme mit viel Sensorik, verschiedenen Eingabe- und Visualisierungsmöglichkeiten. Die Interaktion mit dem Benutzer kann sehr verschieden ausfallen. Sprachsteuerung, Gestenerkennung und Trackingbasierte Steuerung sind nur ein Bruchteil der Möglichkeiten in AR. Sollten diese Techniken eingesetzt werden, müssen sie durch aufwendige Algorithmen ausgewertet werden. Der Trackingaufwand in AR-Systemen ist um ein vielfaches höher als in VE. Es sind nicht nur Positionsdaten vom Anwender gefragt, auch die Umgebung und eventuell auch Eingabegeräte müssen getrackt werden, was wiederum effizientere Algorithmen als in VE fordert.

1.2.3 Motivation

Abbildung 1.1: Das Realität-Virtualität Kontinuum nach Milgram

Trotz der Schwierigkeiten, die mit einem AR-System einhergehen, ist die Entwicklung in Richtung MR (Mixed Reality[1] Abbildung 1.2.3) sinnvoll. Mit einem geschickten AR-System kann der Benutzer Vorteile der Realität mit denen der Virtualität vereinen. Sollten sich dem Anwender schwierige Aufgaben stellen, kann er sie, mit Hilfe von AR, effizienter lösen. Weitere Motivation wird im folgenden Abschnitt anhand von Beispielen gegeben.

[1] Mixed Reality stellt das Kontinuum zwischen Realität und Virtualität dar. Also die Mischwelt beider Möglichkeiten. Ist beispielsweise ein reales Objekt in eine virtuelle Umgebung eingebunden, handelt es sich um erweiterte Virtualität. Wird hingegen die Realität um virtuelles erweitert, ist es erweiterte Realität, AR.

1.3 Welche Anwendungsmöglichkeiten gibt es?

Aus der bisherigen Entwicklung haben sich sechs Anwendungsmöglichkeiten für AR-Systeme hervorgehoben. Diese werden nun vorgestellt.

1.3.1 Medizin

Abbildung 1.2: Versuchsaufbau einer Brustuntersuchung

Anwendungsmöglichkeiten in der Medizin bestehen in der Ausbildung und in der Chirurgie. In der Ausbildung kann AR als Trainingsplattform für angehende Chirurgen verwendet werden. Insbesondere durch haptische Augmentierung einer Trainingsoperation. In der Chirurgie können wichtige Daten auf den Patienten eingeblendet werden, sodass der Chirurg nicht von seinem Eingriff aufsehen muss, um auf die Daten zu zugreifen. Radiologen kann AR die Möglichkeit geben, dem Chirurg eine Art Röntgenblick zu geben, indem sie ihm dreidimensionale radiologische Bilder auf den Patienten augmentieren.
In Abbildung 1.2 wird eine Biopsie an einer künstlichen Brust durchgeführt. Solche Systeme erfordern eine hohe Genauigkeit bei der Augmentierung, da anhand der eingeblendeten Daten die Nadel zur Gewebeentnahme geführt wird.

1.3.2 Produktion, Wartung und Reparatur

Hier kann AR komplizierte Arbeitsvorgänge an Maschinen oder in der Produktion vereinfachen. Die Arbeitsschritte können anhand von 3D-Modellen visualisiert werden und den Anwender unterstützen. Informationen können eingeblendet werden, wie Hinweise auf Hochspannung oder ähnliches, um den Benutzer zu schützen.

1.3.3 Visualisierung und Kommentierung

Im Allgemeinen kann AR dazu verwendet werden Aufgaben, die der Benutzer mit einem bestimmten Objekt zu erfüllen hat, zu notieren und direkt mit dem Objekt zu verbinden. Hat dieses Objekt bestimmte Eigenschaften, die mit bloßem Auge nicht zu erkennen sind, können diese Kommentare darauf hinweisen. Augmentierung kann auch dazu eingesetzt werden Räume oder Objekte zu visualisieren. Soll beispielsweise die Inneneinrichtung geändert werden, kann diese erst in der AR betrachtet werden ehe sie umgesetzt wird. Sollten schlechte Sichtverhältnisse herrschen kann der Raum durch ein 3D-Modell visualisiert werden, um eine bessere Orientierung zu ermöglichen.

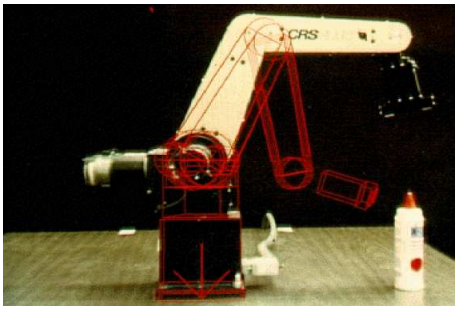


Abbildung 1.3: Robotersteuerung

1.3.4 Robotersteuerung

Bei der Programmierung von Roboterbewegungen kann AR als Simmulationsumfeld dienen um von vornherein Fehler zu vermeiden, die bei der Programmierung entstehen können. Bei Fernsteuerung kann AR zur Visualisierung der Bewegungen benutzt werden (Abbildung 1.3).

1.3.5 Unterhaltung

Abbildung 1.4: Training mit dem Laserschwert

AR kann auch als Unterhaltungs-, oder Spiele Umgebung verwendet werden. Die Realität kann um intelligente, virtuelle Kreaturen erweitert werden, mit denen der Benutzer interagieren kann. Die Interaktion kann auch zu einem Spiel ausgebaut werden, wie in Abbildung 1.4, einem Jedi Trainingprojekt im Zuge einer Studienarbeit an der Universität Karlsruhe (TH).

1.3.6 Militär

Auf diesem Gebiet wird AR schon seit längerem eingesetzt. In Kampfjets werden Fluginformationen über ein HUD (Head Up Display) in das Sichtfeld des Piloten eingeblendet, damit dieser sich auf den Flug konzentrieren kann und nicht mit der Instrumententafel beschäftigt ist. In Kampfhubschraubern sollen so genannte HMS (Helm Mounted Sights) eingesetzt werden. In Zukunft soll die Zielerfassung über diese erfolgen, sodass der Pilot nur noch das Zielobjekt ansehen muss, um es zu erfassen und die Bordkanonen danach auszurichten. Abbildung 1.5 zeigt ein HMS, wie es in heutigen Militärsystemen verbaut ist.

Abbildung 1.5: Ein HMS für den Einsatz beim Militär

1.4 Wie sind aktuelle AR-Systeme aufgebaut?

Der Aufbau aktueller AR-Systeme kann in mehrere Teile untergliedert werden. Auf der Hardwareseite gibt es Darstellungsgeräte, Interaktionswerkzeuge und Trackingmechanismen, die jeweils eine Vielfalt von Umsetzungsmöglichkeiten bieten. Das Gebiet der Trackingmechanismen sondert sich etwas von den anderen Hardwarekomponenten ab, da es kommerzielle Hardwaresysteme gibt und freie Softwarelösungen. Software-seitig gibt es Ansätze zur Augmentierung, Registrierung und zur Kalibrierung. Diese Gebiete sind die Schwerpunkte der AR Entwicklung, da es hier besonders viele Probleme zu lösen gibt.

1.4.1 Hardware

Darstellungsgeräte

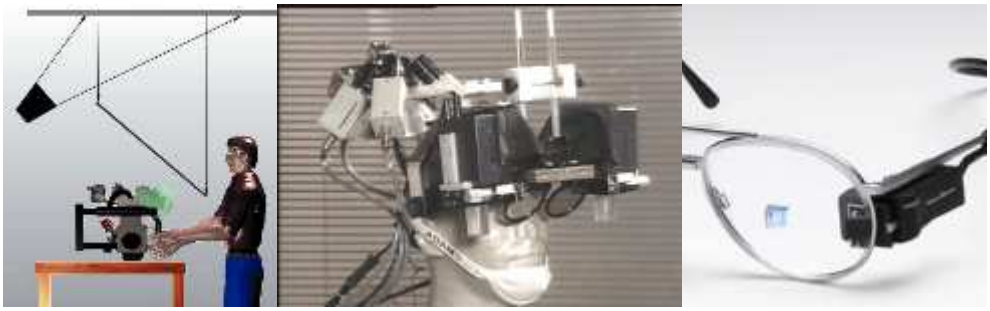


Abbildung 1.6: Optische Durchsichtsysteme, (von links) fixer Aufbau einer reflektierenden Oberfläche mit Projektor, HMD Prototyp, Retinaprojektor

Die Technik, die zur Visualisierung eingesetzt wird, gliedert sich in zwei Gruppen. In die der optischen Systeme, bzw. Sichtsysteme und die der videobasierten Systeme. Optische Systeme projizieren die augmentierten Daten über eine durchsichtige Oberfläche in das Sichtfeld des Anwenders. Hauptkriterium ist die direkte Sicht auf die Realität. Videobasierte Systeme nehmen die Umwelt über Kameras wahr. Die Kombination mit der Augmentierung erfolgt im Videosystem und wird dem Benutzer über Monitorsysteme übermittelt. Eine direkte Sicht auf die Realität ist hier nicht möglich. Diese Technik bietet aber durch verschiedene Kompositionsstrategien mehr Möglichkeiten bei der Darstellung.

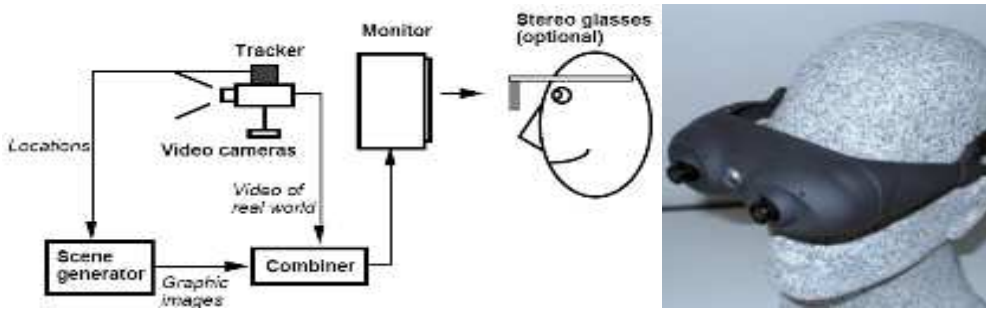


Abbildung 1.7: Videobasierte Durchsichtsysteme, (von links) stationäres Monitorsystem, HMD mit 2 Kameras

I. Optische Systeme

Der Aufbau von Sichtsystemen kann unter verschiedenen Bedingungen erfolgen. Wichtig ist dabei die Frage der Platzierung der Projektionsebene und das Tracking der Kopfposition des Anwenders. Die Projektionsebene kann eine halbverspiegelte Fläche sein oder direkt die Netzhaut des Benutzers. Die halbverspiegelte Oberfläche kann entweder direkt vor den Augen des Benutzers am Kopf angebracht sein oder fix an einem Ort im Arbeitsumfeld aufgebaut sein. Wird die Kopfposition getrackt, kann die Visualisierung angepasst werden, um zu der gegenwärtigen Kopfposition und dem Blickwinkel zu passen. Ohne Tracking kann die Augmentierung nur von einem bestimmten Punkt aus betrachtet werden (wie die Pilotenkanzel in Kampfjets) oder sie ändert nie ihre Position im Sichtfeld des Benutzers (HMD ohne Tracking). Die letzte Situation entspricht der der Retinaprojektoren, die schwache Laser einsetzen, um die Augmentierung direkt auf der Netzhaut vorzunehmen. Abbildung 1.6 zeigt eine Auswahl solcher Systeme.

II. Videobasierte Systeme

Diese Technik basiert darauf, die Umwelt über eine, oder mehrere Kameras aufzunehmen, um die Augmentierungsdaten zu erweitern und auf einem Monitorsystem darzustellen. Bei der Kombination kann beispielsweise Chromakeying[2] eingesetzt werden, was eine effiziente Lösung ist. Oder es können Pixel für Pixel Vergleiche vorgenommen werden, falls Tiefeninformationen vorliegen und Überdeckungen von virtuellen und realen Objekten aufgelöst werden sollen. Wie auch bei optischen Systemen kann das Darstellungssystem direkt vor den Augen des Anwenders angebracht sein oder an einer festen Position im Raum. Das Kamerasystem hingegen kann entweder das Sichtfeld des Benutzers wiedergeben, indem es, wie das Darstellungssystem, in der Nähe der Augen angebracht ist, oder einen fremden Ort, der für den Benutzer nicht zugänglich ist. Diese Telerealität ist bei HMDs vielleicht nicht ganz sinnvoll, dafür aber bei fixen Monitorsystemen, die dem Anwender ein Fenster in die Realität des Kamerasystems bieten.

III. Vergleich der beiden Techniken

[2] Chromakeying ist eine Technik, die gerne zur Effektgenerierung verwendet wird. Das virtuelle Objekt wird hierbei auf einer (beispielsweise) grünen Hintergrundfläche generiert. Das Grün wird bei der Kombination mit dem Realvideostrom als transparent verarbeitet. Somit dürfen weder Realstrom, noch Erweiterung diese Farbe enthalten. Das Resultat ist eine vollständige Überdeckung des Realstroms an den Erweiterungen.

Die Frage, welche Methode zur Darstellung besser geeignet ist um AR-Systeme zu verwirklichen, ist nicht leicht zu klären. Beide Systeme haben ihre vor, bzw Nachteile. Diese sollen im folgenden aufgezeigt werden.

Pro Optische:

Sicherheit: Bei einem Ausfall des Visualisierungssystems ist dem Benutzer weiterhin eine Sicht auf die Realität möglich. Zwar ist die Sicht verdunkelt, aber nicht unmöglich. Diese Eigenschaft macht die optischen Systeme besonders geeignet für kritische Anwendungen, bei denen ein Verlust der Sicht die Sicherheit gefährdet.
Einfachheit: Das augmentierende System muss nur einen Videostrom bearbeiten, den der Erweiterung. Dadurch wird das System entlastet und eine geringere Latenz wird ermöglicht.
Auflösung: Die Auflösung, mit der der Benutzer seine Umwelt wahrnimmt, wird nicht eingeschränkt. Bei optischen werden nur die dargestellten Objekte in der Auflösung eingeschränkt. Sollen beispielsweise feine Beschriftungen an Kabelsträngen erkannt werden, ist ein Videosystem ungeeignet, weshalb bei Boeing optische Systeme entwickelt werden, um als Produktionshilfe bei Kabelbäumen zu dienen.
Offset: Dieser Punkt betrifft eher HMD Systeme, die dem Benutzer seine gewohnte Umwelt zeigen sollen. Optische Systeme haben keine Offsetprobleme, da das Bezugssystem Auge erhalten bleibt.

Kontra Optische:

Unflexible Kompositionsstrategie: Optische Systeme können virtuelle Objekte nur einblen-

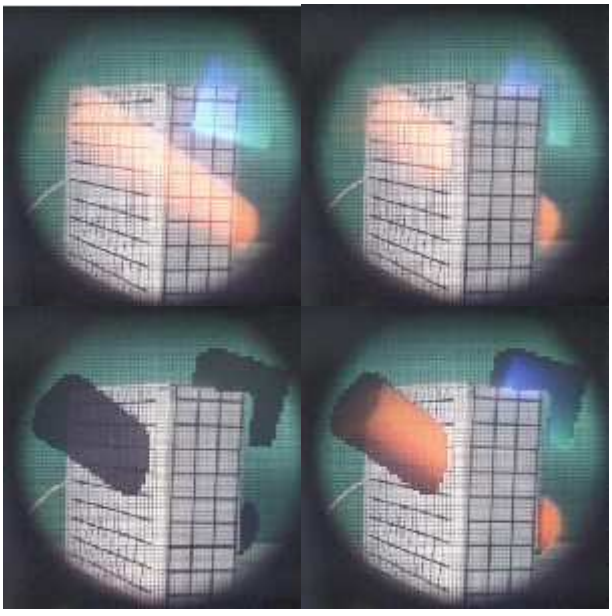


Abbildung 1.8: Auflösung von Geistbildern: (von oben links, zeilenweise) Geistbild, Einberechnung der Tiefe, Abdunkelung der Zeichenflächen, Kombiniertes Bild

den ohne den Hintergrund ausblenden zu können. Dadurch entstehen so genannte Geistbilder wie in Abbildung 1.8. Um diese zu verhindern müsste vor die Projektionsfläche ein Filter an-

gebracht werden, der den Hintergrund des einzublendenden Objekts verdunkelt. Ein solches System ist derzeit nicht erhältlich.
Synchronisation: Die Augmentierung kann nicht hinreichend mit der Realwelt synchronisiert werden. Zwar haben optische Systeme den Vorteil der geringen Systemlatenz, aber bei Bewegungen ist dennoch eine deutliche Verzögerung zu sehen, welche die Illusion der Augmentierung zerstört.
Bildverbesserung: Ist eine Verbesserung des Realweltbildes nötig, kann sie bei optischen Systemen nicht durch Digitalfilter erreicht werden. Es sind schwere Linsen und Filtersystem nötig, die das Darstellungssystem komplizieren und auch im Gewicht erhöhen, welches bei HMDs besonders wichtig ist.

Pro Videobasiert:

Flexibilität der Kompositionsstrategien: Videobasierte System beherrschen verschiedene Kompositionsstrategien, wie das erwähnte Chromakeying oder Pixel für Pixel Vergleiche, Virtualität mit Realität zu Kombinieren. Das Resultat ist ein schlüssigeres Bild als es bei optischen Systemen möglich ist.
Filterung: Die eingehenden Daten des Videosystems können durch digitale Filter verbessert werden und dem Benutzer ein breiteres, unverzerrtes Gesichtsfeld ermöglichen. Diese Möglichkeit besteht bei optischen Systemen nicht.
Synchronisation: Neben einer Filterung, die auf die eingehenden Videodaten angewendet werden können, kann der Datenstrom mit dem der Augmentierung angeglichen werden, damit diese zeitlich stimmig wiedergegeben werden.
Zusätzliche Registrierungsinformation: Dem augmentierenden System stehen nicht nur die Daten des Tracking Systems zur Verfügung, sondern auch die Kamerabilder, die ausgewertet eine bessere Positionierung der virtuellen Objekte ermöglicht.
Helligkeit/Kontrast: Weiterhin können die Videodaten ausgewertet werden, um bei dem augmentierten Objekt die Helligkeit und den Kontrast einzustellen.

Kontra Videobasiert:

Sicherheit: Bei einem Ausfall des Visualisierungssystems ist dem Benutzer keine Sicht auf die Realität möglich. Videobasierte Systeme schneiden den Benutzer ganz von der Realität ab (ob jetzt fern oder lokal), sodass der Anwender prinzipiell blind ist.
Zusätzliche Latenz: Videobasierte Systeme haben mit Verzögerungen bei Aufnahme und Kombination zu kämpfen.
Auflösung: Bei Videosystemen gibt es gleich zwei Einschränkungen in der Auflösung. Die der Kameraauflösung und die des darstellenden Systems. Umgebungsdetails können dadurch nicht erkannt werden, was einschränkend wirken kann, besonders wenn sie für eine Aufgabe des Anwenders von Wichtigkeit sind.
Offset: Bei Video HMDs besteht ein Offset zwischen den Augen, dem normalen Blickwinkel, und dem Kamerasystem. Die Kameras (beispielsweise bei stereoskopischer Aufnahme) befinden sich meist in einer anderen Höhe, haben einen anderen Abstand zueinander als die Pupillendistanz des Betrachters und so weiter. Diese Abweichungen können durch Spiegel- oder Linsensysteme behoben werden, was aber zu aufwendig wäre, oder von der Softwareseite durch Transformationen oder andere Softwarefilter realisiert werden müsste. Was die Video-HMD-Systeme wiederum komplizierter macht.

IV. Probleme optischer und videobasierter Systeme

Beide Systeme haben Probleme mit der Fokussierung und dem Kontrast. In Videobasierten zeigt sich das Problem der Fokussierung so, dass die virtuellen Objekte nach dem Lochkamera-Prinzip generiert werden, also in jeder Tiefenebene scharf dargestellt werden. Störend kann das wirken, wenn das Objekt nicht in der betrachteten Ebene ist, aber dennoch scharf gezeichnet wird. Abhilfe kann ein angepasstes Rendersystem schaffen in Kombination mit einer Autofokuskamera. In optischen Systemen werden generierte Objekte immer in der selben Distanz zum Betrachter gezeigt, sodass auch hier Fokussierungsprobleme herrschen.
Ein weiteres Problem ist die Bewegungsfreiheit. Beide Systeme erfordern eine Verkabelung des Benutzers, was ihn in seiner Mobilität einschränkt. Im Gegensatz zu VE, wo der Benutzer eher dazu animiert wird, herumzufliegen, muss der AR-Benutzer laufen können.
Weitere Probleme sind Registrierung und Kalibrierung, die später genauer besprochen werden.

Interaktionswerkzeuge

Für den Einsatz in AR sind Eingabegeräte geeignet, die ausreichend mobil sind und den Benutzer in seiner Bewegungsfreiheit nicht, oder wenig einschränken. Da es beispielsweise nötig sein könnte durch eine Tür zu gehen (AR ist ja schließlich selbst mobil), sollten die Hände des Benutzers nicht nur für Eingabegeräte vereinnahmt sein, sodass eine Betätigung des Türgriffs möglich ist. An diesem Beispiel ist zu sehen, dass selbst triviale Alltagsaufgaben zu einem Problem werden können. Es ist aber auch hier die Entscheidung des Entwicklers, wie viel Mobilität er dem Benutzer einräumt. Es werden Datenhandschuhe, Touchpanels (auch PDAs), Zauberstäbe, PIPs (Personal Interaction Panel, Studierstube), und ähnliche Eingabegeräte verwendet. Es sind aber auch stationäre Eingabegeräte wie 3D-Maus, Phantom und ähnliches einsetzbar. Die meisten Möglichkeiten kommen aus dem Bereich der VE und sind bereits bekannt. In Verbindung mit AR auftretende Interaktionswerkzeuge wie Zauberstäbe und PIPs funktionieren mit Trackingsoftware. Diese erkennt Marker auf den Eingabegeräten oder die Eingabegeräte selbst und trackt diese zur Interaktion. Mit Hilfe eines solchen Systems wurde eine Interaktionskonfiguration entwickelt, die robust und intuitiv erlernbar sein soll ([Veigl '02]).

I. ARToolkit zu Interaktion

Bei ARToolkit handelt es sich um die bereits erwähnte Trackingsoftware, die bestimmte Marker aus Videodaten extrahieren und identifizieren kann. Die Konfiguration ist bereits ein vollständiges AR-System bestehend aus:

- 2D Touchpad
- Fahrradhandschuhe mit Drucksensor am rechten Daumen
- Optischer HMD Helm mit Firewirekamera und Trägheitssensor

Wie in Abbildung 1.9 zu sehen, sind beide Hände sind mit ARToolkit-Markern versehen, sodass sie, wenn sie im Blickfeld der Kamera sind, getrackt werden können. Die Fahrradhandschuhe stellen eine robuste Basis für die Anbringung des Drucksensors und der Marker dar und lassen dabei dem Anwender noch genug Fingerspitzengefühl. In Verbindung mit den Markern kann somit ein 3D-Cursor berechnet werden, falls der Drucksensor betätigt wird. Die gewonnenen

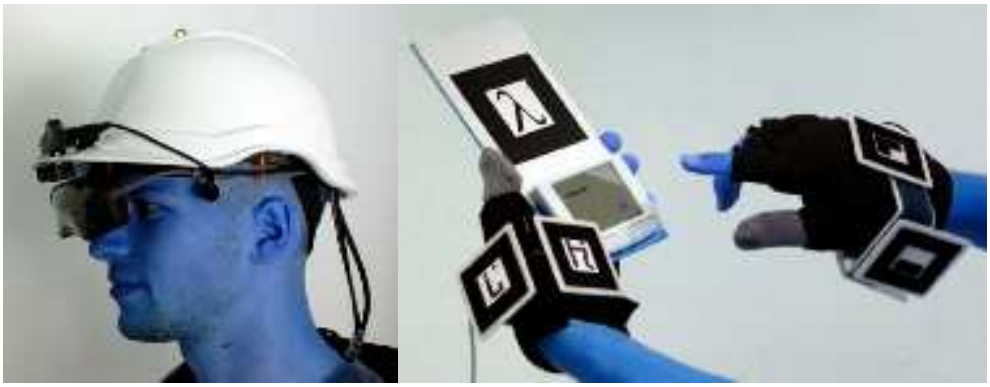


Abbildung 1.9: Aufbau der ARToolkit-Interaktion

3D-Daten sind ausreichend genau, da der Abstand von Handgelenk zu Daumen nur geringen Schwankungen unterliegt. Am linken Handgelenk ist das Touchpanel angebracht, der auch mit einem ARToolkit Marker versehen ist, der als Monitor zum System fungiert. Das Touchpanel wird zur direkten Systeminteraktion verwendet.
[Veigl '02] ist eine Weiterentwicklung des PIP. Während aber bei der PIP Realweltstützen in Form von Tafel und Stift erforderlich sind, kann die Interaktion hier Freihand erfolgen. Bei einen Versuch mit 1500 Versuchspersonen hat sich das System als robust und intuitiv nutzbar erwiesen. Eine Verbesserung des Systems könnten kabellose Verbindungen bringen. In der Zwischenzeit jedoch müssen haltbare Kabel herhalten.

Tracking

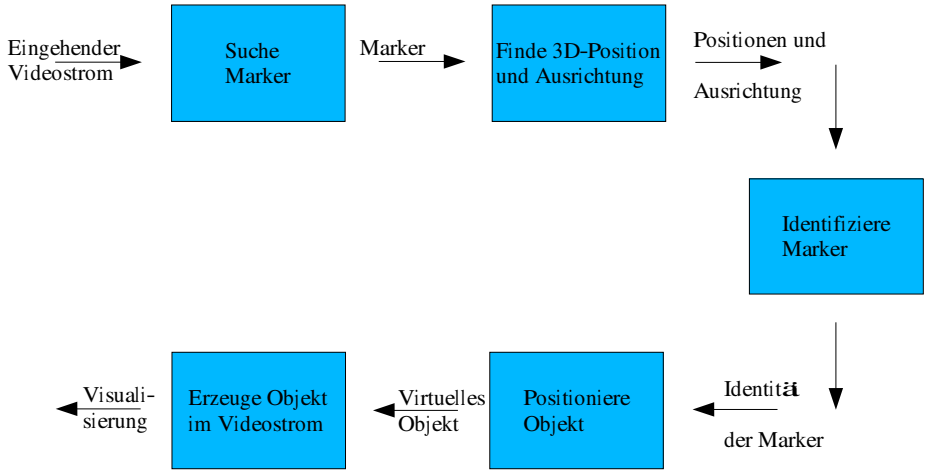


Abbildung 1.10: Schritte im visuellen Tracking mit Markern

Das Trackingsystem wird bei AR-Systemen besonders stark gefordert. Es muss eine sehr hohe Genauigkeit bei möglichst großer Reichweite aufweisen und dabei den Echtzeitanforderungen des Systems genügen. Wie auch bei den anderen Hardwaregebieten stellt sich auch hier dem Entwickler ein breites Spektrum an Möglichkeiten.

- Optisches Tracking (mit/ohne Marker)

- Trägheitssensor (Gyroskop)
- Sender/Empfänger Systeme
- GPS
- Differentielles GPS
- Neigungssensoren
- ...

Das Problem aktueller optischer Systeme ist, dass nur eine, oder wenige Kameras eingesetzt werden, was das Tracking von Markern deutlich erschwert. Aktive Marker, die beispielsweise leuchten, erleichtern zwar das Tracking, erfordern aber eine Verkabelung des zu trackenden Objektes. Wird ein Marker verdeckt und kein weiterer Marker ist mehr im Sichtfeld verliert das System die Orientierung, womit auch eine geschickte Platzierung von Markern erforderlich ist. Trägheitssensoren, wie beispielsweise ein Gyroskop, geben nur relative Positionsdaten wieder und funktionieren nur bei Bewegung fehlerfrei. Bei Stillstand fangen sie an zu driften und müssen neu kalibriert werden. Sender/Empfänger Systeme, wie beispielsweise Ultraschallsender mit mehreren Mikrofonen zur Positionsbestimmung, sind zwar genau, brauchen aber eine präparierte Umgebung. GPS ist inzwischen für jedermann erhältlich aber leider nicht genau genug. Die Genauigkeit liegt in etwa bei 10m-20m. Abhilfe schafft da differentielles GPS, was eine Genauigkeit im cm Bereich erlangt. Der Vorteil dieser Systeme liegt in der großflächigen Verfügbarkeit und in der absoluten Positionsbestimmung. Nachteilig ist hingegen die Ungenauigkeit. Genauere Systeme hingegen haben eine stark eingeschränkte Reichweite, weshalb GPS durchaus eine Option für Outdoortracking ist. Neigungssensoren haben wie Gyroskope das Problem, dass sie relative Positionsdaten wieder geben.
Es gibt viele kommerzielle Systeme, welche die eine oder andere Technik umsetzen. Keine der bekannten Methoden alleine ist dazu geeignet weiträumiges Tracking zu ermöglichen. Vielmehr ist eine Kombination verschiedener Ansätze nötig. Vorgeschlagen wird beispielsweise eine Kombination von GPS, Neigungssensoren und elektronischem Kompass, um die fehlenden Neigungs- und Rotationsdaten zu der Positionserkennung hinzufügen zu können. Ein weiterer Ansatz schlägt eine Kombination von optischem Tracking und Gyroskop vor. Im Innenbereich wird das System durch Marker unterstützt, außerhalb erfolgt eine Vorausberechnung der Bewegung anhand der Videodaten. Dazu werden EKF (Extended Kalman Filter) eingesetzt. Mit dieser Vorausberechnung kann dann das Gyroskop rekalibriert werden um seine Einsatzdauer zu erhöhen.
Zum Tracking werden auch freie Softwarelösungen zur Verfügung gestellt, wie beispielsweise Opentracker und ARToolkit. Das ARToolkit stellt bereits ein vollständiges AR-Softwaresystem mit Trackingsoftware, Videocapturesoftware und Darstellungssoftware. Die Marker, die für den Einsatz mit ARToolkit gedacht sind, können beliebig ausgedruckt werden. Wurden sie zuvor mit eindeutigen Bildern versehen, kann sie das System später identifizieren. Das Tracking der AR-Software gliedert sich wie in Abbildung 1.10.

1.4.2 Software

Auf der Softwareseite von AR-Systemen ist eine genaue Differenzierung eher schwierig, da die Komponenten stark ineinander einwirken. Es kann eine Einteilung in augmentierende, registrierende und kalibrierende Software gefasst werden. Ziele und Probleme dieser Komponenten werden in diesem Abschnitt besprochen.

Augmentierung

Was in dieser Ausarbeitung oftmals als Augmentierung bezeichnet wurde, ist nichts anderes als die Visualisierung der virtuellen Daten. Je nach System ist die Augmentierungssoftware damit beschäftigt Bilddaten zu generieren (optische Systeme) oder zusätzlich noch mit Videodaten zu kombinieren (videobasierte Systeme). Bei der Bildgenerierung wird auf die Registrierungsdaten zugegriffen. Obwohl es vielleicht so aussieht, ist die Visualisierung kein Schwerpunkt in AR-Systemen. Die Entwickler begnügen sich mit Texteinblendungen und Gittermodellen, die für ihre Studien durchaus ausreichend sind. Derzeit liegen die Schwerpunkte der Entwicklung beim Tracking und bei der Registrierung. Erst wenn diese hinreichend gelöst sind, ist eine Verbesserung der Visualisierungssoftware sinnvoll.

Registrierung

Durch die Registrierung soll eine visuell schlüssige Komposition virtueller und realer Objekte ermöglicht werden. Dazu muss für das zu augmentierende Objekt eine Transformation gefunden werden, die, in Bezug zum Betrachter, die richtige Rotation, Neigung und Position hat. Da hier die Daten des gesamten AR-Systems zusammenkommen, sammeln sich auch die Fehlerquellen. Und da am Ende der Registrierung die Visualisierung steht, werden sie besonders häufig sichtbar. Der menschliche Sehsinn ist stark ausgeprägt. Fehler in der Positionierung die vielleicht nur 5mm betragen, können bereits störend wirken. Es gibt zwei Arten von Fehlerquellen. Statische und dynamische.

I. Statische Fehlerquellen

Statische Fehler können aus Verzerrungen im optischen System, Fehlern im Trackingsystem, mechanischen Fehlern und aus falschen Sichtparametern entstehen. Verzerrungen im optischen System entstehen bei allen Kamerasystemen. Durch geschickte Optiken kann das verzerrte Bild geglättet werden, was aber ein kompliziertes (schweres) Linsensystem erfordert. Realistischer ist eine digitale Lösung durch Bildbearbeitungssoftware (Softwarefilter). Fehler im Trackingsystem werden durch die geringe Genauigkeit, die kurze Reichweite und eine hohe Datenrate erzeugt (Abschnitt 1.4.1). Diskrepanzen zwischen Modell und Wirklichkeit wirken sich in mechanischen Fehlern aus. Ist beispielsweise die Projektionsfläche eines optischen HMDs nicht 20mm (Modell) sondern 22mm (Wirklichkeit) vom Projektor entfernt, kommt es zu diesen Fehlern. Fehlerhafte Sichtparameter können durch ungeschickte Kalibrierung entstehen, bei der Parameter, wie Pupillendistanz, Parallaxewinkel, Gesichtsfeld und ähnliches, bestimmt werden.

II. Dynamische Fehlerquellen

Dynamische Fehler treten durch Verzögerungen im System auf. Derzeit gibt es kein System, das nicht Verzögerungen einbringt und somit zu diesen Fehlern führt. Da AR ein mobiles System ist und viel mit Bewegungen des Anwenders arbeitet, fallen diese Latenzfehler besonders auf. Der Zeitpunkt des Trackings und das Aufnehmen der zugehörigen Videodaten ist bereits mit Verzögerungen behaftet. Kommen noch Bearbeitungen des Videostroms hinzu, kommt es zu deutlich sichtbaren Latenzfehlern, wie in Abbildung 1.11 deutlich wird. Hier soll das eingeblendete graue Trapez sich bündig zur Stabspitze (am unteren Bildrand) bewegen. Auf dem linken Bild ist die Szenerie in Stillstand, auf dem rechten in Bewegung zu sehen.

Abbildung 1.11: Dynamische Fehler werden erst bei Bewegung sichtbar

Meist akkumulieren sich auch die Fehler, da die Grundsysteme, auf denen AR aufgebaut wird, nicht für Echtzeitfähigkeit ausgelegt sind. Viele Systeme sind eher für den Durchsatz hoher Datenströme geeignet. Systemverzögerungen sind derzeit das größte Problem in AR. Um dem entgegen zu wirken, kann eine Umsetzung der folgenden Lösungsvorschläge helfen.

Verringerung der Systemlatenz: Der Einsatz von Echtzeitsystemen kann hilfreich sein, würde aber AR-Systeme zu sehr spezialisieren. Besser ist es, Verzögerungen im System zu verringern, was in Zukunft durch leistungsfähigere Komponenten möglich sein wird. Eine Eliminierung der Systemlatenz hingegen ist unmöglich.
Verringerung der sichtbaren Latenz: Zur Verringerung dieser Latenz wird folgender Ansatz vorgeschlagen:

1. Bei der Aufnahme erfasst das Kamerasystem ein größeres Blickfeld als nötig
2. Der Szenengenerator berechnet das Bild für dieses größere Blickfeld
3. Die Kopfposition des Benutzers wird im letzten Moment abgegriffen
4. Die große Szene wird zurechtgeschnitten und dargestellt

Durch eine geschickte Staffelung der Arbeitsschritte kann somit die sichtbare Latenz verringert werden.
Anpassen des zeitlichen Ablaufs: Wie bereits bei Abschnitt 1.4.1 erwähnt, ist eine zeitliche Abpassung der Videoströme möglich. Der erweiternde Videostrom richtet sich in seinem Timing dabei nach dem eingehenden Videostrom.
Vorausberechnung: Die eingehenden Trackingdaten der Kopfposition des Benutzers können zur Vorausberechnung der künftigen Kopfposition benutzt werden. Diese Vorausberechnung kann bei der Bilderzeugung eingesetzt werden.

Kalibrierung

Bei der Kalibrierung soll eine Anpassung des Trackings und der Bildgenerierung in Verhältnis zum Benutzer vorgenommen werden. Dabei werden verschiedene Sichtparameter des Benutzers durch Trackingdaten der Kopfposition ermittelt. Interessant sind Daten wie Pupillendistanz, Parallaxewinkel (Schielung), Gesichtsfeld, verschiedene Offsets und so weiter. Durch eine korrekte Kalibrierung sind genauere Augmentierungen möglich, die räumliche Wirkung

von virtuellen Objekten wird verbessert, wie auch die Plastizität. Besonders bei optischen Darstellungssystemen ist eine genaue Kalibrierung nötig, da die virtuellen Objekte in das normale Sichtfeld des Anwenders eingeblendet werden. In Videosystemen ist keine so genaue Kalibrierung nötig, da die Umwelt dort wie auf einer Leinwand erlebt wird, die in einer gewissen Distanz zum Benutzer schwebt. Dort beschäftigt sich die Kalibrierung eher damit Verzerrungen des Kamerasystems aufzuheben.

1.5 Gibt es eine allgemeine Benutzeroberfläche?

Diese Frage stellt sich, wenn ein Vergleich zu VE gezogen wird. VE sind spezielle Umgebungen, die um eine Aufgabe herum aufgebaut werden. Es sind stationäre Highendsysteme die nur von qualifiziertem Personal verwendet werden können. Außerdem sind sie sehr teuer (angefangen bei 25.000Euro bis 1.000.000Euro). AR-Systeme hingegen verwenden mobile Endgeräte, wie Laptops in Verbindung mit handelsüblichen Darstellungsgeräten und Eingabegeräten (wenn überhaupt), die in der Summe ein Bruchteil von aktuellen VR-Systemen kosten (1.000Euro bis 10.000Euro). Es stellt sich deshalb die Frage ob AR-Systeme auf einen Anwendungsbereich spezialisiert werden sollen, oder ob eine allgemeine Oberfläche, ähnlich den Desktopsystemen der PCs, entwickelt werden soll. Ein solcher Ansatz ist beispielsweise das Studierstube Projekt der TU Wien ([Schmalstieg '02]).

1.5.1 Studierstube

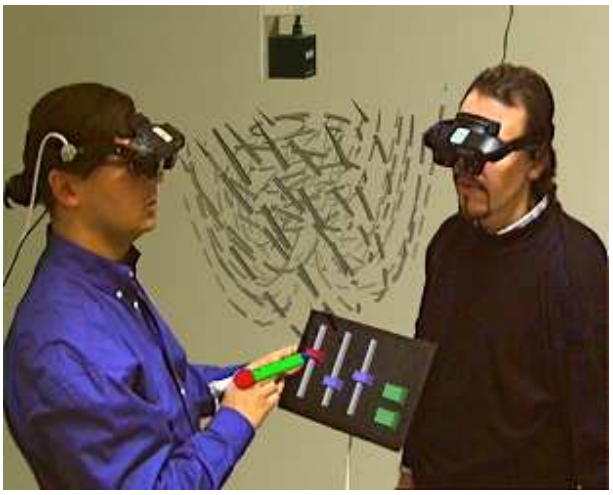


Abbildung 1.12: 2 Personen bearbeiten ein 3D-Datensatz in der Studierstube Umgebung

Das Ziel dieses Projekts ist der Entwurf einer 3D Benutzerschnittstelle, die, äquivalent zur 2D-Oberfläche aktueller PCs, eine Plattform für den täglichen Gebrauch bietet. Sie soll Systemzugriffe ermöglichen, Anwendungen verwalten können, Multitasking und Multiuser fähig sein und Zusammenarbeit zwischen Anwendern ermöglichen. Im Zuge dieser Entwicklung wurde ein Eingabesystem entworfen, die PIP, das eine zweihändige Eingabe erfordert. Die Konfiguration besteht aus einer mit Markern versehenen Tafel und einem getrackten Stift. Sie wurde gewählt weil sie eine vertraute Kombination für den Menschen ist (Papier und Bleistift) und ein haptisches Feedback liefert. Anwendungen bestehen in diesem System aus einer 3D-Repräsentation

der Anwendungsdaten innerhalb einer Box, welche dem Fenstersystem der 2D-Oberflächen nachempfunden ist. Die Anwendungen können verschoben, verkleinert, vergrößert und iconifiziert werden (wobei sie durch ein Icon auf der PIP dargestellt werden). Zur Zusammenarbeit kann die PIP weitergereicht werden, wobei persönliche Daten in der Augmentierung des Mitarbeiters nicht angezeigt werden (womit auch ein gewisser Datenschutz gewährt wäre).
Das Studierstubeprojekt verwendet zudem ein Konzept, das sich Locales nennt. Ähnlich den grafischen Oberflächen von PCs, die mehrere Desktops unterstützen, kann es im Studierstube System mehrere Locales geben, die jeweils Anwendungen oder Replikate bestehender Anwendungen enthalten. Die Replikate werden vom System synchronisiert. Mehrere Darstellungssysteme können sich ein Locale teilen, wie es beispielsweise bei der Zusammenarbeit vorkommen kann. Es können aber nicht mehrere Locales auf einem Monitorsystem gezeigt werden. Durch dieses System wird das Multitasking und die Multiuserfähigkeit erweitert.
Insgesamt unterstützt das Studierstube System viele Eigenschaften einer allgemeinen Benutzeroberfläche, was vor allem durch das Multitasking hervorgehoben wird. Ein besonderer Vorteil ist auch die mögliche Zusammenarbeit mehrerer Benutzer. Auch ein Fernzugriff wird ermöglicht, womit die Anwender nicht einmal den selben Raum teilen müssen, um zusammenarbeiten zu können. Leider handelt es sich bei dem Projekt noch um einen Prototypen. Es werden Aspekte der Multiuser-, Multitasking-, Multilocalefähigkeit beachtet aber für Mobilität ist das System derzeit noch nicht ausgelegt.

1.6 Ausblick

Trotz des günstigen Hardwareaufbaus von AR-Systemen sind derzeit keine kommerziellen Pakete erhältlich. Das liegt nicht zuletzt daran, dass es noch zu viele Softwareprobleme gibt, wie beispielsweise die Registrierung und die Benutzerschnittstelle. Hinzu kommen Trackingprobleme, vor allem im Outdoorbereich. Eine Erhöhung der Genauigkeit in diesem Bereich würde viele aktuelle Probleme lösen. Derzeit werden die meisten zivilen AR-Systeme als Prototypen und Versuchskonfigurationen in Forschung und Entwicklung eingesetzt. Das erste kommerzielle AR-System könnte der von [Azuma '97] angesprochene Prototyp von Boeing zur Herstellung von Kabelbäumen sein.
Im Militär werden bereits HMS in Kampfjets und Hubschraubern eingesetzt. Auch wenn diese Systeme nicht gleich für den zivilen Einsatz zugänglich sind, liefern sie zumindest Ideen und die Gewissheit, dass eine robuste Umsetzung der AR-Technik möglich ist. Besonders positiv ist auch das hohe Forschungsinteresse, das sich seit den letzten Jahren fortsetzt. Bis aber ein Augmented Reality System realistische virtuelle Objekte fehlerlos in die Realität einfügt, wird noch einige Zeit vergehen.

Literaturverzeichnis

[Azuma '97] R. Azuma. A survey of augmented reality. 1997.

[Schmalstieg '02] Werner Pugathofer Michael Gervautz L Miguel Encarnacao Zsolt Szalavari Gerd Hesina Anton Fuhrmann Dieter Schmalstieg. The Studierstube Augmented Reality Project. 2002.

[Veigl '02] Dieter Schmalstieg Gerhard Reitmayr Florian Ledermann Andreas Kaltenbach Stephan Veigl. Two-Handed Direct Interaction with ARToolkit. 2002.

[Özbek '03] Christopher S. Özbek. Spielerische Evaluierung eines Augmented Reality Systems. 2003.

2 Bewegungsvorhersage

Thomas Hettler

2.1 Einführung

2.1.1 Motivation

Bei der Aufnahme von Motion-Capturing-Daten kann nicht davon ausgegangen werden, dass die Daten fehlerfrei und absolut präzise sind.
Eine Nachbearbeitung ist nötig, um die daraus resultierenden Störungen (ruckartige Bewegungsanimationen, unnatürliche Gelenkwinkel) zu beheben.
In vielen Fällen (z.B. Virtual Environments) liegen nicht die kompletten Daten vom Anfang bis zum Ende der Aufnahme vor, sondern es muss direkt aus den alten und aktuellen Messwerten eine Nachbearbeitung erfolgen, um einen realistischen und natürlichen Bewegungsablauf zu generieren.
Des Weiteren besteht Bedarf an einer Technologie, die es ermöglicht, verschiedene Aufnahmen zu verketten. Dabei sollte auch der Übergang zwischen den Aufnahmen in der Bewegungsanimation flüssig und realistisch sein.

2.1.2 Nachbearbeitung der Daten mit Kalman-Filtern

In
[Sul et al. '98] stellen die Autoren eine Technik vor, die diese Anforderungen erfüllt. Zur Erzeugung von animierten Bewegungsabläufen wird dabei der Kalman-Filter verwendet, wobei die Eingabe die Messdaten und die Ausgabe die Daten zur Animation des Modells sind.
Um einen natürlichen Bewegungsabblauf zu erhalten, werden physikalische Gegebenheiten (z.B. die Massenträgheit oder Beschränkungen an den Gelenken des Modells) berücksichtigt.

2.2 Körpermodell

Zunächst wird die Modellierung des menschlichen Körpers beschrieben, wie sie in [Sul et al. '98] verwendet wurde.
Der Körper mit seinen Bewegungsmöglichkeiten wird durch eine Menge starrer Segmente (Brustkorb, Oberarm, Unterarm, ...), die durch Gelenke verbunden sind, repräsentiert.
Bildet man einen Graphen mit den starren Segmenten als Knoten und den Gelenken als Kanten, so ist dieser zyklenfrei und kann als Baum aufgefasst werden.
Eine Pose des Modells ist eindeutig bestimmt durch Translation und Rotation des Wurzelsegments (z.B. des Beckens), sowie durch die Rotationen jedes einzelnen Kindsegments bezüglich seines Elternsegments.
Neben Informationen über die Rotation zum Elternsegment besitzt jedes Segment außer dem Wurzelsegment Daten über Beschränkungen dieser Bewegung.

Abbildung 2.1: Körpermodell

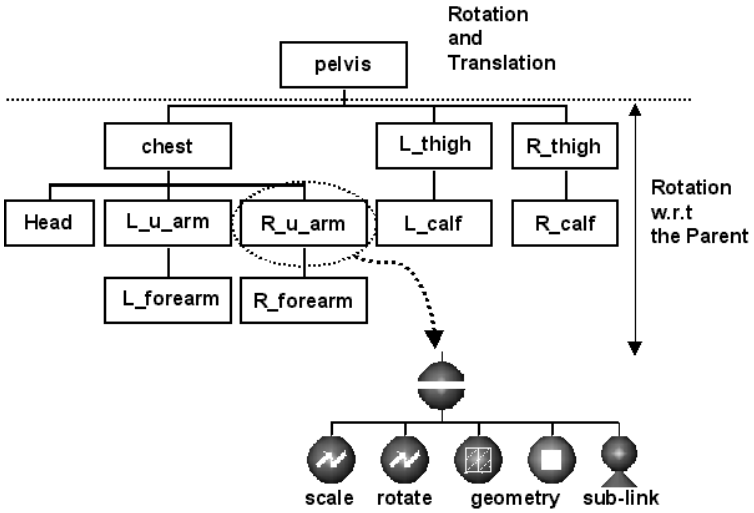


Abbildung 2.2: Segmente und Knoten als Baum

Diese Beschränkungen beziehen sich auf die Anzahl der rotativen Freiheitsgrade des Gelenks und seiner maximalen Auslenkungen bezüglich dieser Freiheitsgrade (siehe Abschnitt 6).

2.3 Eine Einführung in den Kalman-Filter

Im Jahr 1960 veröffentlichte R.E. Kalman sein berühmtes Paper [Kalman '60], welches eine rekursive Lösung für das Problem der linearen Filterung von diskreten Daten beschrieb.
Später wurde das Verfahren zum Extended Kalman-Filter erweitert, um Daten auch nicht linear filtern zu können.
Seitdem wurde ausgiebig Forschung mit dem Filter betrieben und es wird in einer Vielzahl von Anwendungen eingesetzt.
Im Folgenden wird der einfachere lineare Fall betrachtet.

2.3.1 Gegebenheiten für den Einsatz des Filters

Es werde ein Vorgang der realen Welt durch fehlerbehaftete Messungen überwacht. Die Aufgabe des Filters ist es, von einer Messsung auf eine möglichst präzise Schätzung des Zustands des Vorganges zurückzuschließen.
Der Zusammenhang von physikalischem Zustand und der Messung sei gegeben durch:

$$z_t = H \cdot s_t + v_t \tag{2.1}$$

- z_t ist der Messvektor zum Zeitpunkt t.
- s_t ist der Zustandsvektor des Vorganges zum Zeitpunkt t.
- v_t ist der Messfehlervektor zum Zeitpunkt t.
- H ist die Matrix, welche vom Zustand auf die Messung abbildet.

Die Zustände des Vorgangs zu den Zeitpunkten der Messungen seien nun nicht unabhängig voneinander, sondern es gebe eine ungenaue Vorhersagefunktion, die vom letzten Zustand auf den nächsten Zustand abbilde. Diese Vorhersagefunktion sei aus der Natur des Vorganges bekannt.

$$s_t = A \cdot s_{t-1} + w_{t-1} \tag{2.2}$$

- s_t ist der Zustand zum Zeitpunkt t.
- $A \cdot s_{t-1}$ ist die Vorhersagefunktion (Matrix A) angewendet auf den letzten Zustand.
- w_{t-1} beschreibt den Fehler der Vorhersagefunktion zwischen s_{t-1} und s_t. Dieser Fehler kommt zum Beispiel durch äußere Einflüsse auf den Vorgang oder durch eine zu grob modellierte Vorhersagefunktion zustande.

Der Fehler der Messung sei mit Kovarianzmatrix R normalverteilt.
Der Fehler der Vorhersagefunktion sei mit Kovarianzmatrix Q normalverteilt.

2.3.2 Funktionsweise

Der Filter besteht aus einer Menge Gleichungen, die in derselben Reihenfolge immer wieder abgearbeitet werden.
Ein solcher Zyklus lässt sich in zwei Teile, Vorhersage (“Time Update”) und Korrektur (“Measurement Update”), unterteilen.

Vorhersage

$$\hat{s}_t^- = A \cdot \hat{s}_{t-1} \tag{2.3}$$

$$P_t^- = A \cdot P_{t-1} \cdot A^T + Q_{t-1} \tag{2.4}$$

Im Vorhersageteil wird aus dem alten geschätzten Zustand $\hat{s}_{t-1}$ [1] mithilfe der Vorhersagefunktion eine a priori Schätzung $\hat{s}_t^-$ [2] des nächsten Zustandes erzeugt (2.3).
P_t^- ist die Kovarianzmatrix der Differenz zwischen wahrem Zustand s_t und dem a priori geschätzten Zustand $\hat{s}_t^-$.

[1] Das $\hat{}$ soll andeuten, dass es sich um Schätzungen handelt
[2] Hochgestelltes $^-$ bedeutet “a priori”

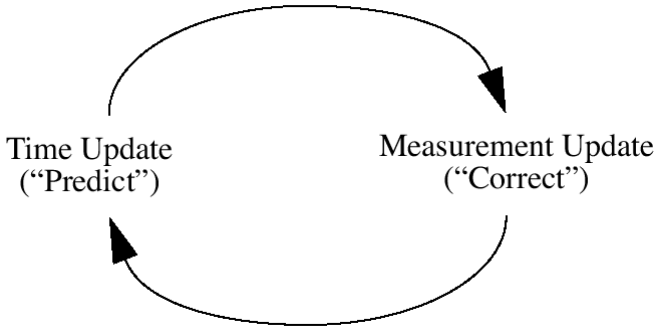


Abbildung 2.3: Der Zyklus des Kalman Filters

Korrektur

$$K_t = P_t^- \cdot H^T \cdot \left(H \cdot P_t^- \cdot H^T + R_t\right)^{-1} \tag{2.5}$$

$$\hat{s}_t = \hat{s}_t^- + K_t \cdot \left(z_t - H \cdot \hat{x}_t^-\right) \tag{2.6}$$

$$P_t = (I - K_t \cdot H) \cdot P_t^- \tag{2.7}$$

Im Korrekturteil wird aus den vorhandenen statistischen Daten in (2.5) der sogenannte "Kalman Gain" K_t berechnet.
Anschliessend wird in (2.6) die Differenz zwischen eingehender Messung z_t und der erwarteten Messung $\hat{z}_t = H \cdot \hat{s}_t^-$ gebildet.
Diese Abweichung wird Residuum ("innovation" oder "residual") genannt. Das Residuum wird durch den Kalman Gain K_t gewichtet und zu der a-priori-Schätzung des Zustandes $\hat{s}_t^-$ hinzuaddiert.
Auf diese Weise erhält man a posteriori eine Schätzung $\hat{s}_t$ des Zustandes aus Messung und Vorhersage und der nächste Zyklus kann beginnen.

Anmerkungen zum Kalman Gain

Zu beachten ist, dass die Gleichungen (2.4)(2.7)(2.5) nur dazu dienen, iterativ eine möglichst gute Gewichtung (Kalman Gain) zwischen Vorhersage und Messung zu errechnen.
Die eingehenden Messdaten haben dabei keinerlei Einfluss auf die Entwicklung der Gewichtung.
Im Falle, dass Messfehlerkovarianz R und Vorhersagefehlerkovarianz Q konstant sind, konvergiert der Kalman Gain und könnte sogar offline vorberechnet werden.

Um die Rolle des Kalman Gains deutlich zu machen, werden im folgenden zwei Fälle betrachtet:

- Die Messung ist sehr präzise. Daraus folgt eine kleine Messfehlerkovarianzmatrix R. Für den Kalman Gain K_t gilt wegen (2.5):

  $$\lim_{R_t \to 0} K_t = H^{-1}$$

  In (2.6) eingesetzt, ergibt sich:

  $$\hat{s}_t = H^{-1} \cdot z_t$$

Die Korrektur vertraut also auf die Messung und kaum der Vorhersage.

- Die a-priori-Fehlerkovarinazmatrix P_t^- ist klein und es wäre sinnvoll, die Vorhersage gegenüber der Messung bei der Gewichtung in (2.6) zu bevorzugen:

$$\lim_{P_t^- \to 0} K_t = 0$$

In (2.6) eingesetzt, ergibt sich:

$$\hat{s}_t = \hat{s}_t^-$$

Die Korrektur vertraut also auf die Vorhersage.

Initialisierung

Für den ersten Zyklus des Filters werden eine alte Zustandsschätzung $\hat{s}_{t-1}$ und eine Kovarianz P_{t-1} zwischen dieser alten Schätzung $\hat{s}_{t-1}$ und dem alten tatsächlichen Wert s_{t-1} benötigt. Sie müssen passend gewählt werden. Eine von der Realität abweichende Wahl wird allerdings in den folgenden Zyklen automatisch iterativ angepasst.

2.4 Filterung translativer Bewegungen

Dieser Abschnitt befasst sich mit der Anwendung des Kalman Filters auf die Translationsbewegungen des Körperwurzelsegments (das einzige Segment, bei dem Translationen gemessen werden).
Um Verwirrung zu vermeiden werden die Notationen und Bezeichner aus dem vorhergehenden Abschnitt übernommen.

2.4.1 Der Zustandsvektor

Der Zustand des Wurzelsegments bezüglich Translationen wird angegeben durch

- einen Ortsvektor $(s_x, s_y, s_z)^T$
- einen Geschwindigigkeitsvektor $(v_x, v_y, v_z)^T$

  Die Einheit der Geschwindigkeit ist Längeneinheiten pro Messintervall.

Zusammen ergibt sich:

$$s = (s_x, s_y, s_z, v_x, v_y, v_z)^T$$

2.4.2 Messung und Zustand

In diesem Fall ist der Unterschied zwischen Zustand s und Messung z deutlich ersichtlich. Der Zustand enthält auch Informationen über die Geschwindigkeit, während die Messung nur Informationen über die Position liefert.

$$z_{\text{fehlerfrei},t} = \begin{pmatrix} I_{3\times3} & 0_{3\times3} \end{pmatrix} \cdot s_t = H \cdot s_t$$

$H = (I_{3\times3} 0_{3\times3})$ beschreibt den Zusammenhang zwischen Messung und Zustand.

2.4.3 Die Vorhersagefunktion

Bei der Vorhersage wird einfach davon ausgegangen, dass das Wurzelsegment, der Newtonschen Massenträgheit folgend, seine Geschwindigkeit in Betrag und Richtung beibehält.

$$\hat{s}_k^- = \begin{pmatrix} \hat{s}_{x,k-1} + \hat{v}_{x,k-1} \\ \hat{s}_{y,k-1} + \hat{v}_{y,k-1} \\ \hat{s}_{z,k-1} + \hat{v}_{z,k-1} \\ \hat{v}_{x,k-1} \\ \hat{v}_{y,k-1} \\ \hat{v}_{z,k-1} \end{pmatrix} = \begin{pmatrix} I_{3\times3} & I_{3\times3} \\ 0_{3\times3} & I_{3\times3} \end{pmatrix} \cdot \hat{s}_{k-1} = A \cdot \hat{s}_{k-1}$$

2.4.4 Der Vorhersagefehler

Im behandelten Fall kommt der Fehler w_{t-1} (siehe Gleichung 2.2) durch Beschleunigungen zustande, die ihre Ursache in Wechselwirkungen mit den anliegenden Kindsegmenten und äußeren Kräften (Schwerkraft, Wechselwirkungen mit der Umwelt) haben.
Die Vorhersagefehler-Kovarinzmatrix Q_{t-1} könnte also vor der Aufnahme durch statistische Daten über die zu erwartenden translativen Beschleunigungen des Wurzelsegments abgeschätzt werden.
Die zu erwartenden Beschleunigungen sind sicher von der Lebendigkeit der aufzunehmenden Szene abhängig und die Vorhersagefehlerkovarianzmatrix Q_{t-1} kann dementsprechend angepasst werden.

2.4.5 Der Messfehler

Der Messfehler kommt durch Sensorfehler bei der Aufnahme zustande. Der Aufbau des Filters erlaubt es während seiner Anwendung die Messfehler-Kovarianzmatrix R_t zu verändern, um beispielsweise eine veränderte Entfernung zu den Sensoren auszugleichen.

2.4.6 Einbau in den Filter

Wenn man die vorherigen Matrizen und Werte in den Filter einbaut, tritt ein Problem auf.
In (2.6) wird deutlich, dass die Messung keinen Einfluss auf die Geschwindigkeiten im neuen Zustandsvektor hat.
Im Korrekturteil ist daher eine weitere Gleichung erforderlich, die die Geschwindigkeiten aktualisiert, nachdem $\hat{s}_t$ bestimmt wurde.

$$\hat{s}_t = \begin{pmatrix} \hat{s}_{x,t} \\ \hat{s}_{y,t} \\ \hat{s}_{z,t} \\ \hat{s}_{x,t} - \hat{s}_{x,t-1} \\ \hat{s}_{y,t} - \hat{s}_{y,t-1} \\ \hat{s}_{z,t} - \hat{s}_{z,t-1} \end{pmatrix} = \begin{pmatrix} I_{3\times3} & 0_{3\times3} \\ I_{3\times3} & 0_{3\times3} \end{pmatrix} \cdot \hat{s}_t + \begin{pmatrix} 0_{3\times3} & 0_{3\times3} \\ -I_{3\times3} & 0_{3\times3} \end{pmatrix} \cdot \hat{s}_{t-1}$$

2.5 Filterung rotativer Bewegungen

Ebenso wie die Translation des Wurzelsegments werden die Ausrichtungen sämtlicher Gelenke mithilfe des Kalman-Filters nachbearbeitet.

2.5.1 Der Zustandsvektor

Der Zustand jedes einzelnen Gelenks wird angegeben durch

- Die Ausrichtung des am Gelenk anliegenden Kindsegments bezüglich seines Elternsegments als Quaternion q.
- Die momentane Drehachse und die Winkelgeschwindigkeit als Eulerparameter Ω.

Als Gesamtzustand ergibt sich also:

$$s = (q, \Omega)^T = (q_w, q_x, q_y, q_z, \omega_x, \omega_y, \omega_z)^T$$

2.5.2 Messung und Zustand

Für die Messung gilt:

$$z_{\text{fehlerfrei},t} = H \cdot s_t$$

mit

$$H = \begin{pmatrix} I_{4\times4} & 0_{4\times3} \end{pmatrix}$$

2.5.3 Die Vorhersagefunktion

Es wird davon ausgegangen, daß sich die momentane Drehung weiter fortsetzt (Rotationsträgheit).

$$\hat{s}_t = \begin{pmatrix} w \otimes q_{t-1} \\ \hat{\Omega}_{t-1} \end{pmatrix} \tag{2.8}$$

w ist dabei daß zu $\hat{\Omega}_{t-1}$ gehörende Quaternion und $\otimes$ die Quaternionenmultiplikation.
Die Vorhersagefunktion ist in diesem Falle nicht linear, d.h. es existiert keine Vorhersagematrix A. Im erweiterten Kalman-Filter (EKF) wird die nichtlineare Gleichung ähnlich wie bei Taylorpolynomen erster Ordung linearisiert:
Anstatt der Matrix A wird die Jacobimatrix A_t der Vorhersagefunktion verwendet. Sie muss in jedem Zyklus des Filters aktualisiert werden.
A_t wird nur in (2.4) benötigt, die Gleichung sieht dann folgendermaßen aus (2.9).

$$P_t^- = A_t \cdot P_{t-1} \cdot A_t^T + Q_{t-1} \tag{2.9}$$

Als Vorhersagefunktion (2.3) kann die nichtlineare Funktion verwendet werden (2.8).

2.5.4 Zusatzbemerkung

Ähnlich wie bei der Filterung von Translationen muss auch bei Rotationen am Ende der Korrektur in jedem Zyklus die momentane Drehrichtung und -geschwindigkeit $\hat{\Omega}_t$ aktualisiert werden, weil die momentane Drehrichtung und -geschwindigkeit nicht gemessen und daher bei der Gewichtung (2.6) vernachlässigt wird.

2.6 Einbinden von Beschränkungen an den Gelenken

Die Einschränkungen von Gelenken des menschlichen Körpers sind im Körpermodell nicht hart kodiert.
Die Autoren von [Sul et al. '98] verwenden stattdessen eine andere Methode. Sie führen Fehlerfunktionen ein, welche abhängig vom Zustandsvektor des jeweiligen Gelenks sind.
Bei Einhaltung der Beschränkungen liefern diese Fehlerfunktionen minimale Werte, ansonsten wachsen sie mit dem Ausmaß der Überschreitung.
Mithilfe der partiellen Ableitungen der Fehlerfunktionen lassen sich in einem Gradientenverfahren künstliche Messungen generieren, die in den Filter einfliessen und die gefilterte Bewegung einer Einhaltung der Beschränkungen annähert.
Im Folgenden sei $w^{'}$ die Eulerparameterdarstellung zum Quaternion q im Zustandsvektor. Zu der momentanen Drehrichtung und -geschwindigkeit Ω gelten ähnliche Fehlerterme.

2.6.1 Zwei rotative Freiheitsgrade

Ein Gelenk habe zwei Freiheitsgrade und es seinen n_1, n_2 die zugehörigen Vektoren.
Es stehe ein Vektor v auf n_1, n_2 senkrecht ($n_1 \cdot v = n_2 \cdot v = 0$).

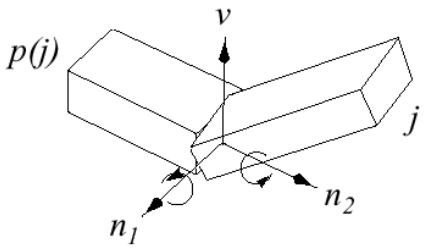


Abbildung 2.4: Zwei rotative Freiheitsgrade

Der Fehler E_{2DOF} ist dann gegeben durch:

$$E_{2DOF} = w^{'} \cdot v$$

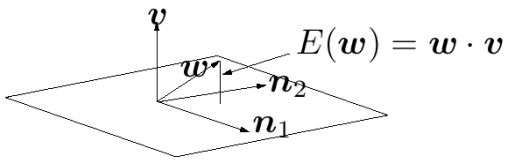


Abbildung 2.5: der Fehler

2.6.2 Ein rotativer Freiheitsgrad

Das Gelenk habe nur einen Freiheitsgrad. Dementsprechend gibt es zwei Vektoren v_1, v_2 senkrecht zu n.

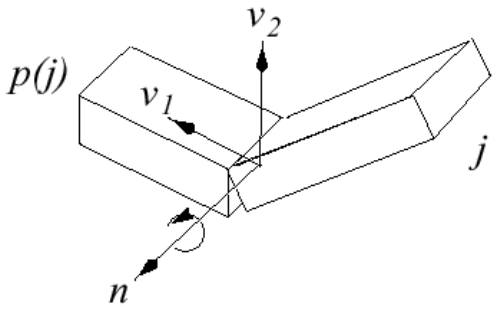


Abbildung 2.6: Ein rotativer Freiheitsgrad

Der Fehler E_{1DOF} ist dann gegeben durch:

$$E_{1DOF} = \begin{pmatrix} w^{'} \cdot v_1 \\ w^{'} \cdot v_2 \end{pmatrix}$$

2.6.3 Grenzwinkel

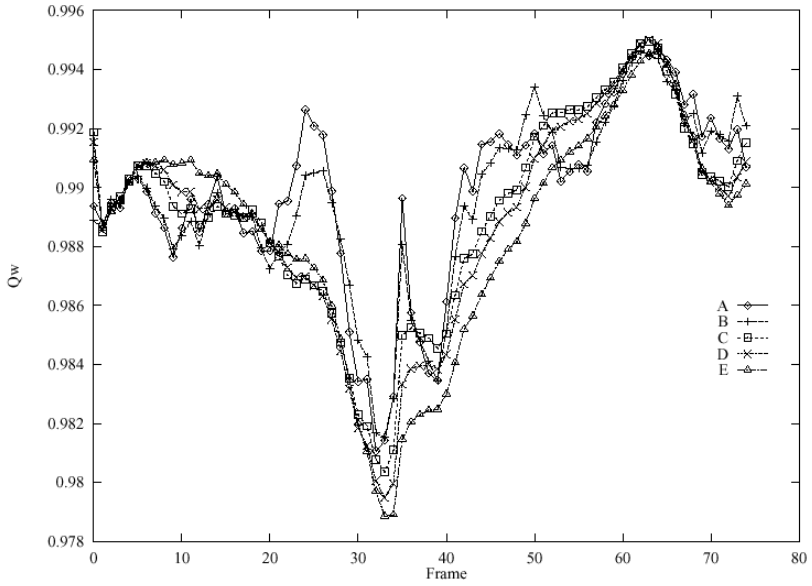
Der Rotation bezüglich einem Freiheitsgrad mit zughörigem Vektor n sei nun nur zwischen zwei Winkeln θ_1 und θ_2 möglich.
Der Fehler E_{Winkel} ist dann gegeben durch:

$$E_{Winkel} = \max\left(0, \theta_1 - w^{'} \cdot n, w^{'} \cdot n - \theta_2\right)$$

2.7 Experimente

In diesem Abschnitt werden Experimente aus [Sul et al. '98] vorgestellt. Die Autoren verwendeten ein eigenes System zur Aufnahme.

2.7.1 Messunsicherheiten



Kurve	Messunsicherheit
A	0.0 (keine Filterung)
B	0.001
C	0.1
D	0.2
E	0.5

Abbildung 2.7: Verlauf des ersten Quaternionenteils q_w eines Brustsegments bei verschiedenen Messunsicherheiten

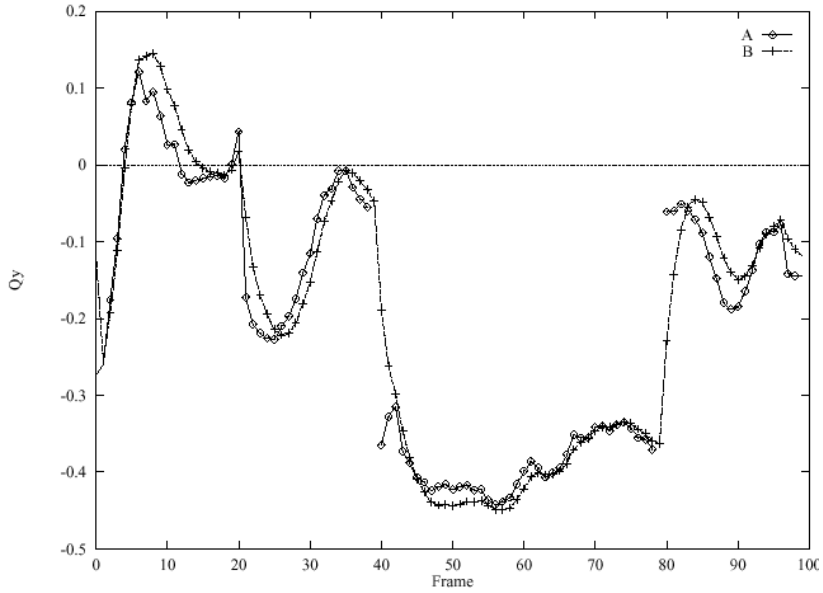
Abbildung 2.7 zeigt wie mit ansteigender Messunsicherheit (Messfehlerkovarianz) die Daten geglättet werden.
So wird beispielsweise bei hoher Messunsicherheit einem aussergewöhnlichen Wert wie bei Frame 35 wenig Vertrauen geschenkt, sondern die Vorhersage stärker gewichtet.

2.7.2 Verkettung von Aufnahmedaten

In Abb. 2.8 wurde zwischen das 40te und 41te Frame der Aufnahme einer gehenden Person eine zweite Aufnahme, die einer Bückbewegung, eingefügt.
Auf die entstandene Kurve (A) wurde der Filter angewendet um einen nahtlosen, natürlichen Übergang zu erhalten (B).
Der Filter sorgt dabei dafür, dass keine unnatürlich hohen Beschleunigungen auftreten und die Beschränkungen an den Gelenken eingehalten werden.

2.7.3 Gegenüberstellung gefiltert - ungefiltert

Abbidung 2.9 stellt Ausschnitte einer Bewegungsanimation, welche der beschriebenen Nachbearbeitung unterzogen wurde, der ohne Nachbearbeitung gegenüber.
Insbesondere an den Beinen ist ersichtlich, dass ohne Filterung starke Veränderungen, die wahrscheinlich durch Messfehler zustande kommen, im Übergang von einem Frame zum nächsten auftreten.
Auch seltsame Gelenkwinkel wie in Frame 47 wurden durch die Nachbearbeitung entfernt.



Kurve A: simple Verkettung von Aufnahmen
Kurve B: Ergebnis der Filterung

Abbildung 2.8: Transition

Frame 45 bis 48
oben: keine Nachbearbeitung der Aufnahme.
unten: Bewegungsanimation mit gefilterten Daten

Abbildung 2.9: Vergleich von gefilterter und ungefilterter Bewegungsanimation

2.8 Fazit und Ausblick

In [Sul et al. '98] wurde eine Methode zur Nachbearbeitung von Daten aus einem Motion-Capturing-Verfahren vorgestellt, welches leistungsfähig genug ist, um in Echtzeit angewandt zu werden.
Diese Methode erfüllt drei Aufgaben

- Glättung der ruckartigen Bewegungen, die durch Sensorfehler hervorgerufen werden.
- Einbeziehung der Beschränkungen des menschlichen Körpers an den Gelenken.

- Generierung flüssiger Übergänge bei der Verkettung von Datensequenzen aus verschiedenen Aufnahmen.

Alle drei Aufgaben werden mithilfe des Kalman Filters [Kalman '60] bewältigt.
Die Erstellung animierter Bewegungsabläufe lässt sich durch eine Gewichtung der Gelenkbeschränkungen, die Unsicherheit der Messung beziehungsweise die Unsicherheit der Bewegungsvorhersage beeinflussen.
Die Autoren von [Sul et al. '98] planen ihr Verfahren zu erweitern und Wechselwirkungen innerhalb des Körpermodells zu berücksichtigen, indem sie als weitere Messungen einbezogen werden.
Des Weiteren ist geplant, eine Übertragung von Bewegungen auf anders proportionierte Körpermodelle zu ermöglichen.

2.9 Anhang: Rotationen im dreidimensionalen Raum

Dieser Abschnitt handelt von Rotationen um den Ursprung im dreidimensionalen Raum. Im Gegensatz zu Rotationen in der Ebene ist die Verknüfung von Drehungen in drei Dimensionen nicht mehr kommutativ.
Rotiert man beispielweise in einem orthonormalen Koordinatensystem einen Punkt der x-Achse zuerst um 90° um die y-Achse und anschließend um 90° um die z-Achse, so befindet sich der Punkt auf der z-Achse. Vertauscht man die Reihenfolge der Einzeldrehungen, so befindet sich der Punkt auf der y-Achse.
Im Folgenden werden 2 Arten der Darstellung einer Rotation angeführt:

2.9.1 Eulerparameterform

Eine Rotation wird als Vektor $\Omega = (\omega_x, \omega_y, \omega_z)$ angegeben. Die Richtung der Drehachse ist durch die Ausrichtung von Ω bestimmt. Der Betrag von Ω ist dabei der Winkel der Drehung im Bogenmaß.

2.9.2 Quaternionen

Ein Quaternion $q = (q_w, q_x, q_y, q_z)$ ist ein 4-Tupel aus einem Realteil und drei Imaginärteilen. Ist dieses Quaternion normiert, so lässt es sich als Rotation interpretieren.
Ein besonderer Vorteil von Quaternionen ist, dass das Ergebnis der nichtkommutativen Quaternionenmultiplikation $r \otimes q$ genau die Rotation angibt, die durch die Hintereinanderausführung der zu q und r gehörenden Rotationen entsteht.

$$r \otimes q = \begin{pmatrix} r_w q_w - r_x q_x - r_y q_y - r_z q_z \\ r_w q_x + r_x q_w + r_y q_z - r_z q_y \\ r_w q_y - r_x q_z + r_y q_w + r_z q_x \\ r_w q_z + r_x q_y - r_y q_x + r_z q_w \end{pmatrix}$$

2.9.3 Umrechnung: Eulerparameter ↦ Quaternion

$$q = \begin{pmatrix} \cos\left(\|\Omega\|/2\right) \\ \sin\left(\|\Omega\|/2\right) \cdot \frac{\omega_x}{\|\Omega\|} \\ \sin\left(\|\Omega\|/2\right) \cdot \frac{\omega_y}{\|\Omega\|} \\ \sin\left(\|\Omega\|/2\right) \cdot \frac{\omega_z}{\|\Omega\|} \end{pmatrix}$$

2.9.4 Umrechnung: Quaternion ↦ Eulerparameter

$$\Omega = \frac{2 \arccos q_w}{\sin\left(\arccos q_w\right)} \cdot \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix}$$

Literaturverzeichnis

[Kalman '60] Emil Kalman, Rudolph. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering*, 82, Nr. Series D, S. 35–45, 1960.

[Sul et al. '98] ChangWhan Sul, SoonKi Jung und Kwangyun Wohn. Synthesis of Human Motion Using Kalman Filter. 1998. http://link.springer.de/link/service/series/0558/papers/1537/15370100.pdf (gesehen 2003).

[Welch & Bishop '02] Greg Welch und Gary Bishop. An Introduction to the Kalman Filter. 2002. http://www.cs.unc.edu/~welch/kalman/ (gesehen 2003).

3 Echtzeitsimulation

Özdem Heper

3.1 Einleitung

n diesem Papier geht es um zwei neue Verfahren der Computergrafik. Zum einen, ein Verfahren zur Simulation von Brüchen, welches auf der Methode der Finiten Elemente beruht. Zum anderen ein Verfahren zur dreidimensionalen Modellierung von Flammen, welches auf der Navier Stoke'schen Gleichung der inkompressiblen Fluidsysteme aufbaut. Außerdem werden drei gängige Modelle der Brandsimulation erklärt(Zonen-, Feld-, CFD-Modelle)
In meinem Seminar möchte ich zuerst den Bedarf an Simulationstechnik erklären, hierbei möchte ich verschiedene Anwendungsgebiete ansprechen und die Probleme die auftreten, wenn eine Simulation Echtzeit ablaufen soll.

3.2 Simulation

3.2.1 Was sind Simulationen:

Simulation *die*, **1.** Verstellung - **2.** *Medizin:* bewusste Vortäuschung von Krankheit. - **3.** Darstellung technischer, biologischer, ökonomischer u. a. Prozesse oder Systeme, durch (math.) Modelle. Simulationen erlauben Untersuchungen oder Manipulationen, deren Durchführung am eigentlichen System zu gefährlich, zu teuer oder anderweitig nicht möglich ist.[1]

Für die Betrachtungen in dieser Ausarbeitung sind Simulationen, welche mathematische Modelle darstellen interessant. Sie bieten Rückschlüsse auf die Realität, bzw. die Vorberechnung von Ereignissen. Eingesetzt werden Sie, falls ein Realversuch zu gefährlich,zu teuer, bzw. das Realobjekt bei einem Versuch irreparabel beschädigt oder zerstört werden könnte.
Mögliche Anwendungsgebiete von Simulationen sind z.B. Produktentwicklung / -optimierung (Crashtests, Aerodynamik usw.), Sicherheitstechnik / Gebäudesicherheit (Brand- / Erdbebensimulation), Wissenschaft (Wetter / Natur / Astrologie) und Unterhaltung (Spezialeffekte).
Dies ist natürlich nur ein Bruchteil der Anwendungsmöglichkeiten, denn der Einsatz von Simulationen ist heute zu vielfältig, und ihr Einfluss auf unser Leben zu groß. Sie würde den Rahmen dieser Ausarbeitung sprengen, wollte man alle aufzählen.

3.2.2 Echtzeit Simulationen

Um eine Simulation in Echtzeit zu berechnen gibt es momentan 2 Möglichkeiten.

[1] Quelle: www.brockhaus.de

- **Abstraktion/Vereinfachung:**
  Das zu Simulierende wird soweit abstrahiert, bzw. die Algorithmen soweit verbessert, dass die Berechnung und Darstellung des nächsten Zeitabschnitts zeitgleich oder schneller zum Realereignis steht. Ein Nachteil ist, daß die Simulation ungenau wird und nur grobe Aussagen treffen kann (Bsp. Feldmodell bei Brandsimulationen).

- **Pure Rechengewalt:**
  Die Parallelisierung des Algorithmus, welches die Simulation darstellt, ermöglicht anfänglich meist nur einen Leistungszuwachs um den Faktor 1,2 - 1,5 pro zusätzlichem Prozessor, der aufgrund des steigenden Kommunikationsaufwandes sich nicht beliebig fortsetzen läßt. Ein weiterer Nachteil ergibt sich aus den hohen Kosten.

Echtzeitsimulationen wären praktisch, sind aber in den meisten Fällen nicht erforderlich. Beispielsweise könnte bei einem Crashtest schnell verschiedene Materialeigenschaften für verschiedene Karosserieteile durchgespielt werden, um ein Maximum an Sicherheit zu garantieren. Da die Entwicklung eines Fahrzeugs in der Regel ein längerer Prozess ist, genügt es, wenn die Simulation die Echtzeitanforderung nicht erfüllt. Man kann die nötigen Berechnungen auch über größere Zeiträume, während der Entwicklungsphase durchführen.
Die Konzepte die ich in meinem Seminar vorstellen möchte erfüllen alle nicht die Echtzeitanforderung und sind auch nicht zur wissenschaftlichen Forschungen geeignet. es handelt sich um visuelle Effekte, die in der Unterhaltungsindustrie Verwendung finden sollen.
Jedes Frame der vorgestellten Animationen würde auf einem derzeitigen (P4 2GHz) Heimcomputer im Durchschnitt 4 Minuten gerendert werden. Eine weitere Möglichkeit ist, zu hoffen das die Computerindustrie in den nächsten Jahren immer leistungsfähigere Rechner auf den Markt bringt, bzw. bessere Technologien (Stichwort Quantencomputer usw.) entwickelt um aufwändige Simulationen auch für Heimanwender zu ermöglichen.

3.2.3 Methoden der Animation:

Es gibt drei Methoden der Computeranimation und jedes hat Vor- und Nachteile.

- **1. Keyframing:**
  Hier werden die Eckdaten der Bewegung des zu animierenden Objektes manuell eingegeben und die Zwischenschritte interpoliert. Nachteil: sehr hoher Arbeitsaufwand bei der Erstellung einer Animation.

- **2. Motion Capturing:**
  Die Bewegungen eines Realobjekts bzw. einer Person werden aufgezeichnet und extrahiert. Nachteil: hohe Datenfülle, Unüberschaubarkeit der Bewegungen.

- **3. Prozedurale Methoden:**
  Animationen werden mittels Algorithmen erzeugt. Nachteil: sehr hoher Rechenaufwand.

Verfahren 1 und 2 haben den Vorteil, dass eine Animation sofort in Echtzeit ausgeführt werden kann. Gegenpol dazu ist der hohe Arbeitsaufwand, um die Animation überhaupt zu erzeugen. Verfahren 3 ist für die Modellierung sehr günstig. Der Modellierer muss zum Beispiel einfach

ein Objekt als zerbrechlich definieren, die Randdaten festlegen und schon ist das Objekt in der virtuellen Welt zerbrechlich. Großes Manko dieser Methode ist der meist hohe Rechenaufwand.
Achtung: Prozedurale Methoden müssen nicht physikalisch begründet sein. Falls die physikalische Korrektheit der Simulation keine Rolle spielt ist es auch möglich über prozedurale Methoden sureale Effekte zu erzielen. Dies ist beispielsweise bei Filmen der Fall (z.B. Matrix [Champagne '03]). Hier kommt es eher auf die optische Wirkung an und die Effekte sind darauf ausgerichtet, den Kinobesucher zu beeindrucken.

3.2.4 physikalisch basiertes Modellieren

Ein physikalisch basiertes System ist ein Teilaspekt der prozeduralen Systeme. Die Zielsetzung ist hierbei, physikalische Bedingungen im virtuellen Raum zu implementieren, so daß physikalisch Effekte wie Gravitation, Aerodynamik, Thermodynamik, Impulserhaltung usw. auch in dieser virtuellen Umgebung gelten. Dadurch lässt sich die Modellierung komplexer Animationen, wie z.B. beim Zerbrechen eines Objekts durch einfache Definition von Materialeigenschaft und Randdaten erstellen. Beispielsweise wurde im Film Antz 1994 eine Wasseroberfläche prozedural animiert. Die folgenden Simulationen verwenden zum einen die Methode der Finiten Elemente (Bruch Simulation) und die Navier Stoke'schen Gleichung für inkompressible Fluid Stoffe (Feuer). Beide Verfahren finden auch bei Brandsimulationen und bei der Wettervorhersage Verwendung.

3.3 Virtuelle Brüche

Abbildung 3.1: Tipsy Turvy -IBM 1989

Bruchsimulationen sind nun keine Neuerfindung in der Computersimulation. Bereits seit 1988 (damals bei IBM) wurde das Utah Teapot zertrümmert (Abb.3.1).
Unschön an dieser Simulation war die starke Fragmentierung an der Bruchstelle und die geringe Palette an simulierbaren Materialeigenschaften. Die starke Fragmentierung entstand, da ein Knoten einfach in 2 Knoten aufgetrennt wurde. Mitarbeiter an diesem Projekt war damals auch Grag Turk der später auch den Stanford Hasen eingescannt hat, welches als Versuchsopfer für O'Briens Bruchexperimente hinhalten musste, doch dazu später mehr.

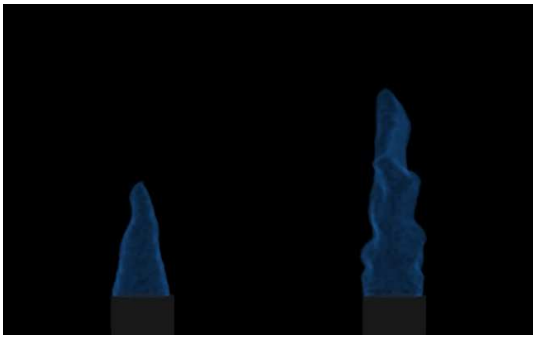


Abbildung 3.2: Eine Mauer wird von einem Objekt getroffen

3.3.1 Die Finite Element Methode

Die Methode der Finiten (Endlichen) Elemente teilt einen Körper in eine diskrete Anzahl von Punkten und deren Verknüpfungen auf. Die neue Bruchmethode von Prof. James O'Brien und Jessica K. Hodgins ist eine solche Finite Elementen Methode [O'Brien '99]. Hierbei wird ein 3D Objekt zu Thetraedern zerlegt, welche unsere Finiten Elemente darstellen. Wird nun eine Kraft auf die Struktur des Objektes ausgeübt (sprich ein Knoten wird ausgelenkt) dann wird diese Kraft weiter verfolgt und ihr Einfluss auf die Nachbarknoten ermittelt. Diese Auslenkung wird bis zu einer bestimmten Schwelle als elastisch betrachtet, d.h. vor einem Bruch verhält sich das Objekt erst elastisch und kehrt in seine Ausgangsform zurück falls die wirkende Kraft nicht ausreichen sollte um das Objekt zu brechen. Wird jedoch der Schwellwert überschritten, wird ein Bruch initialisiert. Dies ist der gleiche Ansatz wie beim Bruch des Utah Teapot, doch das besondere bei der Methode von O'Brien ist, die neue Bruchmethode.
Soll nun ein Bruch stattfinden wird durch den Thetraeder, in dem der Bruch initialisiert wur-

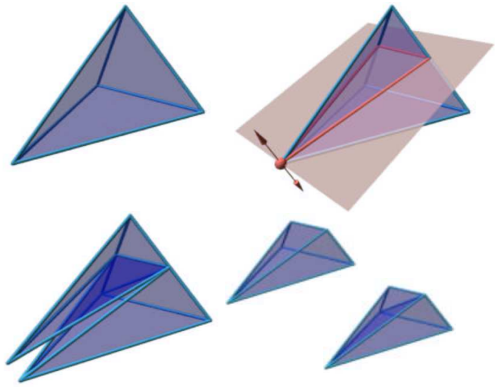


Abbildung 3.3: Die Bruchprozedur an einem Thetraeder aus dem Objekt

de, eine Bruchebene gelegt. Dieser wird dabei in ein kleineren Thetraeder und einem Polyeder aufgespalten (Abb.3.3). Zuletzt wird der entstandene Polyeder wieder zu zwei Thetraeder aufgeteilt. Dabei erhöht sich die Anzahl der elementaren Thetraeder in diesem speziellen Finiten Element um drei.

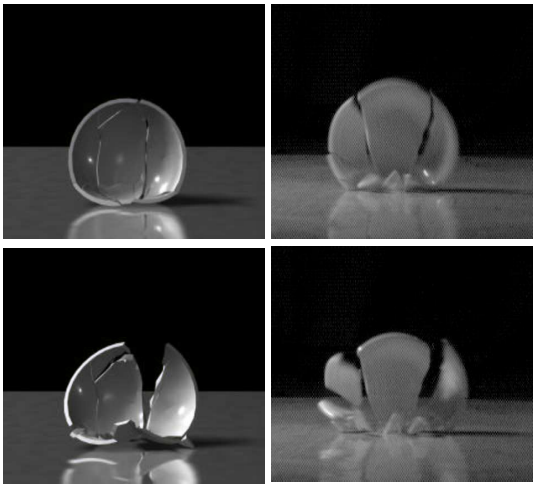


Abbildung 3.4: Gegenüberstellung Simulation - Realität

In Abbildung 3.2 wird eine Mauer von einer in das Bild pendelnden Kugel getroffen und zerbricht. Die Simulation beginnt mit einer Anzahl von 338 Knoten und 1109 Elementen (Thetraedern) nach dem Bruch sind es 6892 knoten und 8275 Thetraedern. Es lassen sich nun visuell schlüssige Brüche simulieren, wie bei der Gegenüberstellung einer Realaufnahme und der Simulation einer Schale mit ähnlichen Eigenschaften zu sehen ist (Abb.3.4).
O'Brien erweiterte 2000 dieses System um eine weitere Schwelle für die Plastizität [O'Brien '00].

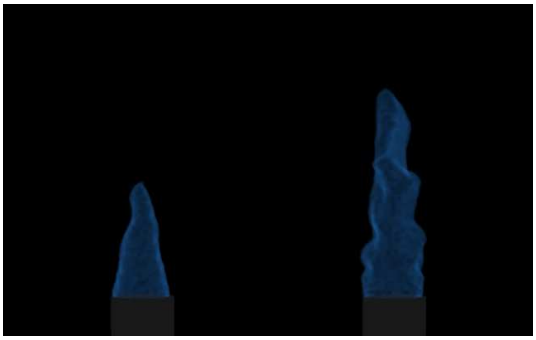


Abbildung 3.5: Demonstration verschiedener Materialien

Wird jetzt ein Knoten ausgelenkt verhält sich das Material bis zur ersten Schwelle elastisch und kehrt beim Nachlassen der wirkenden Kraft zur Urform zurück. Wird die erste Schelle überschritten verformt sich das Material bis zu einer neuen, zweiten Schwelle nun bleibend. Erst mit dem Überschreiten diese Schwelle wird der Bruch initalisiert. Wo es früher nur möglich war kristalline Brüche zu simulieren ermöglicht dieses Verfahren nun eine breite Materialpalette (Abb.3.5). Es lassen sich jetzt sogar Materialien wie Knetmasse oder Stoffe simulieren.

3.3.2 Der Stanford Hase

Der Stanford Hase ist in Wirklichkeit ein etwa 20 cm großer Ton Hase [Turk '00]. Er wurde 1994 im Rahmen eines Laserscan Projektes der Universität Stanford in die digitale Welt übertragen und muss seit dem die schlimmsten Dinge über sich ergehen lassen. Bisher wurde er

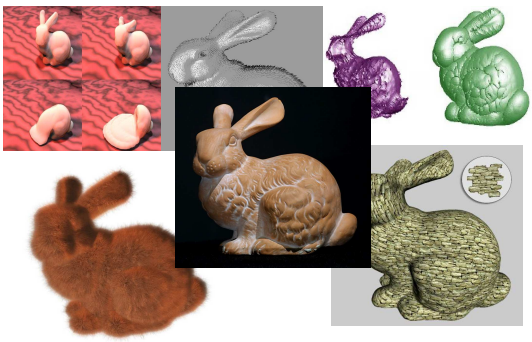


Abbildung 3.6: Mitte: orginal Stanford Hase

versteinert, ausgehungert, gemästet, er wurde extremer Hitze ausgesetzt und eingeschmolzen, musste sehr wirksame Haar-/bzw. Fellaufbaupräparate einnehmen und hatte sonst schon ein schweres Leben. Selten wurde einem Computerwesen soviel Hass entgegengebracht wie dem Stanford Hasen, nun soll er in Stücke geschlagen werden (Abb.3.6).

Dabei kommt aus der linken hinteren Ecke unseres Versuchsraumes ein schweres Objekt

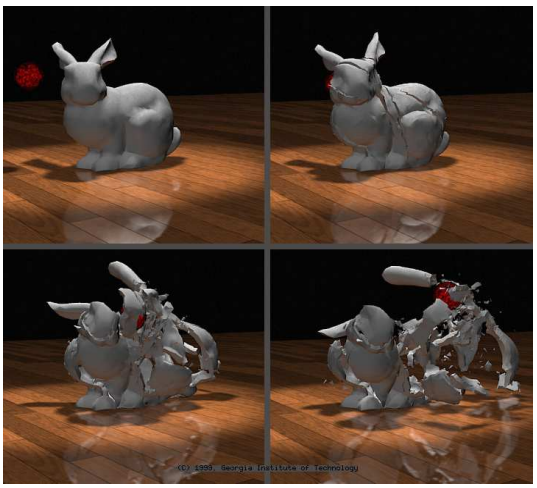


Abbildung 3.7: Hase zerbricht

mit großer Masse und Geschwindigkeit angeflogen und trifft das Versuchstier. Dieser wird dann, falls Materialeigenschaften des Hasen bzw. des Projektils richtig gewählt sind, zerbersten und in mehreren Stücken verschiedener Größe und Geschwindigkeit sich im Raum verteilen(Abb.3.7).

Resultat:

Der Hase wird buchstäblich zerfetzt. O'Brien hat diese Simulation mit verschieden Eigenschaften von Hase und Projektil durchgespielt und erreicht so verschiedene Effekte. Einmal scheint der Hase aus Porzelan zu bestehen und zerbricht trocken, einmal wie aus Wackelpudding, mal zerfällt der Hase in wenige Bruchstücke, mal in Tausende oder er wird von der Wucht des Aufpralls komplett mitgerissen usw.

3.3.3 Fazit

Dieses neue Verfahren ist überwiegend für den Unterhaltungsbereich gedacht, da die doch recht globale Deklaration der Materialeigenschaften für einen wissenschaftlichen Einsatz zu oberflächlich ist. Es wird von einem homogenen Material ausgegangen und mögliche innere Strukturen werden nicht berücksichtigt. Besonders auffallend war für mich bei der recht eindrucksvollen Demonstration Realität/Simulation, dass die simulierte Schale trotz des sehr realistischen ersten Bruchs nicht weiter zerbricht. Sie zerfällt in drei Teile, ähnlich wie Eierschalen, welche auf dem Untergrund abrollen, während in der slow motion aufnahme der echten Schale diese drei Teile weiter zerbrechen.

3.4 Virtuelle Flammen

Da der Mensch über ein sehr ausgeprägtes Sehsinn verfügt, fallen uns unnatürliche Bewegungen sehr schnell auf. Bisher war es üblich, Flammen durch immer zufällig neu permutierter Anzeige einer Flammenaufnahme oder durch Filmschleifen zu visualisieren. Dies hat aber den Nachteil, dass Feuer ein Gefahrenindikator ist und vom Menschen noch exakter beobachtet wird, als andere Ereignisse.
Es ist deshalb sehr schwer eine realistische Flammensimulation zu erzeugen. Ron Fedkiw und Henrik Wann Jenson von der Universität Stanford haben es dennoch versucht und hatten Erfolg.
Sie versuchen keine optische Täuschung zu erzeugen, wie es bisher üblich war, sondern modellieren eine echte virtuelle Flamme.
Dabei gehen sie davon aus, das jede reale Flamme aus drei Teilen besteht.

- Der Brennmittelfluss
- 2.Reaktionszone (Blue Core)
- 3.Heiße Gasprodukte/die Schwarzkörper Emission (blackbody radience)

Modeliert wird in einem Voxelraum, d.h. der Raum ist in Quadranten eingeteilt in denen unterschiedliche Temperaturen herrschen. Ausserdem verhalten sich Brennmittel und Schwarzkörper Emision nach der Navier Stoke'schen Gleichung für inkompressible Fluidsysteme.

3.4.1 Navier Stoke'sche Gleichung

Bis zum Anfang des 19. Jahrhundert war der Glaube verbreitet, daß Fluide inkompressibel sind und Körper reibungsfrei umströmen. Von inkompressibel spricht man, wenn das Medium einer Volumenänderung (ideel) einen grossen Widerstand entgegensetzt. Diese Aussage betrifft auch die Fluiddynamik. Als kompressibel werden die Fluide dann betrachtet, wenn die Strömungskinetik bzw. -kräfte zu einer Dichteänderung führen.
Der Berücksichtigung des Einflusses der Reibung in Strömungsfeldern sind den Arbeiten aus den Jahren 1827-1845 von Navier, Cauchy, Poisson und St. Venant zu verdanken. Diese Physiker führen eine unbekannte molekulare Funktion zur Beschreibung der Reibung ein. Stokes verwendet diesbezüglich die Viskosität. Die Navier-Stoke'sche Gleichung findet heute überall dort Verwendung, wo Strömungen von Gasen oder Flüssigkeiten (also Fluiden) modelliert werden sollen. So z.B. auch bei Brandsimulation, Wettervorhersage, Stömungsmodellen von Tragflächen, Schiffsrümpfen, Turbinen, Fahrzeugen usw. Wegen der Komplexität der Navier Stoke'schen Gleichung lassen sich ähnlich den CFD-Modellen (Siehe Kap.3.5.3) die Fluide nur

für eine finite Anzahl diskreter Volumina berechnen. Einen kurzen Überblick findet man unter [unbekannt '01]. Der Umstand, daß in einem Voxelraum modelliert wird kommt dem entgegen.

3.4.2 Brennstofffluss

Bei einer gewönlichen Gasflamme ist der Treibstoff eine unsichtbare gasförmige Strömung. Sie vermischt sich mit dem Sauerstoff der Umgebungsluft, erwärmt sich bis es die Reaktionstemperatur erreicht hat und reagiert in der Reaktionszone (blue core). Das Vermischen wird nicht simuliert, da sich durch das vermischen nur die Reaktionsgeschwindigkeit des Treibstoffs verzögert. Grösse der Austrittsöffnung und Reaktionsträgheit nehmen direkten Einfluss auf die Form der Flamme.

3.4.3 Reaktionszone

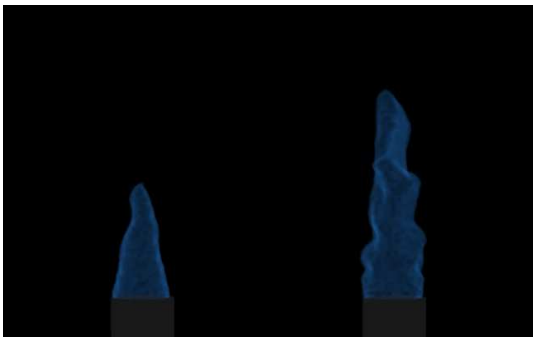


Abbildung 3.8: Die Blue Core
links: grosse Reaktionsgeschwindigkeit bzw. vorgemischtes Gasgemisch
rechts: kleine Reaktionsgeschwindikkeit, bzw. nicht vorgemischstes Gas

Die Reaktionszone ist eine blaue kegelförmige Zone in der der zugeführte Treibstoff reagiert (Abb.3.8). Grösse und Form der Reaktionszone ist abhängig von der

- Injektionsgeschwindigkeit
- Größe der Austrittsöffnung des Treibstoffgemischs
- Reaktionsgeschwindigkeit des Treibstoffs

Durch die Variation dieser Faktoren wird die Form der Flamme beeinflusst. Möchte man eine große turbulente Flamme kann man entweder Grösse der Austrittsöffnung oder Austrittsgeschwindigkeit vergrößern oder man verringert einfach die Reaktionsgeschwindigkeit.

3.4.4 Heiße Gasprodukte

Die Reaktionszone emittiert einen zweiten Fluss aus Schwarzkörper Partikeln. Dieser Strom ist um einen gewissen Faktor größer als der Brennmittelzufluss und hat Einfluss auf die Flammenfülle (Abb.3.9).
Dieser Faktor errechnet sich durch die Dichten von Treibstoff (p_f) und der Dichte des heißen Gasproduktes (p_h):

$$F = p_f / p_h$$

Abbildung 3.9: unterschiedliche Dichten des heißen Gasprodukts

3.4.5 Schwarzkörper Emission

physikalisch

Schwarzkörper ist ein physikalischer Begriff. Es sind Objekte die kein Licht reflektieren, bei Erwärmung aber Licht emittieren können. Sie wurden Anfang des 20. Jahrhunderts von namenhaften Physikern geprägt und stellt den Anfang der Quantenphysik dar.
Wärme tritt in zwei Formen auf. Zum einen als Materialeigenschaft, welche an materielle Substanzen gebunden ist zum anderen als Wärmestrahlung, die man spürt, wenn man sich einem warmen Objekt nähert. Ist das Objekt heiß genug, strahlt es sogar Strahlungen im sichtbaren Frequenzbereich aus. Ende des 19., Anfang des 20 Jahrhunderts galt es den Zusammenhang zwischen Temperatur des Objekts und der ausgestrahlten Strahlung zu finden.
Empirische Methoden, die korrekte Strahlungsformel zu finden, führten nicht zum Erfolg. Erst Max Plank gelang im Dezember 1900 mit der Einführung eines neuen physikalischen Begriffs, des Quantums, der Durchbruch [Neundorf '03].
Schwarzkörper sind in der Physik eine Idealvorstellung, denn jedes Objekt reflektiert in gewissem Grad Licht. Mit Schwarzkörpern lässt sich zum Beispiel die Wärmestrahlung von Lebewesen oder das Leuchten eines Glühfadens in einer Glühbirne erklären. Der durch den elektrischen Strom geheizte Wolframdraht einer Glühlampe ist die wohl verbreitetste technische Anwendung dieses physikalischen Phänomens.

im Modell

Die Schwarzkörper Emission, die von der Blue Core emittiert wird, durchwandert beim Aufsteigen Voxel unterschiedlicher Temperatur und erwärmen sich oder kühlen ab. Dadurch werden sie zum Leuchten angeregt und bilden so die sichtbare Flamme. Löscht man die Schwarzkörper Partikel nicht, nachdem sie die Leuchttemperatur verloren haben kann man auch das aufsteigen von Ruß simulieren.

3.4.6 Fazit

Mit dieser Simulation lässt sich nun nahezu jede Flamme simulieren. Will man ein brennbares Objekt erzeugen, wird der Brennmittelfluss von der Oberflächennormalen emittiert. So wird z.B. ein brennbares Objekt von den Flammen umschlossen und die Flammen sehen äußerst natürlich aus (Abb.3.10).

Abbildung 3.10: Brennbares Objekt wird duch Flamme geschwenkt und fängt Feuer

3.5 Brandsimulation

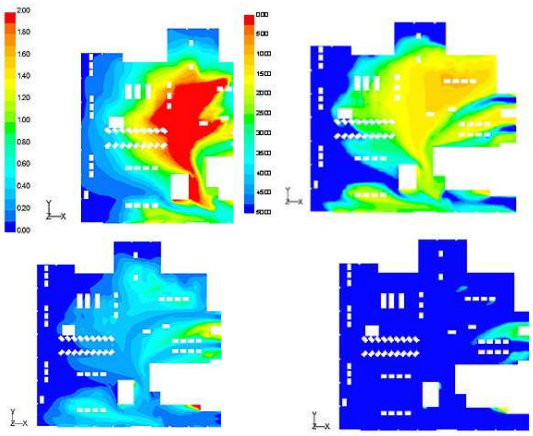


Abbildung 3.11: Verrauchung und Sichtweite vor und nach der Optimierung mit Brandsimulationen

Ziel einer Brandsimulation ist der vorbeugende und abwehrende Brandschutz. Es stellt beim Bau eines neuen Gebäudes einen eigenen Planungsbereich dar, bzw. ermöglicht die Erhöhung der Sicherheit alter Gebäude.
Dabei geht es nicht nur um die Simulation der Flammenausbreitung, sondern auch um die Analyse der Rauchausbreitung .
Diese Rauchausbreitung kann bei definierten Brandszenarien durch Simulation sehr genau berechnet werden. Diese Berechnungen basieren auf folgenden Grundlagen:

- Strömungssimulation (CAD Modell, Ergebnisse)
- brandspezifische Modelle und Berechnungen im Kontext mit der Strömungssimulation (CFD)
- Gebäudesimulation (Wandoberflächentemperaturen,Wärmeströme über Begrenzungsflächen)

Im Rahmen der Untersuchungen werden die Absaugstellen (Lage, Art und Parameter), Entrauchungsluftmengen sowie das Betriebsregime variiert, bis die für das definierte Brandereignis optimalen Entrauchungsluftmengen und -strategien für die geforderte Zielstellung erreicht sind.

Zielstellungen können sein:

- Vorherbestimmung der erforderlichen Sichtweiten für die Fluchtwege und Sicherung der Fluchtzeiten
- Aussagen über die zu erwartenden Auslösezeiten / Auslösetemperaturen von Sprinkleranlagen
- Bestimmung der Temperaturen an speziellen Bauteilen, wie Stahlkonstruktion und ob diese Bauteile einen besonderen Schutz benötigen.

Mit Hilfe der Brandsimulation werden in der Regel geringere Entrauchungsluftmengen und damit kleinere und kostengünstigere Entrauchungsanlagen ermöglicht.
Am Beispiel der Gepäckförderanlage im Terminal 2 des Flughafen München sind aufgrund der Komplexität nach konventioneller Auslegung in einem Brandabschnitt 80 m x 100 m zwei Anlagen mit je 10-fachen Luftwechsel vorgesehen worden. Dabei wurden die Anlagen so ausgelegt, daß der Rauch aus jedem Bereich auf kürzestem Weg abgeführt werden soll.
Bei der Simulation dieses Konzeptes stellte sich heraus, daß große Bereiche verrauchen.
Durch Änderung der Anordnung der Absaugstellen, Veränderung der Abluftmengen und Anpassung der Fahrweise der Entrauchungsanlagen an die tatsächlichen Verhältnisse konnte die Anlage sicherer gemacht werden.
Darüber hinaus wurden die Luftmengen für jede der zwei Anlagen auf ein drittel des ursprünglich Vorgesehenen reduziert. Das führte zu einer deutlichen Senkung der Investitionskosten.

3.5.1 Zonenmodelle

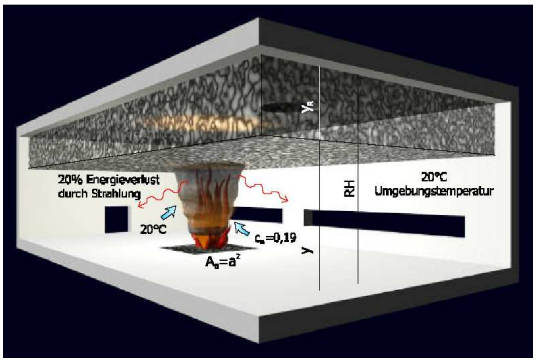


Abbildung 3.12: Unterschiedliche Reaktionsgeschwindigkeiten des Gasgemischs

Das Zonenmodell ist das Einfachste und auch das Älteste Brandsimulationsmodell. Sie wurde in den 50 Jahren entwickelt und genügt heutzutage nur um grobe aussagen über die Rauch- bzw. Hitzeausbreitung zu treffen.
Dabei wird ein Raum in 2-20 Zonen eingeteilt in denen homogene Zustände der Temperatur und des Drucks angenommen werden. Jede Zone ist Volumenvariabel, d.h. die Volumina der

jeweiligen Zonen werden wärend der Simulation den Umgebungsbedingungen angepasst.
Zur Simulation dieses Systems werden in Zeitabschnitten von einer Sekunde Energie- und Massebilanzen aufgestellt und die Volumina entsprechend neu eingeteilt (Abb.3.12). Die Simulation kann Rückschlüsse über die Temperatur, bzw. der Rauchausbreitung in dem betreffenden Raum geben (Bis zu welcher Höhe herrscht welche Temperatur? Wie weit ist der Raum verraucht?).

3.5.2 Feldmodelle

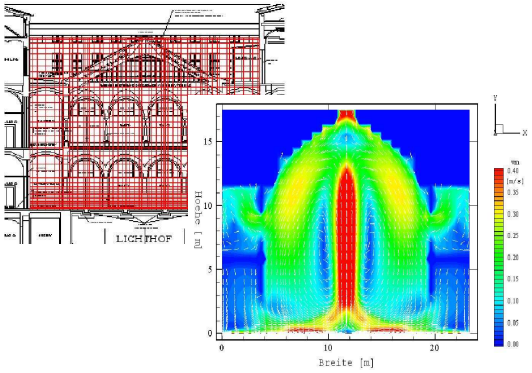


Abbildung 3.13: Feldmodell eines Atriums/Geschwindigkeitsfelder

Das Feldmodell teilt den Simulationsraum in kleinere Kontrollvolumina auf in denen wieder homogene Zustände angenommen werden. Die Kontrollvolumina sind jedoch im Gegensatz zum Zonenmodell volumeninvariabel.
Die Voluminas sind auf Hexaeder beschränkt. Um den Zustand in den Voluminas zu ermitteln werden nun jede zehntel Sekunde nichtlineare partille Differenzialgleichung gelöst und zwar für jedes einzelne Volumina. Die Gleichungen basieren auf der Strömungsmechanik und den Kontinuitätsbedingungen wie Impuls-,Energie und Masseerhaltung.

Im Gegensatz zum Zonenmodell ist diese Simulationstechnik viel aufwändiger, kann aber genauere Aussagen über das Geschwindigkeitsfeld, Druck, Dichte, Temperatur, Gaskonzentrationen, Turbulenzgrößen usw. treffen.
Im Beispiel wird ein Feldmodell eines Atriums simuliert in der die Luftmassen duch die Körperwärme von 400 Personen die sich darin befinden erwärmen (Abb.3.13).

3.5.3 CFD-Modelle

CFD-Modelle (Computational Fluid Dynamics) sind ähnlich wie Feldmodelle. Sie unterscheiden sich nur in der Einteilung des Raumes, denn man ist hierbei nicht auf Hexaeder beschränkt. Man kann je nach Bedarf ein Gebiet feiner vernetzen, z.B. falls eine detailiertere Darstellung erforderlich ist oder eine grobere Vernetzung, falls ein Gebiet für die Simulation nicht von Belang ist. CFD-Modelle werden seit 1993 auch von der Formel 1 benutzt um die Aerodynamik der Fahrzeuge zu verbessern. Damals war Benetton das erste Team das diese Technologie zur Optimierung ihrer Boliden verwendete [Sandkühler '02]

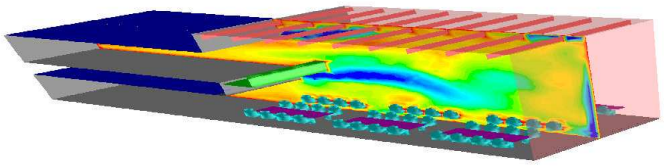


Abbildung 3.14: CFD Modell der Hypovereinsbank Frankfurt

3.5.4 erweiterte Lindenmayer-Systeme

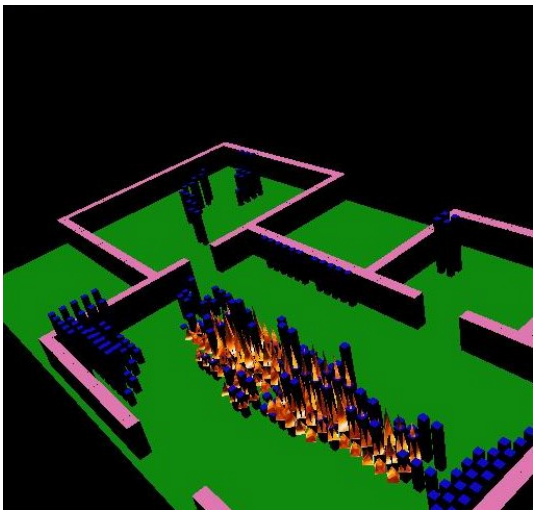


Abbildung 3.15: Java3D Anwendung des neuen Brandsimulations Konzeptes

Lindenmayer-Systeme wurden 1968 von Astrid Lindenmayer eingeführt und sollten ursprünglich den Wachstum primitiver Mikroorganismen beschreiben. Ein neues Konzept der Brandsimulation beruht auf der Verwendung von erweiterten Lindenmeyer Fraktalen [Tomasz Zaniewski '03]. Dabei wird davon ausgegangen, dass sich Flammen wie Lindenmeyer-Fraktale verhalten und sich ihre Ausbreitung mit diesem Modell simulieren lassen.
Schwerpunkt dieses Modells ist die Ausbreitung der Flammen. Thermodynamik, Grad der Verrauchung usw. werden nicht berücksichtigt. Vielmehr soll dieses Verfahren eine Architektur auf ihre Sicherheit testen und so den sichersten Teil des Gebäudes ermitteln.

3.6 Fazit

Momentan gibt es, abgesehen von einigen wenigen nicht komerziellen AR / VR Systemen, kaum Simulationen welche die Echtzeitanforderung erfüllen. Auch scheint es keinen akuten Bedarf zu geben. Bei der Produktentwicklung werden Echtzeitsimulationen aber nicht verwendet, da sie aus bereits erwähnten Gründen überflüssig sind.
Die Flammensimulation ließe sich zur Ausbildung von Feuerwehrmännern oder Soldaten einsetzen, ob es aber sinvoll ist z.B. Feuerwehrmänner mit VR Brandsimulation auszubilden ist fraglich. Es gibt bereits bessere Möglichkeiten mit echten Flammen und echtem Rauch realistische Szenarien zu generieren.

Tatsächlich gibt es aber bereits Programme die zum Training von Soldaten eingesetzt werden sollen. Das amerikanische Militär forscht nach solchen Möglichkeiten und ist vermutlich eine der wenigen Institutionen, die sich eine solche Anlage leisten kann. Vielmehr dienen solche Programme der Rekrutierung neuer Soldaten als einer Trainingsmöglichkeit.

Literaturverzeichnis

[Champagne '03] Jennifer A. Champagne. Making Mega Matrix. 2003. http://millimeter.com/ar/video_making_mega_matrix/ (gesehen 08/2003).

[Fedkiw '01] Ronald Fedkiw. Physically Based Modeling and Animation of Fire. 2001.

[Neundorf '03] Wolfgang Neundorf. Die Physik in der Sackgasse? Wissenschaft und Kritik. 2003. http://www.neundorf.de/Kritik/Physik/Quanten2/quanten2.html (gesehen 08/2003).

[O'Brien '99] James F. O'Brien. Graphical Modeling and Animation of Brittle Fracture. 1999.

[O'Brien '00] James F. O'Brien. Graphical Modeling and Animation of Ductile Fracture. 2000.

[Sandkühler '02] Christian Sandkühler. Aerodynamik. 2002. http://www.f1-plus.com/Deutsch/magazin/2002/02_tk_aerodynamik.html (gesehen 08/2003).

[Stefan Martens '] Peter Vogel Stefan Martens. EDV-gestützte Brandsimulation. http://www.fluent.com/worldwide/germany/support/papers.htm (gesehen 08/2003).

[Tomasz Zaniewski '03] Shaun Bangay Tomasz Zaniewski. Simulation and Visualization of Fire using Extended Lindenmayer Systems. In *Computer graphics, virtual reality, visualisation and interaction in Africa archive Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, S. 39–48. ACM Press, 2003.

[Turk '00] Greg Turk. The Stanford Bunny. 2000. http://www.gvu.gatech.edu/people/faculty/greg.turk/bunny/bunny.html (gesehen 08/2003).

[unbekannt '01] unbekannt. CFD - Analyse und was sie kann. 2001. http://www.asd-online.com/ger/ger_jsmindex.htm?cfd-analyse.htm (gesehen 08/2003).

4 Haptisches Rendering

Marco Lepper

4.1 Einleitung

Diese Arbeit befasst sich mit der Simulation und **Wiedergabe von Tast- und Kraftinformationen** in einer virtuellen Umgebung. Hierbei spricht man auch von **haptischer Wiedergabe** bzw. **haptischem Rendering**, wobei mit **Haptik die Lehre des Tastsinns** bezeichnet wird [Duden]. Es geht also darum, gezielt die Kraft- und Tastsinne so anzusprechen, dass der Eindruck von realer haptischer Empfindung entsteht.
Zunächst wird in Abschnitt 2 der Einsatz von haptischem Rendering motiviert und Anwendungsbereiche angeführt. In Abschnitt 3 wird Einblick genommen in die Funktionsweise des menschlichen Tast- und Kraftsinns. Abschnitt 4 beschäftigt sich mit der Hardwareseite: es werden Methoden zur Stimulation des menschlichen Tast- und Kraftsinns und einige Geräte zur Wiedergabe von haptischen Informationen vorgestellt. Abschnitt 5 schließlich befasst sich mit der Softwareseite: es geht um die Berechnung der in einer virtuellen Umgebung auftretenden Kräfte, die haptisch wiedergegeben werden sollen. Abschnitt 6 versucht sich an einer Bewertung des Nutzens haptischer Wiedergabe und zieht ein Fazit.

4.2 Motivation

Herkömmliche Interaktion mit einer virtuellen Umgebung ist größtenteils nur mit Hilfe eines grafischen Wiedergabegeräts wie eines Bildschirmes möglich. Teils wird zusätzlich auch noch der Hörsinn angesprochen, Sinne wie Geschmacks- und Geruchssinn werden bis heute nicht berücksichtigt.
Die Wiedergabe von haptischen Informationen ist mit einem erhöhten Aufwand an Hardware und Software verbunden, es werden geeignete Ausgabegeräte benötigt und diese Geräte müssen natürlich sinnvoll angesteuert werden. Wodurch wird dieser Mehraufwand gerechtfertigt?

4.2.1 Gegenüberstellung Sehsinn - Tast- und Kraftsinn

Vergleichen wir das Sehen und Fühlen anhand des Erforschens eines realen Objekts.

Sehsinn

Allein durch Betrachten eines Objekts erhält man Informationen über die Form und das Volumen des Objekts. Optische Eigenschaften wie die Farbe eines Objekts, Beschriftung oder Transparenz können sogar ausschließlich über das Auge erfasst werden. Über weitere Eigenschaften wie z.B. das Gewicht oder die Glätte der Oberfläche können nur Annahmen gemacht werden, die auch stark von persönlichen Erfahrungen abhängig sind.

Tast- und Kraftsinn

Erforscht man ein Objekt bei geschlossenen Augen mit der Hand allein über den Tast- und Kraftsinn, erschließen sich ein anderes Spektrum an Informationen. Zum einen gewinnt man wie beim Betrachten auch einen Eindruck über Form und Volumen. Dann aber lassen sich andere Eigenschaften erfühlen: die Oberflächenbeschaffenheit, Härte bzw. Elastizität, Temperatur bzw. Temperaturleitfähigkeit und Gewicht. Außerdem lässt sich durch Manipulation des Objekts feststellen, inwieweit bewegliche Teile oder Funktionen vorhanden sind.

Schluss der Gegenüberstellung

Kombination von visuellem mit haptischem Feedback verspricht eine Erweiterung des Spektrums an Informationen, die aus einer virtuellen Umgebung gewonnen werden können. Speziell Temperatur, Elastizität, Gewicht und Oberflächenbeschaffenheit eines Objekts lassen sich rein visuell gar nicht feststellen oder können nur erraten werden.

4.2.2 Weitere Motivation

Das Hinzufügen von haptischem Feedback zu einer virtuellen Umgebung steigert die Realitätsnähe, was dem Benutzer ermöglicht, ihm bekannte Konzepte zur Interaktion zu verwenden. Die Manipulation eines Objekts/der Umgebung mit den Händen mit Hilfe von Kraft und Berührung ist für den Menschen intuitiv. Hingegen ist z.B. das Ergreifen eines Objekts, ohne Widerstand zu spüren, ungewohnt und erfordert erhöhte Konzentration, um das gewünschte Ergebnis zu erreichen.

4.2.3 Ziele

Die Nutzung von haptischem Rendering hat mehrere Ziele. Die Bedienbarkeit einer virtuellen Umgebung soll verbessert werden. Damit einhergehend ist eine Erhöhung der Produktivität des Benutzers erwünscht. Virtuelles Training soll dadurch einen besseren Trainingseffekt erzielen.

4.2.4 Anwendungen

Es gibt vielfältige Anwendungsmöglichkeiten von haptischer Wiedergabe.
[Alakärppä et al. '98]

Im Bereich der Teleoperationen ermöglicht sie genauere Bewegungen des Operators, der Berührungen intuitiver wahrnimmt. Das ermöglicht eine Erhöhung der Sicherheit, z.B. beim Einsatz in Wiederaufbereitungsanlagen für Brennstäbe oder bei der Bedienung des Telearms des Space Shuttle.

Das Training von realen und besonders kritischen Abläufen erfordert eine möglichst große Realitätsnähe, damit das Gelernte später routiniert ohne böse Überaschungen umgesetzt werden kann. Auch lassen sich Extremsituationen so besser simulieren. Nur in Flugsimulatoren können angehende Piloten ohne Gefahr auf Situationen vorbereitet werden, in denen extreme Kräfte auf sie einwirken.

Auch ermöglicht haptische Wiedergabe den realistischeren Umgang mit noch nicht real existierenden Objekten. So könnte z.B. die Ergonomie eines Produktes getestet werden, was be-

sonders bei großem Fertigungsaufwand und/oder hohen Materialkosten Zeit und Geld spart.

Weiter kann Sehbehinderten mit Hilfe von haptischer Wiedergabe eine virtuelle Umgebung zugänglich gemacht werden.

Der Entertainmentbereich hat natürlich auch Verwendung für die Haptik. Bekannt sind sogenannte Force Feedback Geräte wie Computerjoysticks und Gamepads, die mit Vibrationen und Kräften auf den Benutzer einwirken können.

Konkretes Anwendungsbeispiel: VEST

Bei VEST (Virtual Endoscopie Surgery Training) handelt es sich um ein Trainingsgerät für minimal invasive Operationen. [(MEDTECH) '01]
Bei dieser Art von Operation werden sogenannte Endoskope durch kleine Schnitte in den Körper eingeführt. Die Endoskope sind stabähnlich und besitzen kleine Kameras und/oder Greif- und Schneidewerkzeuge an der Spitze. Es gibt starre und flexible Endoskope. Der Vorteil dieser Operationsart liegt darin, dass keine großen Schnitte durchgeführt werden müssen, um das zu behandelnde Organ zu erreichen, sondern dem Körper nur kleine Wunden zugefügt werden. Nachteilig ist aber, dass der Chirurg auf das in der Regel 2-dimensionale Kamerabild angewiesen ist, um innerhalb des Körpers zu navigieren. Dieses Bild ist stark vergrößert und verleitet dadurch natürlich zur Überreaktion.
Bei dem geringen Bewegungsspielraum des Arztes gefährden übersteuerte Bewegungen der Endoskope natürlich die Gesundheit wenn nicht gar das Leben des Patienten.
VEST stellt dem Trainierenden eine realistische Umgebung zur Verfügung. Es sind mehrere Endoskope vorhanden, die die Operation eines virtuellen Körpers innerhalb einer Box ermöglichen. Ein Bild der Simulation wird ausgegeben. Die Endoskope werden zusätzlich auch durch Motoren angesteuert, so dass z.B. die Festigkeit eines Organs bei Ausüben von Druck wiedergegeben werden kann.

Abbildung 4.1: Vest

Abbildung 4.2: Nahaufnahmen von Vest

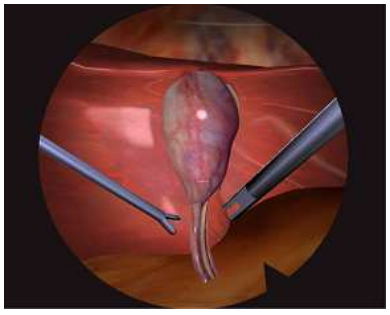


Abbildung 4.3: Bildschirmsicht

4.3 Haptische Wahrnehmung

Um Haptik simulieren und haptische Informationen wiedergeben zu können, ist ein Verständnis über die Funktionsweise dieser Sinneswahrnehmung beim Menschen hilfreich. [Böhme & Sotoodeh '99, Rainer Zwisler '98]

4.3.1 Die Wahrnehmungsarten

In der Haptik werden zwei Wahrnehmungsarten unterschieden. Die taktile und die kinästhetische Wahrnehmung.

Taktile Wahrnehmung

Unter der taktilen Wahrnehmung werden die Eindrücke zusammengefasst, die über die Rezeptoren in der Haut gewonnen werden. Rezeptoren sind für ganz bestimmte Reize ausgelegt und senden über die Nervenbahnen elektrische Signale an das Gehirn, sobald eine entsprechende Reizung vorliegt. Je stärker die Reizung desto höher die Frequenz der Signale. Im Gehirn werden diese Signale dann interpretiert.

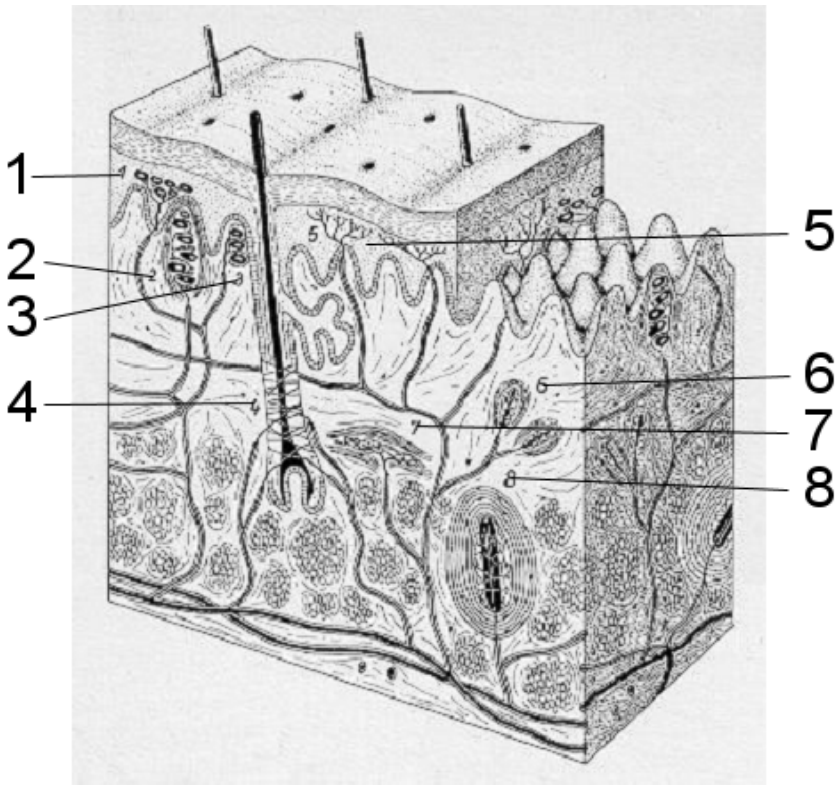


Abbildung 4.4: Hautquerschnitt mit Sinnesrezeptoren

Taktile Wahrnehmungen sind:

- Berührung (4)
- Druck (1, 2, 3)
- Vibration (8)

- Kälte (6)
- Wärme (7)
- Schmerz (5)

Die Rezeptoren für Berührung sind sehr empfindlich und so dafür geeignet, den bloßen Kontakt mit einem Gegenstand zu registrieren. Allerdings sind sie nicht geeignet, Intensitäten bei höheren Drücken zu unterscheiden.
Hierfür sind weitere Rezeptoren vorhanden. Die Druckrezeptoren unterscheiden sich untereinander in den Intensitätsbereichen, die sie auflösen können. Interessant ist, dass auf der Handinnenseite verschiedene Drücke nur bis zu einem minimalen Abstand von 0,9 mm noch getrennt erkannt werden. Das Auflösungsvermögen hängt von der Rezeptorendichte ab (siehe Tab. 4.1).
Die Rezeptoren für Vibrationen können Frequenzen bis 1 kHz auflösen, Vibrationen mit höherer Frequenz werden nur als konstanter Druck wahrgenommen. Die Auflösung liegt bei der Hand im Bereich 10-60 Hz bei 3-4 mm, ab 60 Hz kann eine Vibration nur noch auf 20 mm genau lokalisiert werden.
Die Kälterezeptoren sind schneller äußeren Einflüssen ausgesetzt als die Wärmerezeptoren, da jene tiefer in der Haut liegen. Die Wärmerezeptoren sind auch eher Instrument der Regulierung der Körpertemperatur, da in diesem Temperaturbereich die größte Empfindlichkeit vorliegt.
Rezeptoren für den Schmerz registrieren alle genannten taktilen Wahrnehmungsarten, sobald der Reiz besonders intensiv wird. Auch ist die Anzahl der Schmerzrezeptoren und damit ihre Verteilungsdichte besonders hoch (siehe Tab. 4.1). Diese Rezeptoren dienen zur Erkennung von Gefahren für den Körper.

	Wärme	Kälte	Druck	Schmerz
Innenseite	0,4	6	15	203
Rückseite	0,5	7	14	188

Tabelle 4.1: Punktdichte der Hautsinne der Hand pro cm^2

Kinästhetische Wahrnehmung

Die kinästhetische Wahrnehmung ermöglicht die Erkennung von externen Kräften und deren Intensitäten, wie z.B. das Gewicht eines Objekts, den Greifwiderstand oder den Widerstand einer Wand.
Kinästhetische Wahrnehmung erfolgt über die Auswertung der Informationen über die Anspannung von Muskeln und die Stellung der Gelenke.
Diese Wahrnehmung ist unterscheidet sich von Individuum zu Individuum und hängt stark vom körperlichen Zustand und der eigenen Erfahrung ab.

4.4 Hardware

In diesem Abschnitt werden die Möglichkeiten betrachtet, bestimmte haptische Reize zu erzeugen, Anforderungen an haptische Wiedergabegeräte gestellt und einige Endgeräte vorgestellt.

4.4.1 Erzeugung haptischer Reize

Wie in Kapitel 3 gezeigt, umfasst die Haptik eine Vielzahl an Wahrnehmungen, die für die haptische Wiedergabe aus einer virutellen Umgebung heraus angesprochen werden müssen. Hierbei spricht man bei der Rückgabe von taktilen Informationen auch von **Touch Feedback** und bei der Rückgabe von kinästhetischen Informationen von **Force Feedback**.
Möglichkeiten für Touch Feedback und Force Feedback werden nun im einzelnen vorgestellt. [Böhme & Sotoodeh '99, Rainer Zwisler '98, Alakärppä et al. '98]

Touch Feedback: Druck

Verschiedene Drücke können wie gesehen auf der Handinnenseite noch bis zu einem Abstand von 0,9 mm unterschieden werden. Ein Gerät zur Ausgabe von Drücken sollte für größtmögliche Realitätsnähe diese Auflösung bieten.
Eine Möglichkeit wären in einer Matrix angeordnete dünne Stifte (siehe Abb. 4.5), die einzeln ausgelenkt werden können. Auch wäre die Verwendung von kleinen pneumatischen Kissen denkbar, die natürlich in hoher Frequenz und mit geringer Latenz gefüllt und geleert werden können müssen. Die elektrische Stimulation der Nervenbahnen wird laut der Literatur zur Zeit aus Sicherheitsgründen nicht weiter in Betracht gezogen.

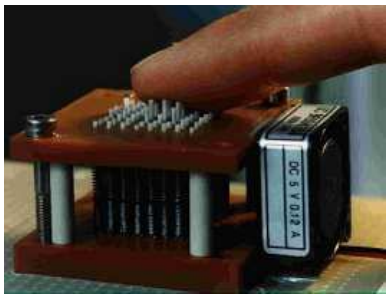


Abbildung 4.5: Stiftmatrix

Abbildung 4.6: piezoelektrischer Summer neben Cent-Münze

Touch Feedback: Vibration

Vibrationen können erzeugt werden durch kleine Lautsprecher, piezoelektrische Summer (siehe Bild) und Unwuchtmotoren.
Vibrationen können ja im Bereich über 60 Hz nur noch auf 20 mm genau lokalisiert werden, was zur realistischen Simulation die Verwendung von relativ wenigen und billigen Standardelementen ermöglicht (siehe Größe piezoelektrischer Summer in Abb. 4.6).

Touch Feedback: Temperatur

Temperaturänderungen an der Hautoberfläche können mit Hilfe von Peltierelementen (siehe Abb. 4.7) und kleinen Heizspiralen vorgenommen werden.
Besonders klein dimensionierte Peltierelemente scheinen mir eher zur Abkühlung geeignet, könnten aber im Gegensatz zu Heizspiralen sowohl zur Erzeugung von Wärme als auch Kälte genutzt werden.

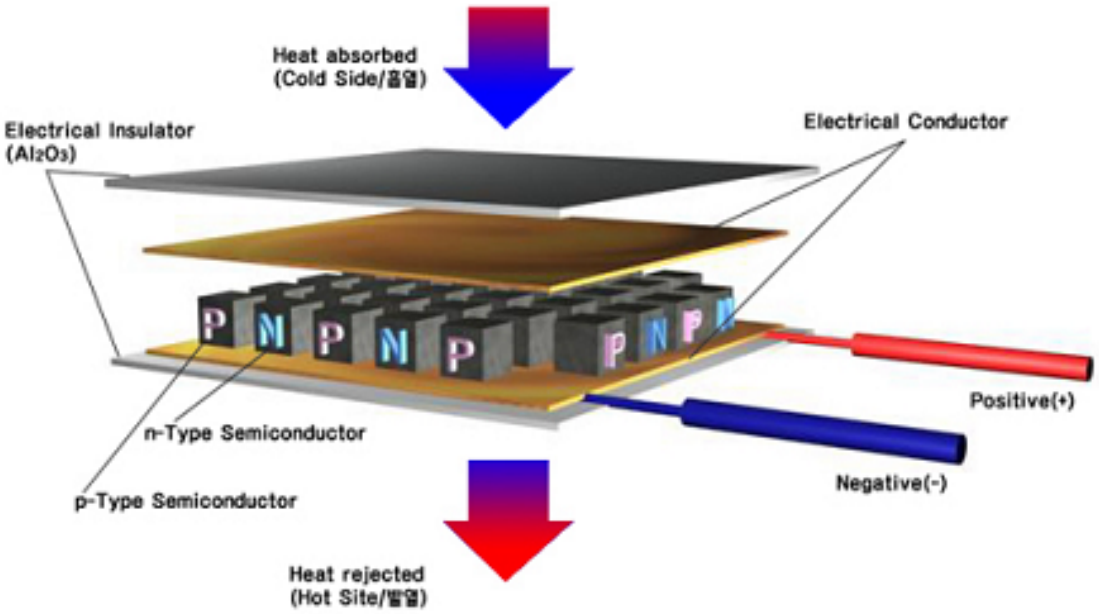


Abbildung 4.7: Schema eines Peltierelements

Force Feedback

Es gibt zwei Arten von Force Feedback: benutzerbezogenes und umgebungsbezogenes.

Beim benutzerbezogenen Force Feedback werden Kräfte relativ zum Körper des Benutzers wiedergegeben. So könnte z.B. Greifwiderstand wiedergegeben werden, aber nicht das Anlehnen an eine virtuelle Wand. Diese Force Feedback Variante ist in der Regel auf vom Benutzer getragene Exoskelette angewiesen (siehe Abb. 4.8 und Abb. 4.9). Diese können z.B. mit Hilfe von Pneumatik oder Seilzügen Kräfte relativ zu den Befestigungspunkten am Körper erzeugen.

Abbildung 4.8: Force-Feedback-Handschuh

Abbildung 4.9: Percro, ein Exoskelett für den Arm

Abbildung 4.10: Force-Feedback-Arm Phantom

Umgebungsbezogenes Force Feedback nutzt ein Wiedergabegerät, dass Kräfte relativ zum Verankerungspunkt in der Umgebung wiedergibt. Ein Beispiel hierfür wäre der Force-Feedback-Arm in Abb. 4.10.

Schmerz

Der Einsatz von Schmerz als Informationsträger ist in der Realität nur in hochsensiblen Bereichen als letztes Mittel zur Abwehr extremer Gefahren denkbar. Beispiel wäre der Einsatz in der Überwachungszentrale eines Kernkraftwerks, falls eine Kernschmelze bevorsteht und das Überwachungssystem nach dem Fehlschlagen aller automatischen Schutzmechanismen das Ausbleiben von Gegenmaßnahmen seitens des Überwachungspersonals feststellt.

In der Literatur kam an einer Stelle der Vorschlag auf, Schmerzen einzusetzen, um Personen bei Fehlentscheidungen in Simulationen zu bestrafen. Dies soll eine Konditionierung bewirken, die solche Fehlentscheidungen zukünftig vermeidet. Dieser Vorschlag wird allgemein als ethisch sehr bedenklich eingestuft, was auch meiner Meinung entspricht. Und der alles andere als unbeschwerte Umgang mit solch einer Simulation dürfte die Produktivität und den Trainingseffekt eher senken als steigern. [Rainer Zwisler '98]

4.4.2 Anforderung an haptische Ausgabegeräte

Um eine möglichst realistische haptischer Wiedergabe zu erreichen, muss neben einer realistischen Erzeugung der haptischen Reize natürlich auch gelten, dass der Benutzer das Ausgabegerät als nicht störend empfindet. Um den haptischen Eindruck nicht zu schmälern, muss

ein entsprechendes Ausgabegerät einen geringen Eigenwiderstand bei Bewegungen aufweisen. Gleichzeitig muss das Gerät auch über eine möglichst geringe Trägheit verfügen, da Beschleunigungen sonst wegen dem erhöhten Kraftaufwand als nicht realistisch empfunden werden. Weiter sollte das Gerät ergonomisch sein, schlechter Tragekomfort schränkt den Benutzer im Umgang mit der virtuellen Umgebung ein. [Alakärppä et al. '98]

4.4.3 Ausgabegeräte

Im Folgenden werden einige Ausgabegeräte vorgestellt und hinsichtlich ihrer Tauglichkeit untersucht.

Cybertouch

Cybertouch ist ein Upgrade für den Datenhandschuh der Immersion Corporation und besteht aus insgesamt 6 Vibratoren, die an der Oberseite der vorderen Fingerglieder und an der Unterseite der Handfläche angebracht werden (siehe Abb. 4.11). Diese Vibratoren können einzeln angesteuert werden und sollen Eigenschaften wiedergeben wie die Oberflächenbeschaffenheit eines virtuellen Objektes beim Darrüberfahren. Es können Frequenzen zwischen 0-125 Hz erzeugt werden. Die Positionierung der Vibratoren scheint mir nicht geeignet für eine realistische haptische Wiedergabe. Durch die Anbringung der Vibratoren auf der Fingeroberseite können nicht wie erwünscht Vibrationen an der Fingerunterseite simuliert werden. Auch sind bei einem Auflösungsvermögen von 3-4 mm bei Frequenzen unter 60 Hz ein Vibrator pro Finger viel zu wenig. [Corporation '03]

Abbildung 4.11: Datenhandschuh mit Cybertouch-Upgrade

Abbildung 4.12: Datenhandschuh mit Cybergrasp-Update

Abbildung 4.13: Mobil-Upgrade für Cybergrasp

Cybergrasp

Cybergrasp ist ein Exoskelett für den Datenhandschuh der Immersion Corporation (siehe Abb. 4.12). Hier können über Seilzüge auf die einzelnen Finger Kräfte relativ zum Handgelenk gewirkt werden. Ausgenommen ist der kleine Finger. Mit diesem Gerät kann z.B. der

Widerstand beim Greifen eines Objekts simuliert werden. Allerdings kann wegen der Kraftübertragung durch Seilzüge kein Widerstand beim Öffnen einer Faust erzeugt werden. Und die maximal erzeugbare Kraft von 12 N pro Finger kann recht einfach überwunden werden, so dass die Finger doch in ein virtuelles Objekt eindringen können, das eigentlich fest sein sollte. Für Cybergrasp ist ein Mobilitäts-Upgrade erhältlich, das dem Benutzer ermöglicht, die externen Komponenten am Rücken zu tragen (siehe Abb. 4.13). Der Benutzer ist immer noch per Kabel mit einem Rechner verbunden, aber so wird zumindest eingeschränkt der Einsatz in augmented Reality ermöglicht. [Corporation '03]

Cyberforce

Cyberforce ist ein weiteres Upgrade für den Datenhandschuh der Immersion Corporation und kann mit Cybergrasp kombiniert werden. Es handelt sich dabei um einen Roboterarm, der auf der einen Seite in der Umgebung verankert ist und auf der anderen Seite am Handgelenk des Benutzers befestigt wird. Es handelt sich also um ein umgebungsbezogenes Force-Feedback-Gerät, dass gleichzeitig auch als Tracker genutzt werden kann (siehe Abb. 4.14 und Abb. 4.15). Theoretisch sollte dieses System dem Benutzer ermöglichen, z.B. die Hand auf einem virtuellen Tisch abzulegen. Allerdings erscheint mir die maximal erzeugbare Kraft von 8,8 N viel zu gering, um der Gewichtskraft entgegen wirken zu können, so dass der Widerstand leicht überwunden werden kann und damit das Eindringen in virtuelle feste Objekte ermöglicht wird. Und die minimal erzeugte Gegenkraft von 6,6 N verletzt wiederum das Gebot eines möglichst geringen Eigenwiderstands. Der Benutzer wird deutlich mehr Kraft aufwenden müssen als gewohnt, um seine Hand oder den ganzen Arm in (virtuellem) freiem Raum zu bewegen. [Corporation '03]

Abbildung 4.14: Umgebung mit zwei Cyberforce-Geräten

Abbildung 4.15: Cyberforce gekoppelt mit Cybergrasp

Abbildung 4.16: PHANToM

PHANToM

Bei PHANToM handelt es sich um einen umgebungsbezogenen Feedback-Roboterarm, der gleichzeitig als Eingabegerät dient. Das Tracking und die haptische Wiedergabe besitzt 6 Freiheitsgrade. Die Bedienung des Geräts erfolgt durch Greifen und Bewegen des Armendes, die haptischen Reize werden folglich über die benutzte Hand erfasst. Dieses Gerät ist natürlich nicht dazu geeignet, in der virtuellen Umgebung eine ganze Hand zu repräsentieren. Eine

geeignete Repräsentation wäre entweder nur ein Punkt oder ein einfaches Werkzeug wie ein Stift. [Technologies ']

4.5 Software

Nachdem nun Methoden erläutert wurden, einzelne haptische Reize zu erzeugen, und einige Wiedergabegeräte für Haptik vorgestellt wurden, interessiert natürlich die Ansteuerung dieser Geräte. Diese muss über die eingesetzte Software erfolgen, die die virtuelle Umgebung bereitstellt. Zunächst werden in diesem Kapitel die Voraussetzungen für die Simulation erläutert, dann wird auf mögliche Algorithmen und deren Probleme eingegangen.

4.5.1 Voraussetzungen

Damit aus einer virtuellen Umgebung heraus haptische Wiedergabegeräte angesteuert werden können, müssen entsprechende Treiberschnittstellen vorhanden sein. Die Übergabe von Kraftvektoren, Vibrationsfrequenzen und Temperaturwerten sollte möglich sein.

Weiter ist wichtig, dass die Berechnung der Ausgabewerte wie Kraftvektoren in Echtzeit, also mit geringer Latenz, erfolgt. Ist die Latenz zu hoch, stimmt das visuelle Bild nicht mit den haptischen Reizen überein, was den Wert der haptischen Ausgabe deutlich mindert. Auch sollte die Frequenz der Berechnungen bei mindestens 1000 Hz liegen, da ja bis zu dieser Schwelle noch Vibrationen gefühlt werden können.

Natürlich ist auch eine Erweiterung der Eigenschaften der einzelnen Objekte in der virtuellen Umgebung notwendig. Es werden Informationen über die Oberflächenbeschaffenheit benötigt, wie z.B. ein Reibungskoeffizient oder eine haptische Textur, die die Oberfläche moduliert. Auch sind Informationen über die Elastizität bzw. die Härte eines Objekts nötig. Weiterhin muss die Masse und eventuell auch die Temperatur bzw. Wärmeleitfähigkeit eines Objekts bekannt sein. Besitzt das Objekt bewegliche Teile, so werden auch Informationen über die Reibung zwischen diesen Teilen benötigt.

Eine weitere Voraussetzung ist die geeignete Repräsentation des haptischen Ein-/Ausgabegeräts in der virtuellen Umgebung. Es bietet sich an, jedem Ausgabepunkt eines haptischen Geräts eine Koordinate im Raum zuzuweisen. Dieser Punkt sei als **Tool-Tip** bezeichnet. Ein PHANToM-Gerät müsste z.B. nur durch ein Tool-Tip repräsentiert werden, da es nur ein haptisches Ausgabesignal an der Hand des Benutzers gibt. Bei Cybergrasp wiederum müsste jede Fingerspitze, die an das Exoskelett gebunden ist, ein eigenes Tool-Tip erhalten.

Sei weiterhin festgelegt, dass Kontakt zwischen einem Element des haptischen Ein-/Ausgabegeräts und einem virtuellen Objekt besteht, wenn das entsprechende Tool-Tip innerhalb dieses Objekts liegt (siehe Abb. 4.17).

4.5.2 Berechnung des haptischen Feedbacks

Die Berechnung des haptischen Feedbacks erfolgt in mehreren Schritten [Beier et al. '03]:

1. Die Tool-Tip-Positionen müssen bestimmt werden (Tracking)

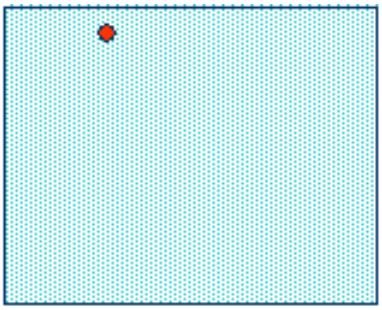


Abbildung 4.17: Kollision des Tool-Tips (rot) mit einem Objekt

2. Kollisionen zwischen dem haptischen Gerät und der virtuellen Umgebung müssen erkannt werden.

3. Die Richtung der auszugebenden Kraft muss bestimmt werden.

4. Die Stärke der auszugebenden Kraft muss bestimmt werden.

5. Der bestimmte Kraftvektor muss über die Treiberschnittstelle ausgegeben werden.

Diese Schritte werden ständig wiederholt. Die Frequenz der Wiederholung sollte wie gesagt mindestens 1 kHz betragen.

4.5.3 Ein einfaches Beispiel

Betrachtet wird die Kollision eines Tool-Tips mit einem virtuellen Objekt.
[Beier et al. '03]
Das Objekt liege in geometrischer Repräsentation vor.

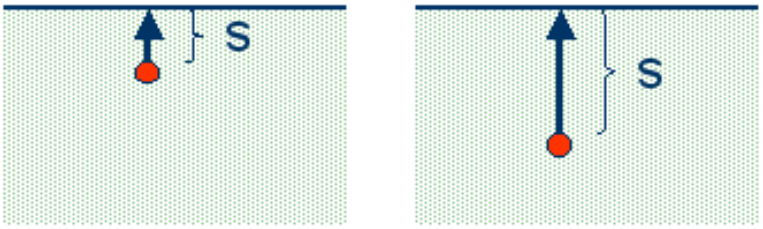
Berechnung des Kraftvektors

Das Tracking der Eingabegeräte (Schritt 1) und die Kollisionserkennung (Schritt 2) waren Themen vorangegangener Seminararbeiten, hierauf wird nicht weiter eingegangen. Es wird angenommen, dass die Tool-Tip-Position bereits bekannt ist und eine Kollision erkannt wurde.
In Schritt 3 erfolgt nun die Berechnung der Richtung der auszugebenden Kraft. Diese soll der auf das Objekt ausgeübten Kraft möglichst so entgegenwirken, dass die Tool-Tip-Position auf die Objektoberfläche zurückgestellt wird. In diesem Beispiel soll die Kraftrichtung der Richtung der Oberflächennormalen durch die Position des Tool-Tips entsprechen. Diese Oberflächennormale erhält man, indem man den Punkt auf der Objektoberfläche bestimmt, der dem Tool-Tip am nächsten liegt. Der gewünschte Richtungsvektor zeigt dann also von der Tool-Tip-Position zum gefundenen Oberflächenpunkt.
Nun muss in Schritt 4 die Kraftstärke bestimmt werden. Es ist einleuchtend, dass die auszugebende Rückstellkraft desto stärker sein muss, je tiefer das Tool-Tip in das Objekt eindringt (siehe Abb. 4.18).

Kraftstärke: $F = k * s$

Hierbei ist s der Abstand des Tool-Tips zur Objektoberfläche und k eine Federkonstante.

k sollte für harte Objekte nicht unendlich, aber ausreichend hoch gewählt werden. Bei elastischen Objekten muss k entsprechend verringert werden.

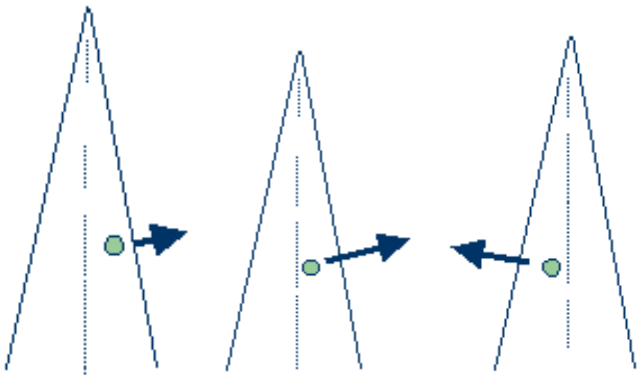


(a) (b)

Abbildung 4.18: unterschiedlich tiefes Eindringen des Tool-Tips

Probleme bei diesem Ansatz

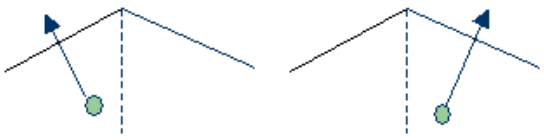
Leider hat der beschriebene Algorithmus einige negative Eigenschaften.
In einem geometrisch repräsentierten, aus Polygonen zusammengesetzten Objekt gehört zu jedem Polygon ein Bereich an Koordinatenpunkten, die diesem Polygon am Nähesten liegen. Zu einem Tool-Tip gibt es in solch einem Bereich also immer den gleichen Richtungsvektor für die wiederzugebende Kraft.
Dringt das Tool-Tip immer tiefer in ein Objekt ein, kann es zum Durchdringen des Objekts kommen (siehe Abb. 4.19). Das Tool-Tip befindet sich plötzlich in einem Bereich, der dem gegenüberliegenden Polygon näher ist, was dazu führt, dass die berechnete Kraft das Tool-Tip auf dieser Objektseite wieder heraus drückt. Dieser Effekt des Durchdringens ist bei festen Objekten nicht erwünscht und wirkt auf den Benutzer sehr irritierend.



(a) (b) (c)

Abbildung 4.19: unbeabsichtigtes Durchdringen eines Objekts

Es kann aber auch zu weniger ausgeprägten Sprüngen der Kraftrichtung kommen, wie in Abb. 4.20 zu sehen. Hier wandert das Tool-Tip von links nach recht. Zu Beginn liegt es näher am linken Polygon, im nächsten Berechnungsschritt liegt es näher am rechten Polygon. Dies führt wieder zur abrupten Änderung der Kraftrichtung, die Kante zwischen den beiden Polygonen kann gar nicht wahrgenommen werden, was für den Benutzer sehr verwirrend ist.



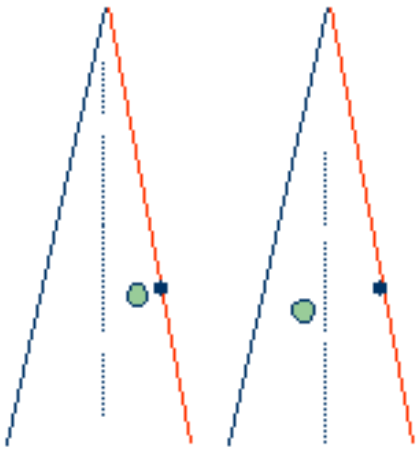
(a) (b)

Abbildung 4.20: unterschiedlich tiefes Eindringen des Tool-Tips

Constraints und God-Objekt

Ein Lösungsansatz für das Verhindern von ungewolltem Durchdringen eines Objekts ist die Einführung von constraints (Beschränkungen) und einem God-Objekt [Beier et al. '03]. Das God-Objekt ist eine Koordinate in der virtuellen Umgebung. Solange das Tool-Tip sich in freiem Raum befindet, ist die Position des God-Objekt mit der Position des Tool-Tips identisch. Wird das Eindringen des Tool-Tips in ein Objekt erkannt, wird in der ersten Iteration des Algorithmus der Kraftvektor wie gewohnt bestimmt. Allerdings bleibt das God-Objekt nun auf der Objektoberfläche und erhält die Position des zum Tool-Tip am nächsten liegenden Oberflächenpunkts. Das God-Objekt ist nun auf das Polygon beschränkt, in dem es sich in der ersten Iteration befand. Dieses Polygon wird constraint genannt. Für die weiteren Berechnungen der Kraft wird der Algorithmus nun abgewandelt. Das God-Objekt ist immer der zum Tool-Tip nächste Punkt auf der Ebene, die die constraint enthält. Die Kraftrichtung entspricht nun dem Vektor vom Tool-Tip zum God-Objekt, als Entfernung zur Objektoberfläche wird die Entfernung zum God-Objekt benutzt. Befindet sich das God-Objekt außerhalb der constraint, muss ermittelt werden, ob die Gerade durch das Tool-Tip und das God-Objekt ein Polygon schneidet, das Nachbar der aktuellen constraint ist. Wird solch ein Polygon gefunden, geht das God-Objekt darauf über und dieses Polygon wird zum neuen constraint.

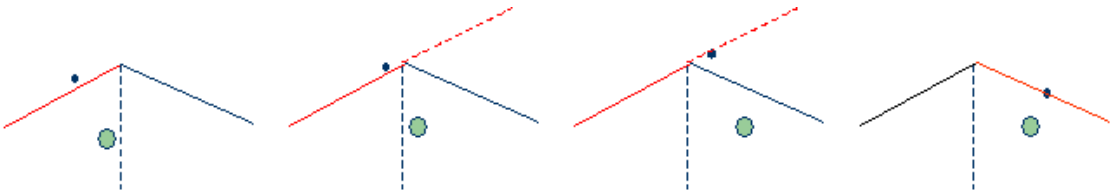
Abb. 4.21 zeigt, dass mit Hilfe des God-Objekts und der constraint das Durchdringen von Objekten verhindert werden kann.



(a) (b)

Abbildung 4.21: Durchdringen des Objekts nicht mehr möglich

Abb. 4.22 zeigt aber, dass immer noch abrupte Sprünge der Kraftrichtung möglich sind. In Abb. 4.22(c) geht das God-Objekt vom linken auf das rechte Polygon über, was eben zu einem Sprung in der Richtung des Kraftvektors führt. Die Kante kann auch hier wieder nicht gefühlt werden.

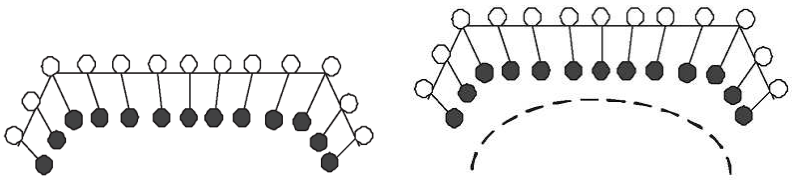


(a) (b) (c) (d)

Abbildung 4.22: God-Objekt wechselt das Polygon

Force-Shading

Die angesprochenen Sprünge der Kraftrichtung können mit Hilfe von Force-Shading verhindert werden. Dieser Ansatz arbeitet auch mit constraints, allerdings ist die Kraftrichtung nicht mehr zwingend senkrecht zum constraint. Vielmehr werden die Oberflächennormalen mit den Winkelhalbierenden der Polygonkanten interpoliert (siehe Abb. 4.23(a)). So gibt es fließende Übergänge im Kraftfeld, allerdings werden Kanten als stark abgerundet empfunden (siehe Abb. 4.23(b)). [Pohl '02]



(a) (b)

Abbildung 4.23: Force-Shading

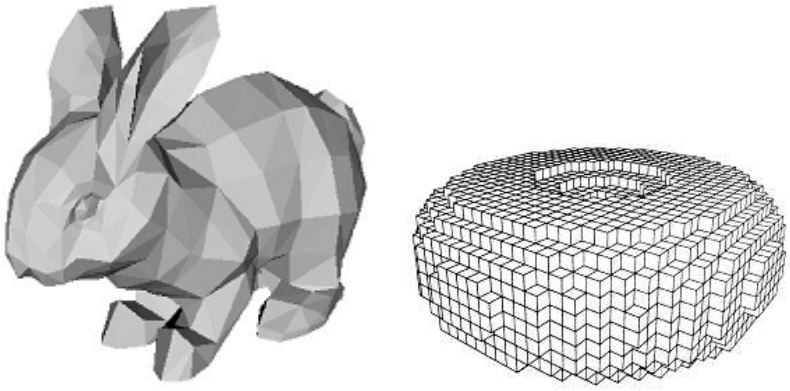
4.5.4 Einschub: verschiedene Arten der Oberflächenrepräsentation

Bevor ein weiterer Ansatz zur Berechnung des haptischen Feedbacks vorgestellt wird, werden an dieser Stelle kurz verschiedene Arten der Repräsentation von Objekten erläutert. [Sinsch '02]

Von geometrischer Repräsentation spricht man, wenn die Objektoberfläche über Polygone definiert wird. Ein Polygon kann ein beliebiges Vieleck sein, üblich ist die Verwendung von Dreiecken, bei einigen Anwendungen z.B. im Computer-Aided-Design-Bereich auch die Verwendung von Rechtecken. Beispiel siehe Abb. 4.24(a).

In einer volumetrischen Repräsentation wird nicht allein die Objektoberfläche sondern das

ganze Objektvolumen definiert. Möglich ist das Zusammensetzen des Objekts aus Würfeln, in diesem Zusammenhang Voxel genannt. Beispiel siehe Abb. 4.24(b).



(a) Polygone (b) Voxel

Abbildung 4.24: verschiedene Objektrepräsentationen

In einer impliziten Repräsentation wird ein Objekt mathematisch über eine Funktion definiert. Diese Funktion wird implizite Funktion oder Potential genannt. Eingabewerte der Funktion sind Raumkoordinaten. Ergibt die Funktion den Wert 0, so gehört der überprüfte Punkt zur Objektoberfläche. Ist der Funktionswert kleiner 0, liegt der Punkt innerhalb des Objekts, ist der Wert größer 0, so befindet sich der Punkt außerhalb des Objekts.

4.5.5 Ein weiterer Ansatz zur Berechnung des haptischen Feedbacks

Im Folgenden wird nun wieder die Kollision des Tool-Tips mit einem Objekt betrachtet. [Kim et al. '02] Das Objekt sei in geometrischer Repräsentation gegeben.
Der Algorithmus beruht darauf , dass die visuelle Darstellung des Objekts weiter auf der geometrischen Repräsentation basiert, die haptische Darstellung des Objekts aber auf einer implizit volumetrischen Repräsentation.

Berechnung der impliziten volumetrischen Repräsentation

Die implizite volumetrische Repräsentation des Objekts wird erzeugt, indem zuerst ein Quader um das Objekt gelegt wird. Dann wird dieser Quader in gleich große Unterwürfel gerastert. Nun soll in den Ecken dieser Unterwürfel das Potential des jeweiligen Punktes gegenüber dem Objekt gespeichert werden. Da das Objekt ja nur geometrisch aber nicht implizit vorliegt, könnte z.B. einfach der Abstand zur Objektoberfläche gespeichert werden. Der Abstand muss mit einem negativen Vorzeichen behaftet werden, wenn der Punkt innerhalb des Objekts liegt.

Kollisionserkennung

Ein Vorteil der gewählten impliziten volumetrischen Repräsentation liegt in der schnellen Kollisionserkennung. Zunächst muss überprüft werden, ob das Tool-Tip innerhalb eines der um ein Objekt gelegten Quader liegt. Der Aufwand beträgt hierfür bei n Objekten Schlechtestenfalls $O(n)$. Wurde ein entsprechender Quader gefunden, lässt sich in $O(1)$ der Unterwürfel bestimmen, in dem das Tool-Tip liegt. Nun müssen an der Stelle des Tool-Tips die Potentiale

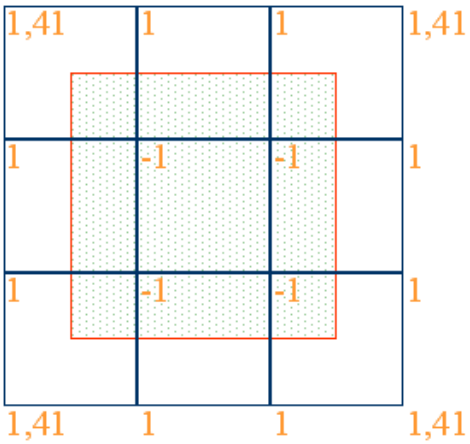


Abbildung 4.25: 2-dimensionales Beispiel für eine implizite volumetrische Objektrepräsentation

der umliegenden acht Würfelecken interpoliert werden, was auch einen Aufwand $O(1)$ bedeutet. Hat das bestimmte Potential ein negatives Vorzeichen, so liegt eine Kollision vor und der auszugebende Kraftvektor muss bestimmt werden.

Berechnung der Kraftrichtung

Um die Kraftrichtung zu bestimmen, werden die Gradienten der Würfelecken benötigt, die das Tool-Tip umgeben. Der Gradient eines Punktes ist ein Vektor, der in die Richtung des stärksten Anstiegs des Potentials zeigt. Dieser Vektor steht somit senkrecht zur Objektoberfläche. Bestimmt werden kann der Gradient einer Würfelecke mit Hilfe der Potentiale der benachbarten Würfelecken. Die berechneten Gradienten werden wieder an der Stelle des Tool-Tips interpoliert und ergeben so die Kraftrichtung.

Berechnung der Kraftstärke

Nun muss die Kraftstärke berechnet werden. Hierfür wird der sogenannte virtuelle Kontaktpunkt benötigt. Diesen erhält man, indem man einer Strecke ausgehend vom Tool-Tip mit der Richtung der bestimmten Kraftrichtung folgt und den ersten Schnittpunkt mit der Objektoberfläche sucht. Hierbei kann wieder mit Hilfe der Interpolation der gespeicherten Potentiale bestimmt werden, ob ein Streckenpunkt noch innerhalb, schon außerhalb des Objekts oder auf der Objektoberfläche liegt.

$$\vec{F} = (p_c - p_t) * k + \vec{v} * b$$

Der Kraftvektor berechnet sich nun mit der angegebenen Formel. Hier steht p_c für den virtuellen Kontaktpunkt, p_t für das Tool-Tip. Der Vektor $p_c - p_t$ gibt die Kraftrichtung an und beinhält den Abstand zwischen beiden Koordinaten. Die Federkonstante k muss wieder entsprechend der Elastizität des Objekts gewählt werden. In der angegebenen Formel wird außerdem noch die Geschwindigkeit $\vec{v}$ berücksichtigt, mit der sich das Tool-Tip bewegt. Je größer die Geschwindigkeit, desto größer auch die Rückstellkraft. Weiter ist noch ein Viskositätsfaktor b vorhanden, um Oszillationen zu verhindern.

Reibung

Bis jetzt wird bei der Berechnung des Kraftvektors noch keine Reibung berücksichtigt, so dass sich die Objektoberfläche spiegelglatt anfühlt. Reibung kann simuliert werden, indem der virtuelle Kontaktpunkt in Richtung des virtuellen Kontaktpunkts der vorherigen Iteration zurückgestellt wird. Benötigt wird also ein Rückstellvektor $\vec{F_r}$.

Die folgenden Berechnungen werden in Abb. 4.26 veranschaulicht.

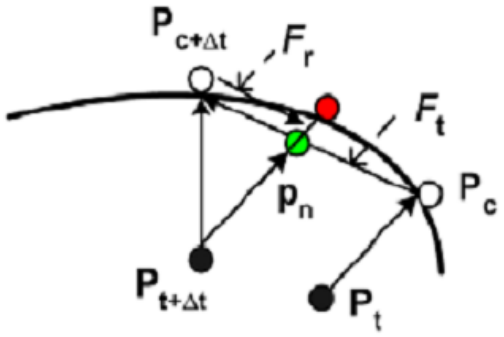


Abbildung 4.26: der neue virtuelle Kontaktpunkt $p_{c+\Delta t}$ wird zurückgestellt

$$\vec{F_r} = (p_c - p_{c+\Delta t}) * f_r$$

Der Rückstellvektor $\vec{F_r}$ zeigt vom neuen virtuellen Kontakt $p_{c+\Delta t}$ zum alten virtuellen Kontaktpunkt p_c . Der Vektor $p_c - p_{c+\Delta t}$ wird mit dem Wert eines Reibungsterms f_r multipliziert. Dieser Wert muss kleiner 1 sein, so dass der neue virtuelle Kontaktpunkt nicht ganz zum alten virtuellen Kontaktpunkt zurückgestellt werden kann.

$$f_r = f_c * (1 + (\|p_{c+\Delta t} - p_{t+\Delta t}|) * d)$$

In den Reibungsterm fließt der Reibungskoeffizient f_c der Objektoberfläche ein. Berücksichtigt wird auch, wie weit entfernt sich das Tool-Tip von der Objektoberfläche am virtuellen Kontaktpunkt befindet. Je tiefer das Tool-Tip sich im Objekt befindet, desto größer wird die Reibung. Dies fließt über den Teilterm $(\|p_{c+\Delta t} - p_{t+\Delta t}|)$ in den Reibungsterm ein. Die Tiefenkonstante d wird hierbei mit dem Abstand von der neuen Tool-Tip-Position zur Position des neuen virtuellen Kontaktpunktes multipliziert.
Mit Hilfe des Rückstellvektors $\vec{F_r}$ lässt sich jetzt der neue virtuelle Kontaktpunkt zurücksetzen.

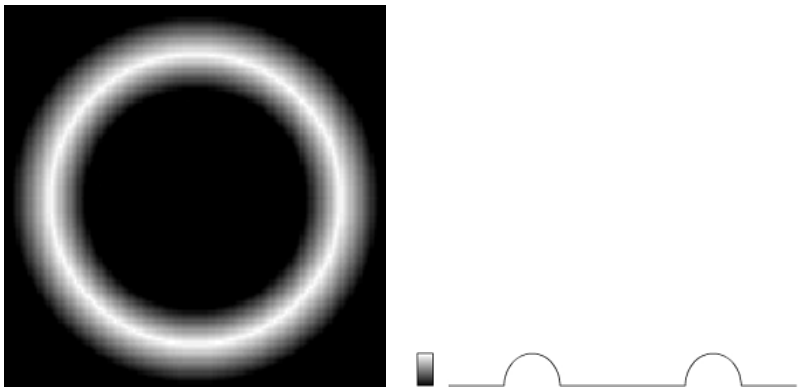
$p_n = p_{c+\Delta t} + \vec{F_r}$ Der zurückgesetzte virtuelle Kontaktpunkt p_n liegt nicht zwingend auf der Objektoberfläche, so dass eine Halbgerade ausgehend von der neuen Tool-Tip-Position $p_{t+\Delta t}$ durch den zurückgesetzten virtuellen Kontaktpunkt p_n gebildet und anschließend der Schnittpunkt dieser Halbgeraden mit der Objektoberfläche bestimmt werden muss. Dieser gefundene Schnittpunkt wird nun als neuer virtueller Kontaktpunkt verwendet und über ihn der auszugebende Kraftvektor berechnet.

4.5.6 Haptische Texturen

Im Folgenden werden noch kurz einige Methoden zur haptischen Texturierung vorgestellt [Pohl '02]. Haptische Texturen werden verwendet, um auf einfache Weise die Oberfläche eines Objekts modulieren zu können, ohne die Struktur des Objekts selber ändern zu müssen

Bump-Mapping

Beim Bump-Mapping liegt die haptische Textur als Grafik vor, die sogenannte Bump-Map. Die Grafik enthält Grauwerte, die als Höheninformationen interpretiert werden und eine Height-Map ergeben. Abb. 4.27(b) zeigt im Querschnitt die Höheninterpretation der Textur in Abb. 4.27(a). Die Textur wird um das Objekt gelegt und die Objektoberfläche muss entsprechend der Höheninformationen der Textur moduliert werden.



(a) Bump-Map

(b) Höheninterpretation im Querschnitt

Abbildung 4.27: Bump-Mapping

Lattice-Textur

Lattice-Texturen liegen in einer impliziten volumetrischen Repräsentation vor. Es handelt sich hierbei wieder um einen Wurfel, der wiederum in gleichgroße Unterwürfel unterteilt ist, in deren Ecken Änderungsvektoren gespeichert sind (siehe Abb. 4.28). Der Hauptwürfel wird um das zu modulierende Objekt gelegt und die Oberflächenpunkte werden entsprechend der an ihrer Stelle interpolierten Änderungsvektoren verschoben.

4.5.7 Kinästhetische Simulation

Zur realistischen Erzeugung haptischen Feedbacks gehört natürlich auch die Berücksichtigung von Kräften im kinästhetischen Bereich.
So spielt beim Heben eines Objekts natürlich die Gewichtskraft eine Rolle.
Generell muss bei jeder Beschleunigung eines Objekts die aufzuwendende Kraft beim Benuzter als Widerstand spürbar sein.

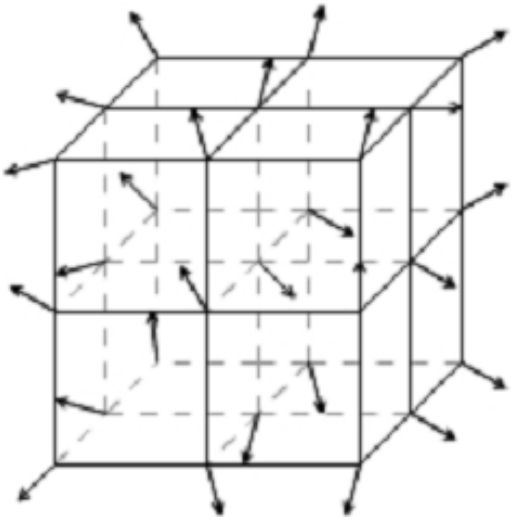


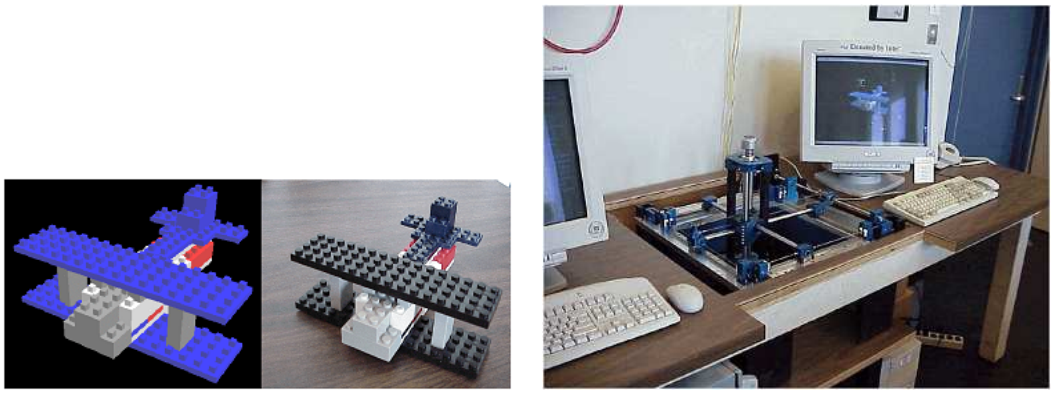
Abbildung 4.28: Lattice-Textur

4.6 Bewertung und Fazit

Nach der Vorstellung einiger Methoden zur Berechnung und Ausgabe von haptischem Feedback wird im Folgenden der Nutzwert von haptischer Wiedergabe betrachtet und abschließend ein Fazit gezogen.

4.6.1 Untersuchung zum Nutzen haptischer Wiedergabe

Im nun vorgestellten Versuch wurde anhand des Baus eines Lego-Modells untersucht, inwieweit haptisches Feedback bei Training in einer virtuellen Umgebung den Trainingseffekt steigert [Adams et al. '99].



(a) Modell virtuell und real

(b) Trainingsumgebung

Abbildung 4.29: Bilder zum Lego-Versuch

Die Versuchspersonen wurden in drei Gruppen eingeteilt, wobei darauf geachtet wurde, in jeder Gruppe ungefähr den gleichen Erfahrungsquerschnitt im Umgang mit Lego-Modellen zu erhalten.
Alle Gruppen erhielten eine zweiminütige Einweisung, in der der Aufbau des Lego-Modells

gezeigt wurde. Gruppe 1 und Gruppe 2 trainierten anschließend 30 Minuten lang den Aufbau des Modells in einer virtuellen Umgebung, wobei nur Gruppe 1 haptisches Feedback hatte. Die Interaktion mit der Trainingsumgebung erfolgte über ein (für Gruppe 1 haptisches) Ein-/Ausgabegerät mit drei Freiheitsgraden (siehe Abb. 4.29(b)), visuelles Feedback über einen Bildschirm war für beide Gruppen vorhanden.
Nach der Trainingsphase bzw. im Fall der Gruppe 3 direkt nach der Einweisung baute jeder Teilnehmer das Modell fünf mal, wobei die jeweils benötigte Zeit gemessen wurde. Die einzelnen Ergebnisse sind in Abb. 4.30 zu betrachten.

LEGO™ Model Completion Times in Seconds

treat. group	match. triad	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	test mean	pre-test mean
1	1	80	64	57	62	78	68	68
1	2	92	80	72	80	67	78	73
1	3	109	212	105	93	100	124	92
1	4	233	170	113	95	90	140	102
1	5	172	134	90	88	88	114	110
2	1	111	104	74	80	67	87	62
2	2	145	180	97	78	68	114	80
2	3	173	101	92	80	88	107	94
2	4	188	144	101	100	94	125	98
2	5	293	158	209	83	75	164	106
3	1	235	109	104	82	80	122	70
3	2	279	151	112	107	114	153	87
3	3	194	157	93	107	97	130	94
3	5	240	199	148	142	146	175	124

Abbildung 4.30: Testergebnisse

Es zeigte sich, das die Gruppen mit vorherigem Training gerade die ersten Modelle deutlich schneller bauten als die Gruppe ohne Training, bei den letzten Modellen glichen sich die Werte teilweise wieder an. Bei den Gruppen mit Training war diejenige mit haptischem Feedback zwar nur leicht aber immerhin messbar schneller als die Gruppe ohne haptischem Feedback. Das sah in der Trainingsphase deutlich anders aus. Hier baute die Gruppe 1 im Vergleich zur Gruppe 2 in der gleichen Zeit im Schnitt doppelt so viele Modelle in der virtuellen Umgebung. Diese Ergebnisse sind allerdings zu relativieren. Zum einen war die Versuchsgruppe mit 15 Teilnehmern viel zu klein, um statistisch zuverlässige Werte ermitteln zu können. Außerdem wurde mit der Hilfe von Haptik der Umgang mit der virtuellen Umgebung zwar deutlich vereinfacht, aber die Probanden benötigten selbst so im Training ungefähr 5-10 so lange wie in der Realität, um ein Modell zu bauen.

4.6.2 Fazit

Betrachtet man die vorangegange Untersuchung, so scheint virtuelles Training mit haptischer Wiedergabe bei einfacheren Aufgaben nur eine geringe Verbesserung des Trainingeffekts zu bewirken. Allerdings scheint der Umgang mit einer virtuellen Umgebung durch haptisches Feedback deutlich vereinfacht zu werden. Dadurch bedingt dürfte der Trainingseffekt bei komplexeren Anwendungen mit Hilfe von haptischer Wiedergabe gesteigert werden.

Abschließend bleibt noch zu erwähnen, dass es bis jetzt noch keine auch nur annähernd ideale haptische Ausgabegeräte gibt. Vor allem die Ausgabe aller haptischer Reize am ganzen Körper, ohne Bewegungseinschränkungen, ist noch Zukunftsmusik.

Literaturverzeichnis

[Adams et al. '99] Richard J. Adams, Daniel Klowden und Blake Hannaford. Virtual Training for a Manual Assembly Task. 1999. http://www.haptics-e.org/Vol_02 (gesehen 08/2003).

[Alakärppä et al. '98] Juha Alakärppä, Heikki Kurki und Sauli Ojalehto. 1998. http://www.ratol.fi/lisenssi/virtuaali/vr8.ppt (gesehen 08/2003).

[Beier et al. '03] Florian Beier, Oliver Schuppe und Timo Stich. Haptik. 2003. http://www-li5.ti.uni-mannheim.de/vipa/endosim/vr-seminar/ (gesehen 08/2003).

[Böhme & Sotoodeh '99] Denis Böhme und Masoud Sotoodeh. Haptic und Forcefeedback. 1999. http://www.informatik.uni-bremen.de/~nostromo/haptik/index.html (gesehen 08/2003).

[Corporation '03] Immersion Corporation. 2003. http://www.immersion.com (gesehen 07/2003).

[Doerrer '01] Dipl.-Ing. Christoph Doerrer. Allgemeine Informationen zu dem Forschungsthema Sensor- und Aktorsysteme für haptische Displays. 2001. http://www.emk.e-technik.tu-darmstadt.de/~doerrer/Forschung/Allgemein/allgemein.html (gesehen 08/2003).

[Howe '] Robert Howe. Introduction to Haptic Display: Tactile display. http://haptic.mech.nwu.edu/intro/tactile/ (gesehen 07/2003).

[Kim et al. '02] Laehyun Kim, Anna Kyrikou, Gaurav S. Sukhatme und Mathieu Desbrun. An Implicit-based Haptic Rendering Technique. 2002. http://www-grail.usc.edu/Haptic (gesehen 08/2003).

[(MEDTECH) '01] Forschungszentrum Karlsruhe Programm Medizintechnik (MEDTECH). Virtual Endoscopic Surgery Training (VEST). 2001. http://www-kismet.iai.fzk.de/KISMET/VestSystem.html (gesehen 08/2003).

[Pohl '02] Christof Pohl. Haptische Texturierung. 2002. http://www-li5.ti.uni-mannheim.de/vipa/endosim/seminar/ (gesehen 08/2003).

[Rainer Zwisler '98] Rainer Zwisler. Virtuelle Realität und die Rolle von Haptik. 1998. http://www.zwisler.de/scripts/haptics/haptics.html (gesehen 08/2003).

[Sinsch '02] Rainer Sinsch. 3D Algorithmen in der grafischen Datenverarbeitung - Einführung und Grundlagen. 2002. http://homepages.fh-giessen.de/~hg11001/mm/3DAlgorithmen_EinfuehrungUndGrundlagen.pdf (gesehen 08/2003).

[Site '01] Tom's VR Site. Survey of Haptic Interfaces: Force Feedback. 2001. http://home-3.12move.nl/~sh290334/dbase_force/ (gesehen 08/2003).

[Technologies ’] SensAble Technologies. http://www.sensable.com (gesehen 07/2003).

[Theisinger ’02] Rolf Theisinger. Haptische 3D-Benutzungsoberfläche. 2002. http://www.agc.fhg.de/agc/publication/pdf/Diplomarbeiten_final/Diplomarbeit_Theisinger.pdf (gesehen 08/2003).

[Zachow et al. ’96] Stefan Zachow, Johann-Habakuk Israel und Kai Köchy. VR Ausgabegeräte. 1996. http://cg.cs.tu-berlin.de/~kai/tablet/vr_out.html (gesehen 08/2003).

5 Interaktive Animation

Markus Schwarz

5.1 Einführung

Menschliche Bewegungsdaten sind eine Vorraussetzung für die Kontrolle virtueller Charakter in 'Virtual Reality' und 'Augmented Reality' Umgebungen sowie bei Computeranimationen, in Filmen und Spielen. Eine realistisch wirkende Animation zu erstellen ist keine einfache Aufgabe. Der Gang eines Menschen zum Beispiel ist nicht nur ein einfaches „einen Fuß vor den anderen setzen“, sondern eine individuelle Folge von Bewegungen des ganzen Körpers.
Neben dem Wunsch möglichst realistische Bewegungsabläufe zu generieren, ist auch die Bearbeitung vorhandener Bewegungsdaten ein erstrebenswertes Ziel. Schließlich soll sich ein Avatar in einer virtuellen Umgebung „einleben“, sich darin bewegen und interagieren. Dazu müssen Bewegungsdaten dynamisch bearbeitet und an die jeweilige Situation angepasst werden. Neben der eigentlich Animation ist auch die Kontrolle eines virtuellen Charakters von großer Bedeutung.
Diese Ausarbeitung soll einen kleinen Einblick über Verfahren geben, die es uns ermöglichen, realistische Bewegungen aufzunehmen, auszuwerten, zu bearbeiten und anzuwenden.

5.2 Motion Capture

5.2.1 Rückblick

Beim 'Motion Capture', kurz MoCap, werden reale Bewegungen eins zu eins auf den virtuellen Charakter abgebildet. Somit wird die Bewegung nicht künstlich erzeugt, sondern direkt der Natur entnommen, wodurch Bewegungen sehr realistisch wirken.
Wie bereits bekannt gibt es verschiedene MoCap-Systeme die eingeteilt werden können in optische, magnetische, mechanische, akustische, bildverarbeitende und Hybrid-Systeme.
Die beim MoCap-Prozeß gewonnen Daten können auf verschiedene Weise repräsentiert werden. Für Animationen wird im Allgemeinen ein detailliertes Modell eines Skeletts verwendet. Dieses Modell erlaubt eine einfache weitere Bearbeitung der Daten und ist nicht gebunden an einen einzigen Körper. Das Skelett wird dargestellt durch die Länge der Gliedmaßen und die Winkel der Gelenke. Eine Bewegungssequenz besteht dann aus der (Flug-)Bahn des Wurzelgelenks (zum Beispiel dem Becken) und den relativen Gelenkwinkeln der einzelnen Köperteile. Diese Daten werden dann in einer Bewegungsdatenbank abgelegt, damit sie später wiederverwendet und bearbeitet werden können.

Egal welches MoCap-System eingesetzt wird, es ist unmöglich die Marker bzw. Sensoren direkt an den Gelenken oder Knochen anzubringen. Bei markerbasierten Systemen muss deswegen auf den Abstand zwischen Marker und dem echten Skelett geachtet werden. Dies ist vor allem bei Muskeln und Gelenken schwierig, da hier die stärksten Abweichungen auftreten.

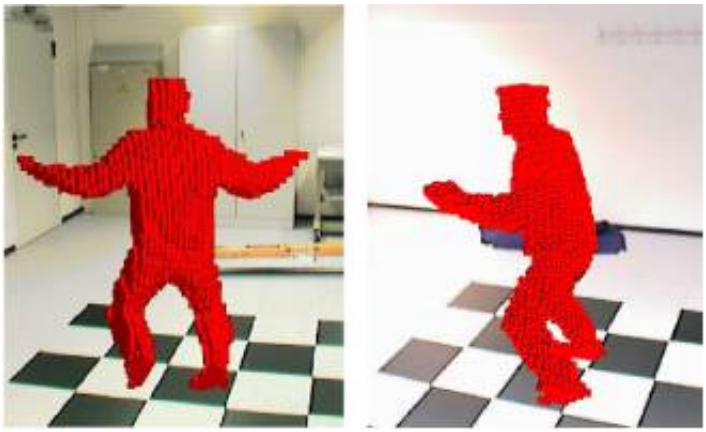


Abbildung 5.1: Visuelle Hülle eines Menschen zurückprojiziert in 2 Kameraansichten

5.2.2 Extraktion des menschlichen Skeletts

Das folgende Verfahren ist für markerlose optische MoCap-Systeme entworfen worden [Theobalt et al. '01]. Es ermöglicht das Extrahieren des Skeletts anhand einer farbbasierten Merkmalserkennung und der visuellen Hülle eines Menschen.
Wie bereits aus dem Seminarvortrag 'Motion Tracking' bekannt können hervorstechende Merkmale wie die Hände, der Kopf (Gesicht) und die Füße anhand ihrer Farbwerte ausfindig gemacht werden. Auch die Berechnung der visuellen Hülle anhand mehrerer Kamerabilder wurde dort bereits erläutert. Die visuelle Hülle wird dargestellt durch eine voxelbasierte Rekonstruktion des Menschen (siehe Abbildung 5.1).

Algorithmus

Der Algorithmus schätzt die Gelenkparameter zu jedem Zeitpunkt t anhand der aufgenommen Bewegungssequenz ab (siehe auch Abbildung 5.2).
Er benutzt als Eingabe:

1. die visuelle Hülle
2. die Standorte des Kopfes, der Hände und der Füße
3. die Parameter aus dem vorhergehenden Schritt *t-1*

Die Gelenkparameter zum Zeitpunkt t=0 sind wegen der Startposition der Person bekannt. Die Ausdehnung des Modells wird angepasst an die Ausdehnung der Person, die vor den Kameras agiert. Dies geschieht durch ein vorheriges Ausmessen der Länge der Gliedmaßen. Diese Daten werden dann ebenfalls in das System eingebracht.

Der menschliche Körper wird nun modelliert als ein zweischichtiges Bewegungsskelett. Die erste Schicht des Modells besteht aus 7 Gelenken und 10 Knochensegmenten. Die Arm- und Beinsegmente der ersten Schicht sind in der Länge variabel. Die zweite Schicht verfeinert die erste Schicht durch Ober- und Unterarmsegmente sowie durch Ober- und Unterschenkelsegmente. Die Länge dieser neuen Segmente (Ober-, Unterarm, Ober-, Unterschenkel) ist konstant und durch die Initialisierung bekannt. Zusammen mit der ersten Schicht bilden sich Dreiecke

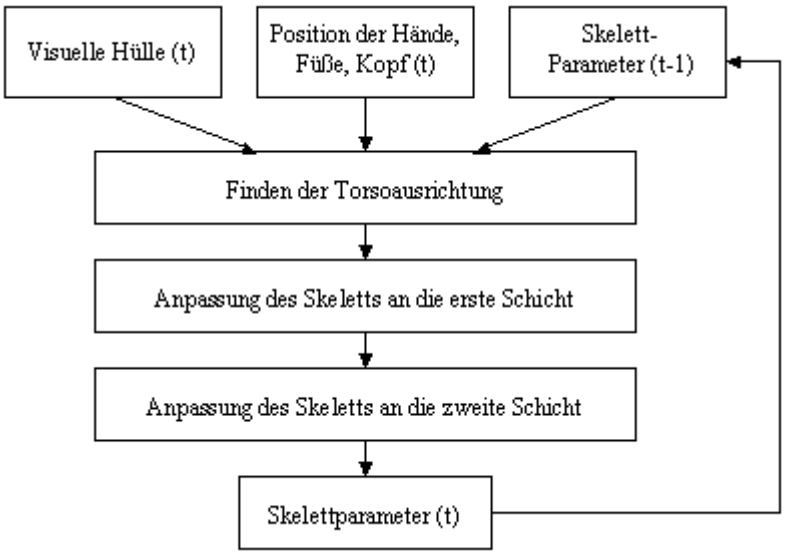


Abbildung 5.2: Übersicht der Skelettextraktion

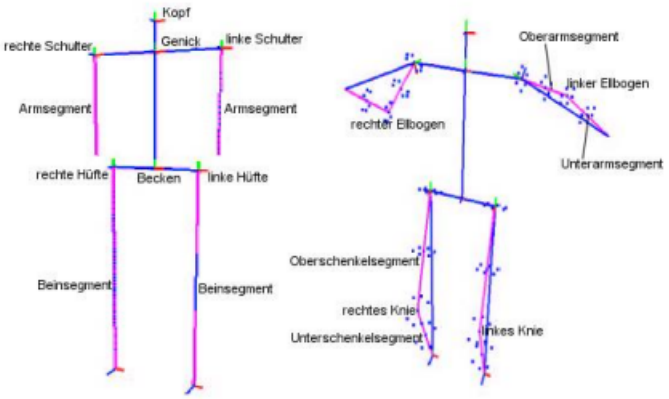


Abbildung 5.3: links: erste Schicht des Skelettmodells; rechts: zweite Schicht des Skelettmodells

an den Beinen und Armen aus den neuen Segmenten plus den alten Segmenten (siehe Abbildung 5.3). Die erste Schicht hat insgesamt 24 Freiheitsgrade, also 6 für das Beckengelenk (Translation auf x-, y-, z-Achse; Rotation um x-, y-, z-Achse) und nochmal 3 für jedes Weitere Gelenk. Die zweite Schicht erweitert die Erste durch 4 Freiheitsgrade (1 für jedes neue Ellbogengelenk bzw. Kniegelenk).
Zu jedem Zeitpunkt t durchläuft die Skelettextraktion folgende Punkte (siehe auch Abbildung 5.2)

1. Finden der Torsoausrichtung (Torso = Rumpf, also menschlicher Oberkörper ohne Kopf und Arme)
2. Anpassung des Skeletts an die erste Schicht
3. Anpassung des Skeletts an die zweite Schicht

a. Torsoausrichtung: Das erkennen der Schulterposition direkt aus den optischen Daten ist schwierig, da die Schultern keine erkennbaren hervorstechenden Merkmale besitzen. Die visuelle Hülle jedoch kann benutzt werden um die Schulterposition und die Torsoausrichtung zu bestimmen. Da sowohl Schulterposition als auch Torsoausrichtung zum Zeitpunkt *t-1* bekannt sind, und man davon ausgehen kann, das Änderungen hier nur gering sind, lässt man die aus dem vorherigen Zeitpunkt bestimmten Daten mit einfließen um das Ergebniss zu verbessern.

b. Anpassungen des Skeletts an die erste Schicht: Aus der farbbasierten Merkmalsverfolgung sind die Positionen des Kopfes, der Hände und der Füße zu jedem Zeitpunkt bekannt. Es wird angenommen das die Hüftknochen parallel zum Boden sind. Jetzt kann man zusammen mit der Torsoausrichtung und der Schulterposition die erste Schicht berechnen. Es wird sowohl die Distanz zwischen der Schulter und den Handgelenken als auch die Distanz zwischen der Hüfte und den Fußknöcheln berechnet, welches die Länge der Arm- bzw. Beinsegmente der ersten Schicht ergibt (siehe Abbildung 5.3).

c. Anpassungen des Skeletts an die zweite Schicht: Die Volumendaten, also die visuelle Hülle, wird nun benutzt um die letzten vier Freiheitsgrade der zweiten Schicht zu bestimmen. Die Länge der Arm- und Beinsegmente aus der zweiten Schicht sind konstant. Die direkte Distanz zwischen der Schulter und den Handgelenken bzw. der Hüfte und den Fußknöcheln sind aus der ersten Schicht bekannt. Hieraus lassen sich direkt die Winkel der Ellbogen und Kniegelenk bestimmen. Einzig offen bleibt noch die Rotation des gesamten Arm- bzw. Beinsegments aus der ersten Schicht. Um diese Rotation zu finden wird der Punkt des Ellbogen- bzw. Kniegelenks auf der visuellen Hülle gesucht (siehe Abbildung 5.4). Dadurch ist die Rotation genau bestimmt.

Beispiel eines fertig angepassten Skeletts siehe Abbildung 5.5

5.2.3 Fazit

Bei anderen MoCap-System kann sich die Extraktion des Skeletts auch einfacher gestalten, da keine visuelle Hülle oder ähnliches berechnet werden muss, sondern direkt auf günstig platzierte Marker zurückgegriffen werden kann. Wie bereits bekannt können 'Motion Capture Daten' auch gefiltert werden (z.B. Kalman Filter) um falsch sitzende oder nicht sichtbare Marker zu korrigieren.

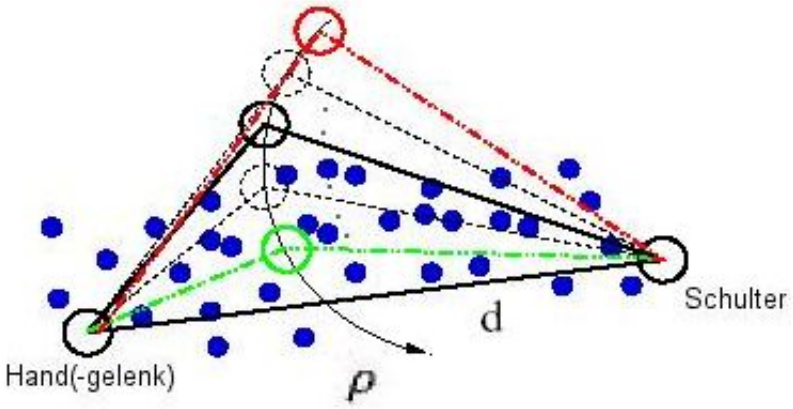


Abbildung 5.4: Die kleinen Kugeln repräsentieren die Voxel der visuellen Hülle. d ist der in der ersten Schicht berechnete Abstand zwischen Schulter und Handgelenk. p ist die Rotation. Der grüne Kreis repräsentiert die wahrscheinlichste Position des Ellbogengelenks.

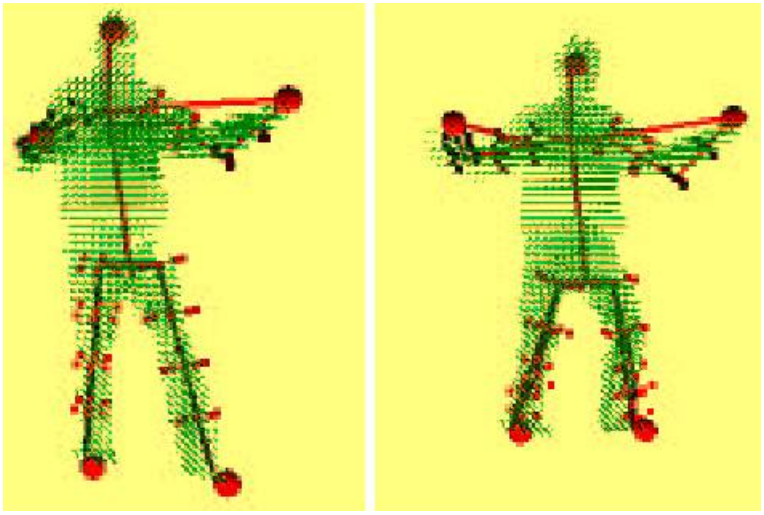


Abbildung 5.5: Zwei Beispiele des Skelettmodells nach der Anpassung an die zweite Schicht. Rote Kugeln markieren die farbbasierten Merkmale (Hände, Kopf, Füße).

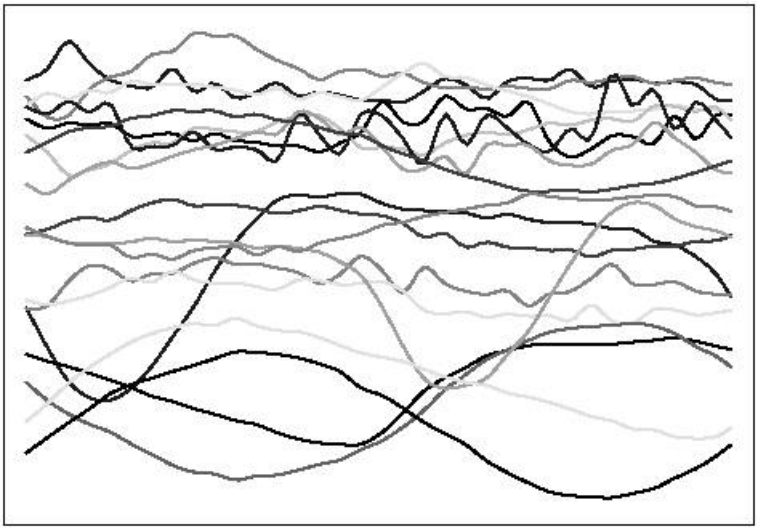


Abbildung 5.6: Bewegungskurven eines gehenden Menschen.

MoCap ist ein zuverlässiger Weg um realistische Bewegungen zu erhalten. Da es jedoch sehr teuer und zeitaufwendig ist neue Bewegungen aufzunehmen, wäre es wünschenswert, die vorhandenen Daten modifizieren und neue Bewegungsabläufe dynamisch erzeugen zu können, ohne dabei die Realitätsnähe der Animation zu gefährden.

5.3 Motion Warping

5.3.1 Einleitung

Eine einfache Möglichkeit, um vorhandene Bewegungen zu bearbeiten, bietet der Ansatz des 'Motion Warping' [Witkin & Popovic '95]. Die Bewegung eines gehenden Menschen zum Beispiel hat die Bewegungskurven wie in Abbildung 5.6.
Bewegung wird beschrieben durch eine Menge von Bewegungskurven, wobei jede Kurve gegeben ist durch die Parameter des Modells (z.B. Gelenkwinkel des rechten Unterarms). Diese Kurven sind eine Funktion über die Zeit.
Erwünschenswert ist nun das Ableiten neuer Bewegungskurven aufgrund einer Menge von Bedingungen, die vom Benutzer gesetzt werden. Allerdings müßen diese Bedingungen so geartet sein, das die abgeleiteten neuen Kurven ähnlich der originalen Kurven sind, und zwar in der Hinsicht, daß die feinen Details der Bewegung erhalten bleiben.

5.3.2 Algorithmus

Jede Kurve wird unabhängig bearbeitet so daß wir im folgenden nur eine einzelne Kurve $\theta(t)$ betrachten. Wie im herkömmlichen Keyframing enthalten die Bedingungen eine Menge von Tupeln (θ'_i, t_i) wobei jedes Tupel den Wert angibt, den θ' (das ist die neue Bewegungskurve = 'warped motion curve') zur spezifizierten Zeit t_i annimmt, mit $1 \leq i \leq$Anzahl der Keyframes. Die neue 'gewarpte' Bewegungskurve wird transformiert duch folgende Formel

$$\theta'(t) = a(t) * \theta(t) + b(t)$$

wobei $a(t)$ eine Skalierungs- und $b(t)$ eine Offsetfunktion ist. Diese beiden Funktionen müßen der folgenden Gleichung genügen

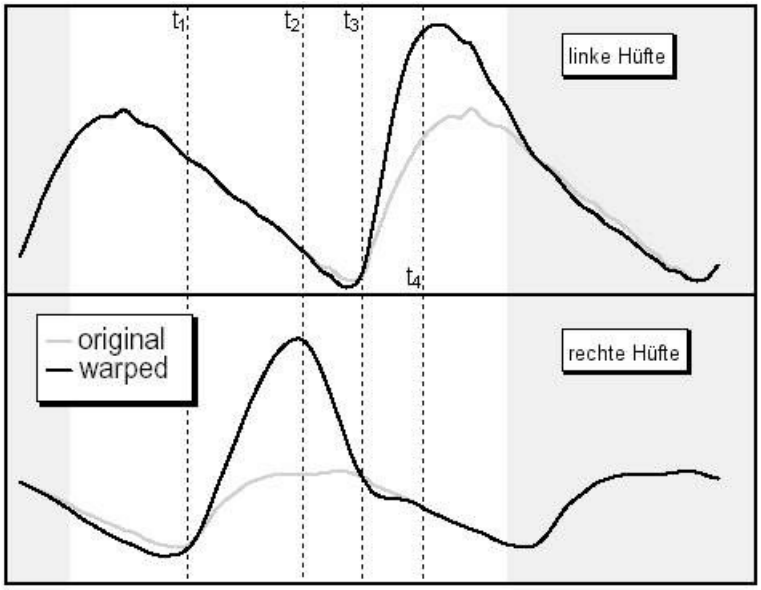


Abbildung 5.7: Originalkurven und neue Bewegungskurven für die linke und die Rechte Hüfte. Die vertikalen Linien kennzeichnen die Keyframes.

$$\theta'_i(t_i) = a(t_i) * \theta(t_i) + b(t_i) \ , \forall i$$

Dies bedeutet einfach, daß die Punkte der definierten Tupel (θ'_i, t_i) auf der neuen Bewegungskurve θ' liegen müßen. Ein Problem ist nun, das die Werte von a und b nicht eindeutig bestimmt werden können an den Keyframes zum Zeitpunkt t_i. Aus diesem Grund muß der Benutzer an jedem Keyframe i entscheiden, ob die neue Kurve im Punkt i durch Skalierung oder Offsetverschiebung berechnet werden soll. Wurde Skalierung gewählt, wird $b(t_i)$ konstant gehalten und man erhält $a(t_i) = \frac{\theta'_i(t_i) - b(t_i)}{\theta(t_i)}$ durch Umstellung der vorherigen Formel. Wurde Verschiebung gewählt, halten wir $a(t_i)$ konstant und lösen nach $b(t_i)$ auf.
Häufig ist die Skalierung nicht erwünscht, zum Beispiel bei einer Translation oder Rotation der gesamten Bewegung. Skalierung der Gelenkwinkel ist zum Beispiel nützlich zur Übertreibung. In diesem Fall wird die Offset-Funktion auf null gesetzt.
Sobald alle Werte von $a(t_i)$ und $b(t_i)$ errechnet wurden, können $a(t)$ und $b(t)$ durch einen interpolierenden Spline konstruiert werden.

5.3.3 Fazit

Das genannte Verfahren macht keinen großen Unterschied zum gewöhnlichen Keyframing: Der Benutzer wählt ein paar Frames aus und markiert sie als Keyframes. An diesen Keyframes kann der Benutzer nun die Haltung des Modells verändern, woraus die Bedingungen (θ'_i, t_i) resultieren. Außerdem muß der Benutzer angeben, ob eine Skalierung oder eine Offsetverschiebung, welche standardmäßig verwendet wird, stattfinden soll. Das ändern der Haltung muß natürlich nicht jede Bewegungskurve beeinflußen, schließlich geht ein Mensch nicht zwingenderweise anders wenn er den Arm hebt.
Abbildung 5.7 zeigt die Bewegungskurven der linken und rechten Hüfte. In Abbildung 5.8 sehen wir zusätzlich die Offsett- und Skalierungsfunktion der rechten Hüfte. Hier wird die Skalierungsfunktion als konstant eins angesehen, da nur eine Offsetverschiebung stattfinden soll. Der Punkt (θ'_2, t_2) auf der gewarpten Kurve im oberen Bild repräsentiert die vom Benutzer ein-

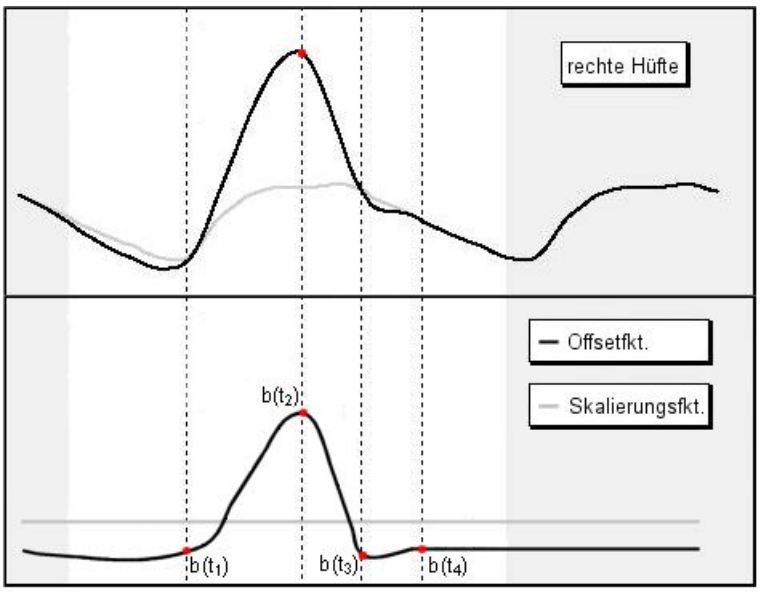


Abbildung 5.8: Oberes Bild: Bewegungskurve der rechten Hüfte. Unteres Bild: Zugehörige Offset- und Skalierungsfunktion.

gestellte Position der rechten Hüfte zum Zeitpunkt t_2. Die Punkte (θ'_1, t_1), (θ'_3, t_3) und (θ'_4, t_4) auf der gewarpten Bewegungskurve sind identisch mit der originalen Bewegungskurve, da hier der Benutzer die Haltung der Hüfte nicht verändert hat. Die roten Punkte $b(t_i)$ im unteren Bild sind die errechneten Offsetwerte zwischen der originalen Kurve θ und den Punkten auf der gewarpten Bewegungskurve. Aus diesen Punkten läßt sich nun die Offsetfunktion $a(t)$ berechnen und die neue Bewegungskurve θ'. Man sieht hier sehr schön, daß die Offsetfunktion an den Stellen null ist, an denen die Originalkurve und die gewarpte Bewegungskurve identisch sind. Abbildung 5.9 zeigt nun ein paar ausgewählte Frames von der originalen und der gewarpten Bewegungssequenz. Übrigens entspricht der nicht ausgegraute Teil der Hüftbewegungskurven aus Abbildung 5.7 der Bewegungssequenz aus Abbildung 5.9.

Neben dem Ändern der Bewegungskurven kann 'Motion Warping' auch eingesetzt werden, um mehrere Clips ineinander überzublenden. Ebenso können Zeitschleifen eingebaut werden, um bestimmte Teile einer Sequenze zu wiederholen.

5.4 Bewegungsgraph

5.4.1 Einleitung

Eine andere Methode, um realistische Bewegungen aus einer Sammlung von MoCap-Daten zu erzeugen ist das erstellen eines Bewegungsgraphen ('Motion Graph') [Kovar et al. '02]. Dieser Graph enthält sowohl Originaldaten als auch automatisch generierte Übergänge. Die Knoten des Graphen dienen als Entscheidungspunkte für den nahtlosen Übergang zu einem neuen Bewegungsknoten. Eine Bewegungssequenz ist dann nichts anderes als ein Pfad auf dem Graphen. Durch die Benutzung von Algorithmen aus der Graphentheorie kann ein Pfad generiert werden, der den Benutzereingaben genügt und uns Kontrolle über die Bewegung des Charakters gibt. Es ist außerdem möglich zusätzliche Bedingungen an eine Bewegung zu knüpfen,

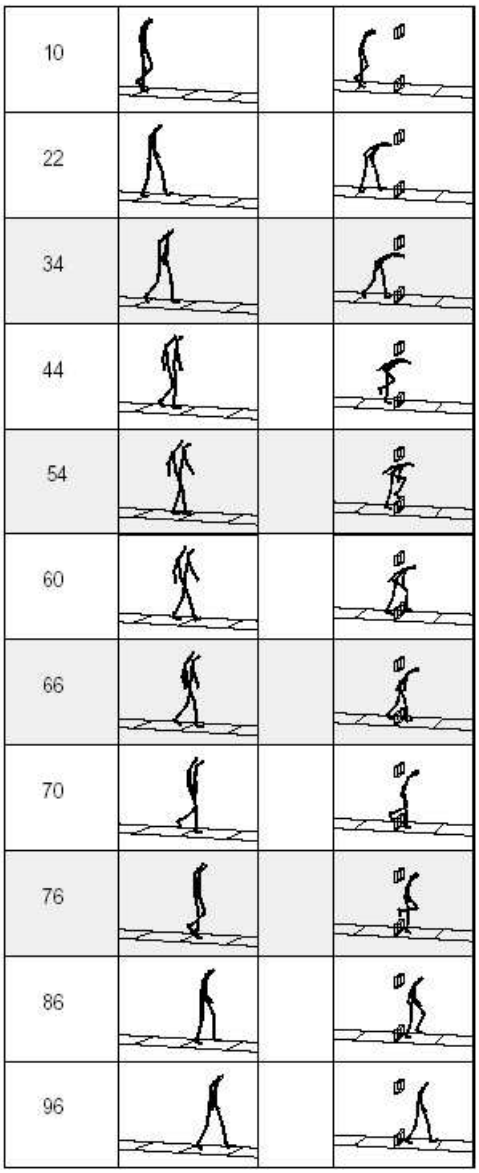


Abbildung 5.9: Ausgewählte Frames einer originalen und einer gewarpten Bewegungssequenz. Die zeit läuft von oben nach unten. In jeder Zeile wird die Framenummer, der Originalframe und der gewarpte Frame gezeigt. Die vier grauen Zeilen kennzeichnen die Keyframes.

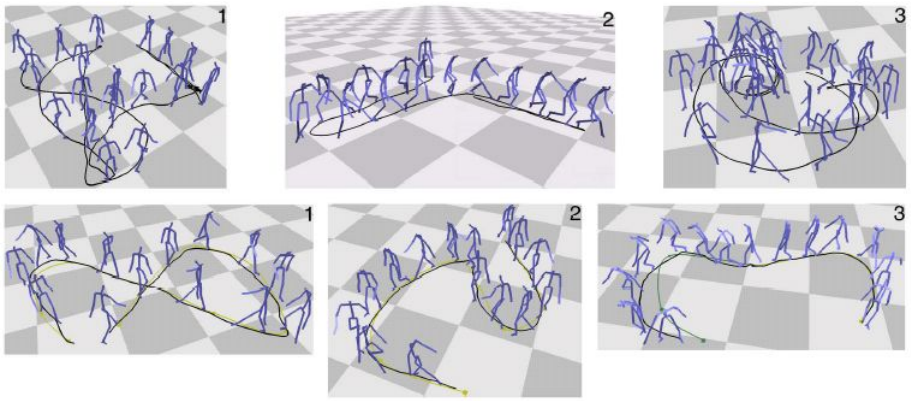


Abbildung 5.10: Die oberen Bilder zeigen originale Bewegungsdaten; zwei Gehbewegungen und eine schleichende Bewegung. Die schwarzen Bahnen zeigen die Wege denen der Charakter folgt. Die unteren Bilder zeigen Bewegungssequenzen erzeugt aus dem Bewegungsgraphen, welcher aus diesen Beipsielen erstellt wurde. Dabei gibt die gelbe Bahn den vom Benutzer spezifizierten Weg an, und die schwarze Bahn den tatsächlich gegangenen Weg. Bild 3 zeigt eine Bewegung entlang dem Weg, wobei der Charakter bei der Hälfte des Weges von normalem gehen auf schleichen umstellt.

also von welchem Typ zum Beispiel eine Bewegung ist. Angenommen man hat einen Clip, in dem der Charakter normal über eine Fläche läuft, und einen Clip, in dem der Charakter eine schleichende Bewegung ausführt. Aus diesen beiden Clips kann ein Graph erzeugt werden, dessen Kanten mit dem Bewegungstyp 'Schleichen' bzw. 'Normal gehen' versehen sind. So kann man den Charakter an verschiedenen Punkten entlang eines Weges zwingen zu schleichen und an anderen Stellen normal zu gehen (Beispiel siehe Abbildung 5.10).

5.4.2 Erzeugung des Bewegungsgraphen

Der Bewegungsgraph ist ein gerichteter Graph wobei eine Kante einem Ausschnitt (Clip) einer Bewegung entspricht. Jede Kante für sich gesehen ist also selber wieder ein kleiner Clip. Knoten dienen als Entscheidungspunkte die solche Clips verbinden. Ein einfacher Bewegungsgraph besteht aus zwei Knoten mit einer Kante die diese verbindet. Dabei ist der erste Knoten der Beginn des Clips und der Zweite das Ende des Clips. Dementsprechend kann ein Clip auch durch einfügen eines oder mehrer Knoten geteilt werden (siehe Abbildung 5.11 oben). Ein viel interessanter Graph benötigt viel mehr Knoten und Verbindungen. Es können nämlich Übergänge zwischen den Clips gefunden werden, deren Daten sehr ähnlich zueinander sind. Dadurch ist es nun möglich einen nahtlosen Übergang zwischen diesen Clips zu erzeugen (siehe Abbildung 5.11 unten).

Kandidaten für einen möglichen Übergang finden

Der einfachste Ansatz zur Lokalisierung solcher Übergangspunkte besteht in der Berechnung einer Distanz zwischen den (Körper-)Haltungen des Skeletts.
Wie auch immer ist dieser einfache Weg nicht unbedingt der beste, da zum Beispiel

- manche Gelenkwinkel eine viel größere Auswirkung auf das Gesamtbild haben wie andere (z.B. Hüftrichtung gegenüber der Brustrichtung)

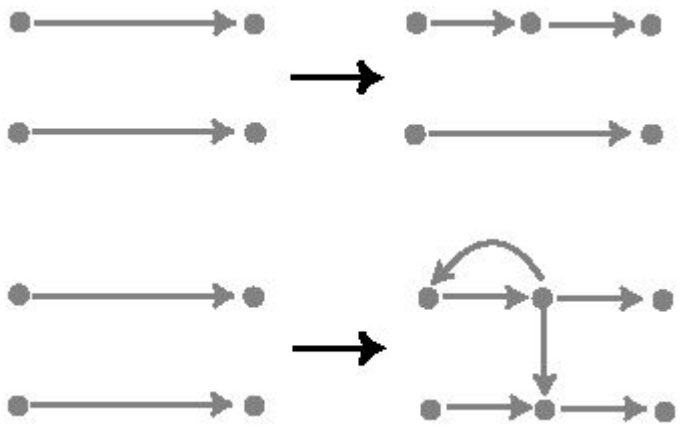


Abbildung 5.11: Man stelle sich einen Bewegungsgraphen vor, der aus zwei initialen Clips erstellt wurde. (oben) Man kann trivialerweise einen Knoten einfügen um den anfänglichen Clip in zwei kleinere Clips zu teilen. (unten) Man kann ebenso einen Übergang einfügen welcher zwei verschiedene Clips oder verschiedene Teile eines Clips verbindet.

- ein nahtloser Übergang nicht nur im Unterschied der Körperhaltung liegt, sondern auch Gelenk-Geschwindigkeiten/Beschleunigung und andere mögliche Bedingungen höherer Ordnung beachtet werden müssen.

Es muss hier bemerkt werden, dass das Skelett eigentlich nur Mittel zum Zweck ist. In einer typischen Animation wird nämlich z.B. ein Polygon-Netz über das Skelett gelegt. Dieses Netz ist alles was man sieht und insofern ist es ein natürlicher Ansatz zu sagen, man nimmt dieses Netz als Vergleich dazu, wie ähnlich sich zwei Frames sind.
Ein besseres Ergebnis wird erzielt, wenn man die Distanz zwischen den Punktwolken zweier Frames misst. Idealerweise wird diese Punktewolke bestimmt durch das Netz, welches den Charakter definiert.

Wichtig für die Berechnung der Distanz zwischen zwei Frames ist die Wahl des Koordinatensystems. Je nachdem wie das Skelett gedreht oder verschoben ist kommt eine andere Distanz zwischen den Frames raus. Gesucht ist also die minimale Distanz zweier Frames, die durch eine geeignete Transformation des Koordinatensystems des einen auf das andere ermöglicht wird. Die Wahl des Koordinatensystems ist nichts anderes als eine Verschiebung der Position (x, z) und Drehung θ des Modells um seine vertikale Achse. Für die Berechnung der Drehung θ und den Koordinaten x und z, sowie der minimalen Distanz, gibt es jeweils eine geschlossene Formel (siehe [Kovar et al. '02]).

Man errechnet die minimale Distanz für jedes Paar von Frames aus der Datenbank. Durch setzen eines Schwellwertes für diese minimale Distanz können alle Übergänge rausgefiltert werden, die nicht den Anforderungen des Benutzer genügen. Verschiedene Arten der Bewegung benötigen jedoch unterschiedliche Genauigkeitsanforderungen. Zum Beispiel braucht eine normale Gehbewegung eines Menschen einen sehr niedrigen Schwellwert für die Übergänge, da man jeden Tag Menschen gehen sieht und deswegen einen sehr scharfen Blick dafür hat, wie eine natürliche Gehbewegung aussehen sollte. Im Gegensatz dazu haben die wenigsten Menschen ein Auge dafür, wie man sich im Ballet bewegen muss. Hier kann also ein viel größerer Schwellwert gewählt werden.

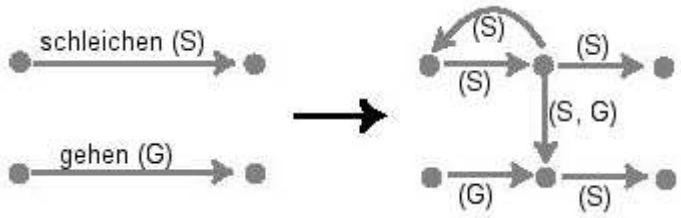


Abbildung 5.12: Dieser Graph entspricht demjenigen in Abbildung 5.11 unten, nur das hier zusätzlich die Bewegungstypen mit eingeflossen sind. Die Originaldaten enthalten einen Clip, in welchem eine Person normal geht, und einen Clip, in welchem eine Person schleicht. Übergängen von einem zum anderen Clip werden mit der Vereinigung der Bewegungstypen gekennzeichnet.

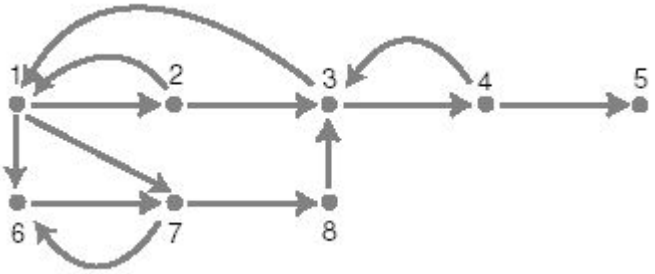


Abbildung 5.13: Ein einfacher Bewegungsgraph. Knoten 4 ist eine 'sink' und Knoten 5 ist ein 'dead end'.

Übergänge erzeugen

Ist ein Übergang zwischen zwei Frames A und B gefunden der, den Schwellwertanforderungen genügt, so erzeugt man Frames, also einen neuen Clip zwischen A und B, die einen nahtlosen Übergang gewährleisten. Dazu wird das Wurzelgelenk linear und die Gelenkrotationen kugelförmig interpoliert. Der Übergang zwischen Frame A und B ist eine neue Kante im Bewegungsgraphen und wird ebenfalls mit einem Bewegungstyp gekennzeichnet. Es gilt: Bei einem Übergang von einem Frame mit einem Satz von Bewegungstypen $L1$ und einem anderen Frame mit einem Satz von Bewegungstypen $L2$, das der Übergang selbst die Vereinigung dieser Bewegungstypen ist, also $L1 \cup L2$ (siehe Abbildung 5.12).

Graph kürzen

In seinem jetzigen Zustand kann der Graph keine Garantie für einen endlosen Bewegungsablauf übernehmen, da es Knoten geben kann (genannt 'dead end') die nicht Teil eines Zyklus sind. Andere Knoten (genannt 'sinks') sind zwar Teil eines oder mehrer Zyklen, können aber nur einen kleinen Bruchteil der Knoten im Graphen erreichen (siehe Abbildung 5.13). Schließlich gibts es noch Knoten die eingehende Kanten haben so das es keine ausgehende Kante gibt die mit demselben Bewegungstyp versehen ist. Dies ist gefährlich da eine logische Fortführung der Bewegung unmöglich ist. Zum Beispiel kann es dazu führen das ein Charakter, welcher gerade eine „Balett"-Bewegung ausführt, keine andere Möglichkeit hat als eine „Box"-Bewegung auszuführen. Dementsprechend sollten nun alle Knoten entfernt werden die obigen Beschrei-

bungen entsprechen. Sollte nach dem löschen solcher Knoten die Anzahl der Frames unter einem gewissen Wert liegen, muss eine Warnung ausgegeben werden und nötigenfalls eine Neuberechnung der Übergänge (siehe Kapitel 5.4.2) mit einem größeren Schwellwert erfolgen.

5.4.3 Bewegung extrahieren

An diesem Punkt ist man mit der Konstruktion des Bewegungsgraphen fertig.
Gesucht ist jetzt ein Pfad durch den Bewegungsgraphen der den Benutzereingaben entspricht. In folgenden Fällen entspricht die Benutzereingabe einem skizzierten Weg dem der Charakter möglichst genau folgen soll. Mit Weg ist hier nicht ein Weg auf dem Graphen gemeint sondern ein Weg über eine virtuelle Ebene bzw. über ein virtuelles Gelände!
Dieses Problem kann formuliert werden wie folgt:

- Geben ist ein Weg W den der Benutzer spezifiziert.
- Gesucht ist eine Bewegung des Charakters entlang diesem Weg W.

Die Grundidee hierzu ist das Abschätzen eines Weges W' auf dem sich der Charakter bewegen könnte. Dabei muss ständig gemessen werden wie groß die Abweichung vom eigentlich Weg W ist. Eine einfache Art um W' zu bestimmen ist die Projektion des Wurzelgelenks zu jedem Zeitpunkt auf den Boden. Da es bei komplizierten Graphen keine einfache Möglichkeit gibt für ein Paar von Knoten einen optimalen Pfad zu finden, muss man sich auf lokale Suchmethoden beschränken die versuchen, einen akzeptablen Pfad in einer vernünftigen Zeit zu finden. Bei der Abschätzung eines geeigneten Pfades durch den Graphen müssen noch die Bedingungen (Bewegungstypen) der Kanten im Graph berücksichtig werden. Das bedeutet, das bei einer 'schleichenden' Bewegung nur in demjenigen Teilgraphen gesucht werden darf, dessen Kanten mit der Beschriftung 'schleichen' markiert sind.
Interessanter ist das Setzen verschiedener Bewegungstypen auf verschiedenen Teilen des Weges. Zum Beispiel möchte der Benutzer, das der Charakter auf der ersten Hälfte des Wegen normal geht und auf der zweiten Hälfte schleicht. Es soll nun der Fall zweier verschiedener Bewegungstypen auf einem Weg betrachtet werden; die Verallgemeinerung auf eine höhere Anzahl ist dann trivial.
In diesem Fall wird der Weg in zwei kleinere angrenzende Wege W1 und W2 geteilt, wobei ein Übergang des Bewegungstyps T1 zu T2 stattfindet. Ist der Charakter schon auf dem Weg W2 dann ist der Algorithmus identisch für den Fall eines einzelnen Bewegungstyps. Ist der Charakter noch auf dem Weg W1 dann wird die verbleibende Distanz des Weges W1 überprüft. Ist die Distanz oberhalb eines gewissen Schwellwertes, so werden weiterhin nur Kanten mit dem Bewegungstyp T1 betrachtet. Ist die Distanz unterhalb dem Schwellwert wird die Suche sowohl nach Typ T1 als auch nach Typ T2 erlaubt und man leitet über zu dem Bewegungstyp T2 wenn eine entsprechende Kante mit der Beschriftung T2 gefunden wurde.

5.4.4 Resultat und Beispiele

Wie schon im oberen Teil der Abbildung 5.10 gezeigt hat die Originalbewegung einen gleichmäßigen Anteil an Bewegungs-Variationen, die Geradeaus-Gehen, scharfe Kurven und leichte Kurven beinhalten. Dennoch ist dieses Verfahren auch bei einer weniger großen Datenbank sinnvoll. Wie Abbildung 5.14 zeigt kann eine vorhandene Datenbank vergrößert werden durch hinzufügen der gespiegelten Bewegung. Aus dem Originalclip und dem gespiegelten Clip kann nun ein größerer Bewegungsgraph erstellt werden.

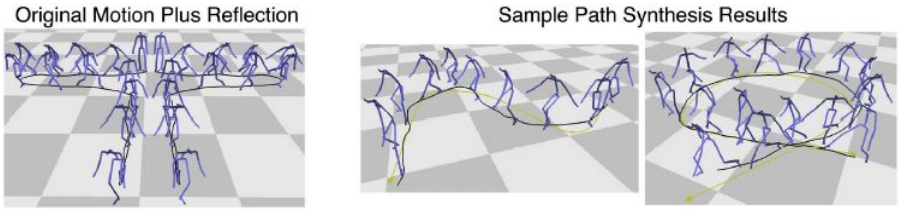


Abbildung 5.14: Das Bild ganz links zeigt die Originalbewegung und ihre Spiegelung. Die beiden rechten Bilder zeigen den daraus generierten Weg. Die gelbe Linie ist der vom Benutzer skizzierte Weg, die schwarze Linie ist die Approximation des aktuellen Weges unseres Charakters.

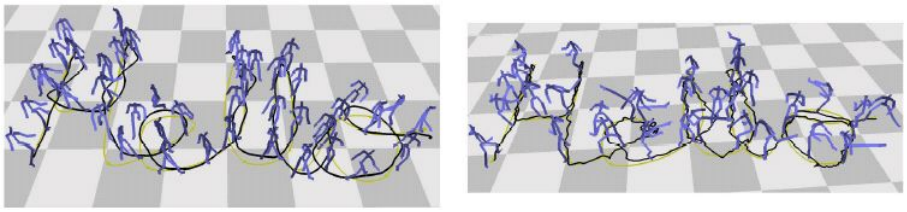


Abbildung 5.15: Diese Bilder zeigen nun einen komplizierten Weg der aus den Originaldaten (hier nicht gezeigt) generiert werden kann. Das linke Bild zeigt ein normales gehen entlang dem skizzierten Weg welches dem Wort „Hello“ entspricht. Das rechte Bild zeigt eine Kampfkunst-Bewegung, die demselben Weg folgt.

Abbildung 5.15 zeigt die Anpassung an einen viel komplizierteren Weg. Die Berechnung des Pfades für das linke Bild mit der normalen Gehbewegung dauerte 58.1s, wobei die Animation selbst 54.9s lang ist. Die Kampfkunst-Animation des rechten Bildes ist 87.7s lang, wobei die Berechnung in nur 15.0s erfolgte. Im Allgemeinen kann man sagen, daß die generierte Animation sehr viel länger oder näherungsweise gleich lange ist wie die Berechnung der Animation. Beide Bewegungsgraphen hatten annähernd 3000 Frames (100s).

Abbildung 5.16 schließlich zeigt einen Weg der mehrere Bedingungen bezüglich der Bewegungstypen enthält. Im ersten Abschnitt der beiden Wege muß der Charakter normal gehen, im Zweiten muß er schleichen und im Dritten muß er Kampfkunst-Bewegungen ausführen. Der Charakter folgt nicht nur dem Pfad, sondern schaltet auch zur richtigen Zeit über zu einem anderen Bewegungstyp. Dieses Beispiel benutzte eine Datenbank mit annähernd 6000 Frames (200s).

Die Berechnungen erfolgten alle auf einem 1.3GHz Athlon. Bei einen Bewegungsgraphen mit ungefähr 6000 Frames (siehe Abbildung 5.16) dauerte der Vergleich aller Frames ca. 25 Minuten. Danach benötigte der Benutzer noch ca. 5 Minuten um die Übergangs-Schwellwerte zu setzen. Und es dauerte weniger als eine Minute um die Übergänge zu erzeugen und den Graphen zu kürzen.

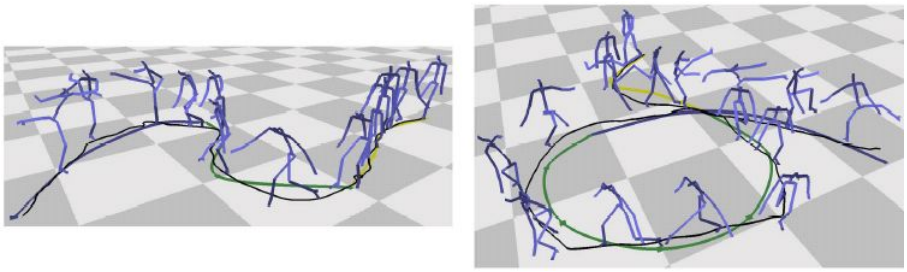


Abbildung 5.16: In beiden Bilder folgen die Charakter einem Weg, auf dem sie normal gehen, schleichen und Kampfkunst-Bewegungen ausführen. Die erwünschten Übergänge der Bewegungstypen werden angezeigt durch eine Änderung der Farbe des Weges.

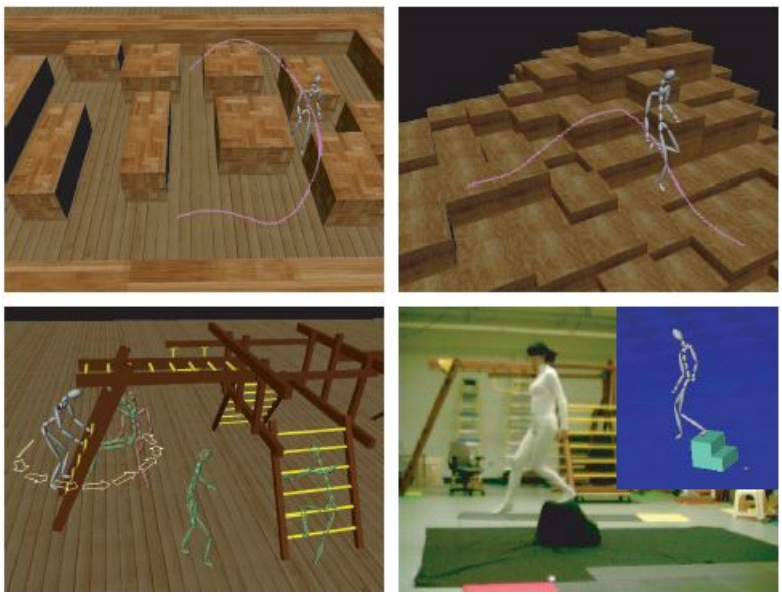


Abbildung 5.17: Echtzeit Kontrolle des Avatars. (oben) Der Benutzer kontrolliert den Avatar durch zeichnen eines Weges in das Labyrinth bzw. auf das unebene Gelände. (unten links) Der Benutzer wählt aus einer Menge von Möglichkeiten in der Spielplatz-Umgebung. (unten rechts) Der Benutzer kontrolliert den Avatar durch ausführen einer Bewegung vor einer Kamera. Nur in diesem Fall hinkt die Bewegung des Avatars um ein paar Sekunden nach.

5.5 Interaktive Kontrolle eines Avatars

5.5.1 Einleitung

Ähnlich der vorherigen Arbeit mit Bewegungsgraphen möchte ich hier ein Konzept aus [Lee et al. '02] vorstellen welches allerdings noch mehr auf die Kontrolle eines Avatars eingeht. Die Animation und Kontrolle eines Avatars ist keine einfache Aufgabe da ein großes Repertoire vorhanden sein muss wie sich ein Avatar verhalten soll. Deswegen muss der Benutzer eine einfache Möglichkeit haben sein Verhalten zu bestimmen. Dazu werden drei verschiedene Benutzer-Schnittstellen eingeführt die eine intuitive Kontrolle ermöglichen sollen: das skizzenbasierte, das auswahlbasierte und das videobasierte Interface (siehe auch Abbildung 5.17).

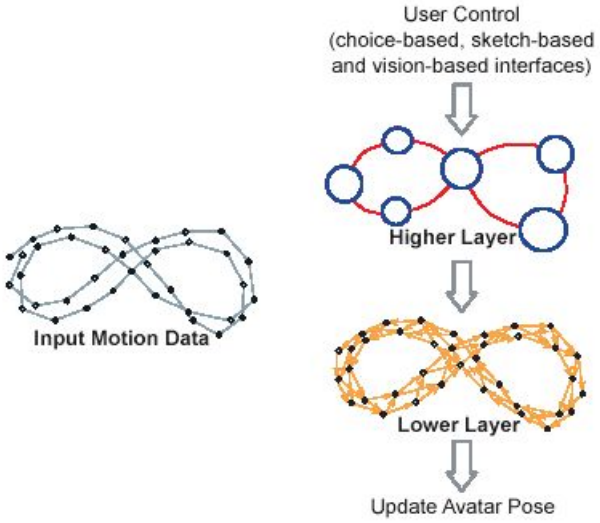


Abbildung 5.18: Zweischichtige Struktur zur Repräsentation von menschlichen Bewegungsdaten. Die 'niedere Schicht' erhält die Details der ursprünglichen Bewegungsdaten, während die 'Höhere Schicht' eine Verallgemeinerung dieser Daten darstellt. Die 'Höhere Schicht' ermöglicht eine effiziente Suche und diebessere Visualisierung möglicher Benutzeraktionen.

Im auswahlbasierten Interface wählt der Benutzer eine Möglichkeit (Richtung, Ort, Verhalten) wie sich der Avatar bewegen soll. Im skizzenbasierten Interface gibt der Benutzer einen Weg vor, dem der Avatar folgen soll. Dies entspricht also etwa dem vorherigen Kapitel 'Bewegungsgraph'. Im videobasierten Interface führt ein Benutzer vor einer Kamera eine Aktion aus. Es wird dann die beste passende Bewegung aus den Daten gesucht die der Avatar ausführen kann. Damit das durch den Benutzer bestimmte Verhalten auf die Bewegung des Avatars projiziert werden kann müssen die vorhandenen Bewegungsdaten aus der Datenbank entsprechend vorbereitet werden. Die Bewegungsdaten werden hierfür in einer zweischichtigen Struktur abgelegt (siehe auch Abbildung 5.18).
Die 'höhere Schicht' ist ein statistisches Modell das eine Unterstützung des Benutzer Interfaces bereitstellt. Dies geschieht durch Clustering (Gruppierung) der Daten indem ähnliche Haltungen des Charakters bestimmt werden. Die 'niedere Schicht' ist wieder ein Bewegungsgraph, der auf Anweisungen aus der höheren Schicht basiert. Dieser Bewegungsgraph wird im Gegensatz zum vorherigen Kapitel allerdings durch das Markov Verfahren generiert.

5.5.2 'Niedere Schicht': Markov Verfahren

Das Markov Verfahren wird dargestellt durch eine Matrix von Wahrscheinlichkeiten wobei das Element P_{ij} der Matrix die Wahrscheinlichkeit des Übergangs von Frame i zu Frame j beschreibt. Die Wahrscheinlichkeit P_{ij} $(0 \leq P_{ij} \leq 1)$ wird gemessen an der Ähnlichkeit der Frames i und j. Je ähnlicher sich die Frames sind desto wahrscheinlicher ist ein Übergang, also desto eher geht P_{ij} gegen eins.
Da diese Matrix sehr groß werden kann und somit einen Speicheraufwand von $O(n^2)$ für n Frames besitzt, muss eine andere Darstellungsmöglichkeit gefunden werden. Die Matrix wird aus diesem Grund vereinfacht zum Beispiel durch kürzen aller unwahrscheinlichen Übergänge (also diejenigen wo P_{ij} gegen null geht). Dabei entsteht durch weiteres vereinfachen ein sehr

ähnlicher Bewegungsgraph wie im Kapitel 5.4 'Bewegungsgraphen'. Da Bewegungstypen hier nicht berücksichtigt werden sind die Kanten nur mit den Wahrscheinlichkeiten P_{ij} beschriftet. Der Hauptsächliche Unterschied bei der Erzeugung des Bewegungsgraphen hier, im Gegensatz zum vorherigen Kapitel, ist die Funktion, mit welcher die Distanz der Frames berechnet wird.

Wichtig für das Markov Verfahren ist die Wahl des Koordinatensystems. Man unterscheidet zwischen einem relativen und einem festen Koordinatensystem.
In einem festen Koordinatensystem treten Übergänge nur zwischen solchen Bewegungssequenzen auf, die nahe im dreidimensionalen Raum beieinander liegen.
In einem relativen Koordinatensystem dagegen werden auch Übergänge erlaubt, welche nicht direkt im dreidimensionalen Raum beieinander liegen. Das relative Koordinatensystem ignoriert effektiv eine Translation in der horizontalen Ebene und eine Rotation um die vertikale Achse bei der Überprüfung, ob zwei Bewegungen ähnlich sind oder nicht.
Im vorherigen Kapitel 'Bewegungsgraphen' wurde das relative Koordinatensystem zugrunde gelegt.
Die Entscheidung welches Koordinatensystem nun verwendet wird hängt von der Strukturierung der Umgebung ab. Angenommen man hat ein oder mehrer feste Objekt mit denen der Avatar interagieren soll, dann ist es sinnvoller ein festes Koordinatensystem zu verwenden. Zum Beispiel soll sich der Charakter auf einen Stuhl setzen, dies kann er natürlich nicht an einer beliebigen Stelle im Raum tun, sondern nur an derjenigen Stelle an der sich auch der Stuhl befindet. Deswegen ist hier ein festes Koordinatensystem zu bevorzugen.

5.5.3 'Höhere Schicht': Statistisches Modell

Auch wenn die 'niedere Schicht' die Bewegungsdetails hält und den Avatar mit einer breiten Auswahl an Bewegungsmöglichkeiten versorgt, so ist die resultierende Struktur doch sehr schwer zu durchsuchen, vor allem für ein einfaches Benutzer Interface. Die 'Höhere Schicht' ist eine Verallgemeinerung der Bewegungsdaten welche die Verteilung der Frames und Übergänge erfasst. Grundlage für diese Verallgemeinerung ist die Cluster Analyse. Es werden Cluster (Gruppen) aus den ursprünglichen Bewegungsdaten gebildet. Diese Cluster erfassen Ähnlichkeiten zwischen den Frames, nicht aber die Verbindungen/Übergänge zwischen ihnen. Allerdings werden Verbindungen zwischen Clustern erfaßt. Zu jedem Frame wird so eine Datenstruktur gebildet die Cluster Baum genannt wird. Die gesamte 'Höhere Schicht' wird dann Cluster Wald genannt (siehe Abbildung 5.19).

Cluster Baum

Jeder Frame in der Datenbank hat seinen eigenen Cluster Baum. Dieser spiegelt das Verhalten, welches direkt zu diesem Frame verfügbar ist, wieder. Diese Anordnung erlaubt je nach Interface Technik festzustellen, welches Verhalten relevant ist. Ist der Avatar zum Beispiel weit vom Stuhl entfernt dann ist ein Verhalten welches mit dem Stuhl interagiert nicht relevant, obgleich es interessant werden könnte wenn der Avatar sich dem Stuhl nähert.

Ein Cluster Baum für einen Frame r wird berechnet wie folgt:
Der Algorithmus beginnt mit einem Knoten bestehend aus dem Cluster zu dem r gehört. Ist man beim Frame i und geht zum Frame j, dann wird überprüft, ob i und j vom selben Cluster k sind, falls nein, fügt man das Cluster, dem Frame j angehört, dem Baum als Kind-Knoten zum aktuellen Knoten hinzu. In beiden Fällen werden die Kinder des Frames j weiter

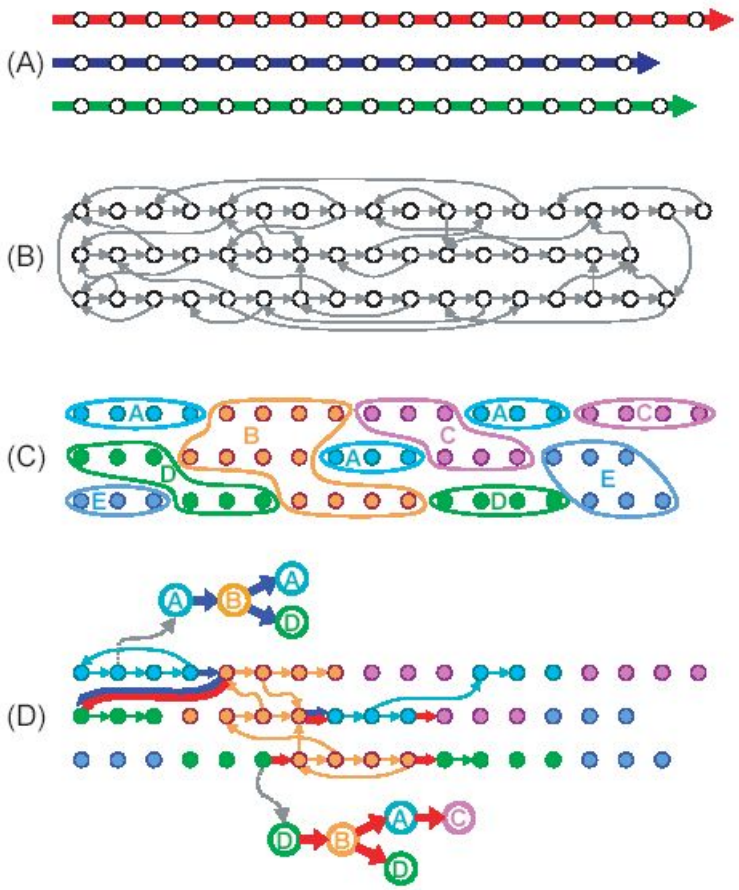


Abbildung 5.19: Bewegungsdatenvorverarbeitung. (A) Die Bewegungsdatenbank besteht ursprünglich aus einer Menge von Clips mit mehreren Frames. (B) Viele Frame-zu-Frame Übergänge werden generiert die einen gerichteten Graphen formen. (C) Ähnliche Bewegungen werden in Gruppen (Clustern) zusammengefaßt. (D) Um Verbindungen über Clustergrenzen hinweg zu erfassen, erstellen wir einen Cluster Baum für jeden Frame durch traversieren des Graphen, um Cluster zu finden die in einer maximalen Tiefe erreichbar sind. Überquert ein Übergang (dicker Pfeil) eine Clustergrenze, dann wird ein neuer Knoten zum Cluster Baum hinzugefügt. Bemerkung: Frames innerhalb deselben Clusters können verschiedene Cluster Bäume haben.

rekursiv traversiert (=durchgegangen). Die Rekursion endet, wenn der Baum eine maximale Tiefe erreicht hat. Diese Tiefe ist zeitgebunden (siehe Abbildung 5.19).

Cluster Pfad

Ein Cluster Pfad ist ein Pfad von der Wurzel des Cluster Baums zu einem seiner Blätter. Besitzt ein Cluster Baum k Blätter, so gibt es k Cluster Pfade. Jeder dieser Pfade repräsentiert eine Sammlung möglicher Aktionen des Avatars. Für jeden gegebenen Cluster Pfad kann man eine am größten wahrscheinliche Sequenz von Frames finden. Diese Sequenz liefert eine nützliche Referenz für die verschiedenen Interface-Techniken um den Avatar zu steuern.

Praktischer Nutzen des Clusterns

Clustering wurde nicht in all den Beispielen benutzt; die Beispiele mit dem Spielplatz und dem Treppchen machen Gebrauch vom Clustering, während das Labyrinth und das unebene Gelände ohne es auskommen. Der nutzen des Clusterings hängt sowohl von der Reichhaltigkeit der Datenbank als auch vom gewählten Benutzer Interface ab. Für das skizzenbasierte Interface im Labyrinth und dem unebenen Gelände reicht eine Suche im Graphen der 'niederen Schicht' aus um einen Weg zu finden der am besten zum skizzierten Weg passt. Dies funktioniert dann wie bei „Bewegungsgraphen“ aus dem vorherigen Kapitel. Für das auswahlbasierte Interface auf dem Spielplatz wurde Clustering genutzt um die Anzahl der Möglichkeiten, die dem Benutzer präsentiert werden, klein zu halten. Für das videobasierte Interface am Beispiel mit dem Treppchen wurde ebenfalls Clustering verwendet um das finden einer passenden Avatar Aktion zu einer Benutzer Aktion zu ermöglichen. Benutzeraktionen werden dann nämlich nur mit 10 Pfaden im Cluster Baum verglichen statt mit Million von Pfaden im Graphen der 'Niederen Schicht'.

5.5.4 Den Avatar kontrollieren

Auswahlbasiert

Bei diesem Interface bekommt der Benutzer kontinuierlich einen Satz Aktionen präsentiert aus denen er wählen kann (siehe Abbildung 5.20 unten). Bewegt sich der Avatar so ändert sich auch die Anzeige der Wahlmöglichkeiten passend zum Kontext. Wählt der Benutzer eine Aktion wird diese ausgeführt. Findet keine Auswahl durch den Benutzer statt wird die Bewegung des Avatars bestimmt durch die Wahrscheinlichkeit der nachfolgenden Bewegung. Um den Benutzer nicht zu verwirren sollte er nicht mehr als 4 Wahlmöglichkeiten zu jedem Zeitpunkt haben. Dazu wird der Cluster Baum des gerade eingeblendeten Frames verwendet und eine kleine Auswahl an Aktionen angezeigt. Es können zusätzliche visuelle Informationen, wie zum Beispiel der Weg den der Avatar bei der entsprechenden Aktion gehen würde, eingeblendet werden.

Skizzenbasiert

Dieses Interface erlaubt dem Benutzer das zeichnen eines Weges auf dem Bildschirm, z.B. mit der Maus. Dieser zweidimensional gezeichnete Weg wird auf die Oberfläche der virtuellen Umgebung projiziert um dreidimensionale Koordinaten zu erhalten. Wird Clustering verwendet, dann nutzt dieses Interface aus, das es oftmals einen passenden Weg durch den Cluster Baum des aktuellen Frames gibt. Dazu wird eine Bewertung für jeden Cluster Pfad m errechnet,

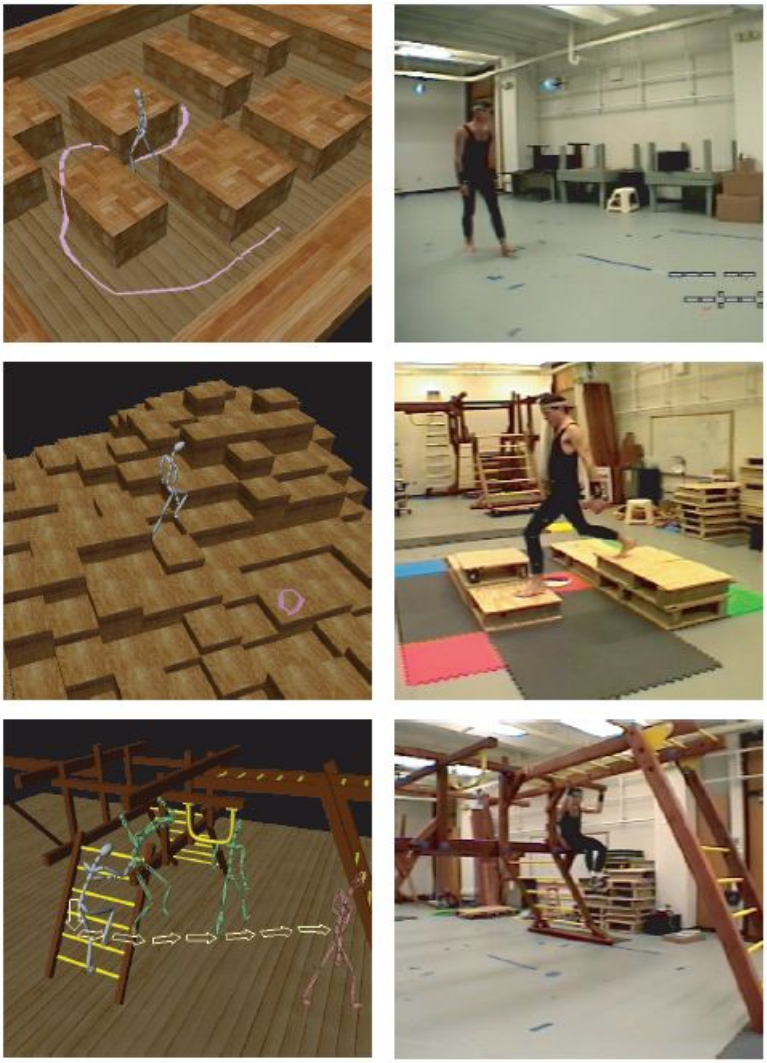


Abbildung 5.20: (Von oben nach unten) Labyrinth, Gelännde und Spielplatz Beispiele. Die linke Spalte zeigt jeweils die Kontrolle des Avatars in der virtuellen Umgebung. Die rechte Spalte zeigt die Originaldaten aufgezeichnet in einem Motion Capture Studio. Die Avatarbewegung im Labyrinth und auf dem Gelände wird kontrolliert durch das skizzenbasierte Interface, whärend auf dem Spielplatz das auswahlbasierte Interface benutzt wird.

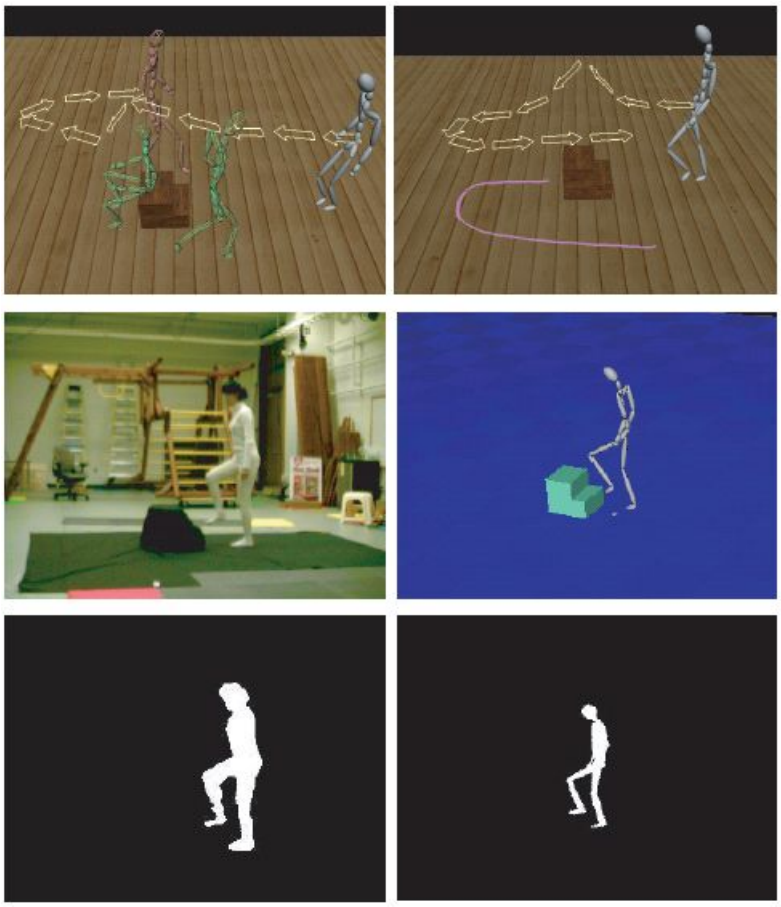


Abbildung 5.21: Treppchenbeispiel. (oben links) Auswahlbasiertes Interface. (oben rechts) Skizzenbasiertes Interface. (mitte und unten links) Der Benutzer führt eine Bewegung vor einer Kamera aus, wobei seine Silhouette aus dem Video extrahiert wird. (Mitte und unten rechts) Kontrolle des Avatars durch das videobasierte Interface und die gerenderte Silhouette passend zur Silhouette des Benutzers.

basierend auf der nachfolgenden Bewegung mit der größten Wahrscheinlichkeit. Die andere Möglichkeit ist der Verzicht auf Clusterings. Hier wird inkrementell ein Pfad durch den Graphen gesucht der am besten den Weg approximiert (siehe Abbildung 5.20 oben/mitte und Abbildung 5.21 oben links).

Videobasiert

In diesem Interface führt ein Benutzer die gewünschte Aktion vor einer Videokamera aus und der Avatar versucht diese Aktion nachzuahmen durch auswählen einer Bewegungssequenz aus der zweischichtigen Datenbank. Aus den Videodaten wird eine Silhouette erzeugt und eine übereinstimmende Silhouette in der Datenbank gesucht. Dies geschieht wieder durch berechnen eines minimalen Cluster Pfades des aktuellen Frames. Wird also keine passende Bewegung in der Datenbank gefunden wird eben die am besten passende Bewegung genommen (siehe Abbildung 5.21 mitte/unten).
Die Auswertung der Videodaten kann zwar in Echtzeit geschehen, aber das Finden einer passenden Aktion für den Avatar muß aus einem Zeitraum von Videodaten erfolgen, weswegen

eine zeitliche Differenz zwischen der Benutzer und der Avataraktion liegt. In diesen Beispielen ist die Bewegung des Avatars um ca. 3s verzögert.

5.6 Ausblick

Zu den hier vorgestellten Verfahren gibt es natürlich noch viele andere Ansätze, ähnliche aber auch grundsätzlich verschiedene. Auch hat man zum Beispiel die Möglichkeit, simple Animationen durch hinzufügen physikalischer Gesetze realistischer zu gestalten.
Nach der Erzeugung realistische Bewegungen ist eine Animation im Normalfall allerdings längst nicht fertig. Das Skelett muß angepaßt werden in Größe oder auch Länge einzelner Gliedmaßen an einen zu visualisierenden Charakter(-körper). Es muß eine visuelle Hülle um das Skelett gelegt werden, zum Beispiel in Form eines Polygon-Netzes. Ein weiteres Forschungsgebiet beschäftigt sich dann mit der Deformation der (Körper-)Oberfläche, also dem Polygon-Netz.

Literaturverzeichnis

[Arikan & Forsyth '02] Okan Arikan und D. A. Forsyth. Interactive motion generation from examples. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, S. 483–490. ACM Press, 2002.

[Faloutsos et al. '01] Petros Faloutsos, Michiel van de Panne und Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, S. 251–260. ACM Press, 2001.

[Gavrila '99] D. M. Gavrila. The Visual Analysis of Human Movement: A Survey. *Computer Vision and Image Understanding: CVIU*, 73, Nr. 1, S. 82–98, 1999.

[Kovar et al. '02] Lucas Kovar, Michael Gleicher und Frédéric Pighin. Motion graphs. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, S. 473–482. ACM Press, 2002.

[Laszlo '00] J. F. Laszlo. Interactive Control of Physically-Based Animation. In ACM-Press (Hrsg.), *SIGGRAPH2000 Proceedings*. 2000.

[Lee et al. '02] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins und Nancy S. Pollard. Interactive control of avatars animated with human motion data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, S. 491–500. ACM Press, 2002.

[Liu & Popovic '02] C. Karen Liu und Zoran Popovic. Synthesis of complex dynamic character motion from simple animations. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, S. 408–416. ACM Press, 2002.

[Theobalt et al. '01] Christian Theobalt, Marcus Magnor, Hans-Peter Seidel und Pascal Schueler. Multi-Layer Skeleton Fitting for Online Human Motion Capture. In *Max-Planck-Institut fuer Informatik in Saarbrücken*, 2001.

[Witkin & Popovic '95] Andrew Witkin und Zoran Popovic. Motion warping. In *Computer Graphics (Proc. SIGGRAPH '95)*, 1995.

6 (Selbst-)Kollision & Reaktion

Marc Diensberg

6.1 Einführung

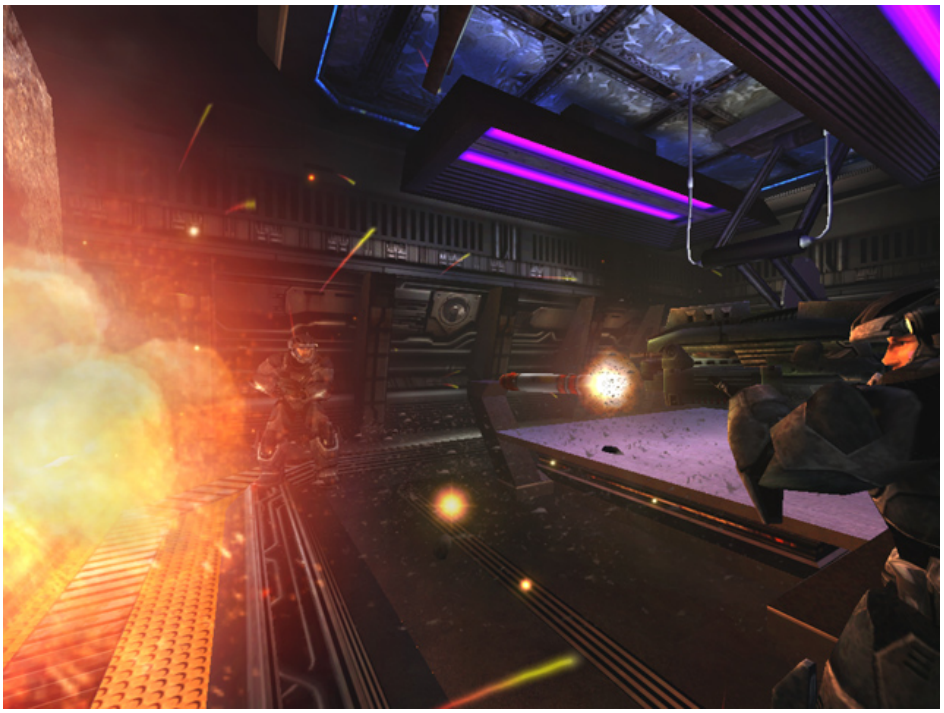


Abbildung 6.1: Kollisionen in Computerspielen

Heutzutage kommen Kollisionserkennungssysteme in vielen Gebieten zum Einsatz. Neben den offensichtlichen Bereichen wie der virtuellen Simulation von realen Umgebungen (z.B. in Flugsimulatoren, Computerspielen (vgl. Abb. 6.1), ...) werden Kollisionserkennungssysteme auch in der Robotik zur Steuerung von (humanoiden) Robotern benötigt, sei es in der Industrie zur Montage von Autos oder in der Weltraumforschung zum Steuern eines Erkundungsroboters wie beispielsweise Pathfinder. Dabei lässt sich die Kollisionserkennung in die Fremd- und die Selbstkollisionserkennung unterteilen.

6.1.1 3D Modelle

Desweiteren kann man Kollisionserkennungssysteme auch anhand ihrer zugrundeliegenden geometrischen Modellierung aufteilen. Dadurch ergeben sich die in Abbildung 6.2 aufgeführten Klassen.

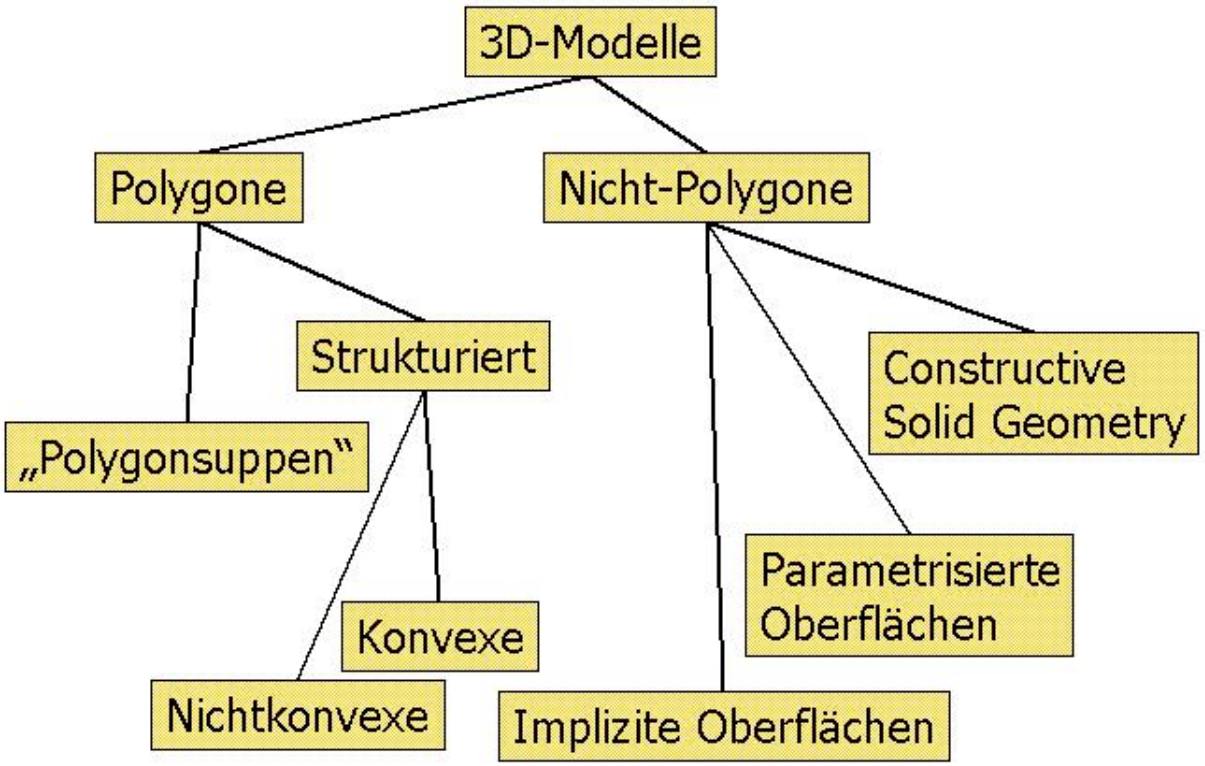


Abbildung 6.2: 3D Modelle

Dabei ist zu beachten, dass bestimmte Algorithmen nur auf bestimmte Typen von 3D-Modellen ausgelegt sind und nicht oder nur mit geringerer Performance mit anderen Typen funktionieren.

6.2 Allgemeine Techniken

6.2.1 Allgemeines

Die naive Methode zur Kollisionserkennung zwischen n Objekten testet alle diese Objekte auf Kollision mit allen anderen Objekten und hat einen Aufwand von $O(n^2)$. Zudem kann ein Objekt (z.B. ein Avatar in einer virtuellen Umgebung) wiederum aus vielen Teilen (in diesem Fall Gliedmaßen) bestehen, die untereinander alle wieder auf Selbstkollisionen untereinander und Fremdkollisionen zu anderen Objekten getestet werden müssen. Die einzelnen Teile selbst können auch wieder aus einigen tausend Polygonen bestehen, so dass für zwei Objekte bereits einige tausend Kollisionstests nötig werden. Daher ist dieser naive Ansatz nicht echtzeitfähig.

6.2.2 Bessere Ansätze

Um den quadratischen Aufwand der naiven Methode zu reduzieren, kann man im ersten Schritt eine virtuelle Umgebung in die bewegten und die stationären Objekte aufteilen. Die stationären Objekte müssen nun nicht mehr auf Kollisionen untereinander getestet werden, was die Anzahl der möglichen Kollisionen bei N bewegten Objekten und M stationären Objekten auf

$$\binom{N}{2} + N * M \tag{6.1}$$

reduziert. Dies ist bereits besser als der Aufwand von $O(n^2)$ des naiven Verfahrens (wobei $n := N + M$), aber noch lange nicht gut genug, um Echtzeitfähigkeit zu garantieren.

6.2.3 Bounding Volume Hierarchies (BVHs)

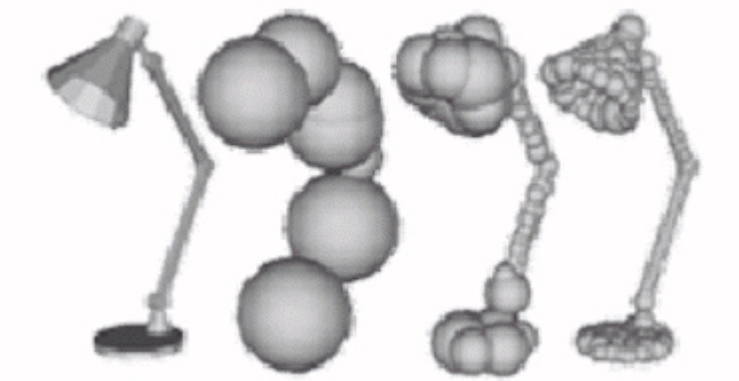


Abbildung 6.3: Hüllkörperhierarchie

Der nächste Ansatz besteht darin, komplexe Objekte durch Hüllkörperhierarchien zu approximieren. Dazu wird für jedes Objekt ein Baum von geometrisch primitiven, konvexen Hüllkörpern aufgebaut, welche das Objekt immer besser annähern. Nun muss ein Kollisionstest die Blätter des Baumes nur noch auf Kollision testen, wenn der Wurzelhüllkörper kollidiert. Dies reduziert den Aufwand für die nötigen Kollisionstests zwischen n bewegten Objekten im Durchschnitt von $O(n^2)$ auf $O(n * \log n)$. Außerdem sind Kollisionstests zwischen Hüllkörpern aufgrund ihrer primitiven geometrischen Form sehr schnell im Vergleich zu Tests zwischen den approximierten Objekten.

Diese Vorteile erkauft man sich durch den erhöhten Speicheraufwand der zur Speicherung des Baumes nötig wird und den Vor- bzw. Neuberechnungsaufwand für die Hüllkörper. Weiterhin nimmt man in Kauf, dass aufgrund der mitunter schlechten Approximation des Objekts durch die Hüllkörper Kollisionen erkannt werden, die keine sind, da nur die Hüllkörper kollidieren. Dies kann im schlechtesten Fall, wenn alle Hüllkörper, aber nicht die approximierten Objekte, kollidieren, sogar zu einem Mehraufwand im Vergleich zum naiven Verfahren führen.

Daher werden jetzt einige Gütekriterien für Hüllkörperhierarchien eingeführt:

- Vorberechnungsaufwand
  Wieviel Aufwand ist nötig, um die Hüllkörperbäume zu erzeugen?

- Aufwand des Kollisionstests
  Wie schnell lassen sich die Hüllkörper auf Kollisionen testen?
- Approximationsgenauigkeit
  Wie gut konvergieren die Hüllkörper des Baumes gegen das Objekt?
  Und daraus resultierend: Wieviele falsche Kollisionen werden erkannt?
- Speicheraufwand
  Wieviel zusätzlicher Speicherplatz wird für die Hüllkörperhierarchie benötigt?
- Neuberechnungsaufwand bei Rotation
  Müssen die Hüllkörper bei Rotationen neu berechnet / angepasst werden?
  Falls ja: Welcher Aufwand ist dazu nötig?

Anmerkung:
Diese Kriterien gelten für starre Körper. Bei nicht starren Körpern kommt neben dem Aufwand für die Neuberechnung bei Rotation noch der Aufwand zur Neuberechnung bei Deformation des Objekts hinzu.

Im folgenden werden nun einige gängige Hüllkörper-Primitive vorgestellt, aus denen sich eine Hüllkörperhierarchie aufbauen lässt und diese werden anhand der oben genannten Kriterien verglichen.

6.2.4 Axis Aligned Bounding Boxes (AABBs)

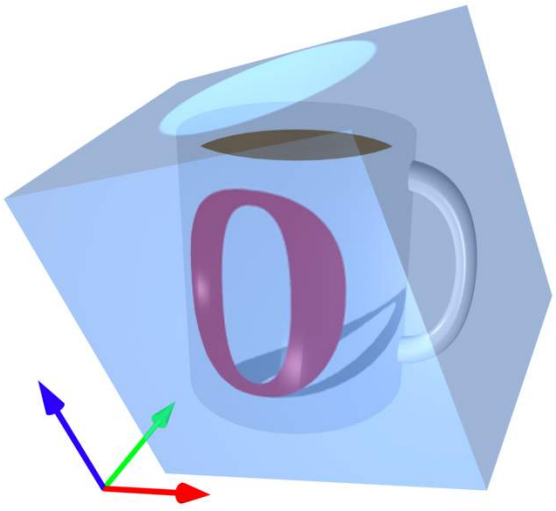


Abbildung 6.4: Achsenparallele Bounding Box

Die wohl einfachste Art ein Objekt anzunähern ist durch eine achsenparallele Bounding Box. Dabei unterscheidet man zwischen zwei Arten von achsenparallelen Bounding Boxes:

Dynamische, achsenparallele Bounding Boxes

Die dynamischen, achsenparallelen Bounding Boxes sind, wie der Name schon sagt, am Weltkoordinatensystem ausgerichtet. Dies macht ihre Berechnung sehr einfach, da man lediglich

den minimalen und maximalen x-, y- und z- Wert des zu approximierenden Objekts bestimmen muss. Die so gewonnenen zwei Punkte bestimmen die Bounding Box eindeutig und somit ergibt sich ein Vorberechnungsaufwand von $O(n)$ bei einem Objekt aus n Primitiven. Auch der zusätzliche Speicheraufwand ist sehr gering, da nur zwei weitere Punkte pro Bounding Box gespeichert werden müssen.

Der Nachteil von dynamischen, achsenparallelen Bounding Boxes ist ihre schlechte Approximationsgenauigkeit und der damit verbundene Mehraufwand durch unnötige Kollisionstests.

Desweiteren müssen dynamische, achsenparallele Bounding Boxes bei jeder Rotation neu berechnet bzw. angepasst werden.

Statische, achsenparallele Bounding Boxes

Statische, achsenparallele Bounding Boxes vermeiden die Neuberechnung bei Rotationen, welche bei dynamischen, achsenparallelen Bounding Boxes nötig ist. Dies geschieht durch das Vorberechnen einer maximalen Bounding Box, welche alle möglichen Rotationen enthält. Natürlich führt dies im Allgemeinen zu einer wesentlich schlechteren Approximation und einem höheren Vorberechnungsaufwand.

Kollisionstests zwischen achsenparallelen Bounding Boxes

Der Kollisionstest zwischen zwei achsenparallelen Bounding Boxes ist denkbar einfach: Zwei dieser Bounding Boxes kollidieren genau dann, wenn alle ihre Projektionen auf die 2 bzw. 3 Weltkoordinatenachsen des $\Re^2$ bzw. $\Re^3$ kollidieren.

Algorithmus:
Seien A und B zwei achsenparallele Bounding Boxes im $\Re^2$ und seien

$a_1 = (a_{x,min} a_{y,min})^T$, $a_2 = (a_{x,max} a_{y,max})^T$

mit $a_{x,min} < a_{x,max}$ und $a_{y,min} < a_{y,max}$

die Eckpunkte der ersten Bounding Box und

$b_1 = (b_{x,min} b_{y,min})^T$, $b_2 = (b_{x,max} b_{y,max})^T$

mit $b_{x,min} < b_{x,max}$ und $b_{y,min} < b_{y,max}$

die Eckpunkte der zweiten.

A und B kollidieren nicht, falls

$b_{x,min} > a_{x,max}$ oder $b_{x,max} < a_{x,min}$

bzw.

$b_{y,min} > a_{y,max}$ oder $b_{y,max} < a_{y,min}$

6.2.5 Oriented Bounding Boxes (OBBs)

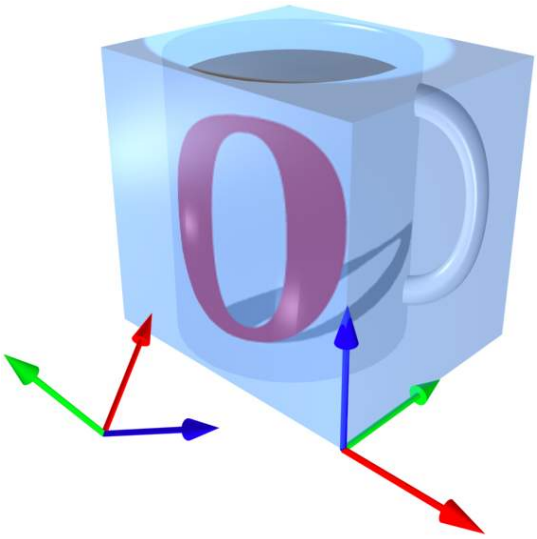


Abbildung 6.5: Orientierte Bounding Box

Orientierte Bounding Boxes sind eine Weiterentwicklung der achsenparallelen Bounding Boxes. Sie haben im Gegensatz zu den achsenparallelen Bounding Boxes den weiteren Freiheitsgrad der Ausrichtung. Dadurch lässt sich zum einen die schlechte Approximation der achsenparallelen Bounding Boxes (besonders der statischen AABBs) vermeiden und zum anderen wird die Neuberechnung bei Rotation unnötig. Diese vermeidet man, da die orientierten Bounding Boxes lokal am Objekt ausgerichtet sind und sich somit automatisch mitdrehen. Da die Ausrichtung lokal und frei wählbar ist, ist generell auch eine bessere Approximation der Objekte möglich (vgl. Abb. 6.4 und Abb. 6.5), was zu weniger falsch erkannten Kollisionen führt.

Dafür nimmt man jedoch den etwas erhöhten Speicheroverhead in Kauf, der für die Speicherung der Orientierung jeder einzelnen Bounding Box nötig wird. Außerdem benötigt die Vorberechnung der orientierten Bounding Boxes etwas mehr Zeit und da nun frei orientierte Quader im Raum auf Kollision gestestet werden müssen, ist der Kollisionstest selbst auch aufwändiger. Bei orientierten Bounding Boxes wird der Kollisionstest mit Hilfe des Separating Axis Theorems gelöst.

6.2.6 Separating Axis Theorem (SAT)

Das Separating Axis Theorem besagt, dass zwei konvexe Objekte im Raum genau dann nicht kollidieren, wenn eine Achse existiert, auf welcher die Projektionen der Objekte nicht kollidieren.

Da theoretisch unendlich viele mögliche Achsen existieren, muss die Anzahl der zu testenden Achsen erst einmal diskretisiert werden. Bei den orientierten Bounding Boxes kann man sich dabei zunutze machen, dass als trennende Achse nur eine der $2 * 3$ Achsen der beiden orientierten Bounding Boxes oder das Kreuzprodukt aller drei Achsen der ersten Bounding Box mit denen der zweiten Bounding Box in Frage kommen. Dadurch läßt sich die Anzahl der nötigen Tests auf maximal $2*3+3*3 = 15$ reduzieren. Sobald eine trennende Achse gefunden

wurde, können die übrigen Tests abgebrochen werden. Falls nach diesen 15 Tests keine Achse gefunden wurde, wird eine Kollision gemeldet.

Algorithmus:
Gegeben seien zwei orientierte Bounding Boxes A und B. Desweitern seien T_a und T_b die Mittelpunkte von A und B, e_i und f_i mit $i = (1, ..., 3)$ die Achsen von A und B, a_i und b_i die Halblängen von A und B in Richtung e_i bzw. f_i mit $i = (1, ..., 3)$ und a die zu testende Achse mit $a \in e_i, f_i, e_i \times f_j, i, j = (1, ..., 3)$

Die beiden Bounding Boxes kollidieren nicht, falls

$$|a * T_a * T_b| > \sum_{i=1}^{3} a_i * |a * e_i| + \sum_{i=1}^{3} b_i * |a * f_i| \tag{6.2}$$

für mindestens ein a gilt.

6.2.7 Discreet Oriented Polyeder (Erzeugung eines k-DOP)

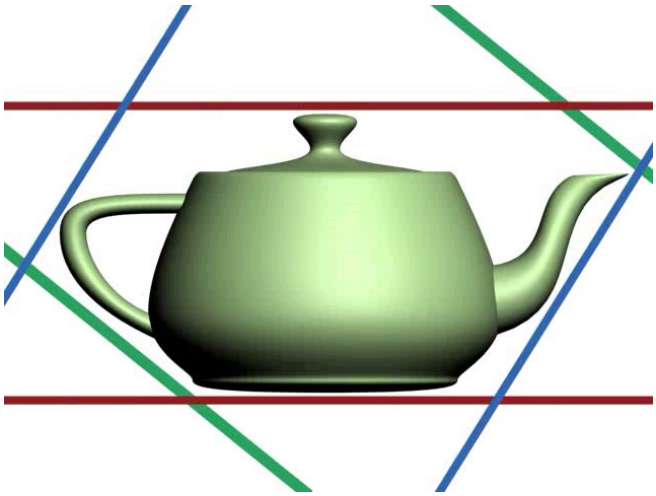


Abbildung 6.6: k-DOP

Die diskret orientierten Polyeder stellen eine Verallgemeinerung der (achsen-) orientierten Bounding Boxes dar, die als 6-DOPs angesehen werden können. Um eine bessere Approximation zu erzielen, werden Flächen mit Hilfe von k - Halbebenen angenähert, d.h. im $\Re^2$ entspricht ein k-DOP einem k-Eck dessen gegenüberliegende Seiten immer parallel sind. Von den hier vorgestellten Bounding-Volumen bieten die k-DOPs die beste Approximationsgenauigkeit.

Wählt man eine feste Ausrichtung für diese Halbebenen, so sind nur $k/2$ Kollisionstests (analog zu den achsenparallelen Bounding Boxes) nötig. Dabei benötigt man für den Kollisionstest lediglich die minimale bzw. maximale Position des k-DOPs entlang der $k/2$ Achsen, also k Skalare. Allerdings muss der orientierte Polyeder bei Rotationen neu berechnet werden.

Bei frei gewählten Halbebenen bedient man sich wie bei den orientierten Bounding Boxes wieder des Separating Axis Theorems und braucht dafür $2 * k/2 + k/2 * k/2 = 1/4 * k^2 + k$

Tests. Ebenfalls analog zu den OBBs müssen hier neben der minimalen bzw. maximalen Ausdehnung auch die Achsen selbst gespeichert werden. Der Vorteil von frei gewählten Halbebenen ist analog zu den orientierten Bounding Boxes die bessere Approximation und der Wegfall der Neuberechnung bei Rotationen.

Algorithmus:
Gegeben seien zwei k - DOPs A und B mit k festen Halbebenen.

Seien weiterhin a^i_{min} und a^i_{max} bzw. b^i_{min} und b^i_{max} die Position von A bzw. B entlang der i-ten Achse mit $i = (1, ..., k/2)$

A und B kollidieren nicht, falls ein i existiert, für welches

$b^i_{min} > a^i_{max}$ oder $b^i_{max} < a^i_{min}$

gilt.

6.2.8 Spheres

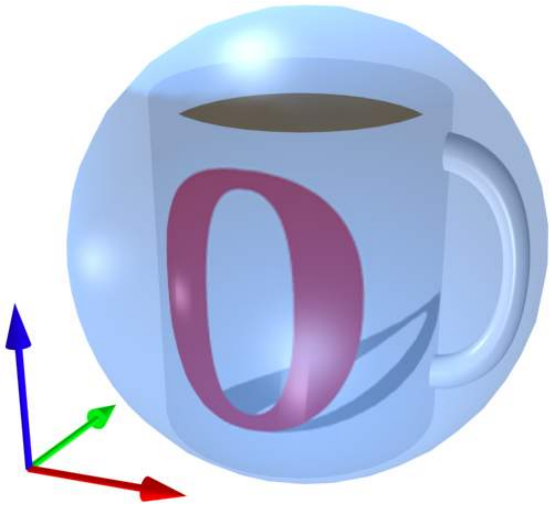


Abbildung 6.7: Kugelhüllkörper

Ein anderer Ansatz zur Annäherung von Objekten durch Bounding-Volumen sind Kugeln. Trotz der schlechten Approximation bieten Kugeln als Hüllkörper viele andere Vorteile. Zum einen haben sie den geringsten Speicherplatzbedarf, da eine Kugel durch einen Mittelpunkt und einen Radius eindeutig bestimmt ist. Zum anderen ist der Aufwand des Kollisionstests sehr gering und sie sind immun gegen Rotationen. Das größte Problem bei Kugeln als Hüllkörper stellt ihre Berechnung dar. Um eine möglichst gute Approximation zu liefern, sollte sich der Mittelpunkt der Kugel im Schwerpunkt des anzunähernden Objekts befinden. Eine Art diesen sinnvoll zu bestimmen ist, eine Bounding Box um das Objekt zu berechnen und den Mittelpunkt als Schnitt der Diagonalen dieser Bounding Box zu definieren.

Algorithmus:

Gegeben seien zwei Kugeln A und B mit den Mittelpunkten M_a und M_b und den Radien r_a und r_b.

Die beiden Kugeln kollidieren genau dann, wenn $(M_b - M_a)^2 < (r_a + r_b)^2$ ist.

6.2.9 Sphere Shells

Kugelschalen sind speziell an Bezier-Patches angepasste Hüllkörper. Sie sind definiert durch den Schnitt zweier ineinander liegender Kugeln mit dem selben Mittelpunkt. Die so entstehende Kugelschale ist desweiteren begrenzt durch einen Kegel, welcher seine Spitze im Mittelpunkt der beiden Kugeln hat. Abhängig vom Öffnungswinkel des Kegels ergibt sich eine kleinere bzw. größere Sphere Shell. Für weitere Details zu Sphere Shells wird auf die Literatur verwiesen.

6.2.10 Kollisionen zwischen Dreiecken

Bei der Kollision von Dreiecksprimitiven können eine Anzahl verschiedener Fälle auftreten:

- Edge - Edge
- Vertex - Face
- Face - Vertex
- Vertex - Vertex
- Edge - Vertex
- ...

Allerdings kann man sich bei der Implementierung auf die ersten drei Fälle beschränken, da sich die übrigen Fälle auf diese zurückführen lassen.

6.2.11 Zellraster-Verfahren

Ein anderer Ansatz, um die Anzahl der möglichen Kollisionen in einer Szene einzuschränken, sind die Zellraster-Verfahren. Wie der Name bereits sagt wird eine Szene hierbei in Zellen aufgeteilt und die Objekte diesen zugeordnet. Liegt ein Objekt auf der Grenze zu einer oder mehreren Zellen, so wird es allen diesen Zellen zugeordnet. Kollisionen können nur zwischen Objekten auftreten, die sich in der selben Zelle befinden.

Das große Problem der Zellraster-Verfahren ist die Wahl der Zellgröße bzw. bei dynamischer Zellgröße die Anordnung der Zellen. Bei fester Zellgröße darf zum einen die Zellgröße nicht zu klein gewählt werden, da die Anzahl der Zellen die Anzahl der Objekte nicht überschreiten sollte und sonst sehr viele Objekte in mehreren Zellen liegen und in allen auf Kollisionen getestet werden müssen. Zum anderen bringt eine zu große Zellgröße ebenfalls keinen Vorteil, da man in diesem Fall kaum Kollisionstests einsparen kann, dafür aber mit dem Overhead für das Zellraster-Verfahren selbst zu kämpfen hat. Bei variabler Zellgröße sind zusätzliche Daten (z.B. über die statistische Verteilung von Objekten in einer Szene) nötig, um die Zellen möglichst gut anzuordnen (Regionen hoher Objektdichte in viele und Regionen niedriger Objektdichte in wenige Zellen aufteilen).

Da sich Objekte in der Szene bewegen können, muss in jedem Schritt die Zugehörigkeit aller Objekte zu ihren Zellen neu bestimmt werden. Unter der Voraussetzung von geometrischer Kohärenz einer Szene kann man diese Zugehörigkeitstests auf die Nachbarzellen der ursprünglichen Position beschränken.

6.2.12 Geometrische Kohärenz

Die geometrische Kohärenz besagt, das sich Objekte zwischen zwei diskreten Zeitpunkten t_1 und t_2 auf einer kontinuierlichen Bahn bewegen und keine Sprünge machen. Auf die Zellraster-Verfahren angewendet bedeutet dies, dass sich ein Objekt vom Zeitpunkt t_1 zum Zeitpunkt t_2 nur in Zellen bewegen kann, die an ihre Ausgangszelle angrenzen. Voraussetzung hierfür ist ein ausreichend kleiner Abstand zwischen t_1 und t_2.

Die geometrische Kohärenz bildet ebenfalls die Basis für alle diskreten Kollisionserkennungsalgorithmen, welche im Kapitel 6.4.1 behandelt werden.

6.2.13 Andere Ansätze

Neben den bisher vorgestellten Methoden zur Reduktion der möglichen Kollisionen in einer Szene gibt es noch andere Verfahren, die z. B. auf einer Sortierung der Objekte einer Szene anhand ihrer Position in einem Baum basieren. Der Grundgedanke hierbei ist, dass Kollisionen nur in Teilbäumen auftreten können. Desweiteren muss aufgrund der geometrischen Kohärenz nicht immer der ganze Baum neu berechnet werden, sodass sich einzelne Objekte lediglich in die angrenzenden Teilbäume bewegen können. Weitere Verfahren und Ansätze sind der Literatur zu entnehmen.

6.3 Kollisionsreaktionen

6.3.1 Allgemeines

Je nach Anwendungsgebiet ist es nicht immer genug, nur aufgetretene Kollisionen zu erkennen und entsprechend darauf zu reagieren. Beispielsweise die Positionierung einer Autotür in einem Auto, oder andere Simulationen realer Montagevorgänge erfordern Kollisionsvermeidung. Dies ist auch besonders in der Robotik wichtig, da dort Kollisionen zum Schaden für den Roboter selbst und seine Umgebung führen können. Dies führt zu einer Klassifikation von Kollisionserkennungssysteme anhand ihrer Reaktion:

- Kollisionsvermeidung
  - Ein Mindestabstand muss immer zwischen Objekten gewährleistet werden
  - Bei Unterschreitung eines Schwellwertes wird die Bewegung gestoppt / geändert (Robotik)
- Bewegungssimulation
  - Bei Kollision prallen Objekte voneinander ab
  - Impuls- / Constraint-basierte Systeme

- Physikalische Simulation
  - Kollisionen führen zur Deformation der Objekte
  - Finite Elemente Methode

6.3.2 Beispiel: Impulsbasierte Kollisionsauflösung

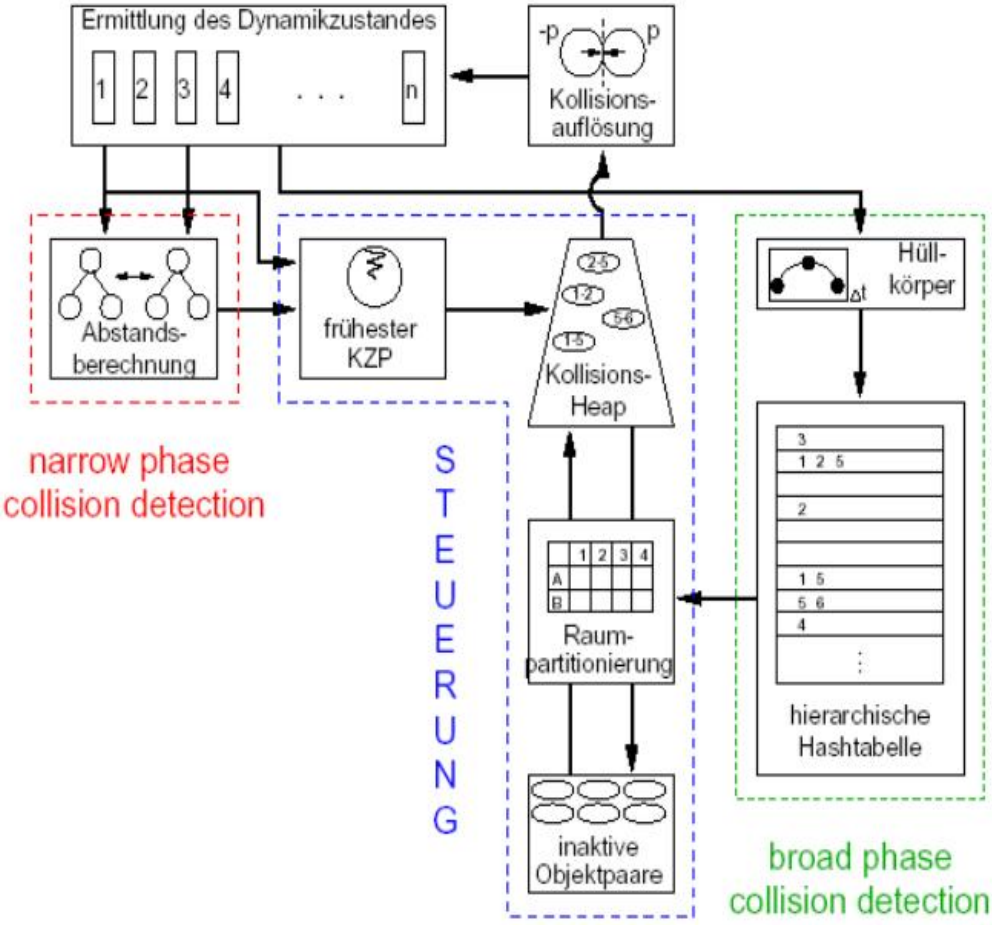


Abbildung 6.8: Impulsbasiertes Kollisionsauflösungssystem

In Abb. 6.8 ist ein Beispiel für ein impulsbasiertes Kollisionsauflösungssystem zu sehen. Wie man erkennen kann, wird die Kollisionserkennung in zwei Phasen aufgeteilt. Zum einen die broad phase, welche auf Hüllkörperhierarchien basiert und zum anderen die narrow phase, in der die wirkliche Kollisionsberechnung auf Primitivenebene (z.B. Dreiecken) gemacht wird.

Impulsbasierte Simulationen sind ein Ansatz, um realistische Bewegungen von Objekten zu simulieren. Im Gegensatz zu constraint-basierten Simulationen, die auf Zwangsbedingungen beruhen wird bei impulsbasierten Simulationen die Bewegung eines Objekts nur abhängig von seinem Impuls und der Schwerkraft modelliert. Das heißt, dass ein Objekt entweder kollidiert und damit entsprechend sein Impuls geändert wird, oder dass es einer ballistischen Flugbahn folgt.

Bei impulsbasierten Simulationen wird jeder Kontakt zwischen Objekten als Kollision im Kontaktpunkt angesehen. Dadurch lassen sich auch Bewegungen eines Objekts entlang eines anderen modellieren, indem man den ständigen Kontakt der beiden Objekte als eine Folge von Mikrokollisionen betrachtet. Nach diesem Prinzip werden die Bewegungen von Objekten im Großen durch die Behandlung der Kollisionen im Kleinen berechnet.

Für weiterführende Informationen sei auf die Literatur verwiesen (z.B. [Mirtich & Canny '95] oder [Lennerz '00]).

6.4 Diskrete & Kontinuierliche Kollisionserkennung

6.4.1 Diskrete Kollisionserkennung

Allgemeines

Die meisten der in der Praxis zu Einsatz kommenden Kollisionserkennungssysteme basieren auf diskreter Kollisionserkennung. Dies bedeutet, dass immer nur einzelne Zeitpunkte t_1, t_2, t_3, ... auf Kollisionen getestet werden. Um zu gewährleisten, dass zwischen diesen Zeitpunkten keine Kollision auftritt, dürfen die Abstände zwischen den einzelnen t_i einen festzulegenden Schwellwert nicht überschreiten. Dieser Schwellwert sollte vor allem von der Geschwindigkeit, mit welcher sich Objekte durch die Szene bewegen, abhängen und mindestens gleich der gewünschten Framerate zur Darstellung der Szene sein. Wenn zu den Zeitpunkten t_i und t_{i+1} keine Kollision vorliegt, ist unter Annahme der geometrischen Kohärenz das Auftreten einer Kollision zwischen t_i und t_{i+1} sehr unwahrscheinlich. Aber es lässt sich immer ein Fall konstruieren, in dem eine Kollision nicht erkannt wird und damit ein Tunnel-Effekt auftritt.

Der Vorteil dieses Ansatzes liegt in seiner Echtzeitfähigkeit, da diese Art von Algorithmen in der Regel deutlich schneller sind als vergleichbare kontinuierliche Verfahren. Dies macht sie ideal für den Einsatz in haptischen Systemen und Echtzeitsimulationen, in denen es vor allem auf eine hohe Geschwindigkeit ankommt.

Für Systeme, bei denen der Zeitpunkt der ersten Kollision gefragt ist (was bei den meisten der Fall ist), muss, um den diesen Zeitpunkt festzustellen, Backtracking durchgeführt werden. Dies ist nötig, da nur diskrete Tests auf Überdeckungen durchgeführt werden.

Wenn nur Zustandsdaten über die aktuellen Positionen, Richtungen und Geschwindigkeiten von Objekten verfügbar sind, kann nachträgliches Backtracking sehr aufwendig werden, besonders bei hoher Penetrationstiefe von Objekten, die aus vielen konkaven und/oder konvexen Polygonen bestehen. Ist eine hohe Genauigkeit unnötig und nur ein kollisionsfreier Zustand wichtig, kann man alternativ auch die Penetrationstiefe zweier Objekte ermitteln (z.B. anhand der Hüllkörper), und diese dann entgegen ihrer Bewegungsrichtung um diese Distanz zurücksetzten.

In der Regel wird aber das Backtracking, ähnlich wie bei den kontinuierlichen Verfahren, direkt während der Kollisionserkennung mit Hilfe von Intervallhalbierungsverfahren durchgeführt. Bei dieser Art des Backtrackings spielt die Penetrationstiefe keine Rolle.

Algorithmus:

Gegeben seien zwei Objekte A und B und ihre Positionen zu zwei diskreten Zeitpunkten t_1 und t_2. Weiterhin seien die Bewegungen von A und B zwischen diesen Zeitpunkten bekannt und es liege zum Zeitpunkt t_1 keine Kollision vor. Gesucht sei der Zeitpunkt der ersten Kollision, der mittels Intervallhalbierung bestimmt werden soll.

Wird zum Zeitpunkt t_2 eine Kollision festgestellt, so wird das Intervall zwischen t_1 und t_2 halbiert und der Zeitpunkt $t_{1.5}$ ($t_{1.5} := t_1 + 1/2 * (t_2 - t_1)$) auf Kollision getestet. Wird auch hier eine Kollision erkannt, so wird das Intervall $[t_1, t_{1.5}]$ weiter unterteilt und wieder auf Kollision getestet, ansonsten wird bei Kollisionsfreiheit das Intervall $[t_{1.5}, t_2]$ unterteilt. Dies wird bis zu einem Schwellwert fortgesetzt. Danach wird das größte t_i, bei dem keine Kollision erkannt wurde, als letzter kollisionsfreier Zeitpunkt und das nächst größere t_i als Zeitpunkt der ersten Kollision definiert wird.

Aufwandsbeispiele

Nachfolgend ein paar Beispiele für die Komplexität von Kollisionstests die auf verschiedenen Arten von Polygonen basieren:

- Konvex - konvex:
  - $O(\log^2 n)$
  - Dopkin / Kirkpatrik 1983 - 1990
- Konvex - nicht-konvex:
  - $O(n \log n)$
  - Dopkin / Herschberger / Kirkpatrik / Suri 1993
  - Doprindt / Mehlhorn / Yvinec 93
  - Schömer 94
- Nicht-konvex - nicht-konvex:
  - $O(n^{2-\varepsilon})$
  - Schömer / Thiel 96

6.4.2 Kontinuierliche Kollisionserkennung

Allgemeines

Wenn genaue Kollisionserkennung gefragt ist, die garantieren kann, dass keine Kollisionen übersehen werden, kommt die kontinuierliche Kollisionserkennung ins Spiel. Sie erkennt aufgrund ihrer Kontinuität jede Kollision und kann somit die geforderten Garantien liefern. Desweiteren ist die Berechnung des frühest möglichen Kollisionszeitpunktes ein elementarer Bestandteil der kontinuierlichen Kollisionserkennung (in der Regel mit Intervallhalbierung). Dadurch vermeidet man ein aufwendiges, nachträgliches Backtracking wie es bei manchen diskreten Verfahren nötig wird (vgl. Kapitel 6.4.1).

Leider sind die meisten kontinuierlichen Verfahren sehr komplex und daher nur eingeschränkt oder gar nicht echtzeitfähig.

Überstrichenes Volumen

Ein Ansatz zur kontinuierlichen Kollisionserkennung ist die Berechnung des Volumens, welches ein Objekt zwischen den Zeitpunkten t_i und t_{i+1} überstreicht. Zur Kollisionserkennung schneidet man dieses Volumen mit den Volumina aller anderen Objekte. Ein nicht leerer Schnitt bedeutet dabei eine mögliche Kollision. So kann man garantiert jede Kollision zwischen t_i und t_{i+1} erkennen.

Dieses Verfahren setzt voraus, dass die Flugbahnen der Objekte zum einen bekannt und zum anderen kontinuierlich sind, da ansonsten das überstrichene Volumen nicht genau berechnet werden kann.

In der Praxis findet diese Methode in dieser Form allerdings kaum Anwendung, da es sehr schwierig ist, das exakte, überstrichene Volumen eines Objekts zu bestimmen, besonders wenn es sich auf einer nicht linearen Bahn bewegt oder rotiert. Daher berechnen Algorithmen, die diesen Ansatz implementieren, oft ein konservatives, überstrichenes Volumen, indem z.B. Hüllkörper zur Volumenberechnung verwendet werden. Dies kann zwar zu einer erhöhten Anzahl von Fehlalarmen führen, dafür wird aber die Berechnung des überstrichenen Volumens stark vereinfacht und damit beschleunigt. Außerdem haben die meisten Algorithmen die Einschränkung, dass die Bewegung eines Objekts zwischen zwei Punkten sich als eine einfache Translation beschreiben lassen muss. Aber auch mit dieser Einschränkung ist dieses Verfahren generell nicht echtzeitfähig.

Weitere Entwicklung

Einen ähnlichen Ansatz wie die oben genannte Berechnung des überstrichenen Volumens machte Canny in seinem 1986 veröffentlichen Algorithmus für Polygone ([Canny '86]). Er parametrisierte die Bewegung der Objekte und berechnete Kollisionen durch das Lösen polynomieller Gleichungen niedriegen Grades. Allerdings war dieser Algorithmus aufgrund seiner hohen Komplexität nicht echtzeitfähig.

In den Jahren 2000/2001 entwickelte dann Redon einen Algorithmus, der ähnlich wie der Ansatz von Canny, die Bewegung eines Objektes parametrisierte ([Redon et al. '00]). Zusätzlich setzte er Hüllkörperhierarchien aus Kugeln ein und schaffte es damit, eine Szene aus bis zu ein paar tausend Polygonen in Echtzeit auf Kollisionen zu testen ([Redon et al. '01]). Leider erlaubt dieser Algorithmus nur ein bewegtes Objekt.

Schnelle, kontinuierliche Kollisionserkennung zwischen starren Körpern

Der Algorithmus von Redon, Kheddar und Coquillart wurde zum ersten Mal auf der Eurographics 2002 (für Details siehe [Redon et al. '02]) vorgestellt. Nach Angaben der Autoren erlaubt er in dieser Implementierung die Simulation von mehreren bewegten Objekten, welche aus tausenden Polygonen bestehen können, in Echtzeit und dies auch bei geringer geometrischer Kohärenz. Die vorgestellte Version war für Polygonsuppen implementiert, soll sich aber auch an parametrisierte und implizite Flächen anpassen lassen.

Der Algorithmus basiert auf orientierten Bounding Boxes. Die Bewegung der einzelnen Objekte zwischen zwei Zeitpunkten muss sich durch eine zeitabhängige Gleichung ausdrücken lassen.

Mit Hilfe dieser zeitabhängigen Position der Bounding Boxes lässt sich ein konservativer, kontinuierlicher Kollisionstest durchführen. Dieser besteht aus zwei Phasen:

Zuerst wird ein konservativer Test auf Basis des Separating Axis Theorems durchgeführt. Das heißt, es wird geprüft, ob für den Zeitraum zwischen t_i und t_{i+1} eine Achse gefunden werden kann, welche die beiden zu testenden Objekte für die ganze Zeitspanne trennt. Dazu muss zunächst der SAT-Test (vgl. Kapitel 6.2.6) in eine zeitabhängige Form gebracht werden. Dies ist nicht weiter schwierig, da eine zeitabhängige Position der Bounding Boxes bereits bekannt ist. Mittels Intervallarithmetik kann man eine obere und untere Grenze für die linke bzw. rechte Seite der Ungleichung (6.2) aus Kapitel 6.2.6 angeben. Sei $[l_1, l_2]$ bzw. $[r_1, r_2]$ diese Grenze für die linke bzw. rechte Seite. Ist $l_1 > r_2$ so trennt die Achse a die beiden orientierten Bounding Boxes für den gesamten Zeitraum. Diesen Test führt man nun für alle 15 möglichen Achsen durch.

Leider lassen sich mit diesem Test nur Achsen ermitteln, welche die beiden zu testenden Bounding Boxes für den gesamten Zeitraum zwischen t_i und t_{i+1} trennen, das heißt, selbst wenn keine solche Achse gefunden wurde, müssen die Objekte noch nicht kollidieren. Um Achsen zu finden, welche die Objekte auf verschiedenen Abschnitten zwischen t_i und t_{i+1} trennen, wird Intervallhalbierung durchgeführt. Dazu wird das Intervall zwischen t_i und t_{i+1} bis zu einem Schwellwert unterteilt und für die einzelnen Teilintervalle eine Trennachse gesucht. Ist der Schwellwert erreicht und es wurden nicht in allen Teilintervallen Trennachsen gefunden, so wird eine Kollision im ersten dieser Teilintervalle gemeldet.

Da dies sehr aufwendig werden kann, wird zunächst im zweiten Schritt eine Heuristik aufgestellt, die festlegt, wann eine Unterteilung des Intervalls vorgenommen wird. Generell gilt, dass eine Achse, welche zwei Bounding Boxes zu einem Zeitpunkt des Intervalls trennt, nicht für das gesamte Intervall gilt, wenn die Geschwindigkeiten der Bounding Boxes relativ zu ihrer Größe zu groß sind. In diesem Fall sollte also eine Unterteilung stattfinden, ansonsten nicht. Dazu müssen zunächst die Geschwindigkeiten der Zentren ($v(T_a)$ bzw. $v(T_b)$) der beiden Bounding Boxes zum Zeitpunkt t_i ermittelt werden. Danach wird die Geschwindigkeit der beiden Zentren relativ zueinander als $v_r = v(T_a) - v(T_b)$ bestimmt. Dies entspricht der Annahme, dass die jeweils zweite Bounding Box statisch ist. Mit $(t_{i+1} - t_i) * |v_r|$ ergibt sich nun die ungefähre Länge des Weges, den die beiden Zentren zwischen t_i und t_{i+1} relativ zueinander zurücklegen. Eine Intervallhalbierung wird durchgeführt, wenn

$$\sum_{j=1}^{3} a_j * |v_r * e_j(t_i)| + (t_{i+1} - t_i) * |v_r| > k * \sum_{j=1}^{3} b_j * |v_r * f_j(t_i)| \tag{6.3}$$

ist. Dabei ist k eine festzulegende Konstante. Schlägt der Unterteilungstest fehl, so wird eine Kollision gemeldet. Versuche haben gezeigt, dass mit $k = 0.2$ die meisten falschen Kollisionsmeldungen wegfallen. Da jede Kollision erkannt wird, aber bei fehlgeschlagenem Unterteilungstest nicht weiter nach einer Trennachse gesucht, sondern direkt eine Kollision gemeldet wird, heißt dieser Test konservativ.

Wurde eine Kollision gemeldet, so wird sie zum Schluss auf Primitivenebene heruntergebrochen und mit fauler Auswertung und Intervall-Arithmetik schnell gelöst. Dabei werden drei mögliche Fälle unterschieden:

1. Edge - Edge:
Eine Kollision zwischen Kante $a(t)b(t)$ und Kante $c(t)d(t)$ tritt auf, falls

$$a(t)c(t) * (a(t)b(t) \times c(t)d(t)) = 0 \tag{6.4}$$

mit $t \in [0, 1]$ ist. Eine Lösung t_k wird nur behalten wenn der entsprechende Kontaktpunkt auf einer der Kanten $a(t_k)b(t_k)$ oder $c(t_k)d(t_k)$ liegt.

2. Vertex - Face / Face - Vertex :
Eine Kollision zwischen einer Ecke $a(t)$ und der von $b(t)c(t)d(t)$ aufgespannten Ebene tritt auf, falls

$$a(t)b(t) * (b(t)c(t) \times b(t)d(t)) = 0 \tag{6.5}$$

mit $t \in [0, 1]$ ist. Auch hier ist eine mögliche Lösung t_k nur gültig, falls der Kontaktpunkt in dem durch $b(t_k)c(t_k)d(t_k)$ gegebenem Dreieck liegt.
Die Gleichungen (6.4) und (6.5) werden mit Hilfe von Intervallhalbierung gelöst. Dabei wird bei einer Intervallhalbierung immer zuerst das erste Intervall weiter untersucht. Wird eine gültige Lösung gefunden wird der Test abgebrochen und das zweite Intervall muss nicht mehr geprüft werden.

Ergebnisse:
Mit diesem Algorithmus ist es den Autoren gelungen, z.B. die Positionierung einer Autotür, bestehend aus ca. 16.000 Polygonen in ein Auto aus ca. 29.000 Polygonen in Echtzeit durchzuführen.

Leider waren keine Daten zur Performance des Systems bei multiplen bewegten Objekten verfügbar.

6.5 Selbstkollisionen

Selbstkollisionserkennung kommt vor allem aus der Robotik. Sie wird besonders bei der Koordination von humanoiden Robotern benötigt, um sicherzustellen, dass diese weder über ihre eigenen Beine stolpern, noch sich selbst und dann eventuell ihrer Umgebung Schaden zufügen.

Dazu müssen gewisse Mindestabstände zwischen den verschiedenen Körperteilen sichergestellt werden. Bei einem komplexen, humaniden Roboter wie dem H7 (vgl. Abb. 6.9) sind dies bereits 435 Tests.

Um diese Zahl etwas zu reduzieren, kann man sich (wie in [Kuffner et al. '02] beschrieben) zunutze machen, dass alle Gelenke nur eine begrenzte Bewegungsfreiheit haben und somit alle möglichen Bewegungen bekannt sind. Dadurch lassen sich die Tests für die Fälle eliminieren, die nicht auftreten können. Mit diesem Trick müssen wir nun nur noch 76 Tests für den gesamten Roboter durchführen.

Im nächsten Schritt wird die Komplexität des Modells reduziert. Als erstes wird die konvexe Hülle für die einzelnen Körperteile berechnet. Um eventuellen Fehlern des Kollisionstest vorzubeugen (z.B. durch Messfehlern der Sensoren usw.) und um Problemstellen des Modells auszugleichen, wird als nächstes das Modell nun noch mit einer konvexen Schutzhülle versehen. Um diese zu erzeugen wird die konvexe Hülle des Modells um einen Schutzabstand vergrößert.

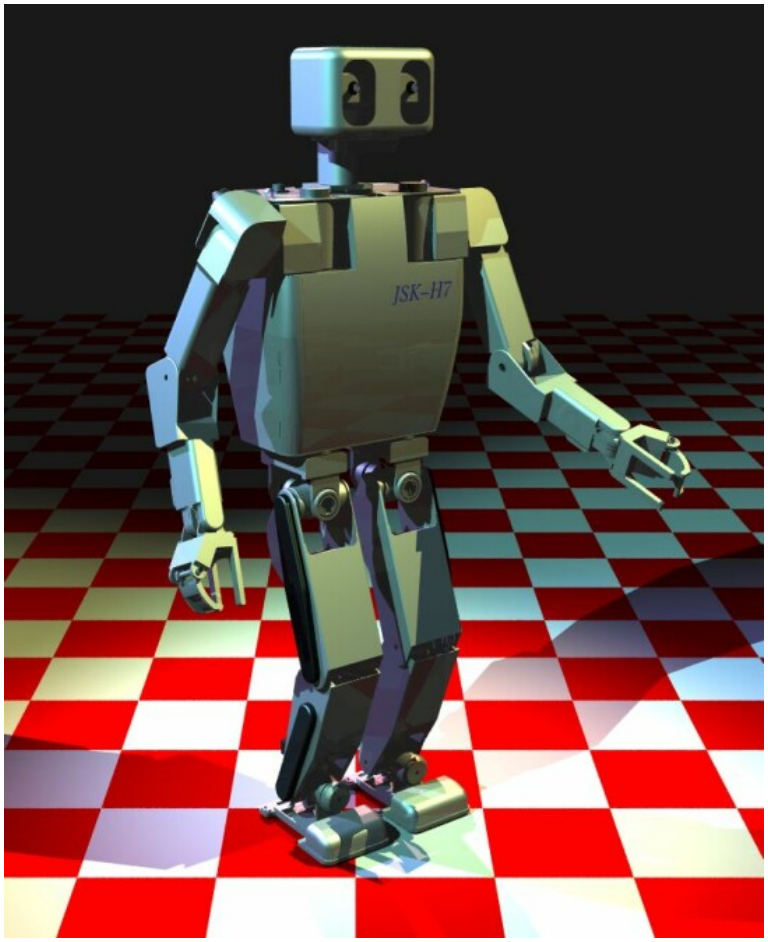


Abbildung 6.9: Modell des H7

Zum Schluss wird diese dann noch weiter durch Bounding Boxes approximiert. Beim Beispiel des H7 lässt sich damit das Modell von ursprünglich 314.588 Dreiecken auf 2702 Dreiecke und dann 432 Dreiecke für die Bounding Boxes reduzieren.

Da die Schutzhülle nur aus konvexen Polygonen besteht, ein Roboter nur kontinuierliche Bewegungen ausführen kann und somit auch die räumliche und zeitliche Kohärenz gewährleistet ist, lassen sich die nötigen Tests zur Abstandswahrung sehr schnell durchführen.

Ergebnisse:
Wie sich der folgenden Tabelle entnehmen lässt, führen die obigen Techniken insgesamt zu einer Zeitersparnis von ca. 5/6.

Zu Testen:	Gelenke:	#Tests	Zeit (msec):
Ganzer Körper:	31	435	2.442
Reduzierter Körper:	31	76	0.429
Beine:	14	79	0.441
Reduzierte Beine:	14	19	0.128

6.6 Ausblick

Zusammenfassend lässt sich sagen, dass es bereits sehr viele gute Ansätze für Kollisionserkennungsalgorithmen gibt, aber leider noch keinen perfekten Algorithmus. Besonders die kontinuierlichen Algorithmen haben sich in den letzten Jahren sehr stark weiterentwickelt und werden zunehmend schneller und damit auch echtzeitfähig.

Literaturverzeichnis

[Canny '86] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. Patt. Anal. Mach. Intell. 8,2*, S. 200–209, 1986.

[Cohen et al. '95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha und Madhav K. Ponamgi. I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments. *Proc. ACM Symposium on Interactive 3D Graphics*, S. 189–196, 1995. ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/collision.pdf (gesehen 07/2003).

[Ehmann & Lin '01] Stephen Ehmann und Ming C. Lin. Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition. *Eurographics*, 20, Nr. 3, 2001. http://www.cs.unc.edu/~geom/SWIFT++/ (gesehen 07/2003).

[Fuhrmann '] Artur Fuhrmann. Approximation konvexer Polyeder in der Hausdorff-Metrik.

[Gottschalk et al. '96] S. Gottschalk, Ming C. Lin und D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. *Siggraph*, 1996. http://www.cs.unc.edu/~geom/OBB/OBBT.html (gesehen 07/2003).

[Hudson et al. '97] T. Hudson, Ming C. Lin, J. Cohen, S. Gottschalk und D. Manocha. V-COLLIDE: Accelerated Collision Detection for VRML. *Proc. of VRML*, 1997.

[Kuffner et al. '02] Kuffner, Nishiwaki, Kagami, Kuniyoshi, Inaba und Inoue. Self-Collision Detection and Prevention for Humanoid Robots. *International Conference on Robotics and Automation*, 2002.

[Lennerz '00] Christian Lennerz. Impulsbasierte Dynamiksimulation starrer Koerper unter Verwendung von Huellkoerperhierarchien, 2000.

[Mirtich & Canny '95] Brian Mirtich und John Canny. Impulse-based Simulation of Rigid Bodies, 1995.

[Redon et al. '00] S. Redon, A. Kheddar und S. Coquillart. An Algebraic Solution to the Problem of Collision Detection for Rigid Polyhedral Objects. *International Conference on Robotics and Automation*, S. 3733–3738, 2000.

[Redon et al. '01] S. Redon, A. Kheddar und S. Coquillart. CONTACT: arbitrary in-between motions for continuous collision detection. *IEEE ROMAN*, 2001.

[Redon et al. '02] Redon, Kheddar und Coquillart. Fast Continuous Collision Detection Between Rigid Bodies. *Eurographics Graphics*, 21, Nr. 3, 2002.

[Schoemer '94] Elmar Schoemer. Interaktive Montageplanung mit Kollisionserkennung, 1994.

[Zachmann '03] Gabriel Zachmann. Triangle and Quadrangle Intersection Tests, 2003.

7 Navigation und Interaktion

Michael Niedermaier

7.1 Einleitung

Um in virtuellen Umgebungen intuitiv und effizient arbeiten zu können, benötigt man Ein- und Ausgabegeräte die aufeinander abgestimmt sind und die möglichst ergonomisch sein sollten. Damit man die Möglichkeiten der Hardware auch ausnutzen kann, bedarf es natürlich auch entsprechender Softwaretechniken. So sollte ein Datenhandschuh nicht nur als dreidimensionaler Mauszeiger dienen, sondern auch zum Beispiel das Greifen ermöglichen.

7.1.1 Inhalt

Die Interaktion und Navigation in "Virtual Environments" befasst sich mit der Schnittstelle zwischen dem Benutzer und dem Computer, beziehungsweise die Schnittstelle zwischen dem Benutzer und seiner virtuellen Repräsentation, dem Avatar. Es werden Hardware- und Software-technische Möglichkeiten in der Realisierung der Schnittstelle erläutert und anhand einiger Beispiele beschrieben.

7.1.2 Ziele

Ziel für die Interaktion und Navigation ist es sich möglichst intuitiv in der virtuellen Umgebung zurecht zufinden und diese so immersiv (möglichst echt) zu erfahren. Damit zusammenhängende Ziele sind die fünf Sinne Sehsinn, Hörsinn, Tastsinn, Geruchsinn und Geschmacksinn möglichst wie in der Realität anzusprechen. Außerdem sollten speziell Eingabegeräte so gestaltet sein, dass sie ohne Aufwand benutzbar sind, den Benutzer nicht behindern und universell einsatzfähig sind.

7.2 Eingabegeräte

Zunächst lassen sich Eingabegeräte unterscheiden in diskrete, kontinuierliche und hybride Eingabegeräte.

Diskrete sind eindimensional und können nur diskrete Signale, wie z.B. bei der Tastatur einen Tastendruck, erfassen.

Kontinuierliche hingegen können Bewegungen erfassen, wie z.B. beim Datenhandschuh, der die einzelnen Finger- und Handpositionen kontinuierlich erfasst.

Hybride sind schließlich noch Eingabegeräte, die sowohl diskret als auch kontinuierlich sind, wie z.B. die Maus. Bei dieser sind die 2D-Bewegungen die kontinuierlichen und die Tasten sind die diskreten Elemente.

7.2.1 Herkömmliche Eingabegeräte

Herkömmliche Eingabegeräte sind unter anderem die Tastatur, die Maus und eventuell noch der Joystick. Jeder der mit Computern arbeitet, kennt diese und kann damit umgehen. Allerdings sind diese nicht besonders gut für virtuelle Umgebungen geeignet, da sie maximal zwei von den möglichen sechs Freiheitsgraden bedienen können.

7.2.2 3D-Desktop Eingabegeräte

[Turner et al. '96] Da die herkömmlichen Eingabegeräte nicht gut geeignet sind für dreidimensionale Umgebungen wurden weitere Desktop Eingabegeräte entwickelt um intuitiver mit den sechs Freiheitsgraden umgehen zu können. Die "Spacemouse" und der "Spaceball" [Gabbard '97] sind wie ein dreidimensionaler Joystick, der sich nicht nur in die drei Dimensionen bewegen lässt, sondern er lässt auch Rotation um alle drei Achsen zu. Somit sind mit diesem Eingabegerät alle sechs Freiheitsgrade bedienbar. Weiter gibt es noch die "Ring Mouse" und die "Fly Mouse" bei denen jeweils die dreidimensionale Position im Raum erfasst wird. "Ring Mouse" ist, wie der Name schon sagt, wie ein Ring den der Benutzer anziehen kann und die "Fly Mouse" ist eine Maus, bei der die dreidimensionale Position erfasst wird. Allerdings ist diese Variante sehr ermüdend, da man ständig die Kraft aufbringen muss, dem Gewicht der Maus entgegenzuwirken. Alle vier hier genannten Eingabegeräte haben zusätzlich noch diverse diskrete Tasten. Außer den erwähnten gibt es natürlich noch viele andere dreidimensionale Desktop Eingabegeräte, die hier nicht erwähnt sind, da sie den Rahmen dieser Arbeit sprengen würden.

7.2.3 Datenhandschuh

Der Datenhandschuh [Gabbard '97] (Abb. 7.1) dient dazu die Hand- und Fingerbewegungen zu erfassen und auf den Avatar zu übertragen. Somit hat man in der virtuellen Welt ein oder auch zwei Hände um mit virtuellen Objekten interagieren zu können. Damit sind mit der virtuellen Hand alle Bewegungen ausführbar, die auch mit der echten, realen Hand machbar sind. Außerdem lassen sich Objekte greifen, Handzeichen machen und er ist sehr intuitiv benutzbar, da man seine Hände genau so wie in der realen Welt einsetzen kann.
Für den Datenhandschuh gibt es optionale Pakete für die Rückkopplung. Diese sind in Abschnitt 9.3.3 genauer erklärt.

7.2.4 Pinch Glove

Der "Pinch Glove" (Abb. 7.2) [Bowman & Wingrave '01] ist ebenfalls ein Handschuh. Allerdings hat dieser nur beliebig angebrachte Kontakte an den Fingern und an der Hand selber, welche bei Berührung ein diskretes Signal auslösen. Diese Variante ist wesentlich billiger als der Datenhandschuh, allerdings lässt sie auch nur diskrete Eingaben zu.

7.2.5 Tracker

[Gabbard '97] Die verschiedene Trackerarten werden hier nicht weiter vorgestellt, da sie in einem Thema zuvor schon ausführlich behandelt wurden.

Abbildung 7.1: Cyberglove

Abbildung 7.2: Pinch Glove

7.2.6 Spezielle Eingabegeräte

Es gibt viele verschiedene Eingabegeräte [Dachselt '00], die speziell für eine Aufgabe oder einen kleinen Aufgabenbereich gut sind. Hier werden ein paar Ausgewählte kurz erläutert.

Der fliegende Teppich [Will '02] besteht aus einer Plattform an der drei Gewichtszellen angebracht sind, die die Gewichtsverlagerungen des Benutzers, der auf der Plattform steht, erkennen können. Mit der Gewichtsverlagerung lässt sich ein virtueller fliegender Teppich steuern. Mit diesem kann man gut durch große Szenarien navigieren ohne sich in der realen Welt zu bewegen. Schlecht geeignet ist dieser allerdings für kleine und exakte Bewegungen. Für Szenarien mit verwinkelten Gängen wäre der fliegende Teppich also nicht so gut geeignet wie normales Gehen.

Weitere Eingabegeräte , die für spezielle Aufgabengebiete geeignet sind, sind z.B. ein Flugzeug- oder Raumschiffsimulator, ein Fahrrad, auf das man sich setzt und mit dem man durch virtuelle Welten fliegen oder fahren kann, ohne dass sich das Fahrrad in der realen Welt von der Stelle bewegt. Des weiteren gibt es z.B. die **"Data Suit"**, die ein Anzug ist, mit dem man die Körperposition erfassen und auf einen Avatar übertragen kann. Ein **„Face Tracker"** ist ein Eingabegerät, das die Gesichtszüge und Gesichtsmimik mit einer Kamera aufnimmt und ebenfalls auf einen Avatar übertragen kann. Um zu vermeiden während des Gehens mit einem Gegenstand oder einer Wand zu kollidieren, gibt es noch z.B. den "Gaitmaster" in zwei Ausführungen, die "omni-directional Treadmill" [Moghaddam & Buehler '93] und die "Torus Treadmill":

Gaitmaster Mit dem "Gaitmaster" (Abb. 7.4) [vrLab '03] lassen sich Treppenstufen oder auch geradeaus Laufen simulieren. Er besteht aus zwei fußgroßen Plattformen. Sobald man einen Fuß zum Gehen hebt, fährt die zuständige Plattform an die Stelle, an der der Benutzer seinen Fuß absetzen wird. Die Information wo dies sein wird, erhält sie durch die virtuelle Welt in der sich der Benutzer befindet und durch die Bewegung die der Benutzer geradeausführen will. Wenn in der virtuellen Welt z.B. Treppenstufen kommen und der Benutzer nach vorne geht, fährt die Plattform so nach oben und vorne, dass der Benutzer genau dann den Fuß auf die Platform setzt, wenn er auch den Fuß auf die virtuelle Treppenstufe setzen würde. Den "Gaitmaster" gibt es als "straight forward type" oder als "omni-directional type". Der erste Typ

Abbildung 7.3: Torus Treadmill

Abbildung 7.4: Gaitmaster

bietet nur die Möglichkeit gerade-aus zu laufen und die Richtung muss man mit einem anderen Eingabegerät bestimmen. Beim zweiten kann man sich in beliebige Richtungen fortbewegen. Allerdings ist dieser dementsprechend aufwändiger und somit teurer. Außerdem ist mit dem "straight forward type" eine Geschwindigkeit von 1,5m/s und mit dem "omni-directional type" nur eine Geschwindigkeit von 0,5m/s möglich.

Die „omni-directional Treadmill" besteht aus zwei Laufbändern, die sich orthogonal in einer Ebene begegnen. Jedes dieser Laufbänder besteht aus 3400 Rollen, mit einem Durchmesser von etwa einem Zentimeter und einer Länge von etwa 5 Zentimeter, die längs zur Laufbandrichtung liegen. Die eine Richtung wird mit dem oberen Laufband ganz normal in Laufbandrichtung erzeugt, die orthogonale Richtung wird dadurch erzeugt, dass das zweite Laufband unter dem anderen liegt und die kleinen Rollen des oberen in orthogonaler Richtung zur oberen Richtung antreibt. So lassen sich beide Richtungen beliebig kombinieren und der Benutzer der auf diesem Laufband läuft, kann sich in beliebige Richtungen fortbewegen, bleibt aber in der realen Welt auf der gleichen Position. Im Vergleich zum "Gaitmaster" lässt sich hiermit nur das Laufen in der Ebene simulieren.

Die "Torus Treadmill" (Abb. 7.3) bewirkt genau das gleiche wie die „omni-directional Treadmill" nur ist sie einfacher aufgebaut. Sie besteht nicht aus lauter kleinen Rollen sondern aus etwa 15 Laufbändern, die quer aneinander gereiht sind und zusammen das große Laufband, in orthogonaler Richtung zu den einzelnen, bilden.

7.2.7 Spracheingabe

[Gabbard '97] Bei Sprache als Eingabe muss man vor dem Einsatz erst einige Überlegungen anstellen. Soll man ständig per Spracheingabe mit dem Computer verbunden sein, oder soll der Computer nur dann Befehle annehmen, wenn ein Knopf gedrückt wird? Soll der Computer auf den Benutzer abgestimmt sein, so muss der Benutzer vor dem Einsatz dem Computer ein paar Texte vorlesen, damit dieser sich an die Stimme „gewöhnt". Wenn man das nicht macht, ist natürlich die Fehlerrate dem entsprechend höher. Dabei muss man sich dann auch gleich überlegen, wie man bei Fehlerkennung weiter verfahren will. Angenommen man befindet sich in einem virtuellen Raum mit anderen Benutzern und will sich mit ihnen auch unterhalten können, dann kann der Computer nur schwer herausfinden, ob ein gesprochener Satz an ihn

oder an einen der anderen Benutzern gerichtet war. Daher benutzt man in verteilten virtuellen Welten selten Sprache als Eingabe.
Computer sind heutzutage zwar in der Lage den gesprochenen Text ins schriftliche zu übertragen, aber wenn es sich nicht um spezielle Befehle handelt, fehlt dem Computer noch das Verständnis über den Inhalt des Textes. Es handelt sich also noch nicht um ein Verständigen oder Kommunizieren mit dem Computer, sondern um ein Befehle geben, die man auch erst lernen, beziehungsweise wissen muss. Der Vorteil von Spracheingabe ist einmal, dass man Befehle geben kann, ohne dazu eine Hand zu benötigen und man kann dem Computer Texte diktieren statt sie mühsam auf einer (virtuellen) Tastatur einzutippen.

7.2.8 Kombinationen

Kombinationen bilden sich ganz automatisch aus den vorhandenen Eingabegeräten. Ob man einfach Tastatur und Maus als Kombination verwendet oder "Spacemouse" und Maus. Es gibt sehr viele Möglichkeiten, wie man die verschiedenen Eingabegeräte kombinieren kann. An dieser Stelle werden nur noch ein paar wenige Beispiele erwähnt. Der "Flex and Pinch" ist eine Kombination aus Datenhandschuh und "Pinch Glove". Das heißt man kann die Finger-/ Handposition und zusätzlich noch den Kontakt zweier Finger als diskretes Signal erfassen. Ein weiteres schönes Beispiel ist eines der vorgestellten zweidimensionalen Laufbänder in einem CAVE (siehe 9.3.1) ausgestattet mit einer "Data Suit" und Datenhandschuhen. Damit kann man durch unendliche virtuelle Welten laufen und beliebig mit Objekten interagieren ohne an irgendeine physische Grenze der realen Welt zu stoßen. Außerdem lässt sich Spracheingabe, wie schon erwähnt, auch gut zu anderen Eingaben kombinieren.

7.3 Ausgabegeräte

7.3.1 Visual

HMD (Head Mounted Display)

Das HMD [Effects '98] ist eines der billigsten und bekanntesten Displays, die räumliches Sehen ermöglichen. Es Besteht aus zwei LCD oder CRT Monitoren, die direkt vor den Augen angebracht sind. Damit kann für jedes Auge ein eigenes Bild erzeugt werden, um einen räumlichen Effekt [Robinett & Holloway '94] zu erzeugen. Der Benutzer trägt es wie einen Helm und kann außer den zwei Monitoren nichts sehen. Damit ist er total abgeschirmt von der realen Welt und kann um so tiefer in die virtuelle Welt eintauchen. Normale HMDs haben allerdings keine besonders gute Qualität und keine sonderlich gute Rundumsicht, was das Gefühl der Immersion in der virtuellen Welt wiederum negativ beeinflusst. Positiv zu bemerken beim HMD ist, dass der Benutzer nicht stark in der Bewegungsfreiheit eingeschränkt wird und mit dem HMD auf dem Kopf sich auch in der realen Welt bewegen kann, was wiederum für spezielle Techniken sinnvoll ist (siehe Techniken/Motion Compression/Redirected Walking). Allerdings sind diese trotzdem recht schwer und passen meist nicht besonders gut.

BOOM (Arm Mounted Display)

Der BOOM (Abb. 7.5) [Gabbard '97] ist ähnlich wie das HMD, nur dass dieser von einem Roboterarm getragen wird. Das bringt Vorteile und Nachteile mit sich.
Einerseits kann man den Benutzer entlasten, da das Gewicht nun von dem Arm getragen wird und man kann daher größere Displays mit besserer Auflösung und besserer Rundumsicht in

Abbildung 7.5: BOOM (Arm Mounted Display)

Abbildung 7.6: Virtual Retinal Display (VRD)

den BOOM einbauen. Voll ausgenutzt werden kann die Rundumsicht dabei allerdings immer noch nicht. Außerdem kann man durch den Arm die Position und Blickrichtung des Benutzers ziemlich exakt erfassen, was sich wiederrum auf den exakten virtuellen Blickpunkt (Viewpoint) übertragen lässt.
Anderseits ist ein großer Nachteil, dass die Bewegungsfreiheit immens eingeschränkt wird. Nicht nur, dass der Benutzer sich nicht mehr frei, wie beim HMD, bewegen kann, sondern er benötigt zudem auch eine Hand für die Kontrolle des BOOM, die dann für die Navigation und Interaktion in der virtuellen Welt nicht zur Verfügung steht.

Brillen

Der Stereo Monitor (Shutter Brille mit Emitter) , meistens als “Shutter Brille“ bezeichnet, ist eine Brille, durch die man abwechselnd mit dem linken oder rechten Auge auf ein Display sieht. Dies kann ein normaler Monitor sein, eine große Leinwand oder auch mehrere Leinwände, die den Benutzer umgeben. Allerdings ist der Einsatz nicht mit einem LCD Bildschirm möglich. Die “Shutter Brille“ lässt die Bilder immer nur für ein Auge durch. Sie ist weiterhin z.B. per Infarot mit dem Display synchronisiert, so dass dieses immer den Blickwinkel abwechselnd vom linken und rechten Auge darstellen kann. Dadurch wird das räumliche Sehen ermöglicht. Als Nachteil lässt sich vermerken, dass zum einen der Monitor die doppelte Frequenz benötigt damit kein Flimmern zustande kommt und es wird, da immer nur ein Auge gleichzeitig sieht, nur die halbe Helligkeit durchgelassen. Für flimmerfreies Sehen sind etwa 60 Herz Bildwiederholfrequenz notwendig. Daher sollte man für den Einsatz mit der “Shutter Brille“ eine Mindestfrequenz von etwa 120 Herz oder mehr benutzen. Die Rundumsicht kann hierbei auch nicht voll ausgenutzt werden, da entweder das zu kleine Display oder die Brille selber das Sichtfeld einschränkt. Außerdem wird das Stereosehen durch das Nachleuchten am Monitor negativ beeinflusst, da dadurch das eine Auge ein Nachleuchten von einem Bild sieht, das gar nicht für das Auge gedacht ist. Dabei gilt, um so besser das Display, um so weniger kommt dieser Effekt zustande. Die “Shutter Brille“ stellt eine billige universelle Lösung für das Stereosehen dar, da sie am Standardarbeitsplatz oder auch in einem CAVE (siehe /Stationär/SSVR) verwendet werden kann.

Das Virtual Retinal Display (VRD) (Abb. 7.6) [Reme '97] wurde 1991 erfunden und sieht aus wie eine Brille, nur dass diese mit zwei Projektoren ausgestattet ist, die das Bild direkt auf die Retina projizieren. Rein theoretisch ist damit ein großes Sichtfeld, eine annähernd perfekte Auflösung, sehr gute Farbtiefe, Stereosehen und Augenbewegungsverfolgung möglich. Außerdem ist sie theoretisch einsetzbar für virtual- und augmented Reality.
Leider ist das noch alles Fiktion und die aktuelle Forschungsarbeit an diesem Display ist noch nicht so weit vorangeschritten. Daher gibt es das VRD nur in Monochrom, mit einer schlechten Auflösung, mit einem kleinen Sichtfeld (etwa 15 Grad), für augmented Reality und für ein Auge. Die Augenbewegung lässt sich heutzutage damit auch noch nicht verfolgen. Jedoch kann man damit rechnen, dass sich bei diesem Forschungsgebiet noch einiges tun wird.

Stationär

Die Surround Screen VR (SSVR) [Gabbard '97] ist nichts anderes als drei bis sechs, jeweils etwa zehn Quadratmeter große Wände, die den Benutzer umgeben und auf die Stereobilder projiziert werden. Um das räumliche Sehen zu ermöglichen muss der Benutzer noch eine "Shutter Brille" oder Polarisationsbrille tragen. Die Projektoren werden über einen oder meist mehrere Rechner gesteuert. Der Benutzer kann sich im Raum frei bewegen und auch reale Objekte können vorkommen. z.B. ein Stuhl auf den man sich dann echt setzten kann. Allerdings wird durch zu viele oder ungünstig platzierte reale Objekte der virtuelle Eindruck geschwächt und das Eintauchen in die virtuelle Welt fällt dem Benutzer relativ genau nicht mehr so leicht. Um das räumliche Sehen zu ermöglichen muss außerdem die Kopfposition und Blickrichtung vom Benutzer erfasst werden um den richtigen Blickpunkt auf den Wänden darzustellen. Der Vorteil von "Surround Screens" liegt darin, dass sie zum einen eine bessere Qualität bieten und die volle Rundumsicht ausnutzen können und zum anderen den Benutzer nicht durch schwere Geräte einschränken. Allerdings bleibt dem Benutzer nur die Bewegungsfreiheit von dem etwa drei auf drei Meter großen Raum. Sobald ein SSVR von mehr als einem Benutzer gleichzeitig benutzt wird, geht der räumliche Effekt verloren oder es kann nur die Sicht eines Beutzers dargestellt werden. Die Bewegungsfreiheit kann allerdings durch spezielle Eingabegeräte, wie z.B. die schon vorgestellte "Omni-directional Treadmill", verbessert werden.
Hier möchte ich noch kurz drei Arten von SSVRs vorstellen. Das **"CAVE (Cave Automated Virtual Environment"** [Symanzik et al. '96] besteht aus einer Frontwand und zwei Seitenwänden die im Winkel von 90 Grad zueinander stehen. Das **"RAVE (reconfigurable advanced visualization environment)"** ist ähnlich wie das "CAVE" nur dass sich die Seitenwände im Winkel variabel von 90 bis 180 Grad zur Frontwand verstellen lassen. Damit kann man eine einzige 30 Quadratmeter große Wand, eine "L-Form" oder aber auch eine "U-Form" wie das "CAVE" erzeugen. Das **"C6"** (Abb. 7.7) besteht aus sechs Wänden, die wie ein Würfel angeordnet sind und auf die die Bilder projeziert werden. Allerdings gibt es diese teuerste Form bisher nur zwei mal auf der Welt.

Workbenches sind ähnlich wie die SSVRs, nur einfacher und somit billiger gehalten. Diese sind dreidimensionale Arbeitsplätze. Man hat hierbei ein bis zwei Displays vor sich, in denen man mit Hilfe von z.B. einer "Shutter Brille" ein dreidimensionales Bild sieht und diesem mit einem speziellen Eingabegerät, wie z.B. mit einem Datenhandschuh oder einem getrackten Stift, interagieren kann. Diese "Workbenches" bieten meist eine hohe Auflösung. Allerdings wird nur ein kleiner Teil des Sichtfeldes ausgenutzt und um für den Benutzer den richtigen Blickpunkt darstellen zu können, ist es wiederum notwendig die aktuelle Position und Blickrichtung des Benutzers zu erfassen. Für spezielle Aufgaben, wie z.B. dreidimensionale Zeichnungen (CAD)

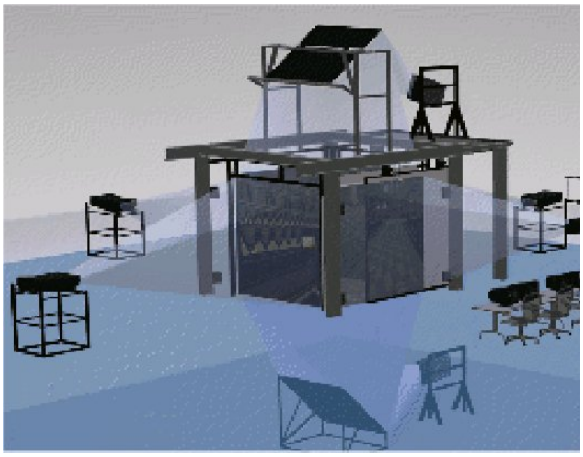


Abbildung 7.7: C6

oder einen Raum mit Möbeln einrichten, sind die “Workbenches“ sehr intuitiv. Man benötigt daher weniger Einarbeitungszeit und man kann effektiver arbeiten als bei einem Standardarbeitsbereich.

7.3.2 Audio

Der Hörsinn kann für verschiedene Aufgaben gut angesprochen werden. Audio kann als einfaches Feedback verwendet werden, z.B. ein Piepston sobald der Benutzer ein Objekt berührt, der dem Benutzer mitteilt, dass er es berührt hat. Außerdem kann Audioausgabe als verstärktes Feedback gut sein. Angenommen der Benutzer klopft an eine Wand, so soll er nicht nur die Kraftrückkopplung spühren, sondern auch das Klopfen hören.
Audio kann zudem hilfreich sein als Computerstimme, die dem Benutzer Tipps gibt oder irgendwelche Texte vorlesen kann. Die natürlichste Art seinen Hörsinn zu benutzen ist die Kommunikation mit anderen Benutzern in der virtuellen Umgebung.

7.3.3 Tastsinn

[Gabbard ’97] Auf eine Aktion folgt eine Reaktion. Ohne diese Reaktion würde ein wesentlicher Teil der Realität fehlen und die virtuelle Welt käme einem viel künstlicher und unnatürlicher vor. Daher gibt es verschiedene Arten den Tastsinn anzusprechen und eine passende Reaktion zu erzeugen. Im Folgenden werden nur Beispiele genannt, die sich auf einen Datenhandschuh beziehen. Denkbar wären natürlich auch Rückkopplungen auf den ganzen Körper oder auf andere Teile des Körpers.

„ground referenced“ (Abb. 7.9) beschreibt eine Kraft, die von einem externen Punkt ausgeht, etwa das Berühren einer Wand. Hierbei ist z.B. die Hand mit einem Computerarm verbunden, der die Hand festhält sobald man die Wand berührt hat. Außerdem ist diese Methode dazu gut die Schwerkraft und somit das Gewicht von hochgehobenen Objekten zu vermitteln.

„body referenced“ (Abb. 7.8) bezeichnet eine Kraft, die von dem eigenen Körper ausgeht. Wie etwa eine Kraftrückkopplung auf die Finger in Bezug auf die Hand. Somit kann ein Benutzer z.B. eine Dose greifen und dabei den Widerstand der Dose beim Schließen der Hand

Abbildung 7.8: „body referenced“ Feedback

Abbildung 7.9: „ground referenced“ Feedback

spüren. So muss er dann, wie in der Realität, eine entsprechend größere Kraft aufbringen um die Dose zusammenzudrücken.

„tactile“ stellt keine wirkliche Kraft dar, sondern eher ein Fühlen. Es handelt sich hierbei um Vibrationsmotoren, die an den Fingern angebracht sind. Sobald man nun mit den Fingern über eine Oberfläche streicht, wird durch die Vibrationen die Rauhigkeit der Oberfläche dargestellt. Um so rauer um so mehr Vibrationen werden erzeugt.

„dermal tactile“ ermöglicht ein Temperaturspüren in den Fingerspitzen. Peltier-Elemente sorgen dafür, dass sich die Temperatur an den Fingerspitzen regeln lässt. Aluminiumplatten übertragen die Temperatur gut auf die Finger und ein Wasserkühlkreislauf sorgt dafür, dass die Peltier-Elemente nicht überhitzen. Mit dieser doch recht aufwändigen Techniken lässt sich das Gefühl von Wärme und Kälte erzeugen. So wird, wenn man in der virtuellen Welt Stahl anfasst, eine dementsprechende Kälte vermittelt. Damit sind Temperaturänderungen von maximal 1,5 Kelvin pro Sekunde möglich und das in einer Temperaturspanne von 8-80°C. Diese Einschränkung ist nicht nur technisch bedingt, sondern dient auch zur Sicherheit des Benutzers. Wenn er sich z.B. einem Feuer nähert, wird so eine Verbrennung der Fingerkuppen verhindert.

Allgemein ist noch zu sagen, dass die Technik noch nicht so weit fortgeschritten ist um die realen Eindrücke auf die virtuelle Welt eins zu eins zu übertragen. Es handelt sich eher um ein Annähern an die reale Welt. Wenn man mehrere oder alle Formen der genannten Rückkopplungen gemeinsam nutzt, kann man jedoch zumindest mit der Hand eine relativ gute Rückkopplung erfahren.

7.3.4 Motionware device (elektrische Impulse hinter dem Ohr)

Dies ist ein Ausgabegerät, das durch elektrische Stimulation am Gewichts- und Orientierungssinn, ein Gefühl der Bewegung erzeugt. Dabei werden elektrische Impulse an den achten Gehirnnerv gesendet. Dies lässt sich gut mit anderen Geräten wie dem schon erwähnten fliegenden Teppich kombinieren. Wenn der Benutzer nicht nur durch die Welt fliegen kann, sondern ihm zusätzlich noch das Gefühl der Bewegung vermittelt wird, kann er viel besser und intensiver in die virtuelle Welt eintauchen. Da allerdings elektrische Impulse direkt an den Gehirnnerv geschickt werden, stehen Benutzer diesem Gerät meist skeptisch gegenüber.

7.4 Techniken

Dieses Kapitel umfasst die softwaretechnischen Aspekte und Interaktionsmöglichkeiten für die Navigation und Interaktion in virtuellen Welten.

7.4.1 Navigation

Die Navigation, siehe auch [Bowman et al. '96], ist allgemein dazu gut sich in virtuellen Welten fortzubewegen. Dies geschieht meist durch eine kontinuierliche Bewegung und nicht durch Teleportation, da diese meist zu erheblicher Desorientierung führen würde. Das Navigieren lässt sich in verschiedene Arten unterteilen:

Beim Erforschen bewegt man sich ohne festes Ziel durch die virtuelle Welt und erkundet die Gegend.

Beim Suchen kennt man entweder die Position von dem Objekt und man bewegt sich zu dem Objekt, oder die Position ist unbekannt. Dann ist es eher ein Erforschen der Umgebung bis man das Objekt gefunden hat. Die dritte Art für das Suchen ist die Möglichkeit, dass man zwar das Objekt nicht sieht, aber die Umgebung kennt und sich gezielt durch die Umgebung bewegt, bis das Ziel erreicht ist.

Manövrieren bezeichnet kleine, genaue Bewegungen um die Ausgangsposition für eine Aufgabe einzunehmen oder den Blickpunkt nur etwas zu verändern.

Lenken ist, wie der Name schon sagt, eine kontinuierliche Bestimmung der Bewegungsrichtung und der Geschwindigkeit. Diese Methode ist vergleichbar mit dem Fahren eines Autos in der Realität.

Ziel-basiert ist eine Art zu navigieren, bei der man einmalig ein Ziel bestimmt und sich automatisch dort hinbewegt. Dabei lässt sich die Bestimmung des Ziels per Karte, direktes Auswählen oder auch durch Auswählen aus einer Liste durchführen.

Die Routenplanung ist ähnlich wie die Ziel-basierte Methode, nur dass man mehrere Punkte in einer bestimmten Reihenfolge eingeben kann, zu denen man sich nacheinander automatisch hinbewegt.

Manuelle Manipulation der Sicht (Viewpoint) beinhaltet zwei Möglichkeiten: "camera in hand" und "fixed object manipulation". Bei ersterer hat man die Kamera unter Kontrolle und fährt oder fliegt mit dieser durch die Szene. Bei der zweiten wählt man ein Objekt aus und kann dann die Szene um dieses Objekt drehen.

Raumkompression

In der Regel hat man das Problem, dass man große virtuelle Szenarien hat, aber nur kleine reale Räume zur Verfügung stehen. Hierbei wird die Raumkompression eingesetzt. Diese bewirkt, dass man sich in kleinen realen Räumen annähernd ohne Grenzen bewegen kann. Bei dem Versuch, eine Person mit verbundenen Augen auf einem geraden Strich laufen zu lassen, wich

diese in einer Kreisform von dem Strich ab. Dies nutzt man bei Raumkompression aus, indem man die virtuelle Sicht dreht, ohne dass der Benutzer etwas davon merkt.

Bei Redirected Walking in Place [Jr. et al. '01] handelt es sich um ein auf-der-Stelle-gehen in einem "CAVE". Allerdings muss auch hierbei die Sicht so gedreht werden, dass der Benutzer nie die offene Rückwand zu Gesicht bekommt. Angenommen ein Benutzer dreht sich um 90 Grad nach links und sieht jetzt die linke Seitenwand, wird die Sicht einfach so gedreht, dass der Benutzer ohne es zu merken sich wieder zur Frontwand dreht, die virtuelle Sicht vom Benutzer aber die gleiche bleibt. Oder ein Benutzer steht zur Frontwand und will sich um 180 Grad drehen und würde dann theoretisch die offene Rückwand sehen. Allerdings dreht sich die virtuelle Sicht dann zwar um die gewünschten 180 Grad, der Benutzer hat sich in der Realität aber nur um z.B. 90 Grad gedreht. Wie stark die Sicht beeinflusst wird, hängt von dem Winkel zwischen Blickrichtung und offener Rückwand ab. Um so mehr man sich der Rückwand zuwendet, um so mehr wird die Sicht gedreht.
Eindeutiger Nachteil des "Redirected Walking in Place" ist, dass man nur auf der Stelle gehen kann, was die Kontrolle über die Bewegungen des Avatars nicht einfach macht. Man kann z.B. nicht einen genau ein Meter großen Schritt machen, da man ja nur auf der Stelle gehen kann. Möglich wäre allerdings auch eine Kombination mit einer "omni-directional Treadmill" um den Nachteil des nicht echten Gehens zu beseitigen.

Redirected Walking [Razzaque et al. '01] ist ähnlich wie "Redirected Walking in Place", allerdings läuft man hierbei wirklich durch den realen Raum mit einem HMD oder ähnlichem auf dem Kopf. Die Ähnlichkeit besteht darin, dass der Benutzer ebenfalls in seiner Bewegung getäuscht wird und nicht die gleichen realen Bewegungen macht, wie der Avatar, sondern es wird ebenfalls die Sicht immer so gedreht, dass man nicht gegen Wände laufen kann. Allerdings ist dazu ein Raum von mindestens drei auf zwölf Meter nötig. Bei Versuchen mit kleineren Räumen musste man entweder die Sicht zu stark drehen, was wiederum zu der so genannten Simulatorkrankheit führte, oder bei zu geringer Korrektur kollidierte die Versuchsperson mit der Wand. Außerdem gilt bei der Wahl der Raumgröße, dass um so größer die virtuelle Welt ist, um so größer sollte der reale Raum sein. Wobei die benötigte Größe des realen Raums logarithmisch abnimmt. In Abb. 7.10 kann man sehen wie sich der Benutzer bei einem Versuch im realen Raum (Abb. 7.10 unten) und wie er sich dabei im virtuellen Raum (Abb. 7.10 oben) bewegt hat.

7.4.2 Manipulation

Manipulation lässt sich in die drei voneinander unabhängige Arten aufteilen: Selektion, Positionierung und Rotation. Die verschiedenen Techniken für die einzelnen Arten lassen sich beliebig kombinieren.

Selektion und Positionierung

Hier werden zuerst ein paar Techniken vorgestellt, die Selektion und Positionierung ermöglichen. Die meisten davon werden auch in [Poupyrev et al. '98] behandelt.

(simple) Virtuelle Hand funktioniert wie ein Mauszeiger im Dreidimensionalen. Diese Methode ist ungenau für weit entfernte und kleine Objekte. Allerdings ist es eine recht intuitive Methode.

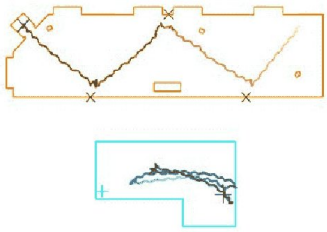


Abbildung 7.10: Redirected Walking

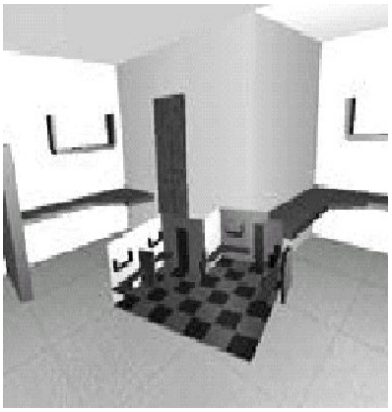


Abbildung 7.11: World in Miniature

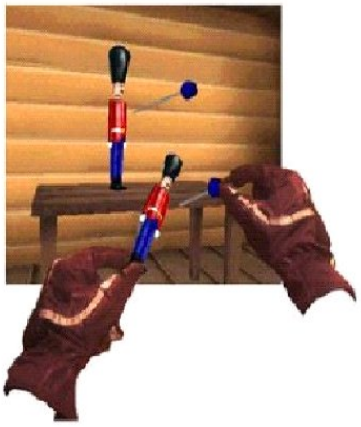


Abbildung 7.12: Voodoo Dolls

Der virtuelle Zeigestab - (Virtual Pointer - Ray-casting (Bolt, 1980)) ist nichts anderes als ein virtueller, unendlich langer Zeigestab mit dem man Objekte selektieren kann. Für weit entfernte, kleine Objekte auch nicht gut geeignet, da der Zeigestab, bei einer winzigen Handbewegung, in der Entfernung eine sehr große Bewegung macht.

Taschenlampe (Liang, 1994) (Flash light) funktioniert ähnlich wie der virtuelle Zeigestab, nur dass ein ganzer Lichtkegel geworfen wird und alles was sich in dem Lichtkegel befindet wird selektiert. Damit lassen sich auch weit entfernte und kleine Objekte gut selektieren, allerdings können auch Objekte, die der Benutzer nicht selektieren will, in den Auswahlbereich fallen.

Aperture (Forsberg, 1996) benutzt einen virtuellen zweidimensionalen Kreis, den der Benutzer beliebig in Grösse und Position ändern kann. Selektiert wird dann alles, was der Benutzer durch den Kreis sieht. Für die verschieden großen und verschieden weit entfernten Objekte lässt sich der Kreis je nach Selektionswunsch vergrößern und verkleinern. Durch die zweidimensionale Auswahl ist die Methode recht einfach und intuitiv benutzbar.

Image plane (Pierce, 1997) bezeichnet eine virtuelle zweidimensionale Wand direkt vor dem Benutzer auf der dieser Objekte, durch auf die Wand klicken, selektieren kann.

Bei Ray-casting (Bowman, 1997) hat man eine Angelleine in der Hand, die einen geraden Strahl zu dem Objekt darstellt. Auf diesem Strahl kann man die Entfernung kontrollieren, allerdings benötigt man dazu ein extra Eingabegerät. Daher hat man dann auch nur noch eine Hand übrig für die Interaktion mit dem Objekt.

Go-Go (Poupyrev 1998) stellt eine intuitive Methode dar, bei der der Arm ab einer gewissen Distanz zum Körper exponentiell gedehnt wird. Diese Distanz oder auch Grenze befindet sich in etwa 60cm Entfernung zum Körper, sie kann allerdings auch an den Aufgabenbereich angepasst werden. Bei dieser Methode kann man bis zu der Distanz mit Objekten ganz normal interagieren und ab der Grenze wird der Arm so gedehnt, dass man auch Objekte die weit entfernt sind noch greifen kann. Positiv an dieser Methode ist der nahtlose Übergang von normaler Interaktion und Interaktion mit einem gedehnten Arm. Allerdings hat man in großer

Entfernung keine gute Kontrolle über die Objekte, da der Blickpunkt dabei ja immer der gleiche bleibt und der Arm durch die exponentielle Dehnung nicht mehr gut kontrollierbar ist.

World-in-Miniature [Stoakley et al. '95] ist eine Art 3D-Karte, wie man in Abb. 7.11 sieht, in der man Objekte manipulieren kann. So kann ein dreidimensionaler Raum in dem man sich befindet auf diese dreidimensionale verkleinerte Darstellung abgebildet werden. Diese muss sich komplett in der Reichweite des Benutzers befindet, damit dieser in der Karte Objekte manipulieren kann. Diese Manipulation wirkt sich dann direkt auf die Objekte in dem großen virtuellen Raum, in dem sich der Benutzer befindet, aus. Kleine Objekte lassen sich damit schwer manipulieren, da sie in der verkleinerten Karte um ein vielfaches kleiner sind. Bei großen Szenarien kann man auch nur Teile der virtuellen Welt in der Karte darstellen lassen.

Non-isomorphic 3D Rotation Techniques (Poupyrev, 2000)

Die grundsätzliche Formel für die Übertragung von Rotation auf die virtuelle Welt lautet: $R_V = {R_R}^k$. Wobei R_V die Rotation in der virtuellen und R_R die Rotation des Eingabegerätes ist. Und k stellt einen Verstärkungsfaktor dar. Wenn der z.B. $k = 1$ gewählt wird, entspricht die reelle Rotation der virtuellen. Nun gibt es zwei Möglichkeiten bei der Rotation: die absolute und die relative Rotation.
Die absolute ist die intuitivere Variante, da sie der Wirklichkeit näher kommt. Bei $k = 1$ dreht sich das virtuelle Objekt genau gleich wie das Eingabegerät und wenn man auf die Ausgangsstellung zurückgeht, geht auch das virtuelle Objekt wieder in die Ausgangsstellung zurück. Allerdings ist man bei der Methode eingeschränkt, was die Rotationsweite betrifft. Wenn man einen Datenhandschuh als Eingabegerät hat, lässt sich das virtuelle Objekt nur so weit drehen, wie auch die Hand sich drehen lässt. Daher gibt es die relative Variante, bei der mit einer Rotation des Eingabegeräts die Rotationsgeschwindigkeit des virtuellen Objekts gesteuert wird. Kehrt man auf die Ausgangsposition zurück, bleibt das virtuelle Objekt zwar stehen, aber es befindet sich dann nicht mehr in der Ausgangsposition. Damit lassen sich unendlich lange Drehungen erzeugen, allerdings ist es nicht so intuitiv und die Rotationen lassen sich auch nicht so genau steuern wie bei der absoluten Rotation. Als Kompromiss kann man natürlich zwischen den zwei Rotationsvarianten manuell hin und her schalten.

Two Handed

Da man aus der realen Welt gewohnt ist beide Hände für bestimmte Aufgaben zu benutzen, sollte man dies auch in der virtuellen Welt ermöglichen, um eine intuitivere Interaktion zu ermöglichen. Nachteil ist allerdings, dass man zum einen erhöhte Kosten hat durch den zusätzlichen Datenhandschuh, zum anderen ist der Implementierungsaufwand wesentlich höher.

Die Fenstergestützte Interaktion stellt ein Beispiel dar, wie man mit zwei Händen sinnvoll in einer virtuellen Welt agieren kann. Man kann es sich vorstellen wie ein Fenster (Abb. 7.13), das sich direkt vor dem Avatar in der virtuellen Welt befindet. Dieses lässt sich verschieben, vergrößern, verkleinern und man kann damit auch zoomen. Verwenden kann man das Fenster indem man durch es hindurch greift, was man andeutungsweise in Abb. 7.14 sehen kann, und so mit den gewünschten Objekten interagiert oder man tippt auf die virtuelle Glasoberfläche des Fensters, um ein Objekt zu selektieren. Man kann also z.B. an ein Objekt hinzoomen und dann, durch das Fenster hindurch, mit dem Objekt interagieren.

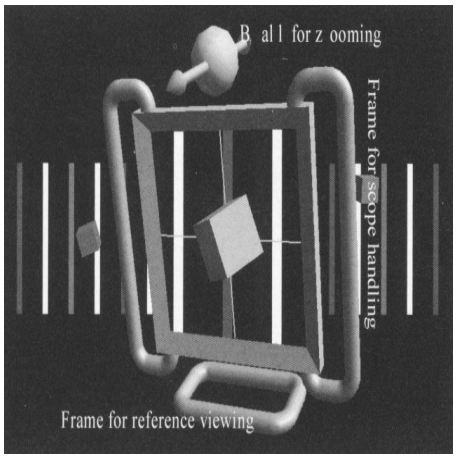


Abbildung 7.13: Fenstergestützte Interaktion

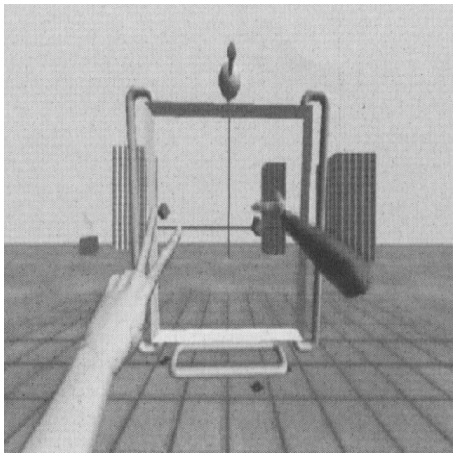


Abbildung 7.14: Fenstergestützte Interaktion (virtuelles Fenster in Benutzung)

Kombinationen der Techniken

Um verschiedene Techniken kombinieren zu können, gibt es zwei Möglichkeiten: die Aggregation und die hybriden Techniken.

Aggregation ist die manuelle Umschaltung auf den gewünschten Modus. Dies geschieht durch ein virtuelles Menü oder anderen Eingabegeräte.

Hybride Techniken stellen die automatische Umschaltung dar. Da Selektion und die Manipulation selber unabhängig voneinander sind, kann man auch automatisch zwischen Selektion und Manipulation umschalten lassen. Meist ist man am Anfang im Selektionsmodus und sobald man ein Objekt selektiert hat wird automatisch in den Manipulationsmodus umgeschalten und man kann das Objekt rotieren oder verschieben. Nach dem Loslassen des Objektes kehrt man dann automatisch wieder in den Selektionsmodus zurück. Bei der Umschaltung von Selektion auf Manipulation wird bei "HOMER (Hand-centered Object Manipulation)" die Hand zum selektierten Objekt gebracht. Bei "Scaled-world grab" hingegen wird der Blickpunkt an das Objekt herangebracht.

Voodoo Dolls (Voodoopuppe)(Pierce et al., 1999) (Abb. 7.12) [Isidoro & Sclaroff '98] ist eine recht neue Technik, die nicht einfach zu realisieren ist, aber in der Anwendung ist sie sehr intuitiv und effektiv. Bei dieser hybriden Technik wird nach der Selektion, eine handgroße Kopie des selektierten Objekts in die Hand gegeben, an dem man nun Manipulationen beliebiger Art durchführen kann. Diese Manipulationen werden dann auch unmittelbar an das echte virtuelle Objekt übertragen. (Dies funktioniert also wie der Mythos der Voodoopuppe in der Realität.) Damit lassen sich auch bewegte Objekte beliebig und in aller Ruhe bearbeiten obwohl diese sich dabei immer noch fortbewegen. Außerdem ist die Grösse und Entfernung des Objekts egal, da man ja eine handgroße Kopie direkt in die Hand bekommt.

7.5 Zusammenfassung

7.5.1 Fazit

Als abschließende Bemerkung kann man zusammenfassen, dass es nicht **eine** beste Technik, Ein- oder Ausgabegerät gibt, sondern für jede Aufgabe gibt es eine Kombination die am sinnvollsten ist. Es lassen sich Techniken schön aus der zweidimensionalen Computer Welt übertragen, wie z.B. Menüs und Knöpfe. Aber auch Techniken aus der realen Welt kommen hier zur Verwendung, z.B. das Gehen oder Autofahren. Zusätzlich lassen sich aber auch Techniken aus der Fantasie übertragen, wie z.B. das Fliegen mit dem fliegenden Teppich oder die Voodoopuppe. Es bietet sich also eine neue Welt in der alles möglich ist, nur ist die Technik noch nicht so weit alles zu realisieren.

7.5.2 Ausblick

Durch die Weiterentwicklung der Computer lassen sich auch bessere virtuelle Welten erschaffen. Mit der immer besser werdenden Technik werden sicher auch neue Ein- und Ausgabegeräte entwickelt, die für bestimmte Aufgabenbereiche besser geeignet sind als alles was es im Moment gibt. Hardware wird zwar weiterentwickelt, aber der Schritt zur perfekten virtuellen Welt ist noch weit entfernt. Um die reale Welt uneingeschränkt zu imitieren, müsste eine perfekte Schnittstelle wie in „Matrix“ implementiert werden, die direkt die Gehirnströme als Schnittstelle verwendet.

Literaturverzeichnis

[Bowman & Wingrave '01] Doug A. Bowman und Chadwick A. Wingrave. Design and Evaluation of Menu Systems for Immersive Virtual Environments. In *VR*, S. 149–156, 2001.

[Bowman et al. '96] Doug A. Bowman, David Koller und Larry F. Hodges. Evaluation of Movement Control Techniques for Immersive Virtual Environments. Technical Report 96-23, 1996.

[Dachselt '00] Raimund Dachselt. Action Spaces - A Metaphorical Concept to Support Navigation and Interaction in 3D Interfaces, 2000.

[Effects '98] The Visual Effects. Vision Research 38 (1998) 2053 – 2066, 1998.

[Gabbard '97] J. Gabbard. A taxonomy of usability characteristics in virtual environments, 1997.

[Isidoro & Sclaroff '98] John Isidoro und Stan Sclaroff. Active Voodoo Dolls: A Vision Based Input Device for Non-rigid Control. Technical Report 1998-006, 1998.

[Jr. et al. '01] Joseph J. LaViola Jr., Daniel Acevedo Feliz, Daniel F. Keefe und Robert C. Zeleznik. Hands-free multi-scale navigation in virtual environments. In *Symposium on Interactive 3D Graphics*, S. 9–15, 2001.

[Moghaddam & Buehler '93] M. Moghaddam und M. Buehler. Control of Virtual Motion Systems, 1993.

[Poupyrev et al. '98] I. Poupyrev, T. Ichikawa, S. Weghorst und M. Billinghurst. Egocentric Object Manipulation in Virtual Environments: Empirical Evaluation of Interaction Techniques. *Computer Graphics Forum*, 17, Nr. 3, 1998.

[Razzaque et al. '01] Sharif Razzaque, Zachariah Kohn und Mary C Whitton. Redirected Walking. Technical Report TR01-007, 13 2001.

[Reme '97] Hat Isaugme Reme. unknown, 1997.

[Robinett & Holloway '94] Warren Robinett und Richard Holloway. The Visual Display Transformation for Virtual Reality. Technical Report TR94-031, 10, 1994.

[Stoakley et al. '95] Richard Stoakley, Matthew J. Conway und Randy Pausch. Virtual Reality on a WIM: Interactive Worlds in Miniature. In *Proceedings CHI'95*, 1995.

[Symanzik et al. '96] Jürgen Symanzik, Dianne Cook, Bradley D. Kohlmeyer und Carolina Cruz-Neira. Dynamic Statistical Graphics in the CAVE Virtual Reality Environment, 1996.

[Turner et al. '96] Russell Turner, Enrico Gobbetti und Ian Soboroff. Head-Tracked Stereo Viewing with Two-Handed 3D Interactionfor Animated Character Construction. *Computer Graphics Forum*, 15, Nr. 3, S. 197–206, 1996.

[vrLab '03] vrLab. Gaitmaster, 2003.

[Will '02] Roger Bostelman Will. A Tool To Improve Efficiency In Large Scale Manufacturing, 2002.

8 Personenverfolgung

Jörn Herwig

8.1 Einführung

Der vorliegende Text befasst sich mit der Verfolgung und Erfassung von Personen und deren Bewegungen im Bereich Virtual Environments. Zunächst werden die Begriffe der Bewegungs-Verfolgung und insbesondere der Personen-Verfolgung kurz anhand der vielfältigen Anwendungsbereiche erläutert. Anschließend werden die Erfordernisse, die das Gebiet der Virtuellen Umgebungen an die verwendete Technik stellt, erörtert. Anhand dieser Erfordernisse erfolgt nun eine Übersicht und Beurteilung der verschiedenen zur Verfügung stehenden Techniken. Daraufhin wird detaillierter auf die Möglichkeiten und Schwierigkeiten einiger ausgewählter Vorgehensweisen aus den Bereichen markerbasierter optischer Verfahren, Gesichtsverfolgung und weiträumigem Einsatz von Augmented-Reality-Anwendungen eingegangen. Abschließend wird noch ein Blick auf die Zukunft dieses Gebiets geworfen.

8.1.1 Virtual Environments und Personen-Verfolgung

Virtual Environments (VE) – Virtuelle Umgebungen, eine Technologie, die es ermöglicht, in eine vom Computer generierte Welt "einzutauchen", sei dies nun zur Produktvisualisierung, zu Forschungszwecken oder einfach als Spiel. Der Begriff beinhaltet schon, dass hierzu mehr nötig ist, als dem Benutzer nur eine (im Rahmen der technischen Möglichkeiten) möglichst perfekte grafische Darstellung zu bieten. Er muss vielmehr von der virtuellen Welt umgeben sein. Diese muss auf die Handlungen des Benutzers reagieren. So muss es eine virtuelle Umgebung ermöglichen, dass der Benutzer in das Geschehen eingreift, sich insbesondere weitgehend frei bewegen und umsehen kann.
Um diese Interaktion zu ermöglichen, müssen die Bewegungen des Benutzers, beispielsweise die Ausrichtung des Kopfes und der Hände, erfasst und in eine dem VE-System verständliche Form gebracht werden. Dieser Prozess wird als Bewegungs-Verfolgung bzw. Motion Tracking bezeichnet.

8.1.2 Bewegungs-Verfolgung

Bewegungs-Verfolgung ist nicht nur für den Bereich der Virtual Environments wichtig, sie tritt in nahezu allen Bereichen unseres Lebens in Erscheinung. Sei es beim Tachometer im Auto, der Navigation von Schiffen per GPS[1], der Überwachung von Flugbewegungen mit Hilfe des Radars oder einer einfachen Computer-Maus, bei all diesen Aufgaben kommen Verfahren zur Ermittlung von Bewegungsdaten zum Einsatz. Alle diese Gerätschaften stellen nichts anderes als "Bewegungs-Verfolger" dar, wobei im Folgenden, aufgrund der Sperrigkeit dieses Begriffs, die entsprechende englische Übersetzung "Tracker" verwendet wird.

[1] Global Positioning System

Für die verschiedenen Zwecke sind unterschiedliche Daten relevant. So ermittelt ein Tachometer nur die Vorwärts-Geschwindigkeit des Fahrzeugs, die GPS-Satellitennavigation dagegen die momentane Position des Empfängers.

Alle Verfahren ermitteln hierbei einen Teil oder alle der möglichen sechs Freiheitsgrade, welche ein fester Körper im dreidimensionalen Raum besitzt, bzw. deren erste (Geschwindigkeiten) oder zweite Ableitungen (Beschleunigungen). Im Einzelnen sind dies drei Positionsangaben und drei Rotationsangaben. Mit Hilfe dieser Werte ist die Position eines Körpers im Raum eindeutig festgelegt. Da außerdem der Zeitpunkt der Messung bekannt ist, können natürlich auch die Änderungen der Werte, also die Geschwindigkeiten und die Beschleunigungen einfach errechnet werden.

Ebenso kann der entgegengesetzte Weg beschritten werden. Durch kontinuierliche Messung der Geschwindigkeiten und der Kenntnis der Ausgangslage kann die Position bestimmt werden. Diese Vorgehensweise weist allerdings eine nicht zu unterschätzende Tücke auf: Messfehler, und seien sie auch noch so klein, führen dazu, dass mit fortschreitender Zeit die errechnete Position immer mehr von der Realität abweicht. Noch gravierender wirkt sich dies aus, wenn die Beschleunigungswerte als Ausgangsdaten herangezogen werden. Ohne regelmäßige, genauere Positionsbestimmungen zur Korrektur der Abweichung können die so ermittelten Positionen erheblich vom korrekten Wert abweichen und taugen nicht einmal mehr als grobe Schätzung.

Besitzt ein Tracker keine Möglichkeit, die Rotations-Ausrichtung eines Messpunkts direkt zu bestimmen, so kann dies durch die Positionsbestimmung von drei Punkten erfolgen. Diese Punkte müssen einen Winkel einschließen, dürfen also keinesfalls auf einer Linie liegen. Außerdem ist zu beachten, dass sich die Punkte nicht gegeneinander verschieben dürfen, ihre Position auf dem Körper zueinander muss konstant bleiben. Die Rotation kann nun aus den ermittelten Positionen berechnet werden. Sind nur zwei Rotationsebenen relevant, so werden nur zwei Messpunkte benötigt.

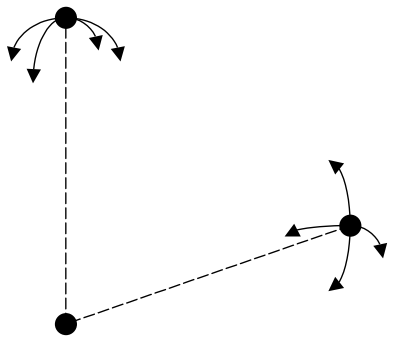


Abbildung 8.1: Rotationsbestimmung mit drei Punkten

Alternativ kann mit Hilfe von mehreren Messpunkten auch der Winkel eines beweglichen Gelenks ermittelt werden. Dazu muss ein Punkt auf oder zumindest nahe dem Gelenk angebracht sein und die beiden anderen Punkte jeweils auf einer der vom Gelenk abgehenden Verbindungen liegen.

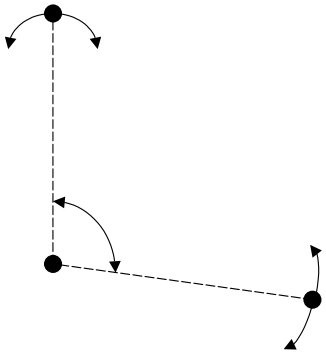


Abbildung 8.2: Winkelbestimmung mit drei Punkten

Bewegungs-Verfolgung beim Menschen

Auch die Bewegungen des menschlichen Körpers können mit geeigneten Techniken erfasst werden. Dies reicht von der einfachen Ortsbestimmung bis zur vollständigen Erfassung aller Bewegungen, inklusive Gestik und Mimik. Eine breite Palette von Anwendungen macht sich diese Möglichkeiten zu Nutze. Abläufe werden vereinfacht, beschleunigt oder überhaupt erst ermöglicht. Unter anderem finden sich Einsatzgebiete in der Industrie, etwa bei Ergonomietests neuer Produkte, der Wissenschaft, z. B. bei der Analyse menschlicher Bewegungsabläufe, beim Film, insbesondere bei Spezialeffekten wie in *Matrix*, in Computerspielen, wo realistischere Bewegungen der Spielfigur wie in *Splinter Cell* ermöglicht werden, oder eben beim Einsatz von Virtual-Environment-Systemen. Meistens wird in diesem Zusammenhang nicht von Bewegungs-Verfolgung bzw. Motion Tracking gesprochen, sondern von Motion Capturing, da die Bewegungsdaten oft zur weiteren Verarbeitung gespeichert, also “aufgefangen” werden.

(a) Film - Matrix Reloaded (b) PC-Spiel - Splinter Cell (c) Virtual Environment

Abbildung 8.3: Anwendungsbereiche von Human Motion Tracking

Eine sehr eingeschränkte Art der Bewegungsverfolgung wurde bereits genannt: die Computer-Maus. Sie erfasst in begrenztem Maße die Bewegungen der Benutzerhand in der zweidimensionalen Ebene des Schreibtisches und setzt diese in Veränderungen der Mauszeiger-Position um. Dieses Vorgehen mag für die Bedienung gebräuchlicher grafischer Benutzeroberflächen weitestgehend brauchbar sein, ist aber keineswegs für eine intuitive Navigation in virtuellen Umgebungen geeignet. Es ist deshalb nötig, andere Verfahren für die Sicht- und Interaktionssteuerung innerhalb von VE-Anwendungen einzusetzen.

Abbildung 8.4: Computer-Maus

8.2 Verfahren

Bevor nun auf die verwendeten Technologien im Einzelnen eingegangen wird, ist es zunächst nötig, die Anforderungen zu kennen, die das Einsatzgebiet Virtual Environments mit sich bringt. Mit Hilfe dieser Anforderungen ist es möglich, die verschiedensten zum Einsatz kommenden Verfahren zu vergleichen und die für den individuellen Anwendungsfall geeignetsten auszuwählen. An dieser Stelle soll ausdrücklich darauf hingewiesen werden, dass keines der vorgestellten Verfahren für jeden Einsatzzweck uneingeschränkt nutzbar ist. Es gibt keinen "Heiligen Gral" der Bewegungs-Verfolgung, keine Technologie, die besser ist als alle anderen. Es ist deshalb nötig, Kompromisse einzugehen, und sei es nur aufgrund eines limitierten Budgets.

8.2.1 Anforderungen

Im Wesentlichen ergeben sich die Erfordernisse an das Messverfahren aus folgenden Kategorien:

1. der angestrebten Annäherung an die Eigenschaften eines idealen Trackers,
2. den erforderlichen Bewegungsdaten,
3. der verwendeten Darstellungstechnik.

Idealer Tracker

Wie bereits erwähnt, gibt es keinen idealen Tracker, der alle Erfordernisse abdeckt. Aber um einen Leistungsvergleich der verschiedenen Ansätze durchführen zu können, ist die gedankliche Konstruktion eines solchen "Allheil-Trackers" durchaus angebracht. Dieser kann als Referenz verwendet werden, um die Stärken, aber mehr noch die Schwächen der realisierbaren Systeme erkennen und einschätzen zu können.
Wie sieht nun ein solcher idealer Tracker aus?

- Er muss klein sein. Das heißt, der Tracker muss leicht zu transportieren sein und, was noch wichtiger ist, er darf dem Benutzer nicht hinderlich oder gar störend sein.

- Er muss eigenständig betrieben werden können. Keine weiteren sperrigen, wartungsintensiven Geräte, wie beispielsweise Kameras oder Magnetfeldgeneratoren, werden benötigt.

- Es werden alle sechs Freiheitsgrade abgedeckt. Sowohl Position als auch Ausrichtung werden erfasst. Ist der Baustein klein genug, dann können, wie bereits erwähnt, auch alternativ drei Positionsbestimmungen für die Rotation herangezogen werden.

- Die erreichbare Genauigkeit muss sehr hoch sein. Dies schliesst die mittlere Genauigkeit und die Stärke von schwankenden Messfehlern (Jitter) ein.

- Der Tracker muss schnell sein. Soll heißen, er weist eine niedrige Latenzzeit auf, um Echtzeitanwendungen zu ermöglichen, und besitzt eine hohe Abtastrate um auch schnelle Bewegungen präzise zu erfassen. Die Leistung darf außerdem nicht von der Anzahl der verwendeten Trackereinheiten beeinflusst werden.

- Er ist immun gegenüber Verdeckung. Es muss also keine direkte Sichtlinie zu irgendeinem Punkt, z. B. Sensoren oder Referenzpunkten, bestehen.

- Umwelteinflüsse haben keine Auswirkungen auf die Funktion oder Messgenauigkeit des Trackers. Dies schließt Lichtverhältnisse, Temperatur, Umgebungsgeräusche, elektromagnetische Felder und vieles mehr mit ein.

- Es kommt zu keinem Abbruch der Verfolgung, egal wie schnell sich der Messpunkt bewegt oder wie weit er sich vom Startpunkt entfernt.

- Die Anbindung an die Empfangsstation muss schnurlos erfolgen.

- Nicht zuletzt muss ein solcher Tracker billig sein und keine Tausende bis Hundertausende von Euro kosten.

Die momentan eingesetzten Techniken erfüllen höchstens drei der geforderten Eigenschaften. Und dies wird wohl auch auf absehbare Zeit so bleiben.

Bewegungsdaten

Weiterhin wichtig ist die Frage, von welchen Bewegungen des menschlichen Körpers überhaupt Daten gebraucht werden. Eine komplette Erfassung aller Bewegungen erscheint übertrieben, wenn z. B. ausschließlich die Ausrichtung des Kopfes bzw. die damit verbundenen genauen Positionen der Augen von Interesse sind.
Die üblichen Aufgaben des Human-Motion-Tracking lassen sich folgendermaßen gliedern:

- Sichtkontrolle

- Navigation innerhalb der virtuellen Umgebung

- Objektauswahl/-manipulation

- Animation von Stellvertretern (Avataren)

Im konkreten Anwendungsfall ist die gleichzeitige Nutzung mehrerer dieser Punkte möglich. Je nach Aufgabe wird eine unterschiedliche Anzahl von Sensoren benötigt. Auch die geforderte Genauigkeit der gelieferten Daten kann variieren. Für die Sichtkontrolle beispielsweise reicht ein Sensor zur Erfassung von sechs Freiheitsgraden aus. Dieser muss aber eine recht genaue Position liefern. Bei der Avatar-Animation dagegen ist die Genauigkeit der Daten weniger von Bedeutung. Vielmehr ist, je nach gewünschter Übereinstimmung der Animation mit der Original-Bewegung, ein Vielzahl von Messpunkten notwendig. Mehr Messpunkte aber stellen erhöhte Anforderungen an die Nachbearbeitung, um aus den Rohdaten die Positionen zu erhalten. Dies kann ein K.O.-Kriterium für den Einsatz im Echtzeitbetrieb sein.

Darstellungstechnik

Einen nicht unwesentlichen Einfluss auf die Erfordernisse eines einzusetzenden Tracker-Systems hat die Wahl der Technik zur letztendlichen Darstellung der virtuellen Welt.
Man kann hierbei drei Hauptgruppen unterscheiden:

1. Sogenannte HMDs*Head Mounted Displays*, die dem Anwender einen Blick aus der "Ich-Perspektive" präsentieren. Meist geschieht dies durch die bekannten Virtual-Reality-Brillen, welche auf den Kopf gezogen werden und für jedes Auge einen kleinen Bildschirm enthalten. Unter diese Kategorie fallen aber auch handgestützte Systeme wie virtuelle "Camcorder" und Ferngläser. Im weiteren wird nun aber ein fest mit dem Kopf verbundenes System angenommen.

Abbildung 8.5: Head Mounted Display

HMDs stellen hohe Ansprüche an die genaue Erfassung der Kopfausrichtung. Dabei ist nicht die absolute Genauigkeit von Belang – eine Abweichung von einigen Grad nach einer kompletten Drehung ist durchaus tolerierbar und wird vom Benutzer auch nicht als störend empfunden, da er ja nur die computergenerierte Sicht als Referenz besitzt – vielmehr müssen Änderungen genau und schnell genug erfasst werden. Kommt es etwa zu einer spürbaren Latenz bei der Erfassung der Bewegung, so scheint die virtuelle Umgebung zuerst mitzulaufen, um dann erst kurze Zeit später wieder still zu stehen. Dieser Effekt kann zur Simulator-Krankheit führen, die, ähnlich der Seekrankheit, aufgrund der Diskrepanz zwischen den Wahrnehmungen des Auges und des Gleichgewichtsorgans zustande kommt.

Vergleichbare Probleme bei der Positionsbestimmung wirken sich weit weniger schlimm aus, besonders wenn nur weit entfernte Objekte im Sichtfeld liegen. Deren scheinbare

Bewegung im Sichtbereich fällt selbst bei schnelleren Positionsänderungen relativ gering aus.

Ein weiteres Problem ist der bereits angesprochene Jitter, also unregelmäßige Schwankungen im Messwert. Besonders, wenn der Benutzer absolut still steht, kann es zu unangenehmem Bildwackeln kommen, vergleichbar einer handgeführten Videokamera. Je nach Amplitude der Störung kann dies ebenfalls zu Übelkeit führen.

Zusammenfassend kann man sagen, dass HMDs Tracker mit geringer Latenzzeit und wenig Störeinflüssen erfordern.

2. Die andere grosse Anzeigenfamilie für VE sind die FSDs*Fixed Surface Displays*. Diese beinhalten alle fest installierten Anzeigen, von virtuellen Arbeitstischen über Projektionswände bis zum komplett umschliessenden CAVE.

Abbildung 8.6: Fixed Surface Display - CAVE

FSDs stellen weit geringere Anforderungen an das Tracker-System als HMDs. Dies liegt hauptsächlich daran, dass die Ausrichtung des Kopfes keinerlei Einfluss auf die angezeigte Szene hat. Alle einsehbaren Blickwinkel sind bereits im dargestellten Bild enthalten. Bei Drehungen des Kopfes ist also keine Anpassung nötig, weshalb verzögerte Daten auch keinen Einfluss auf die Betrachtung haben. Die letzte Aussage stimmt insofern nur eingeschränkt, da sie nur bei zweidimensionalen Projektionen gilt und das oft verwendete Stereo-Rendering außer Acht lässt. Um beiden Augen korrekte Stereobilder präsentieren zu können, ist die Kopfdrehung sehr wohl relevant. Latenzen wirken sich hier allerdings bei weitem nicht so schlimm aus, wie im Fall der HMDs. Im Normalfall kommt es höchstens kurzzeitig zu einer Aufspaltung beider Perspektiven, was aber weit weniger unangenehm auffällt als das Nachziehen der Szene.

Bei der Positionsbestimmung ergibt sich ein geringfügig anderes Bild, gegenüber HMDs. Bei Bewegungen des Kopfes muss die dargestellte Szene aktualisiert werden, bei geringen Veränderungen fällt aber eine gewisse Latenz nicht unangenehm auf. Genau entgegengesetzt den HMDs, sind diese Abweichungen bei FMDs besonders für weit entfernte Objekte eher bemerkbar. Diese scheinen sich gegenüber dem Betrachter in die entgegengesetzte Richtung seiner Bewegung zu verschieben, da die Projektion eigentlich mit dem Benutzer mitlaufen müsste, um die Illusion zu erhalten.

3. Die letzte verbreitete Darstellungsform ist eng verwandt mit den HMDs. Aber anstatt dem Anwender ausschließlich die virtuelle Umgebung zu zeigen, kommen hier durchlässige Anzeigen zum Einsatz, bei denen der Bildinhalt die normale Sicht teilweise überlagert. So können reale und virtuelle Objekte gleichzeitig gesehen werden. Diese Anzeigenform

findet Einsatz in Augmented-Reality-Systemen, um etwa Zusatzinformationen anzuzeigen oder geplante Arbeiten direkt am späteren Einsatzort zu visualisieren.

Abbildung 8.7: Augmented-Reality-System

Verständlicherweise stellt dies enorme Anforderungen an die Genauigkeit und Störungsfreiheit der erfassten Positionsdaten. Anders als bei HMDs hat der Benutzer reale Objekte als Bezugspunkte, so dass geringste Abweichungen vom Soll auffallen. Somit ist neben der Latenz- und Jitter-Problematik als weitere Fehlerquelle die absolute Genauigkeit hinzugekommen.

Andererseits kann es nicht so leicht zur Simulator-Krankheit kommen, da auch bei wackelnden virtuellen Objekten die real vorhandenen Objekte in Ruhe bleiben und somit Gleichgewicht und optische Reize in Einklang stehen.

8.2.2 Technologien

Nachdem jetzt eine qualitative Einstufung möglich ist, folgt nun ein Überblick der unterschiedlichen im Einsatz befindlichen Verfahren.
Bei der Bewegungs-Verfolgung kommt ein breites Spektrum an verschiedensten Messverfahren zum Einsatz. Um noch einmal auf das Beispiel der Computer-Maus zurückzukommen: selbst bei diesem simplen Eingabegerät kommen verschiedene Systeme zum Einsatz. Neben dem klassischen Ansatz einer mechanisch/optischen Abtastung mit Kugel und Fotodioden, kommen immer mehr rein optische Mäuse zum Einsatz. Wenn schon hier zwei Varianten verwendet werden, verwundert es nicht, dass die Anzahl beim Human-Motion-Tracking noch weitaus größer ist. Von akustischen, mechanischen bis zu optischen Verfahren ist hier weitgehend jede messbare physikalische Eigenschaft vertreten.

Elektomechanische

Ein offensichtlicher, aber nur noch wenig genutzter Ansatz ist die Verwendung von elekromechanischer Messung. Die nötigen Bauteile sind gebräuchlich und vergleichsweise billig zu haben. So kommen Potentiometer, Gestänge, Zugseile, Dehnungswiderstände und Glasfasern zum Einsatz. Es werden z. B. Kombination aus Zugseilen an Punkten im Raum und am Körper des Anwenders angebracht. Über die Länge und den Winkel ist eine Positionsbestimmung

möglich. Eine andere Variante ist die Verwendung eine futuristisch anmutenden Exoskeletts. Diese Konstruktion aus Stangen, Potentiometern und Gelenken erlaubt die direkte Messung von Winkeln und Ausdehnungen. Das größte Manko besteht im Fehlen eines Referenzpunkts. Alle Bewegungen können zwar exakt reproduziert werden, aber es kann bspw. nicht bestimmt werden, wie weit entfernt eine Person nach einem Sprung landet.

Abbildung 8.8: Elektromechanisches Exoskellet

Die Vorteile der mechanischen Verfahren sind ihre verhältnismässig geringen Kosten, ihre hohe Genauigkeit und ihre Fähigkeit zur Echtzeiterfassung. Demgegenüber stehen die eingeschränkte Beweglichkeit des Akteurs, die aufwändigen Anpassungen und Kalibrierungen an einzelne Benutzer und die mangelnde absolute Positionierung bei ortsunabhängigen Systemen bzw. die mangelnde Portabilität bei ortsfesten.

Trägheitsbasierte

Ebenfalls eher an die klassische Physik angelehnt sind trägheitsbasierte Systeme. Man misst die auf einen Körper einwirkenden Beschleunigungen und Änderungen des Drehimpulses. Aus diesen und der bekannten Ausgangslage kann, wie schon in der Einleitung erläutert, die aktuelle Position ermittelt werden.

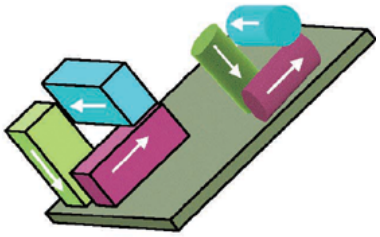


Abbildung 8.9: Kombination von Beschleunigungsmesser und Gyroskopen

Sieht man sich die Vorteile solcher Systeme an, so liest sich diese Aufzählung beinahe wie diejenige des idealen Trackers. Beschleunigungsmesser und Gyroskope zur Ermittlung der Rotationsbeschleunigung sind in Chipform erhältlich. Dies trifft zwar bisher nur auf die Einzelsensoren zu, aber es ist nur eine Frage der Zeit, bis sechs solcher Sensoren auf einem einzelnen

Chip Platz finden. Sie sind störunanfällig gegen äußere Einflüsse, benötigen keine Sichtlinie und sind komplett autonom einsetzbar. Die Latenzzeit ist sehr gering und liegt im Bereich weniger Millisekunden, außerdem ist eine hohe Abtastrate möglich. Die als Dreingabe erhaltenen Geschwindigkeiten können für die Vorhersage der Bewegung genutzt werden. Nicht zuletzt ist der Jitter-Anteil des Signals sehr gering.
Was verhindert nun den Einsatz dieser Technik als Allzweck-Tracker? Ebenfalls in der Einleitung dieser Arbeit kam die Problematik der Positionsbestimmung durch Beschleunigungswerte zur Sprache. Und eben dies trifft auf das Trägheitsverfahren zu. Dieses als Drift bezeichnete Phänomen, rührt von winzigen Messabweichungen her. Kommerziell hergestellte Sensoren der geforderten Größe – es gibt auch sehr viel größere Varianten, die z. B. bei der Schiffsnavigation benutzt werden – besitzen im besten Fall eine Abweichung von etwa einem Milli-g. Also gerade einem Tausenstel der Erdbeschleunigung. Dieser Fehler von nur $0{,}00981\frac{m}{s^2}$ kann nach 30 Sekunden bereits zu einer Positionsabweichung von 4,5 Metern führen.
Alleine sind Trägheitsmesser also nicht einsetzbar. Werden sie aber mit einem Verfahren gekoppelt, das ihn kurzen, regelmäßigen Abständen eine genaue Positionsbestimmung ermöglicht, kann eine erstaunliche Genauigkeit erreicht werden.

Akustische

Auch mit Hilfe von Schallwellen kann eine Positionsbestimmung erfolgen. Normalerweise werden hierbei Ultraschall-Impulse verwendet. Durch entsprechende Positionierung der Lautsprecher und Mikrofone ist anhand der Laufzeit der Impulse eine Entfernungsmessung möglich. Werden mehrere Geräuschquellen oder Mikrofone verwendet, so kann durch Triangulation die Position errechnet werden.
Das Verfahren hat mit dem Auftreten von Schall-Reflektionen zu kämpfen. Bei einem einzelnen Impuls spielt dies noch keine Rolle, da der erste eintreffende Impuls den kürzesten Weg hatte und demnach auf direktem Weg ankam. Nun muss mit dem nächsten Impuls aber solange gewartet werden, bis alle Echos abgeklungen sind, da sonst die Messung verfälscht wird. Dies führt zu einer geringen Abtastrate. Die Echos verhindern auch den Einsatz von Systemen mit kontinuierlichen Schallwellen. Da sich die Impulse nur mit Schallgeschwindigkeit ausbreiten, kommt es außerdem zu einer nicht unerheblichen Latenzzeit bei der Messung. Weitere Einschränkungen sind die Beeinträchtigung durch Umgebungsgeräusche, Luftdruck und Temperatur. Nicht zuletzt verliert das Verfahren mit wachsender Entfernung an Genauigkeit. Einzige positive Eigenschaft ist der unschlagbar niedrige Preis der akustischen Systeme.

Magnetische

Sensoren für magnetische Felder sind klein und um uns herum ist quasi frei Haus das Erdmagnetfeld vorhanden. Es gibt tatsächlich Systeme, die sich dies zu Nutze machen. Nur ist das Magnetfeld der Erde alles andere als homogen. Es ist durchzogen von ortsabhängigen Anomalien, die eine genaue Messung erschweren. Aber durch entsprechende Tabellen ist ein Ausgleich möglich. Es besteht ebenso die Möglichkeit, ein künstliches Magnetfeld zu erzeugen und dieses als Referenz zu benutzen. Aber auch dieses wird von anderen Magnetfeldern, Metallen in der Umgebung und sogar elektrischen Geräten verzerrt. All dies macht eine aufwendige Kalibrierung nötig. Außerdem vermindert die hierdurch nötige Filterung die Abtastrate, die aber dennoch bis zu 140 Hertz betragen kann und somit für die meisten Anwendung völlig ausreicht.

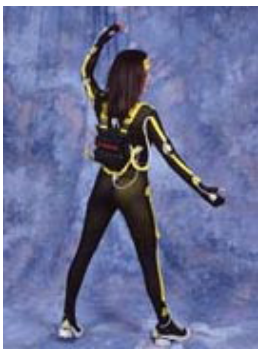


Abbildung 8.10: Magnetische Ganzkörpererfassung

Künstliche magnetische Felder besitzen eine äußerst begrenzte Ausdehnung, da die Feldstärke umgekehrt proportional zur vierten Potenz des Abstands abnimmt. Eine exakte Messung ist deshalb nur auf engem Raum von bis zu 10 Metern technisch realisierbar. Bei einigen Systemen erfolgt die Übertragung der Sensordaten über Kabel. Da diese abgeschirmt sein müssen, um das Magnetfeld nicht zu stören, sind sie recht sperrig und schränken die Bewegungsfreiheit weiter ein. Aber die meisten modernen Systeme verwenden Funkübertragung zur Datenübermittlung und behindern somit die Beweglichkeit nur minimal.
Warum aber gehören magnetische Tracker, trotz dieser Nachteile zu den meistgenutzen Systemen im Bereich des Motion Capturing? Der erste Grund wurde bereits genannt: die Sensoren sind klein und wenig hinderlich. Die Kosten der Systeme sind im Vergleich relativ niedrig. Die grössten Vorteile ergeben sich aber aus der Natur eines Magnetfelds: es kann den menschlichen Körper ohne Schwierigkeiten durchdringen, weshalb auch kein Sichtlinienproblem existiert. Solange sich die Sensoren nicht zu nahe kommen und gegenseitig stören, kann eine hohe Genauigkeit erreicht werden und es ist durchaus möglich mehrere Dutzend zugleich einzusetzen. Anders als beispielsweise bei den folgenden optischen Systemen, können die Daten eindeutig einem Sensor zugeordnet werden, was die Nachbearbeitung deutlich vereinfacht.

Optische

Den von allen Verfahren weitestgehend natürlichsten Weg, in Hinblick auf die menschliche Sinneswahrnehmung, beschreiten die optischen Systeme. Sie werten Bilder aus, die im sichtbaren oder zumindest infraroten Spektrum aufgenommen werden.
Man muss hierbei im Wesentlichen zwei Hauptgruppierungen unterscheiden:

- markerbasierte Verfahren
- markerlose Verfahren

Bei markerbasierten Verfahren werden am Körper der zu verfolgenden Person Markierungen angebracht, die mit einfachen Mitteln vom restlichen Bildmaterial geschieden und separat weiterverarbeitet werden können. Die markerlosen Verfahren dagegen werten das Bild anhand von nicht so leicht zu extrahierenden natürlichen oder künstlichen Merkmalen aus.

Markerbasierte

Bei den verwendeten Marken handelt es sich meist um hochreflektive also passive Kugeln oder aber aktive Lichtquellen in Form von LEDs. Oft wird infrarotes Licht verwendet, da hierbei

die hell leuchtenden Markierungen durch einfache Schwellwert-Trennung aus dem Ausgangsmaterial heraus getrennt werden können.

Abbildung 8.11: Markerbasiertes optisches Verfahren

Das Hauptproblem bei markerbasierten Systemen ist die eindeutige Zuordnung der gefunden Marken zu ihren virtuellen Pendants. Durch Verdeckung kann eine Marke vollständig aus dem Sichtfeld der Kameras verschwinden. Die nötigen Berechnungen führen dazu, dass Echtzeitbetrieb nur mit wenigen Marken machbar ist. Eine Möglichkeit, die Zuordung zu erleichtern, ist der Einsatz aktiver Lichtquellen, von denen immer nur eine zu einem Zeitpunkt aktiv ist. Dadurch sinkt aber, bei einer grossen Anzahl von Markierungen, die Abtastrate erheblich.
Desweiteren sind oft aufwendige Kalibrierungen nötig und die Systeme gehören zu den teuersten Motion-Trackern. Außerdem ist das System anfällig für störende Beleuchtungs-Einflüsse und unerwünschte Reflektionen der Marken. Dies kann man durch eine spezielle Aufnahmeumgebung umgehen, in der wohldefinierte Bedingungen herrschen.
Vorteile sind die sehr große Genauigkeit bis in den Millimeterbereich, die hohe Abtastrate, eine weitgehend uneingeschränkte Bewegungsfähigkeit der Akteure und das große mögliche Arbeitsvolumen.

Markerlos

Die Königsdiziplin der optischen Systeme sind diejenigen, die ohne Marken auskommen. Manche von ihnen benötigen wie die Markenbasierten spezielle Studiobedingungen, wie etwa einen schwarzen Hintergrund. Es gibt aber auch unzählige Varianten, die mit normalen Licht- und Umgebungsbedingungen zurechtkommen. Die Vielzahl der verwendeten Ansätze ist beinahe unüberschaubar. Die Merkmalsextraktion kann auf der Erkennung von Hautfarben, Formen, Kanten und Silhouetten basieren, um nur einige zu nennen.
Aber nicht immer sind nur Kameras auf die zu verfolgende Person gerichtet (der Outside-in-Ansatz), sondern es wird auch der umgekehrte Ansatz (Inside-out) beschritten. Hierbei trägt der Anwender eine oder mehrere Kameras, die anhand der Umgebung die aktuelle Position bestimmen. Dieser Ansatz wird vor allem bei Augmented-Reality-Anwendungen genutzt um größere Gebiete, wie Fabrikationsanlagen, abdecken zu können.
Eine eindeutige Einstufung fällt hier aufgrund der großen Bandbreite verschiedener Ansätze sehr schwer, da auch hier jeder Ansatz seine spezifischen Stärken und Schwächen aufweist.

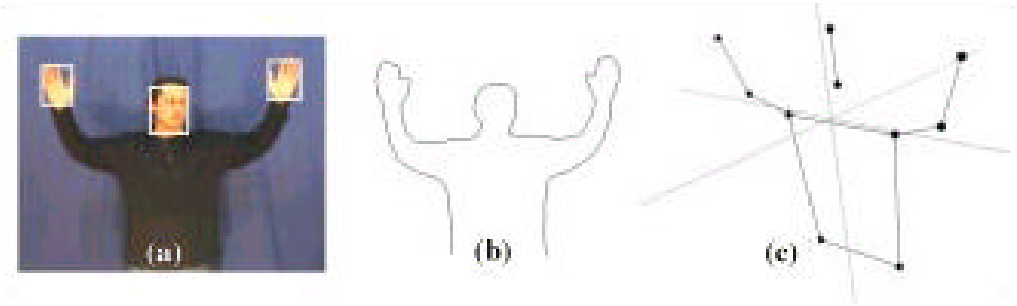


Abbildung 8.12: Markerloses optisches Verfahren

Weiter unten werden deshalb zwei derartige Verfahren genauer beschrieben, um einen Einblick in die Arbeitsweise zu vermitteln.

Weitere Verfahren

Neben den bereits aufgeführten Varianten können auch Radio- und Mikrowellen zur Bewegungs-Verfolgung eingesetzt werden. Das zugrunde liegende Prinzip ist ähnlich den akustischen Systemen, nur haben die elektromagnetischen Wellen den Vorteil einer ungleich höheren Ausbreitungsgeschwindigkeit und somit niedrigerer Latenzzeiten. Dennoch kommen sie sehr selten zum Einsatz.

8.3 Detailbeschreibung ausgewählter Verfahren

8.3.1 Markerbasiertes optisches Verfahren

Die Grundlagen von markerbasierten Verfahren wurden bereits im vorangehenden Abschnitt aufgeführt. Hier soll nun eine etwas detailliertere Beschreibung der weiteren Arbeitsschritte folgen. Dies geschieht am Beispiel von
[Ribo et al. '01].
Die Verwendung von Markern, die sich deutlich vom Hintergrund abheben, erlaubt den Einsatz einer einfachen Schwellwert-Trennung. Dies kann durch Einsatz von Infrarotlicht noch erleichtert werden. Bei passiven Marken erfolgt die Beleuchtung meist durch Lichtquellen, die um die Kamera herum positioniert sind. So wird eine optimale Ausleuchtung und maximale Reflektion des Lichts an den Marken gewährleistet. Das vorgestellte Verfahren arbeitet mit zwei Kameras und passiven, kugelförmigen Markierungen.

Abbildung 8.13: Kalibriertes Stereokamera-System

Die Kenntnis der optischen Eigenschaften beider Kameras und die exakte Anordnung erlauben später eine genaue Rekonstruktion der dreidimensionalen Markerpositionen anhand des vorliegenden Stereobildes.

Zunächst aber müssen in den schwellwertgefilterten Bildern die Mittelpunkte und Ausdehnung der Marker gefunden werden. Es ist sicherzustellen, dass keine Reflektionen als Marker erkannt werden, da dies zu Fehlinterpretationen führen kann. Dazu werden die gefundenen Objekte daraufhin überprüft, ob sie ellipsoide Gestalt besitzen.
Nun hat man die Position und Grösse der in beiden Bildern vorkommenden Marken. Im nun folgenden Schritt werden mit Hilfe dieser Informationen die Positionen der Marken im Raum berechnet.

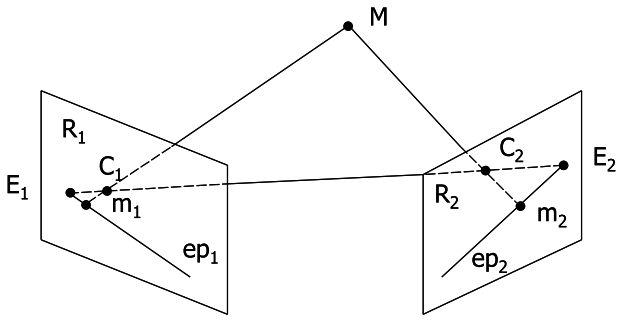


Abbildung 8.14: Extrinsische Kameraparameter

Wie gesagt, ist die exakte Ausrichtung der Kameras für eine Positionsbestimmung unerlässlich. Insbesondere die, durch die Kameraausrichtung bestimmten, extrinsischen Parameter sind hierfür wichtig. Die Gerade durch die beiden optischen Zentren wird hierzu mit den Bildebenen geschnitten. Man erhält hiermit zwei Punkte, die Epipole, je einen für jede Bildebene. Spannt man mit Hilfe der Zentren und eines beobachteten Punkts eine Ebene auf, so schneidet diese die beiden Bildebenen in je einer Gerade, die durch den Epipol geht. Die Berechnung dieser Geraden, der Epipolaren, ist bei Kenntnis des Epipols auch möglich, wenn nur die Position im anderen Bild bekannt ist. Geht man nun von der Position eines beliebigen Markers im linken Bild aus, so können die möglichen Positionen dieses Markers im rechten Bild also auf eine Linie beschränkt werden. Dies Beschränkung ermöglicht die schnelle Suche nach den zusammengehörigen Paaren von Marker-Abbildern. Abschließend muss nur die Position im Raum durch Triangulation errechnet werden.

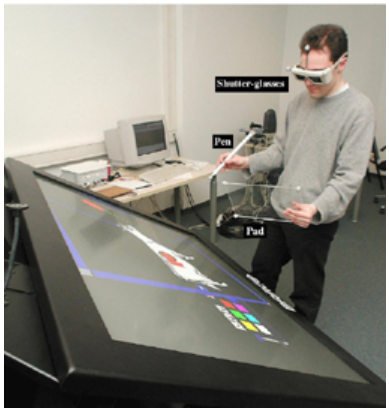


Abbildung 8.15: Verwendung in Verbindung mit virtuellem Tisch

Das vorgestellte Verfahren erlaubt die Verfolgung von bis zu 25 Marken mit 30 Hertz. Diese Zahl und Abtastrate reicht schon für die meisten VE-Anwendungen aus.

Die markerbasierten optischen Systeme für Ganzkörper-Tracking arbeiten nach genau diesem Prinzip. Aufgrund der meist deutlich größeren Anzahl von Marken und Kameras sowie der gewünschten hohen Abtastraten sind diese allerdings noch nicht in Echtzeit realisierbar.

8.3.2 Gesichtsverfolgung

Das menschliche Gehirn besitzt die erstaunliche Fähigkeit, aus den vom Auge gelieferten optischen Reizen Gesichter oder gesichtsähnliche Figuren herauszufiltern und unsere Aufmerksamkeit auf diese zu lenken. Dies mag an einem aus Urzeiten stammenden Instinkt liegen, um auf Gefahr in Form eines Tiers unverzüglich reagieren zu können, aber es spielt wohl auch eine Rolle, dass die Gesichtsmimik einen bedeutenden Anteil an der zwischenmenschlichen Kommunikation ausmacht. Gerade der zweite Aspekt bietet einen interessanten Ansatz für eine verbesserte Mensch-Computer-Interaktion, und dies nicht nur im Bereich virtuelle Umgebungen. Doch um dies zu erreichen, muss der Rechner erst die Fähigkeit erlangen, Gesichter zu erkennen und ihre Bewegungen zu verfolgen.
Der hier vorgestellte Ansatz stammt aus [Bradski '98] und [Nickel '02] und stützt sich vorwiegend auf die Identifikation mittels Farbinformationen.
Als Ausgangsbasis dienen Histogramme, die aus Bildern von Haut gewonnen wurden. Histogramme geben die Häufigkeiten aller Farbwerte im Bild wieder. Der normalerweise bei Kameras verwendete RGB-Farbraum, in dem Farben durch additive Mischung von Rot, Grün und Blau entstehen, ist wenig geeignet für die saubere Erkennung von Haut. Dies rührt daher, dass die Farbwerte der Haut ungleichmäßig im Farbraum verteilt sind. Deshalb kommen andere Farbräume zum Einsatz. Sowohl der chromatische als auch der HSV-Farbraum werden verwendet. Bei beiden können die störenden Helligkeitswerte, die durch die Beleuchtung entstehen, eliminiert werden. Als willkommenen Nebeneffekt können mit Hilfe der so enstandenen Histogramme nicht nur Menschen der Hautfarbe erkannt werden, welche die Vorlage aufweist. Die reinen Farbkomponenten der Hautfarbe aller Menschen stimmen überein. Nur in Helligkeit und Sättigung gibt es offensichtliche Unterschiede.

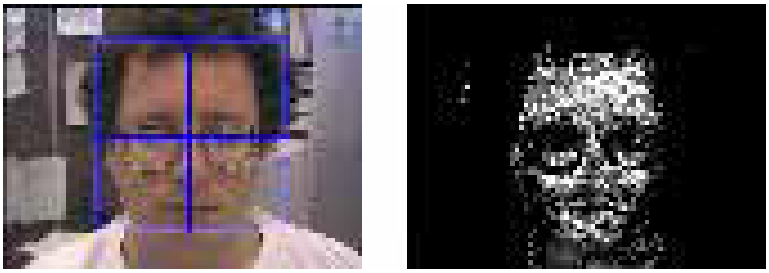
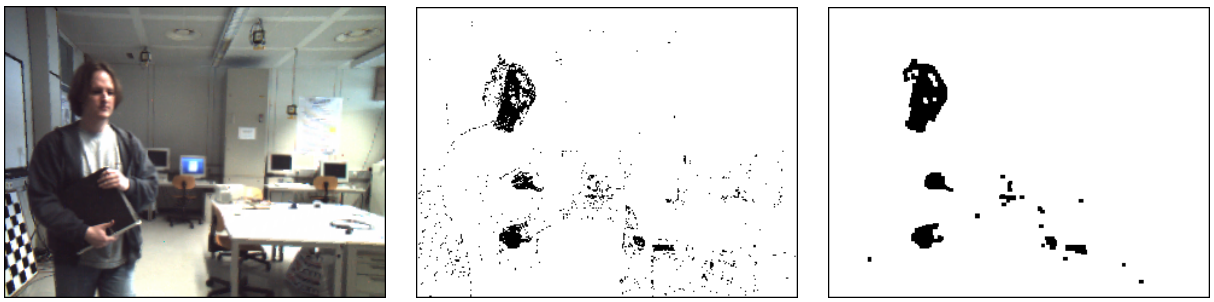


Abbildung 8.16: Wahrscheinlichkeiten für Hautfarbe

Anhand eines gegebenen Histogramms kann man nun in einem Bild die Wahrscheinlichkeiten der einzelnen Bildpunkte berechnen, mit der diese Haut darstellen. Durch Einsatz eines Schwellwert-Verfahrens und einiger Filteroperationen, die der Eliminierung von Streupixeln dienen, erhält man eine Maske, in der alle mit hoher Wahrscheinlichkeit hautfarbenen Bereiche vorhanden sind. An diesem Punkt können zusammenhängende Bereiche erkannt und der jeweilige Mittelpunkt und die Ausdehnung ermittelt werden. Allerdings ist damit noch nicht sichergestellt, dass alle gefundenen Bereich auch wirklich Gesichter darstellen.
Durch die Verwendung einer Stereokamera ist es möglich, noch weitere Informationen zu erhalten. Es kann mit Hilfe der Texturübereinstimmung ausreichend strukturierter Flächen in



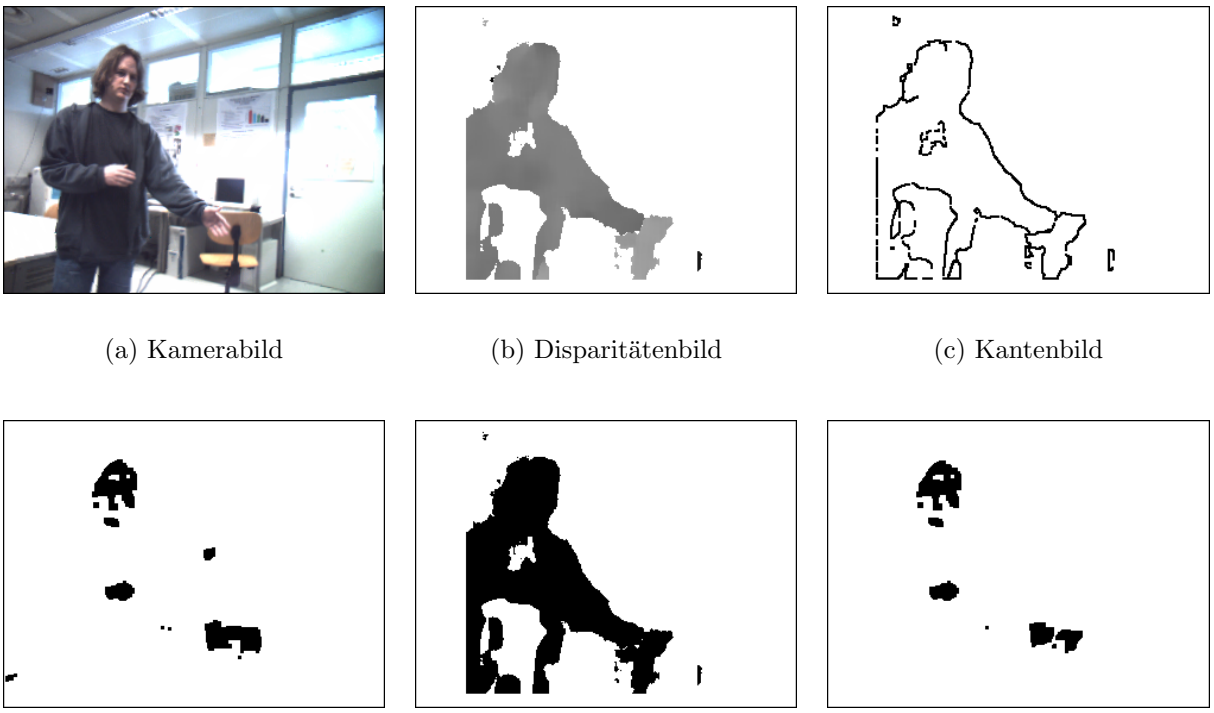
(a) Ursprungsbild (b) Hautfarbmaske (c) Gefilterte Hautfarbmaske

Abbildung 8.17: Erstellung der Hautfarbmaske

(a) Linkes Kamerabild (b) Rechtes Kamerabild (c) Disparitätenbild

Abbildung 8.18: Tiefeninformation

beiden Kamerabildern die Tiefeninformation extrahiert werden. Nun kann man hautfarbene Bereiche, die sich überlappen, in der Wahrscheinlichkeitsverteilung dadurch voneinander trennen, dass man Kanten im Tiefenbild identifiziert und an den entsprechenden Stellen die Wahrscheinlichkeit auf Null setzt.



(a) Kamerabild (b) Disparitätenbild (c) Kantenbild

(d) Hautfarbmaske (e) Tiefenbereichsmaske (f) Kombinierte Maske

Abbildung 8.19: Gesamtablauf

Außerdem kann unter Kenntnis der Entfernung eines Bereichs auf dessen reale Fläche geschlossen werden. Anhand dieses und anderer Merkmale, wie Proportionen und Form, können nun mit Hilfe einer Klassifikation die letztendlich als Gesichter erkannten Bereiche herausgefiltert werden.

Besitzt man nun diese Informationen, können in diesen Gebieten weitere differenzierte Beurteilungen, beispielsweise der Mimik, durchgeführt werden.

8.3.3 Weiträumiges Augmented Reality

Der Einsatz eines Augmented-Reality-Systems stellt hohe Anforderungen an die verwendete Technologie der Bewegungs-Verfolgung. Dies ist besonders der Fall, wenn ein großes Gebiet, etwa ein ganzes Gebäude oder gar Gelände, mit stark variierenden Umgebungsbedingungen erfasst werden muss. Das nun vorgestellte System aus [Naimark & Foxlin '02] erreicht dies durch den Einsatz von künstlichen Vermessungspunkten, die aber kostengünstig herzustellen und nahezu wartungsfrei sind.

Abbildung 8.20: Weiträumige Anwendung von Augmented Reality

Der Tracker besteht im wesentlichen aus einer Schwarz-Weiß-Kamera mit angeschlossenem Rechnersystem, welche vom Benutzer getragen wird und den im Gebäude verteilten Messpunkten. Diese Punkte sind im Wesentlichen Kreisscheiben aus Papier, die mit normalen Druckern erstellt werden können.

Abbildung 8.21: Struktur der Marker

Die Scheiben weisen eine charakteristische Struktur auf, die es dem Tracker ermöglicht, sie zu lokalisieren und eindeutig zu identifizieren. Diese Struktur besteht aus vier konzentrischen Ringen: außen und innen jeweils zwei komplett schwarze Ringe und dazwischen zwei weitere Ringe, die in einer binären Codierung sektorenweise eingefärbt werden. Außerdem existiert eine Verbindungssegment zwischen äußerem und innerem Ring, so dass unabhängig von der Ringcodierung nur eine einzige, komplett verbundene schwarze Fläche vorkommt. Dies erleichtert die spätere Separation der Scheiben vom Hintergrund. Und es gibt noch zwei kleine schwarze Punkte, die mit dem Mittelpunkt einen rechten Winkel bilden, mit deren Hilfe die Orientierung der Scheibe erkennbar ist und einen kleinen weißen Punkt im Mittelpunkt.
Um die Scheiben in den Ausgangsbildern der Kamera zu finden, wird zunächst eine Methode angewandt, um die störenden Einflüsse ungleichmäßiger Beleuchtung zu eliminieren. Dies ist notwendig, da sonst, unter ungünstigen Umständen, die hellen Bereiche einer Marke im

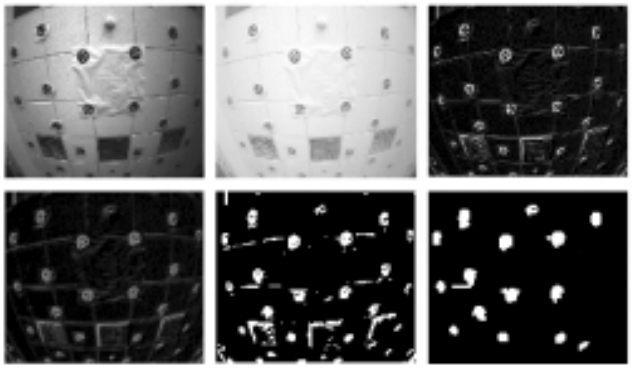


Abbildung 8.22: Erkennung der Marker im Kamerabild

einen Teil des Bildes dunkler sind als der dunkle Teil einer Marke in einem anderen Teil. Erreicht wird dies durch die Anwendung eines Hochpass-Filters, der die üblicherweise fließenden Lichtwechsel im tiefen Frequenzbereich entfernt. Nach anschließender Kontrastverstärkung, Kantendetektion und Entfernung kleiner Störpixel werden aus den übriggebliebenen Kandidaten die besten vier ermittelt. Dies geschieht anhand der Eigenschaften Grösse, Form und Struktur.

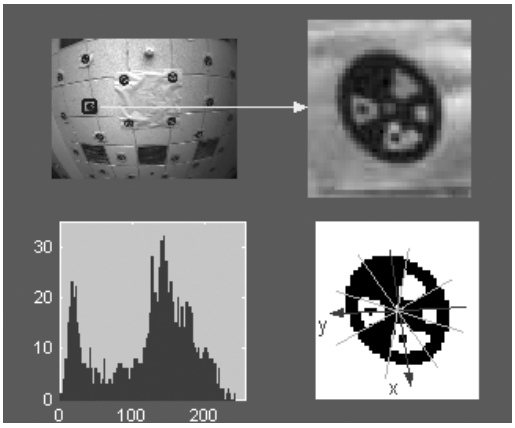


Abbildung 8.23: Auslesen der Markerkodierung

Die so gefundenen Scheiben werden genauer untersucht. Anhand der kleinen Punkte kann die Orientierung der Scheibe ermittelt und die Codierung ausgelesen werden. Mit dieser Codierung ist es nun möglich, die exakte Position der ermittelten Scheiben aus einer Datenbank abzufragen und die Position des Trackers zu bestimmen.

8.4 Ausblick

Wie wird sich das Gebiet der Bewegungs-Verfolgung beim Menschen in Zukunft entwickeln? Die Antwort auf diese Frage kann, wie auf den meisten anderen Gebieten auch, nur eine ungefähre Prognose sein.
Mit ziemlicher Wahrscheinlichkeit wird auch weiterhin eine Vielzahl von verschiedenen Verfahren zum Einsatz kommen, je nach Anwendung, Vorliebe und Budget der Nutzer. Als ebenso sicher kann gelten, dass sich die Tracker in gewissen Bereichen dem idealen Tracker annähern werden: durch Fortschritte bei der Sensorik, der Rechenleistung und den verwendeten

Algorithmen werden die Systeme kleiner, genauer und vor allem billiger. Besonders der letzte Punkt wird dazu führen, dass immer mehr Personen sich den Einsatz von Motion Tracking leisten können. Dies wiederum führt zu einem weiteren Preisverfall und der Exot Bewegungs-Verfolgung wird über kurz oder lang zum Massenprodukt.
Die Frage des vorherrschenden Verfahrens ist weniger leicht zu beantworten. Es ist anzunehmen, dass besonders im Bereich der markerlosen optischen Systeme die eigentliche Revolution erst noch bevorsteht. In keinem anderen Bereich des Motion Tracking wird momentan intensiver geforscht. Aber auch hier wird eine ebenso unüberschaubare Anzahl verschiedener Ansätze entstehen, wie sie im Moment durch die verschiedenen physikalischen Grundlagen gegeben sind.
Aber bis diese Verfahren zur Marktreife kommen, werden wohl weiterhin die magnetischen und markerbasierten optischen Verfahren die bevorzugte Wahl bei kommerziellen Anwendungen sein.

Literaturverzeichnis

[Boyle '] M. Boyle. The Effects of Capture Conditions of the CAMSHIFT Face Tracker. *University of Calgary.*

[Bradski '98] G. Bradski. Computer Vision Face Tracking For Use in a Perceptual User Interface. *Intel Technology Journal*, Q2'98, 1998.

[Foxlin '02] E. Foxlin. Motion Tracking Requirements and Technologies. *InterSense Inc.*, 2002.

[Horber '] E. Horber. Motion Capturing. *Universität Ulm.*

[Naimark & Foxlin '02] L. Naimark und E. Foxlin. Circular Data Matrix Fiducial System and Robust Image Processing for a Wearable Vision-Inertial Self-Tracker. *IEEE International Symposium on Mixed and Augmented Reality (ISMAR 2002), Darmstadt, Germany*, 2002.

[Nickel '02] K. Nickel. 3D-Tracking von Gesicht und Händen mittels Farb- und Tiefeninformationen. *Universität Karlsruhe*, 2002.

[Ribo et al. '01] M. Ribo, A. Pinz und A. Fuhrmann. A new Optical Tracking System for Virtual and Augmented Reality Applications. *IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May 21-21, 2001*, 2001.

[Ulbricht '] C. Ulbricht. Human Motion Capture. *Seminar aus Informatik.*

[Votruba '] P. Votruba. Ablauf eines Motion Capture-Prozesses. *Seminar aus Informatik.*

[Welch & Foxlin '02] G. Welch und E. Foxlin. Motion Tracking: No Silver Bullet, but a Respectable Arsenal. *Motion Tracking Survey*, S. 24–38, 2002.

9 Programmierbare Graphikhardware

Stephan Bergmann

9.1 Einleitung

PC-Grafikkarten haben in den letzten Jahren gewaltige Entwicklungsfortschritte gemacht. Angetrieben nicht zuletzt von der Spieleindustrie, die immer schnellere und flexiblere Grafikhardware fordert, existieren nun - angefangen mit der Geforce3 - Grafikkarten, die programmierbare Einheiten zur Geometrieberechnung und Texturierung enthalten. Heute gibt es, außer im Tiefstpreissegment kaum noch eine Grafikkarte zu kaufen, die nicht über eine mehr oder weniger fortgeschrittene Form dieser Einheiten verfügt. Mit der breiten Unterstützung von DirectX9 seitens der Grafikkartenhersteller sind nun Grafikkarten verfügbar, deren programmierbare Einheiten weit mehr bieten als die ersten Grafikkarten dieser Art. Solche Grafikkarten scheinen, das Ziel fotorealistischer Bilder und kinoreifer Effekte in Echtzeit in greifbare Nähe zu rücken.
Im Folgenden werden nun die programmierbaren Einheiten moderner Grafikhardware vorgestellt. Ein Schwerpunkt liegt dabei auf der Programmierung dieser Einheiten mit Hilfe verschiedenen Schnittstellen und Programmiersprachen.

9.2 Verwendete Abkürzungen

API	Application Programming Interface
DX	Microsoft DirectX
DX8 / DX9	Microsoft DirectX 8.x bzw. 9.x
PS	Pixel Shader
VS	Vertex Shader
Cg	C for graphics
HLSL	High Level Shading Language
ARB	Architecture Review Board
	(Standardisierendes Gremium für OpenGL)
GPU	Graphics Processing Unit (Grafikprozessor)

9.3 Rendering Pipelines

Um räumliche Szenen auf einem gewöhnlichen, zweidimensionalen Bildschirm abzubilden, durchlaufen die Ursprungsdaten, also vor allem Geometrie und Texturen, eine mehrstufige Pipeline ähnlich der Ausführungspipeline in einem Prozessor. Jede Stufe in dieser Pipeline hat eine bestimmte Aufgabe bzw. mehrere Aufgaben und ist auf diese spezialisiert. Diese Pipeline

ist im wesentlichen bei allen 3D Grafiksystemen, die nach dem Rasterisierungsverfahren arbeiten, identisch. Sie kann in Software oder in Hardware implementiert sein, wobei heutzutage ein Großteil der erforderlichen Berechnungen aus Performancegründen von der Grafikkartenhardware übernommen wird. Die Standardpipeline sieht folgendermaßen aus: Die Geometriedaten gelangen zunächst in die *Transform & Lighting* Stufe, in der die Geometriedaten der Objekte in ein gemeinsames Koordinatensystem transferiert und perspektivisch projeziert werden. In dieser Stufe werden auch alle Berechnungen ausgeführt, die einmal pro Vertex (Stützpunkt) ausgeführt werden müssen, wie z.B. Lichtberechnungen für bestimmte Beleuchtungsverfahren oder die Generierung von Texturkoordinaten.
Die nächste Stufe übernimmt die noch übriggebliebenen Berechnungen für die Geometriedaten, wie z.B. das Aussortieren von nicht sichtbarer Geometrie.
Nach dieser Stufe folgt die Rasterisierung der Grafikdaten, d.h. die Bestimmung welche Polygonoberflächen auf welche Rasterpunkte des zweidimensionalen Ausgabegerätes abgebildet werden.
Die Bestimmung der Farbwerte, also das Überziehen der Polygonoberflächen mit Texturen, übernimmt die nächste Stufe der Pipeline. Die Texturen werden hier gefiltert ausgelesen, evtl. gemischt und je nach gewählten Beleuchtungsmodell auch beleuchtet. Auch Spezialeffekte wie z.B. Reflektionen werden in dieser Stufe berechnet. Die Ausgabe der Texturierungsstufe besteht aus Farb- und Tiefeninformation.
Nachfolgend durchläuft das berechnete *Fragment*[1] noch verschiedene Tests, ob es überhaupt in den Bildspeicher geschrieben wird, allen voran den z-Test (Das Fragment wird nur in den Bildschirmspeicher geschrieben wenn an seiner designierten Position noch kein Fragment mit geringerer Tiefe steht.)
Der letzte Schritt ist dann das Schreiben des Farbwerts (evtl. Alpha-verknüpft) in den Bildschirmspeicher und des Tiefenwertes in den z-Speicher.
Die ersten Consumer-Grafikkarten, die eine 3D-Beschleunigung in Hardware implementierten, ersetzten nur die letzten Stufen der Pipeline durch Hardware; d.h. diese Grafikkarten beschleunigten im wesentlichen das Zeichnen von (texturierten) Polygonen. Die Geometrie- und Beleuchtungsberechnungen (T&L[2]) wurden nach wie vor vom Hauptprozessor übernommen.
Die ersten Consumer-Grafikkarten, die eine T&L-Hardwareeinheit besaßen, waren Karten mit dem Geforce-Chip von NVidia. Dieser Grafikchip (und seine Nachfolger) hatte zusätzlich zu den Textureinheiten noch Geometrieeinheiten, die die notwendigen Transformations- und Beleuchtungsberechnungen übernahmen und eine sog. *fixed function pipeline* implementierten.

Fixed Function Pipeline

Durch die Implementierung in Hardware wurden zwar die Geometrieberechnungen um ein Vielfaches beschleunigt, da aber die Pipeline jetzt in Hardware gegossen war, ging auch ein Teil der Flexibilität verloren, die durch eine reine Softwareimplementierung vorhanden war. Die Pipelines waren zwar konfigurierbar, aber nur im Rahmen, der zum Zeitpunkt ihres Entwurfs vorhandenen bzw. vorhergesehenen Fähigkeiten, das heißt z.B., dass man nur noch die in der Grafikkarte implementierten Beleuchtungsmodelle verwenden konnte, wollte man den Vorteil der Hardware T & L Stufe in Anspruch nehmen. Im gleichen Maße wie die Fähigkeiten der Grafikkarten erweitert wurden, wuchs auch der Aufwand, diese zu konfigurieren. Die Grafik-APIs, allen voran die weit verbreiteten APIs DirectX und OpenGL, mußten ständig erweitert werden und die Grafikchiphersteller definierten ihrerseits Erweiterungen, um die Fä-

[1] Ein Fragment ist ein bestimmter Punkt auf der Polygonoberfläche mitsamt seinen Attributen.
[2] Transform & Lighting

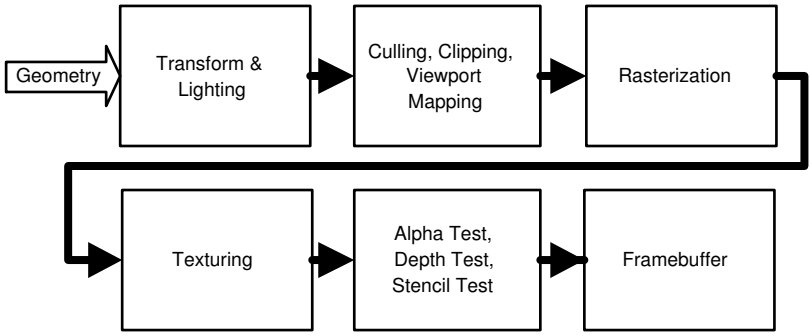


Abbildung 9.1: Fixed-Function pipeline

higkeiten ihrer Chips den Softwareentwicklern zugänglich zu machen. Daraus resultierte ein erhöhter Aufwand für die Softwareentwickler, die nun entweder statt einer mehrere Schnittstellen unterstützen oder auf die erweiterte Funktionalität verzichten mußten.
Um die Konfigurierbarkeit der Hardware besser zugänglich zu machen bzw. zu erweitern, ersetzte man Teile der Grafikpipeline durch mehr oder weniger frei programmierbare Einheiten.

Programmierbare Pipeline

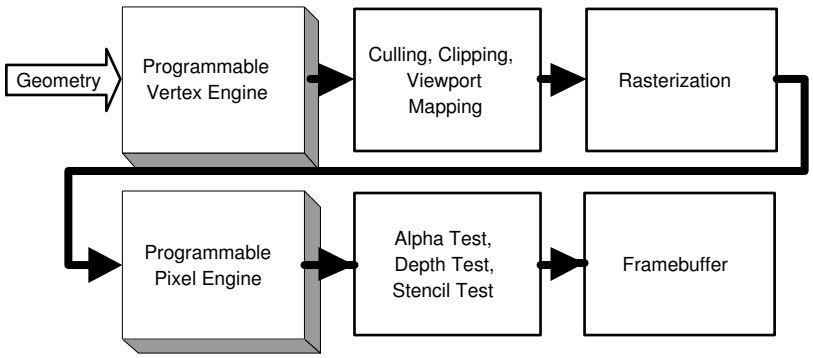


Abbildung 9.2: Programmierbare Einheiten in der Grafik Pipeline

In den heute erhältlichen Grafikkarten werden üblicherweise zwei Stufen der Pipeline durch programmierbare Einheiten ersetzt: Die *Transform & Lighting*-Stufe und die *Texturing*-Stufe. Der Rest der Pipeline ist wie gehabt in Hardware implementiert. Die beiden programmierbaren Einheiten lassen sich üblicherweise unabhängig voneinander auch durch die übliche Fixed-Function pipeline ersetzen, d.h. man muß nicht beide von Hand programmieren, wenn von einer Einheit nur die eingebaute Funktionalität benötigt wird. In beide dieser Einheiten lassen sich mehr oder weniger frei (dazu später mehr) definierte Programme laden, die dann jeweils pro zu verarbeitendem Element einmal ausgeführt werden. Die Programme in der programmierbaren T&L-Einheit, *Vertex-Shader* oder auch *Vertex-Programs* genannt, haben als Ein- und Ausgabe einen Stützpunkt (Vertex) mit Attributen, während die Programme in der Textureinheit, genannt *Pixel-* oder *Fragment-Shader*, manchmal auch *Fragment-Programs*, Attribute eines Fragments (Textur, Texturkoordinaten, Normalen, etc.) erhalten und daraus dessen Farbwert berechnen.

9.4 Vertex Shader

Vertex Shader ersetzen die komplette Transform & Lighting Einheit der Fixed-Function Pipelines und stellen somit eine vorher nicht erreichbare Flexibilität zur Verfügung. Dies bedeutet aber auch, dass der Rest der Pipeline sich drauf verläßt, dass ein Vertex Shader folgende Aufgaben erledigt:

- Transformation der Stützpunkte in das Clipping-Koordinatensystem
- Generierung von Texturkoordinaten (evtl. nur Durchreichen der Daten, die die Applikation liefert)
- Pro-Vertex Lichtberechnungen

Ein Vertex-Shader ist ein Programm, das in der Grafikkarte abläuft und genau einen Stützpunkt als Eingabe nimmt und auch genau diesen Stützpunkt (allerdings transformiert, mit geänderten Attributen, etc.) wieder ausgibt. Ein Stützpunkt kann allerdings mit (fast) beliebig vielen Attributen wie z.B. Farbe, Texturkoordinaten, Normale u.v.a. versehen sein. Das Vertex-Shader Programm wird pro zu verarbeitendem Stützpunkt genau einmal aufgerufen. Ein Vertex-Shader kann allerdings die Anzahl der Stützpunkte nicht verändern, d. h. er kann weder Stützpunkte hinzufügen noch entfernen. Auch Topologieinformationen erhält der Shader nicht, er kann also nicht auf andere außer dem gerade bearbeiteten Stützpunkt zugreifen und hat auch keine Informationen darüber, wie die Stützpunkte zusammenhängen.
Die Ausführungsumgebung von Vertex-Shadern unterscheidet sich nur geringfügig von Grafik-API zu Grafik-API. Im wesentlichen besteht sie aus folgenden Bestandteilen (siehe Abbildung 9.3):

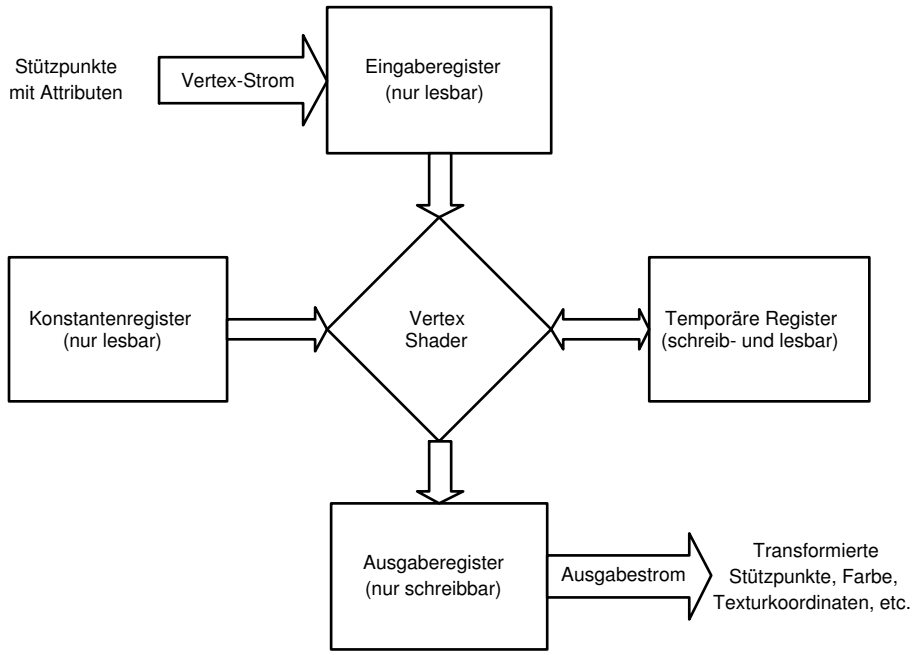


Abbildung 9.3: Die Ausführungsumgebung von Vertex-Shadern

- Eine (gewöhnlich recht kleine) Zahl von nur lesbaren Eingaberegistern, in denen die Attribute des gerade bearbeiteten Stützpunktes stehen

- Ausgaberegister mit festgelegter Semantik, mithilfe derer der Vertex-Shader seinen bearbeiteten Stützpunkt an die folgenden Stufen der Pipeline weiterleitet.
- Nur lesbare Konstantenregister (auch Parameterregister genannt), die von der Applikation gesetzt werden und normalerweise Daten enthalten, die sich nur selten (pro Frame oder Primitivengruppe) verändern (z.B. Transformationsmatrizen)
- Les- und schreibbare temporäre Register, die vom Vertex-Shader Programm beliebig genutzt werden können.
- Addressregister, die gewöhnlich nicht direkt benutzt werden können, sondern zur indirekten Adressierung von Konstantenregistern verwendet werden.

Vertex-Shader wurden erstmals mit DirectX8 in einer weit verbreiteten Grafik-API eingeführt. Die Fähigkeiten und Einschränkungen der Vertex-Shader in den verschiedenen Grafik-APIs sind leicht unterschiedlich und werden im Detail in Abschnitt 9.6 vorgestellt. Vertex-Shader sind zur Zeit noch wesentlich leistungsfähiger als Pixel-Shader, da sie den größeren Befehlssatz besitzen und auch weniger Einschränkungen hinsichtlich Anzahl der Instruktionen und Reihenfolge dieser unterliegen.

9.5 Pixel Shader

Pixel Shader sind das zweite und letzte programmierbare Glied der Rendering Pipeline. Sie ersetzen die *Multitexturing* bzw. *Blending* Stufe früherer Karten und bestimmen die letztendliche Farbe eines Punktes auf einer Polygonoberfläche. Die Farbe des entsprechenden Pixels auf dem Bildschirm kann sich durch Alpha-Blending-, Nebel- und Antialiasingberechnungen, die danach ausgeführt werden, aber noch davon unterscheiden.
Pixel Shader sind (wie schon beschrieben) auch unter dem Namen *Fragment Shader* bzw. *Fragment Program* bekannt. Eigentlich wäre Fragment Shader/Program die passendere Bezeichnung. Ein *Fragment* ist ein Punkt auf einer Polygonoberfläche mit assoziierten Attributen wie (interpolierter) Farbe, Tiefenwert und evtl. mehreren Texturkoordinaten, während ein Pixel der endgültige Farbwert ist, der in den Framebuffer geschrieben wird und der evtl. mit mehreren Fragmenten korrespondiert. Pixel Shader ist jedoch die vorherrschende Terminologie und wird auch in dieser Arbeit verwendet.
Was machen die Pixel Shader aber nun eigentlich? Die Pixel Shader sind wie oben angedeutet zur Bestimmung der endgültigen Farbe einer bestimmten Stelle auf einem Polygon zuständig. Die Pixel-Shader arbeiten wie auch die Vertex-Shader auf einem Strom von Daten, ein Pixel-Shader wird für jedes zu bearbeitende Pixel (genauer Fragment) einmal ausgeführt. Die Auflösung des zu berechnenden Frames bestimmt, wie oft ein Pixelshader für ein Polygon aufgerufen wird.
Die Ausführungsumgebung der Pixel Shader ist ähnlich wie die der Vertex-Shader und unterscheidet sich zwischen den verschiedenen Programmierumgebungen nur geringfügig (siehe Abbildung 9.4). Auch die Ausführungsumgebung von Pixel-Shadern besteht im wesentlichen aus Registern, die teils spezialisiert teils generell verfügbar sind:

- Nur Lese-Register zur Eingabe von Daten aus den vorangehenden Stufen der Pipeline
- Nur-Schreibe-Register mit festgelegter Semantik, in denen der Pixel-Shader seine Ergebnisse (Farbe, Alpha-Wert) an die nochfolgenden Stufen liefert.

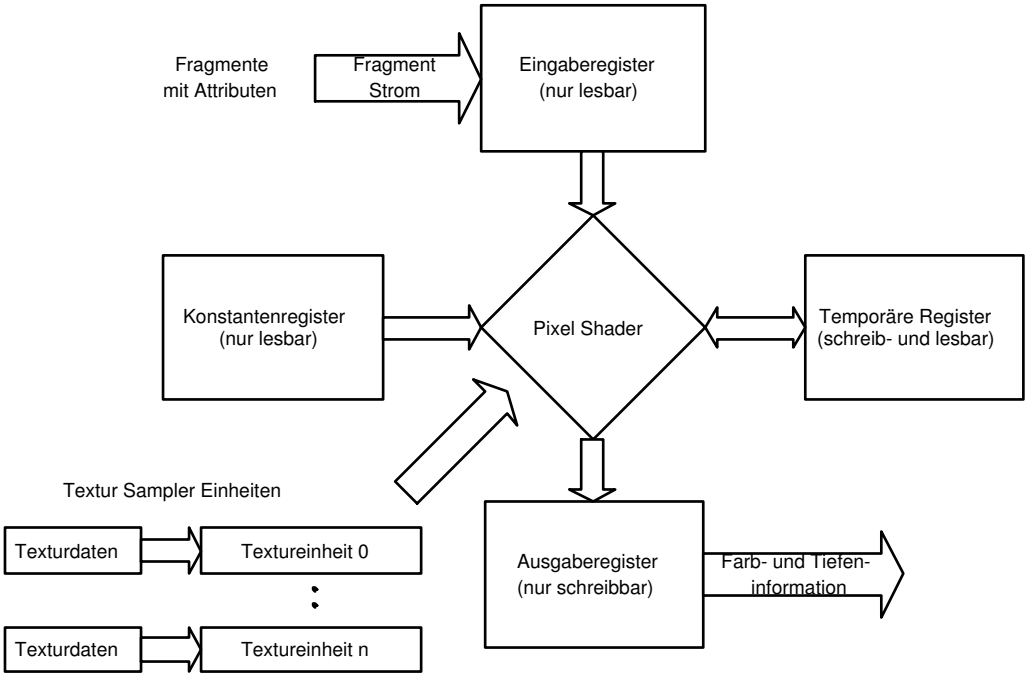


Abbildung 9.4: Ausführungsumgebung von Pixel Shadern

- Konstanten (bzw. Parameterregister), die von den Applikation gesetzt werden und nur gelesen werden können.
- Temporäre Register zur freien Verwendung während der Ausführung.
- Zusätzlich zu den Registern hat ein Pixel Shader Zugriff auf *Texture Sampler* also Textureinheiten, die mittels der vom Pixel-Shader gelieferten Texturkoordinaten (die dieser wiederum ein Eingabedaten bekommen hat) Farbwerte aus den Texturen auslesen und filtern.

Pixel Shader können anders als Vertex-Shader ein Element aus der Pipeline entfernen, d.h. sie können die Verarbeitung eines Fragments beenden und verhindern dass dieses Fragment in den Frame-Buffer geschrieben wird.

9.6 Programmierung

Die zwei am weitesten verbreiteten Schnittstellen zur Grafikprogrammierung auf Consumer-PCs sind DirectX und OpenGL. DirectX ist eine von Microsoft entwickelte Programmierschnittstelle für Systeme unter Windows und stellt einen schnellen und einheitlichen Zugriff auf Multimediakompontenten des PC bereit. Die Grafikprogrammierung ist nur ein (wenn auch großer) Teil von DirectX. Daneben stellt DirectX auch Routinen zur Soundprogrammierung und zum Abspielen von Multimediadateien (Musik, Videos, etc.) bereit. Die zur Zeit aktuelle Version von DirectX ist 9.
OpenGL hingegen beschränkt sich auf die Grafikprogrammierung, ist aber dafür nicht nur auf Windows Plattformen verfügbar, sondern auf einer breiten Palette von Architekturen und Betriebssystemen. Ursprünglich von SGI entwickelt, wurde der Standard von dieser Firma offengelegt und von der Grafikgemeinde und den Grafikkartenherstellern aufgenommen und

unterstützt. Der momentan aktuelle Standard ist OpenGL 1.4, aber es wird für den Sommer 2003 die Festlegung von OpenGL 2.0 erwartet.
Die im weiteren vorgestellten Schnittstellen zur Programmierung der Vertex- und Pixel-Shader bauen ausnahmslos auf einer dieser beiden APIs auf, oder sind bereits in diese integiert.

9.6.1 Programmiersprachen

Assembler

DirectX 8.0 von Microsoft war die erste verbreitete Grafik-API, die eine Sprache zur Programmierung der programmierbaren Stufen der Grafikpipeline integrierte. Diese assemblerähnliche Sprache konnte und kann dazu benutzt werden, Vertex- und Pixelshader zu implementieren. Die Vertexshader ersetzten die *Transform & Lighting*-Stufe der Pipeline vollständig. Die Pixelshader ersetzten die *Multitexturing* bzw. *Texture-Blending* Stufen der Pipeline, boten aber nur wenig mehr als die bisherigen konfigurierbaren Möglichkeiten dieser Stufen.
Aufgrund der eingeschränkten Möglichkeiten der zur Verfügung stehenden Hardware waren die DirectX8 Assembler Sprachen sehr eingeschränkt; so unterscheiden sich die Vertex- und Pixelshader Sprachen stark hinsichtlich ihres Befehlssatzes und der zur Verfügung stehenden Ressourcen.

DirectX 8 Vertex Shader[3] Assembler

Das Ausführungsmodell der DirectX8 Vertex Shader ist im wesentlichen das Standardausführungsmodell, das in Abbildung 9.3 vorgestellt wurde. Es existieren sematisch festgelegt und nur les- bzw. schreibbare Ein- bzw. Ausgaberegister, schreib- und lesbare temporäre Register und nur lesbare Konstantenregister. Sämtliche Register sind vierkomponentige (x, y, z, w) Fließkommaregister.
Der zur Verfügung stehende Befehlssatz besteht aus speziell auf die Grafikberechnung ausgelegten Befehlen, die hauptsächlich als SIMD[4]-Befehle implementiert sind und auf allen Komponenten eines Fließkommaregisters gleichzeitig arbeiten. Diese Befehle sind alle nach dem Schema

```
Instr DestReg, SrcReg [,SrcReg2 [,SrcReg3]]
```

aufgebaut, wobei die Anzahl der Quellregister natürlich von der verwendeten Instruktion abhängt. Jeder Befehl - mit Ausnahme der Makrobefehle (s.u.) - belegt einen Instruktionsslot, von denen DirectX8 Vertex Shader maximal 128 zur Verfügung stellen. Die Abbildung 9.5 gibt einen Überblick über die zur Verfügung stehenden Befehle. Zusätzlich zu diesen Instruktionen stehen dem Programmierer noch *Modifikatoren* und *Register-Masken* zur Verfügung. Die Modifikatoren verändern die Eingangsdaten einer Berechnung bzw. beeinflussen deren Ergebnis. So kann man z.B. den Inhalt eines Quellregisters negiert in die Berechnung einfließen lassen, ohne das Register selbst zu verändern, indem dem Registernamen ein Minuszeichen vornangestellt wird. Masken bestimmen bei Leseoperationen welche Komponenten des Quellregistern gelesen, bzw. in welcher Ordnung vertauscht werden. So dreht z.B. r0.wzyx die Reihenfolge der Komponenten um, während r0.x nur einen Skalar aus dem Quellregister ausliest. Beim Schreiben wird nur in die bezeichneten Komponenten geschrieben. So werden im Zielregister

[3] Auch als Vertex Shader 1.1 bekannt
[4] Single Instruktion - Multiple Data

Instruktion	Beschreibung
add	Addiere 2 Quellregister und speichere das Ergebnis im Zielregister
dp3	Dreikomponentiges Skalarprodukt (Ergebnis wird in alle 4 Komponenten des Ziels repliziert)
dp4	Vierkomponentiges Skalarprodukt (Ergebnis wird in alle 4 Komponenten des Ziels repliziert)
dst	Abstand zwischen 2 Punkten
expp	Berechnet exp(x) mit niedriger Präzision
lit	Berechnet Beleuchtungskoeffizienten
logp	Berechnet den Logarithmus zur Basis 2 mit niedriger Präzision. Quelle ist ein Skalar (z.B. r0.x)
mad	Multipliziert die ersten beiden Quellregister komponentenweise und addiert das dritte hinzu
max	Komponentenweises Maximum der Quellregister
min	Komponentenweises Minimum der Quellregister
mov	Kopiert das Quellregister in das Zielregister
mul	Komponentenweise Multiplikation
rcp	Kehrwert eines Skalars
rsq	Quadratwurzel des Kehrwerts
sge	Setzt Ziel auf 1.0, wenn der erste Quelloperand größer oder gleich dem Zweiten ist, ansonsten ist das Ergebnis 0.0
slt	Das Ziel wird auf 1.0 gesetzt, wenn der zweite Operand kleiner als der Erste ist, ansonsten auf 0.0
sub	Subtraktion

Abbildung 9.5: Befehlssatz der DirectX8 Vertex Shader

r0.xz nur die X und die Z-Komponente geschrieben. Aufbauend auf diesen grundlegenden Befehlen stellt DirectX noch *Makrobefehle* zur Verfügung, die aus mehreren einfachen aufgebaut sind und komplexere Operation wie z.B. Matrix-Vektor Multiplikation ausführen. Diese Makroinstruktionen belegen aber dann auch entsprechend viele Instruktionsslots.
In der Abbildung 9.6 ist ein einfacher Vertex-Shader in dieser Sprache implementiert, der die Vertices transformiert und eine Beleuchtungsberechnung für den diffusen Anteil einer Lichtquelle vornimmt. Als Vorraussetzungen verlangt dieser Shader die Vertex-Position in v0, den Normalenvektor des Stützpunkts in v1 und die entsprechende Texturkoordinate in v2. Die kombinierte Welt-Projektionsmatrix wird in den Konstantenregistern c0 - c3 erwartet, die inverse Transponierte der Weltmatrix in den Registern c4 - c7. Die Lichtrichtung muß im Register c8 gespeichert sein, die diffuse Materialfarbe in c9, eine globale ambiente Beleuchtungsfarbe in c10 und schließlich die Konstante 0.0 in c11.

```
; Transformiere Stützpunkt vom Objektkoordinatensystem ins
; Clipping-Koordinatensystem
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

; Transformiere die Normale ins Weltkoordinatensystem
dp4 r1.x, v1, c4
dp4 r1.y, v1, c5
dp4 r1.z, v1, c6
dp4 r1.w, v1, c7

; Normalisiere den Normalenvektor
dp4 r1.w, v1, v1
rsq r1.w, r1.w
mul r1, r1, r1.w

; Lichtvektor normalisieren
mov r2.xyz, c8
dp3 r2.w, c8, c8
rsq r2.w, r2.w
mul r2, r2, r2.w

; Lichtberechnung durchführen
; farbe = ambient + diffuse * max(0, dot(normal, light direction))
dp3 r3.x, r1, r2
max r3.x, r3.x, c11.x
mad oD0, c9, r3.x, c10

; Texturkoordinate nur durchreichen
mov oT0, v2
```

Abbildung 9.6: Einfache Beleuchtungberechnung mit DirectX8 Vertex Shader Assembler

DirectX8 Pixel Shader Assembler

Die Assemblersprache, die DirectX8 zur Programmierung von Pixel Shadern zur Verfügung stellt, ist primitiv und vielen Einschränkungen z.B. hinsichtlich Programmlänge und Reihenfolge der Befehle unterworfen. DirectX8 unterstützt 5 verschiedene Versionen der Pixel Shader Sprache. Die Versionen 1.0 - 1.3 unterscheiden sich dabei nur in den verfügbaren Instruktionen und in den Beschränkung, die zu höheren Versionsnummern hin lockerer sind. Sie beruhen aber alle auf derselben Ausführungsumgebung. Die Pixel Shader 1.4, die Microsoft mit DirectX Version 8.1 einführte unterscheiden sich im Ausführungsmodell und den Fähigkeiten deutlich. Sie wurden auf Initiative von ATI eingeführt, deren Radeon 8500 Chip diese Shader im Gegensatz zu NVidia's Geforce Karten (Geforce3 unterstützt Pixel Shader bis 1.1, Geforce4 bis 1.3) unterstützt.
Pixelshader unterstützen 2 Arten von Anweisungen: Texturaddressierungs- und Arithmetikan-

weisungen (siehe Abbildung 9.7). Diese Anweisungen dürfen nicht beliebig gemischt werden,

Instruktion	**Beschreibung**
add	Addiere 2 Vektoren
cmp	Vergleiche mit Null
cnd	Vergleiche mit 0.5
dp3	Skalarprodukt (3 Komponenten)
dp4	Skalarprodukt (4 Komponenten)
lrp	Lineare Interpolation
mad	Multiply and Add
mov	Kopiere Quelle nach Ziel
mul	Komponentenweise Multiplikation
nop	Keine Operation
sub	Vektorsubtraktion
tex	Texturelement (Farbe laden)
texcoord	Texturkoordinate als Farbe laden

Abbildung 9.7: Befehlssatz der DirectX8 Pixel Shader 1.3 (nur elementare Texturanweisungen)

sondern zuerst müssen alle Texturanweisungen gegeben werden, danach erst dürfen die Arithmetikanweisungen angewendet werden. Die Ausnahme ist hier der Pixelshader 1.4 bei dem die Ausführung in 2 Phasen aufgeteilt werden kann und Texturanweisungen am Anfang der beiden Phasen stehen dürfen. Die Texturanweisungen greifen auf eine Textureinheit zu und laden die (evtl. gefilterten) Farbwerte eines Texturelements in eines der temporären Register, wo sie mithilfe der arithmetischen Anweisungen gemischt werden können. Sie benötigen dazu die Texturkoordinaten, die von den vorherigen Stufen der Pipeline (Vertex-Shader) oder auch der Applikation selbst kommen und von der Hardware zwischen den Stützpunkten interpoliert werden. In den Pixelshadern bis einschließlich 1.3 war die Zuordnung Textureinheit-Texturkoordinate-Texturregister fest, es existiert eine 1-zu-1 Assoziation, d.h. eine Texturkoordinate[5] konnte nur einmal benutzt werden. Mit den Pixel Shadern 1.4 wurde diese Einschränkung aufgehoben und Texturkoordinaten konnten nun mehrfach und für verschiedene Textureinheiten benutzt werden.
Durch die Einschränkung, dass sämtliche Texturanweisungen vor den Arithmetikanweisungen stehen müssen, waren die Designer gezwungen, viele unterschiedliche Texturanweisungen zur Verfügung zu stellen, die die benötigten arithmetische Operationen (z.B. Matrixmultiplikationen) mit den Texturkoordinaten durchführten. Dieses Design wurde mit den Pixelshadern 1.4 verworfen, weil hier die normalen Arithmetikanweisungen genutzt werden können, um die gewünschten Berechnungen mit den Texturkoordinaten auszuführen. Bei diesem Pixelshadern existieren nur noch die elementaren Texturanweisungen.

[5] die aus mehreren Komponenten (üblicherweise 2-3) besteht

DirectX9 Vertex Shader Assembler

Die Vertex-Shader Assembler Sprache von DirectX9 ist eine Weiterentwicklung der DirectX 8 Assembler Sprache. Die Einschränkungen sind gelockert worden bzw. weggefallen und neue Instruktionen sind hinzugekommen. DirectX9 kennt neben der Version 1.1 der Vertex Shader aus DirectX8 3 neue Vertex-Shader Profile mit unterschiedlichen Fähigkeiten. Diese Profile sind in der Tabelle in Abbildung 9.8 mitsamt ihrer wesentlichen Unterschiede aufgelistet. Zum

Version	Besonderheiten
1.1	128 Instruktionen 96 Konstantenregister
2.0	statische Flußkontrolle erweiterte Arithmetikanweisungen bis zu 256 Instruktionen (aber durch Schleifen mehr ausführbar) 256 Konstantenregister
2.x	dynamische Flußkontrolle verschachtelte stat. Flußkontrolle
3.0	erweiterte Adressierungsmöglichkeiten *Vertex textures* min. 512 Instruktionen (Hardwareabhängig)

Abbildung 9.8: Vertex Shader von DirectX 9

Vergleich wurde auch die schon in DirectX 8 eingeführte Version 1.1 mitaufgeommmen. Eine wesentliche Neuerung der Vertex-Shader ab Version 2.0 ist die Möglichkeit zur *Flußkontrolle* durch neu hinzugekommene Instruktionen. Die Version 2.0 unterstützt statische Flußkontrolle, d.h. Kontrolle des Programmflußes abhängig von den Konstantenregistern. Die Versionen 2.x und 3.0 unterstützen darüber hinaus die dynamische Flußkontrolle, d.h. der Programmlauf kann von Berechnungen innerhalb des Shaders beeinflußt werden. Version 3.0 bietet darüberhinaus erweiterte Adressierungsmöglichkeiten: Neben Konstantenregistern können nun auch Eingabe- und Ausgaberegister über das Adressregister indirekt addresiert werden. Neu eingeführt wurden in dieser Version auch sog. *Vertex Textures*. Das sind Texturen, auf die mittels einer speziellen Instruktion vom Vertex-Shader aus zugegriffen werden kann. Damit ist die Übergabe größerer Datenmengen an einen Vertex-Shader nun möglich. Version 3.0 wird von keiner zur Zeit verfügbaren Consumer-Grafikhardware voll unterstützt. Zur Entwicklung dieser Shader kann man aber auf in DirectX integrierte Softwareemulationen zurückgreifen.

DirectX 9 Pixel Shader Assembler

Die doch sehr eingeschränkten Pixel Shader unter DirectX8 wurden mit der Version 9 deutlich erweitert. Auch hier wurden die Versionen 2.0, 2.x und 3.0 eingeführt. Die wesentlichen Merkmale der Versionen sind in Abbildung 9.9 aufgeführt. Vor allem der Befehlssatz ist in der neuen Version sehr viel umfangreicher; es können nun alle arithmetischen Instruktionen der Vertex-Shader benutzt werden (ab Version 2.0), außerdem kommen ab 2.x Möglichkeiten zur Flußkontrolle hinzu. Die Zahl der verfügbaren Register und die Zahl der erlaubten

Version	Besonderheiten
1.1 1.2 1.3	8 arith. Anweisungen 4 Texturanweisungen
1.4	pro Phase (2 Phasen): 8 arith. Anweisungen 6 Texturanweisungen
2.0	erweiterte arith. Anweisungen 64 arithmetische Anweisungen 32 Texturanweisungen
2.x	statische Flußkontrolle (opt.) dynamische Flußkontrolle (opt.) 96 - 512 Instruktionen
3.0	stat. & dynamische Flußkontrolle min. 512 Instruktionen (Hardwareabhängig)

Abbildung 9.9: Pixel Shader von DirectX 9

Instruktionen wächst mit den Versionsnummern. Auch bei den Pixel-Shadern gibt es noch keine kommerziell verfügbare Grafikkarte, die die Version 3.0 unterstützt, aber hier existieren wiederum DirectX-Bestandteile, die diese Shader in Software emulieren (wenn auch ungleich langsamer als Hardware-Implementierungen).

OpenGL NV_vertex_program

Die OpenGL-Erweiterung NV_vertex_program wurde, wie der Name schon vermuten, läßt von der Firma NVidia entwickelt, um die programmierbare Geometriestufe moderner Grafikkarten (insbesondere natürlich NVidia's Geforce) auch unter OpenGL verfügbar zu machen. Die Fähigkeiten und Leistungsfähigkeit dieser Erweiterung ist im wesentlichen mit denen von DirectX 8 Vertex-Shadern vergleichbar, auch die Ausführungsumgebungen sind dieselben. Die zugehörigen Assemblerprogramme unterscheiden sich nur leicht im Syntax und benutzen dieselben Befehle. Die OpenGL Erweiterung definiert einige wenige Befehle, die nicht in der DirectX8 Assembler Sprache enthalten sind, diese lassen sich jedoch leicht durch DirectX8-Befehle emulieren.

OpenGL ARB_vertex_program

Die ARB_vertex_program Erweiterung ist prinzipiell die offiziell abgesegnete Variante der proprietären NV_vertex_program Erweiterung und ist bis auf wenige Ergänzungen mit dieser identisch. Bei ARB_vertex_program werden die Konstantenregister in zwei Arten aufgeteilt, die sogenannten programmlokalen (*program local*) Register und die Programmumgebungs-Register (*program environment registers*). Die erste Gruppe von Registern behält ihre Werte nur solange das entsprechende Programm auch der aktuelle Shader ist, die zweite Gruppe behält Ihre Werte, so dass diese auch von anderen Vertex-Shader benutzt werden können.

Eine weitere Neuerung dieser Erweiterung ist die Tatsache, dass Register keine festen Namen mehr haben, sondern diese vor der Benutzung deklariert werden müssen. So deklarieren z.B. die folgenden Zeilen

```
ATTRIB pos = vertex.position;
OUTPUT outpos = result.position;
TEMP myTempReg;
ADDRESS myAddressReg;
PARAM mvp[4] = { state.matrix.mvp };
```

ein Attribut-Register namens **pos**, das die Position des gerade bearbeiteten Stützpunktes enthält, ein Ausgaberegister namens **outpos**, das auf das Register mit der transformierten homogenen Position abgebildet wird, ein temporäres Register, ein Addressregister und ein spezielles Parameter-Register, bei dem ein Feature dieser Erweiterung benutzt wird, das *automatic state tracking* (automatische Zustandsverfolgung). Das Register **mvp** wird immer den aktuellen Stand der Model-View-Projection Matrix, die mit den Standard-OpenGL-Befehlen gesetzt wird, verfolgen. Mit dieser Technik lassen sich die OpenGL-Zustandsvariablen komfortabel von Vertex-Shadern aus benutzen, ohne dass der Benutzer diese explizit in die Konstantenregister übertragen muß. Die im obigen Beispiel benutzten Bezeichner, die mit **vertex.** oder **result.** beginnen, sind von der Erweiterung vordefiniert und werden automatisch den richtigen Registern zugeordnet.

OpenGL NV_vertex_program2

NV_vertex_program2 ist NVidias Weiterentwicklung Ihrer früheren Vertex-Shader Erweiterung für OpenGL. Diese Erweiterung zielt konsequent auf die Fähigkeiten von Grafikkarten mit Geforce FX ab und ist in der Funktionalität mit den Vertex-Shadern in der Version 2.x von DirectX9 vergleichbar mit statischer und dynamischer Flußkontrolle, erweiterten Arithmetikbefehlen und denselben Gegebenheiten hinsichtlich der Befehls- und Registeranzahl. Ein kleiner Unterschied zu Version 2.x Vertex-Shadern ist die Tatsache, dass Shaderprogramme in dieser Erweiterung Rückwärtssprünge enthalten dürfen, während DirectX Shader nur Vorwärtssprünge enthalten dürfen.

OpenGL ARB_fragment_program

Die erste Pixel-Shader Erweiterung, die vom ARB für OpenGL offiziell angenommen wurde, ist ARB_fragment_program. Diese Erweiterung entspricht von der Mächtigkeit her den Pixel-Shadern von DirectX9 in der Version 2.0, d.h. sie besitzt mächtige arithmetische Befehle (Sinus, Cosinus, Exponentialrechnung). Sie ist aber nicht so mächtig wie die 2.x Shader (DirectX9), weil ihr die Flußkontrollmöglichkeiten dieser Shader fehlen.

OpenGL NV_fragment_program

NVidias Pixel-Shader Erweiterung für OpenGL zielt wie die NV_vertex_program2 Erweiterung derselben Firma ganz auf die Möglichkeiten des Geforce FX Chips ab und macht diese unter OpenGL verfügbar. Damit entspricht ihre Mächtigkeit der der Pixel-Shader 2.x unter DirectX9. Allerdings werden keine statische Flußkontrollmöglichkeiten unterstützt (hier ist NV_fragment_program etwas weniger mächtig als die Pixel-Shader 2.x). Es existiert aber eine Möglichkeit, if-Bedingungen zu formulieren, wobei jedoch beide Pfade durchlaufen werden

müssen und das Ergebnis eines Pfades dann verworfen wird. Im Bezug auf die Anzahl der erlaubten Instruktionen ist NV_fragment_program etwas weniger restriktiv als das entsprechende DirectX Profil, da 1024 Instruktionen gegenüber 512 erlaubt sind. Auch unterstützt NV_fragment_program verschiedene Genauigkeiten bei arithmetischen Befehlen: Arithmetische Operationen können mit drei verschiedenen Genauigkeiten ausgeführt werden (32bit Fließkomma, 16bit Fließkomma, 12bit Festkomma).
Wie bei ARB_fragment_program und DirectX Pixel Shader 2.x werden auch bei NV_fragment_program komplexe, arithmetische Operationen unterstützt. NV_fragment_program ist die zur Zeit mächtigste, in Consumer-Hardware implementierte und verfügbare Fragment-Shading Sprache.

Hochsprachen

Stanford Shading Language

Die Stanford Shading Language wurde von Mitarbeitern der Stanford University entwickelt und auf der SIGGRAPH 2001 vorgestellt (
[Proudfoot et al. '01]). Die Ziele bei der Entwicklung der Sprache waren, die Programmierung von Echtzeit-Shadern zu vereinfachen und von der darunterliegenden Hardware zu abstrahieren. Diese Sprache wird als Ebene oberhalb von OpenGL realisiert und erweitert diese API um zusätzliche Funktionen zum Umgang mit Shadern.
Der Compiler bzw. die Laufzeitumgebung virtualisiert die Hardware, d. h. es gibt für den Benutzer dieser Sprache keine Beschränkungen in der Anzahl der zur Verfügung stehenden Hardware-Resourcen wie Registern, Eingabe- / Ausgabevariablen oder auch der möglichen Programmlänge. Diese Virtualisierung der Ressourcen vereinfach die Shaderprogrammierung erheblich, da sich der Benutzer nicht um Einschränkungen der verschiedenen Hardwarearchitekturen kümmern muß. Die Entwickler dieser Sprache definieren eine Standardpipeline (Abbildung 9.10), die aus programmierbaren und nicht programmierbaren Einheiten besteht. In

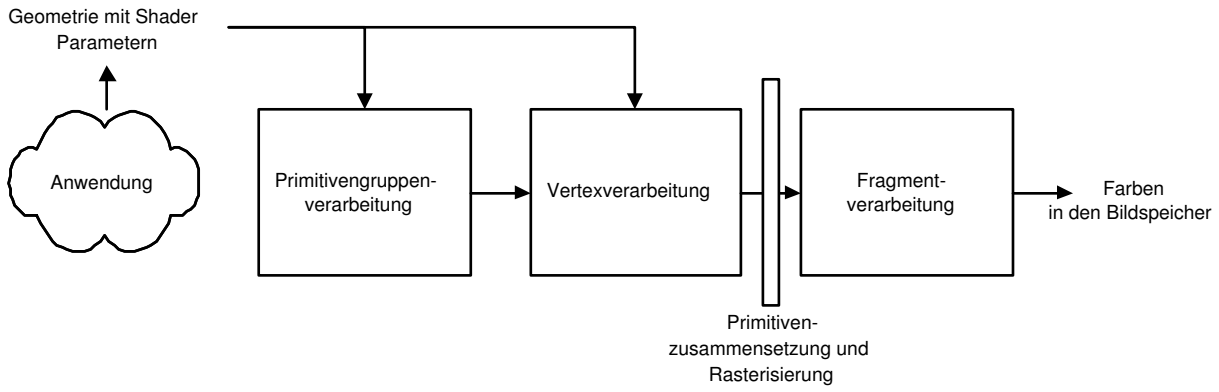


Abbildung 9.10: Pipeline der Stanford Shading Language

dieser Pipeline findet sich auch das Konzept der *Berechnungsfrequenzen*, welches eine zentrale Rolle bei der Entwicklung dieser Sprache spielte. Es gibt vier verschiedene Berechnungsfrequenzen: *constant, primitive group, vertex* und *fragment*. Variablen, die als *constant* gekennzeichnet sind, müssen zur Kompilierzeit schon feststehen, *primitive group* Variablen werden einmal pro sglBegin()/sglEnd()[6] Paar neuberechnet, *vertex* und *fragment* entsprechen pro-Vertex bzw.

[6] Entspricht glBegin() bzw. glEnd() von OpenGL

pro-Pixel Berechnungen. Variablen können entweder explizit mit einer Berechnungsfrequenz versehen werden oder werden vom Compiler mit Standardmarkierungen versehen. Diese Markierungen werden über Operatoren nach festgelegten Regeln weiterpropagiert, wobei die Korrektheit der Ergebnisse garantiert ist.
Der Compiler ist dafür verantwortlich, das Programm auf die von der Hardware vorgesehenen verschiedenen Einheiten (gewöhnlich Vertex- und Pixel-Shader) aufzuteilen und dabei die Korrektheit zu waren.
Es existiert eine Implementierung des Compilers und verschiedene Backends die C-Code, x86 Assembler, oder auch hardwarespezifischen (auf OpenGL aufsetzenden) Sourcecode erzeugen.
Die Abbildung 9.11 zeigt ein kurzes Programmbeispiel, welches diffuse Oberflächenbeleuchtung

```
surface float4 lightmodel_diffuse (float4 a, float4 d)
   {
   perlight float diffuse = dot(N,L);
   perlight float4 fr = select(diffuse > 0, d * diffuse, Zero);
   return a * Ca + integrate(fr * Cl);
   }
```

Abbildung 9.11: Beispielprogramm in der Stanford Shading Language

berechnet. Beachtenswert sind hier die Typmodifikatoren *perlight* und die *integrate* Anweisung. *Perlight* weist den Compiler an, diese Variable für jedes in der Szene vorhandene Licht einmal zu berechnen, die *integrate*-Funktion wird zur Kompilierungszeit in eine Summe umgewandelt, die Berechung wird für jedes Licht durchgeführt, danach werden die Einzelergebnisse aufsummiert.

C for graphics (Cg)

Cg (sprich: C for graphics) ist eine von NVidia entwickelte C-ähnliche High-Level Programmiersprache, die zum Zwecke der Vertex- und Pixel-Shader-Programmierung entworfen wurde. Die Designer bei NVidia lehnten sich dabei an die *Stanford Shading Language (9.6.1)* und an die *Renderman Shading Language* an. Ihr Ziel war, die Entwicklung von Vertex- und Pixel-Shadern zu vereinfachen, ohne sich dabei auf eine spezielle Hardware oder Grafik-API festzulegen. Das Ergebnis ist ein Toolkit, das aus einem Compiler und Runtimes für die 2 meistverbreitetsten Grafik-APIs (DirectX und OpenGL) besteht.

Der Cg-Compiler selbst besteht aus einem generischen Front-End, welches spezielle Back-Ends zur Codeerzeugung unterstützt. Diese Back-Ends werden durch *Profile* ausgewählt. Das Profil wird - wenn gewünscht - entweder durch eine Compileranweisung im Sourcecode selbst oder durch Kommandozeilenparameter beim Compileraufruf übergeben. Die Ausgabe des Compilers sind Assembler-Source Files, deren Syntax vom gewählten Profil abhängt. Es können z.B. DirectX Assembler Sourcen für verschiedene Pixel oder Vertex-Shader Versionen erzeugt werden, die dann direkt über die DirectX-API verwendet werden können. Gleiches gilt auch für OpenGL.

Der zweite Weg Cg in eigenen Projekten zu verwenden, liegt in der Benutzung der Cg Runtimes, die für DirectX und OpenGL existieren. Hier übernimmt die Runtime auch die Aufgabe der Kompilierung der Cg-Quelldateien, so dass diese direkt von Programmen verwendet werden können. Das kompilierte Programm kann dann entweder direkt durch die Cg-Runtime oder auch per-Hand an die Low-Level-API (wieder DirectX oder OpenGL) weitergereicht werden.

Das Beispielprogramm in Abbildung 9.12 zeigt wie ein typisches Cg-Programm aussieht: Es werden zunächst die Ein- und Ausgabetypen deklariert. Der Syntax ist eng an die Sprache C angelegt mit Ausnahme der *Bindings*, die durch einen Doppelpunkt getrennt hinter den Variablennamen stehen. In diesem Beispiel sind das u.a. POSITION, NORMAL und COLOR0. Sie binden die Variable an ein bestimmtes Element des Datenstroms. Im Unterschied zu C sind viele für Vertex und Pixelshader gebräuchliche Datentypen (wie z.B. Vektoren und Matrizen) bereits im Sprachstandard integriert.
Neben den Ein- und Ausgabevariablen gibt es noch die mit *uniform* gekennzeichneten. Sie entsprechen den schon früher erwähnten Konstanten, die den Vertex- bzw. Pixelshadern von der Applikation zur Verfügung gestellt werden können. In diesem Beispiel sind dies Transformationsmatrizen und Beleuchtungseigenschaften (Position und Farbe des Lichts). Cg kümmert sich um die Belegung der Konstantenregister und den Zugriff. Die Applikation kann über die Cg-Runtime herausfinden, in welchen Konstantenregistern das kompilierte Cg-Programm welche Informationen erwartet.
Das eigentliche Programm, das hier in der main-Funktion[7] steht, wird für jedes Element des Datenstroms (hier im Beispiel ist es ein Vertexstrom) einmal ausgeführt, bekommt als Parameter (**Input**) das aktuelle Element des Datenstroms und gibt seine Ausgabe (**Output**) mittels *return* zurück.

DirectX 9.0 High-Level Shading Language (HLSL)

Auch Microsoft erkannte bei der Weiterentwicklung ihres Grafik-APIs die Notwendigkeit, eine Hochsprache zur Programmierung der Vertex- und Pixelshader breitzustellen. Das Assembler der DirectX-Version 8 war zu fehleranfällig und zur schnellen Entwicklung der in moderner Hardware möglichen langen und komplizierten Shader nicht mehr geeignet.
Aus diesen Bemühungen entstand die HLSL für DirectX 9. NVidia und Microsoft begannen die Entwicklung ihrer Shader-Hochsprachen getrennt, aber arbeiteten in späteren Phasen der Entwicklung zusammen und sorgten dafür, dass die Sprachen Cg und die für DirectX 9.0 entwickelte HLSL 100%ig miteinander kompatibel sind.

OpenGL 2.0 Shading Language (GLslang)

Der OpenGL 2.0 Standard ist noch nicht endgültig festgeschrieben, aber vorausichtlich wird dies im Sommer 2003 geschehen. Einer der wesentlichen Gründe für die OpenGL 2.0 Spezifikation war die Einführung einer High-Level Shading Language. Diese Sprache, unter dem Namen GLslang bekannt, wurde von der Firma 3DLabs entwickelt, für OpenGL 2.0 dem ARB vorgeschlagen und im September 2002 angenommen. Zur Zeit ist 3DLabs dabei, die endgültige Spezifikation der Sprache fertigzustellen.
GLslang hat starke Ähnlichkeiten zu Cg, aber auch einige Besonderheiten, die von OpenGL herrühren. Der konzeptionell größte Unterschied ist der Zeitpunkt der Kompilierung: Während Cg vom Cg-Compiler in einen Assembler-Zwischencode kompiliert wird, der je nach Laufzeitumgebung DirectX oder OpenGL spezifisch ist und der dann von der Laufzeitumgebung dem Grafiktreiber der endgültigen Kompilierung für die spezifische Hardware übergeben wird, wird bei GLslang der Sourcecode selbst dem Grafiktreiber zur Kompilierung übergeben. Die Designer erhoffen sich davon ein höheres Optimierungspotential, da der Grafiktreiber so Zugriff auf eine höhere Abstraktionebene erlangt und früher mit Optimierungen ansetzen kann.
Bei der Entwicklung dieser OpenGL Sprache wurde natürlich auf größtmögliche Harmonie

[7] Typischerweise wird der Name *main* verwendet, aber es kann auch ein beliebiger anderer Name sein.

```
// Diffuse Beleuchtung pro Vertex
// Cg Vertex Shader
struct appdata {
    float4 position : POSITION;
    float3 normal   : NORMAL;
    float4 color    : COLOR0;
};

struct v2f {
    float4 HPOS : POSITION;
    float4 COL0 : COLOR0;
};

v2f main(appdata Input,
         uniform float3x3 object_matrix,
         uniform float4x4 objviewproj_matrix,
         uniform float3 lightdir,
         uniform float3 ambientcolor)
{
    v2f Output;
    float diffuse;

    // transformiere Vertices in das Projektionskoordinatensystem
    // mithilfe der fetig Transformationsmatrix
    Output.HPOS = mul(objviewproj_matrix, Input.position);

    // Transformiere die Oberflächennormale in das Weltkoordinatensystem
    // und benutze das Skalarprodukt um die diffuse Lichtintensität
    // zu bestimmen. Wenn die Oberfläche vom Licht wegzeigt, setze die
    // Intensität auf 0

    diffuse = max(0, dot(mul(object_matrix, Input.normal), lightdir));

    Output.COL0.rgb = Input.color.rgb * diffuse + ambientcolor;
    Output.COL0.a = Input.color.a;

    return Output;
} // main
```

Abbildung 9.12: Ein einfacher Cg Vertex Shader

mit OpenGL geachtet. Deswegen finden sich zwei Punkte auf den Entwicklungzielen, die diese Harmonie unterstreichen:

- Die durch GLslang ersetzbare OpenGL fixed function pipeline soll durch GLslang implementierbar sein.

- Der Zugriff auf den OpenGL render state soll von der Sprache aus möglich und einfach zu implementieren sein.

Neben diesen Zielen finden sich auch hier dieselben Ziele wieder, die auch ausschlagebend für die Entwicklung der anderen Shading-Hochsprachen waren, wie

- Leichte Programmierbarkeit
- Portabilität über Hardwaregrenzen hinweg
- Die Flexibilität der Hardware einfacher zugänglich machen
- *Eine* Sprache sowohl für Vertex als auch für Pixel Shader.

Wie die meisten der Shading Languages beruht auch GLslang auf einer erweiterten C-Syntax. Die Sprache C wurde um grafikspezifische Datentypen wie Vektoren und Matrizen und die dazugehörigen Operationen erweitert, weggelassen wurden Zeigeroperationen. Allerdings unterscheidet sich GLslang hinsichtlich der Parameterübergabe von den bisher besprochenen Hochsprachen: Anstatt die Parameterübergabe wie gewöhnliche Funktionsparameter zu handhaben und diese mit *Bindings* an semantisch festgelegte Hardwareregister zu binden, werden Parameter in GLslang in Form von globalen Variablen weitergegeben, die je nach Semantik und Shader nur les- bzw. nur schreibbar sind. So erhält ein Vertex-Shader beispielsweise die Position eines Stützpunktes in der globalen Variablen *gl_Vertex* und muß die transformierte Koordinate in *gl_Position* speichern. Auch Vertex- und Pixel-Shader handhaben ihre Parameterübergabe über gemeinsame Variablen. Die verschiedenen Arten von Shadern befinden sich wie auch z.B. bei Cg in verschiedenen Quelldateien. Allerdings können identisch deklarierte Variablen in Vertex- und Pixel-Shader zu Übergabe von Parametern genutzt werden. Diese Referenzen werden vom Linker aufgelöst, der explizit über einen OpenGL 2.0 Befehl aufgerufen werden muß, bevor Shader benutzt werden können.
Wie schon erwähnt, existiert bei GLslang kein Assembler-Zwischencode, sondern der Quellcode wird direkt über den Aufruf *glCompileShader* kompiliert und kann dann gelinkt und an Objekte gebunden werden.
Die Shader fügen sich nahtlos in die bekannte OpenGL-Schnittstelle ein; so werden die Stützpunkte auch hier mit *glVertex*, *glNormal*, usw. angelegt und werden dann in den globalen Variablen abgelegt, in denen der Shader diese erwartet. Abbildung 9.13 zeigt das schon bekannte Vertex-Shader Beispiel zur diffusen pro-Vertex Beleuchtung in GLslang. Wie man sieht, fehlen die von Cg bekannten Deklarationen von Ein- und Ausgabestrukturen völlig. Für dieses Beispiel reichen die festgelegten globalen Variablen völlig aus.

9.7 Anwendungen

9.7.1 Grafikeffekte

Mit den Vertex- und Pixelshadern lassen sich viele grafische Effekte erzeugen, die vorher nicht, oder nur umständlich bzw. vereinfacht in Echtzeit realisierbar waren. Ein paar einfache Beispiele sollen hier dargestellt werden:

Animation

Vertex-Shader lassen sich u.a. dazu verwenden, Animationen zu berechnen, ohne auf die CPU zurückgreifen zu müssen. Dies kann so aussehen, dass die CPU nur noch die Stützpunkte der Keyframes schickt und die Zwischenschritte dann vom Vertex-Shader interpoliert werden. Dies kann linear, aber auch mittels anderer Funktionen passieren, die die Bewegung weicher werden

```
// Ein Richtungslicht mit diffuser Schattierung
void main (void)
{
   float normalDotVP;
   vec3 color, normal, diffuseLight;
   vec4 position;

   // Stützpunkt ins Clipping-Koordinatensystem transformieren
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   // Normale in Weltkoordinaten transformieren
   normal = gl_NormalMatrix * gl_Normal;
   // Diffusen Beleuchtungsfaktor ausrechnen
   normalDotVP = min (0, dot (normal, gl_Light0[gl_kPosition]));

   // Diffuse Beleuchtung ausrechnen
   diffuseLight = gl_Light0[gl_kDiffuseIntensity] * vec3 (normalDotVP);
   color = gl_Light0[gl_kAmbientIntensity]*gl_FrontMaterial[gl_kAmbientIntensity] +
           diffuseLight * gl_FrontMaterial[gl_kDiffuseIntensity];

   // Farbe in globaler Übergabevariablen speichern
   gl_FrontColor = clamp(vec4 (color, gl_FrontMaterial[gl_kDiffuseAlpha].x), 0, 1);
}
```

Abbildung 9.13: Diffuse Beleuchtung mit GLslang

lassen. Die so gewonnenen Zwischendaten lassen sich aber nicht (oder zumindest nicht ohne größeren Aufwand wieder an den Hauptprozessor übermitteln.

Phong-Shading

Im Gegensatz zum Gouraud-Shading werden beim Phong-Shading die Beleuchtungsberechnungen für jeden Punkt einer Oberfläche ausgewertet und der Punkt entsprechend eingefärbt. Das Phong-Shading wird von den Pixel-Shadern übernommen.
Die für die Auswertung der Lichtgleichung nötigen Oberflächennormalen müssen dem Pixel-Shader übergeben werden. Dafür gibt es zwei Möglichkeiten:

1. Die Normalen werden vom Vertex-Shader in Farb- oder Texturkoordinatenregister geschrieben, sie werden dann von der Hardware automatisch zwischen den Stützpunkten interpoliert.

2. Die Normalen werden als Farbwerte in einer Textur gespeichert und vom Pixel-Shader ausgelesen. Die drei Farbkanale entsprechen dabei den drei Koordinaten.

Bump-Mapping

Mit Hilfe von Bump-Mapping lassen sich Oberflächenunebenheiten simulieren, ohne dass dazu die Geometrie verändert werden müßte. Dazu wird in der Beleuchtungsberechnung für diesen Oberflächenpunkt nicht die eigentliche, sondern eine veränderte Oberflächennormale benutzt, um das Ergebnis dieser Berechnung zu beeinflussen. Dadurch wirkt die Oberfläche als hätte sie Unebenheiten. Die veränderten Normalen werden dem Pixel-Shader als Textur übergeben,

von den Textureinheiten ausgelesen und gefiltert und dann in der Beleuchtungberechnung verwendet.

Reflexion und Brechung

Reflektierende Oberflächen können beim Rasterisierungsverfahren durch eine Umgebungstextur simuliert werden. In dieser Textur ist die Umgebung des Objektes aus der Sicht dieses Objektes gespeichert. Heutzutage wird oft eine *Cube-Map* benutzt. Das ist eine aus sechs Einzeltexuren bestehende Textur, die die Umgebung des Objekts in positive und negative Richtung der drei Koordinatenachsen abbildet.
Ein Vertex-Shader kann nun zu gegebenen Augen- und Normalenvektoren einen Reflektionsvektor berechnen, der dann in der Texturierungsstufe benutzt wird, um die Cube-Map auszulesen. Die Brechung funktioniert ähnlich, es wird statt dem Reflektionsvektor eben der Brechunungsvektor berechnet. Die Anwendungsmöglichkeiten der Shader sind noch weit vielfältiger,

als es die angeführten Beispiele vermuten lassen. Die Shader sind schließlich programmierbare Einheiten und damit sind ihre Anwendungsmöglichkeiten hauptsächlich durch die Kreativität des Programmierers beschränkt.

9.7.2 Raytracing

Das Rasterisieren von Bilder, mittels der von Grafikhardware unterstützten und beschleunigten Techniken in Echtzeit hat, vor allem in den letzten Monaten, immer mehr Erfolge damit, globale Beleuchtungseffekte wie Brechung, Reflektion oder auch Radiosity darzustellen. Erreicht wird dies mittels Tricks wie z.B. *Environment Mapping*, das heißt die Umgebung eines Objekts wird aus Sicht dieses Objektes auf eine Textur gerendert und das Objekt wird dann mit dieser Textur gerendert. Die meisten dieser Effekte können jedoch nur angenähert werden, bei anderen ist auch dies unwahrscheinlich.
Eine Methode, die realistischere Ergebnisse verspricht, ist das *Raytracing*. Die üblichen Renderpipelines können für das Raytracing nicht benutzt werden, weil sich diese Technik grundsätzlich von Rendering der Grafikkarten unterscheidet. Jedoch lag es nahe, die von den programmierbaren Einheiten der neuen Grafikkarten bereitgestellte Rechenleistung zu nutzen und mit Raytracing innerhalb der Grafikhardware zu experimentieren.
Purcell et al. stellen in ihrer Arbeit [Purcell et al. '02] ihren Ansatz vor. Es werden für die eigentlichen Berechnungen nur Pixel-Shader benutzt, da Vertex-Shader nicht auf Texturen zugreifen können, die verwendet werden, um die Masse an Daten für die Shader verfügbar zu machen, bzw. die Ergebnisse der Shader zu speichern.
Zum Zeitpunkt der Arbeit war noch keine Hardware verfügbar, die eine entsprechend mächtige Pixel-Shader-Sprache anbot, also wurde eine Sprache selbst definiert und die Versuche auf einem Simulator durchgeführt. Die benutzte Assembler-Shadersprache hat folgende Eigenschaften:

- Arithmetische Fließkommainstruktionen (vergleichbar mit DirectX 9 Shadern)
- Fließkommatexturen und Bildspeicher
- Programmlänge vergleichbar mit DX9 Pixel Shadern ohne Limits hinsichtlich der Anzahl und Reihenfolge der Textur- und Arithmetikinstuktionen
- Mehr als ein Ausgabewert (2 Fließkommavektoren)

- Direkter Stencilbufferzugriff

Diese Shaderdefinition kommt nahe an die von DirectX9 Pixel-Shadern heran, wird aber von dieser nicht vollständig abgedeckt. In der Arbeit werden zwei Verfahren vorgestellt: eines arbeitet mit mehreren Durchläufen (Multipass) pro Pixel, das zweite setzt voraus, dass Pixelshader dynamische Flußkontrolle unterstützen.
Die Parallelität der Hardware und des Verfahrens wird dadurch ausgenutzt, in dem ein streambasiertes Modell (siehe Abbildung 9.7.2) für ihren Raytracer verwendet wird.

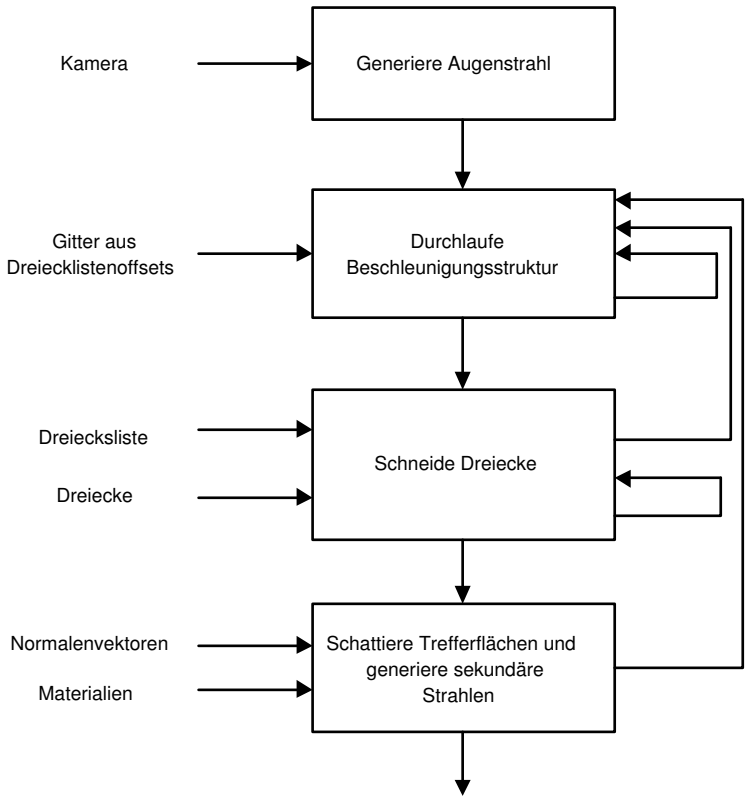


Abbildung 9.14: Streambasierter Raytracer

Um die nötigen Schnittberechnungen zu beschleunigen wird bei Raytracern gewöhnlich eine Datenstruktur verwendet, in der die Dreiecke der Szene nach einer bestimmten Ordnung eingetragen sind. Die Autoren verwenden hier ein dreidimensionalen gleichmäßiges Gitter. Jeder Quader dieses Gitters enthält eine Liste der in ihm enthaltenen Dreiecke. Diese Vorberechnung muß von der Host-CPU ausgeführt werden und ist berechnungsintensiv. Um diesen Ansatz für dynamische Szenen zu verwenden, muß die Vorberechnung effizient für solche Szenen durchgeführt werden können. Die Quader werden in einer 3D-Textur abgelegt, ein Wert von Null bedeutet dabei, dass in diesem Quader keine Dreiecke enthalten sind, anderenfalls befindet sich dort ein Index in eine andere Textur, diesmal eindimensional, in der die Dreieckslisten enthalten sind. In diesen sind Indizes verzeichnet, die wiederum als Index in drei andere Texturen (Erste, Zweiter und Dritter Punkt eines Dreiecks) verwendet werden, um die eigentlichen Koordinaten auszulesen. Die Berechnungen selbst werden von vier sog. Kernels durchgeführt: Der **Eye Ray Generator**-Kernel erzeugt für jeden Pixel einen vom Augpunkt ausgehenden Sichtstrahl und trägt diesen in einer Textur ein. Die Eingabe des **Traversers** ist die oben erwähnte 3D-Textur mit dem Gitter und der vom Generator erzeugte Strahl. Der Traverser

verfolgt nun den Strahl durch das Gitter; findet er einen Quader, der Dreiecke enthält wird dieser zusammen mit dem Strahl ausgegeben und der nächste Kernel, **Intersector** genannt, ist nun dafür zuständig, die Liste der Dreiecke zu durchlaufen und zu bestimmen, ob der Strahl tatsächlich eines der Dreiecke schneidet. Ist dies nicht der Fall, wird der ursprüngliche Strahl wieder vom Traverser weiterverfolgt, anderenfalls werden die Daten an den letzten Kernel, den **Shader** weitergegeben, der nun die eigentliche Schattierung der Oberfläche berechnet. Dies geschieht nach den üblichen Verfahren.
Neben diesem grundlegenden Raytracingverfahren ohne Sekundärstrahlen beschreiben und implementieren die Autoren auch Erweiterungen: Beim *Whitted* Raytracer, der klassischen Raytracern entspricht, werden vom **Shader** je nach Oberflächenbeschaffenheit (Brechung, Reflektion) weitere Strahlen ausgesendet, die nun wieder vom **Traverser** verfolgt werden. Der *Path Tracer* geht etwas anders vor, hier werden die Strahlen zufällig von Oberflächen ausgesandt, und solange verfolgt, bis diese auf Lichtquellen treffen. Ein hybrides Vorgehen wird mit dem *Shadow Caster* implementiert: Die Verdeckung der Oberflächen wird zunächt mit dem normalen Rendering-Verfahren ermittelt, und das Raytracing wird danach benutzt, um Schattenberechnungen durchzuführen.
Mangels verfügbarer Hardware können noch keine Geschwindigkeitsangaben gemacht werden, die Ergebnisse sind jedoch vielversprechend. In einer anderen Arbeit haben Carr et al. die Dreieck-Strahl-Schnittberechnung mit den Pixel-Shadern eine ATI Radeon 8500 implementiert (Pixel Shader 1.4) und kamen so auf eine Zahl von 114 Mio. Schnittberechnungen pro Sekunde. Ein für SSE optimierter Algorithmus schafft auf einer Pentium III 800 CPU etwa 20-40 Mio. pro Sekunde.

9.7.3 Physikalische Berechnungen

Bei der Rechenleistung der programmierbaren Hardware liegt es natürlich nah, diese Rechenleistung auch für Dinge zu nutzen, die eigentlich nichts direkt mit Grafik zu tun haben, aber hohe Rechenleistung benötigen. In diese Kategorie fallen auch physikalische Simulationen und tatsächlich gibt es Ansätze, die Grafikhardware für solche Aufgaben zu mißbrauchen:

CML-basierte visuelle Simulation von Flüssigkeiten und Gasen

In der Arbeit [Harris et al. '02] wird ein Verfahren zur Simulation von Vorgängen in Gasen und Flüssigkeiten wie z.B. Kochen dargestellt, das auf *Coupled-Map Lattices* (CMLs) beruht. CMLs sind eine Weiterentwicklung zellularer Automaten, deren einzelne Zellen im Gegensatz zu diesen reelle Werte speichern können und damit erheblich kleinere Gitter benötigen, um Vorgänge realistisch zu simulieren.
CMLs eignen sich gut, um dynamische Systeme zu simulieren. Wenige grundlegende, lokale Operationen pro Zelle reichen aus, um komplexe globale Effekte zu simulieren. Diese lokalen Operationen berechnen den Zustand einer Zelle neu, Eingabevariablen sind dabei der Zustand dieser und der sie umgebenden Zellen.
Die grundlegende Ähnlichkeit zwischen diesen parallelverarbeitenden Gitternetzen und der möglichen parallelen Ausführung von Pixelshadern in der Rasterebene und die Rechenleistung heutiger GPUs war die Motivation, die Berechnungen in Grafikhardware zu implementieren. Ein Problem dabei war die schwierige Programmierung der Grafikhardware mittels Assemblersprachen. Ein anderes war die beschränkte Rechengenauigkeit der Pixel Shader (Geforce3). Beide Probleme werden in absehbarer Zeit bzw. schon mit heutigen Grafikkarten (Geforce FX, Radeon 9700) gelöst sein, bis dahin ist die Simulation hauptsächlich für visuelle Ausgabe

gedacht, für numerische Ausgabe ist die Genauigkeit zu beschränkt.
Für das Simulationssystem werden Texturen, 2D oder 3D, je nach Dimension des Gitternetzes verwendet, um die Zustände der einzelnen Elemente abzuspeichern. Eine Textur bietet bis zu vier Kanäle, diese werden zur Zustandsspeicherung verwendet. Ein Pixel in der Textur entspricht dabei einem CML-Element. Eine Iteration eines CMLs ist die Neuberechnung aller einzelnen Elemente. Für eine solche Iteration werden evtl. mehrere Durchgänge (*Passes)* benötigt. Ein Durchgang gliedert sich in drei Phasen: Setup der Grafikhardware, Rendern der neuen Zustände, Kopieren des Renderergebnisses in eine Textur.
Beim Setup der Grafikhardware wird ein zu der Projektionsebene paralleles Rechteck gerendert, dabei muß darauf geachtet werden, dass ein Pixel der Datentextur genau einem Pixel im Bildspeicher entspricht, da sonst die Ergebnisse durch die Texturfilterung der Grafikhardware verfälscht werden. Diese Filterung kann aber auch zur Gewichtung zwischen Umgebungselementen benutzt werden. Um die Zustände der umgebenden Zellen auslesen zu können, werden die Texturkoordinaten mittels der Vertex-Shader entsprechend ausgerechnet. Die Pixel-Shader lesen dann die Eingabevariablen aus, berechnen daraus den neuen Zustand und schreiben ihn als Ausgabe in den Bildspeicher, der im dritten Schritt dann wieder in die Textur kopiert wird. Der Gesamtvorgang wird in Abbildung 9.15 dargestellt.

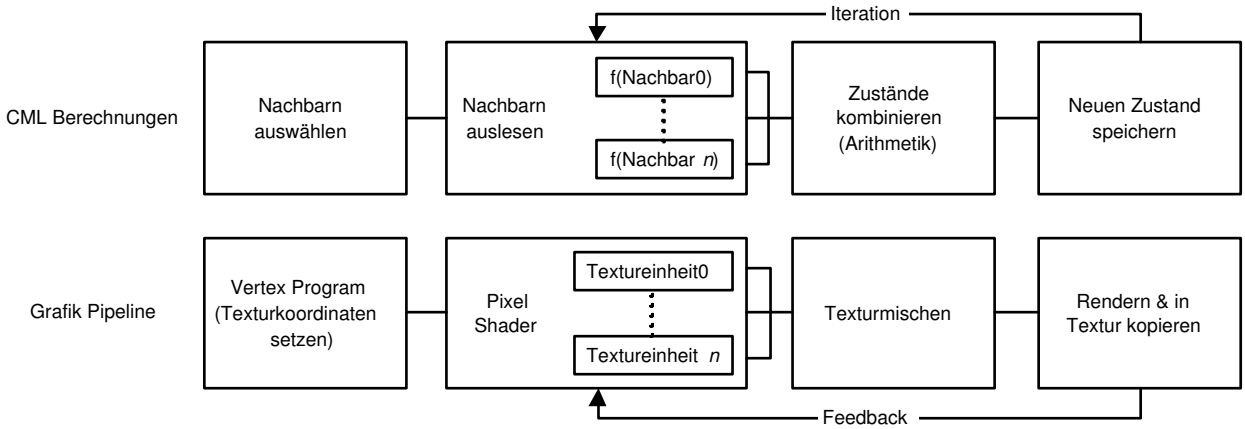


Abbildung 9.15: Übersicht des CML Systems und seiner Implementierung

Abbildung 9.16: CML-System in Aktion: Das linke Bild zeigt das CML-System in eine 3D-Engine integriert, um den Topf zum Kochen zu bringen; das rechte Bild zeigt mehrere Schritte in einer 2D-Simulation kochenden Wassers

9.8 Spezialhardware

9.8.1 PixelFlow

Die PixelFlow-Architektur[Eyles et al. '97], die an der Universität von North Carolina in Zusammenarbeit mit Industrie-Partnern entwickelt wurde, arbeitet wie die hier bisher vorgestellte Grafikhardware mit dem Rasterisierungsverfahren, um Bilder zu erzeugen. Bei der skalierbaren PixelFlow-Architektur geschieht das massiv parallel. Sie besteht aus bis zu 256 *Flow-Einheiten*, die aus einem Geometrie-, einem Rasterisierungsprozessor und der dazugehörigen Infrastruktur bestehen.
Die zu rendernden Geometriedaten werden auf die verfügbaren Flow-Einheiten aufgeteilt und jede dieser Einheiten berechnet ein Bild in der endgültigen Größe, das nur die Ihr zugewiesene Geometrie enthält. Diese Einzelbilder werden über ein *Image-Composition Network* zum endgültigen Ausgabebild zusammengesetzt. Dieses Zusammensetzen geschieht anhand der Tiefeninformationen der berechneten Punkte.
Zur Programmierung der Prozessoren wird eine *PFMan* entwickelte Shading-Hochsprache verwendet, die auf der Renderman Shading Language aufbaut. Die Gesamtprogrammierung des Pixel-Flow Systems geschieht über eine OpenGL-Implementierung mit spezifischen Erweiterungen.

9.8.2 SaarCOR Raytracing Hardware

Die bisher vorgestellte Hardware arbeitet wie alle PC-Grafikkarten nach dem Rasterisierungsverfahren. Dieses Verfahren, dessen Vorteil vor allem die Geschwindigkeit ist, hat auch einige Nachteile. So steigt bei komplexer werdenden Szenen und Effekten der Rechenaufwand stark an. Auch ist es sehr schwierig oder manchmal sogar unmöglich, manche natürlichen Effekte realistisch und in Echtzeit darzustellen. Das Raytracing-Verfahren bietet hier einige Vorteile, so skaliert es bei größeren Szenen wesentlich besser (logarithmisch mit der Anzahl der Dreiecke) und kann komplexere natürliche Phänomene realistischer darstellen. Der Nachteil dieses Verfahrens war aber der erhöhte Rechenaufwand und die benötigten Ressourcen. Es gab zwar Versuche, Raytracing in Hardware zu implementieren, aber die meisten scheiterten an den technischen Möglichkeiten (v.a. Speicherbandbreite).
An der Universität Saarbrücken wurde nun eine Raytracing-Hardwarearchitektur entwickelt, die die Nachteile weitgehende beseitigt und dabei die Vorteile beibehält, die SaarCOR-Architektur [Schmittler et al. '03]. Das größte Problem bisheriger Hardware-Architekturen war die benötigte Speicherbandbreite. SaarCOOR löst dieses Problem durch das ebenfalls an der Uni Saarbrücken entwickelte *Raytracing kohärenter Strahlen*. Dabei wird nicht immer nur ein Strahl verfolgt, sondern immer gleich ein Paket (Im Fall vor SaarCOR sind dies 64) benachbarter Strahlen. Die Szene ist mittels BSP-Baum in verschiedene Unterszenen aufgeteilt. Durchläuft nun einer der Strahlen eine Unterszene, werden alle Strahlen dieses Paketes auf Schnitte mit den in der Unterszene enthaltenen Dreiecken getestet. Durch dieses Verfahren müssen die benötigten Daten nicht so oft aus dem Speicher gelesen werden, sondern können in einem Cache innerhalb des Chips vorgehalten werden, da die räumlich benachbarten Strahlen sehr wahrscheinlich auch dieselben Dreiecke schneiden werden. Außerdem wird dadurch beim Shading der Textur-Cache besser ausgenutzt, da auch benachbarte Texturpixel sehr wahrscheinlich zeitlich sehr nahe nacheinander benötigt werden.
Die Architektur (Abbildung 9.17) ist in 3 Komponenten gegliedert: den *Raytracing-Core (RTC)*, der auch mehrfach vorhanden sein kann, die Strahlerzeugungs- und Shadingeinheit (*Ray Generation and Shading RGS)* und das Speicherinterface (*Memory Interface RTC-MI*).

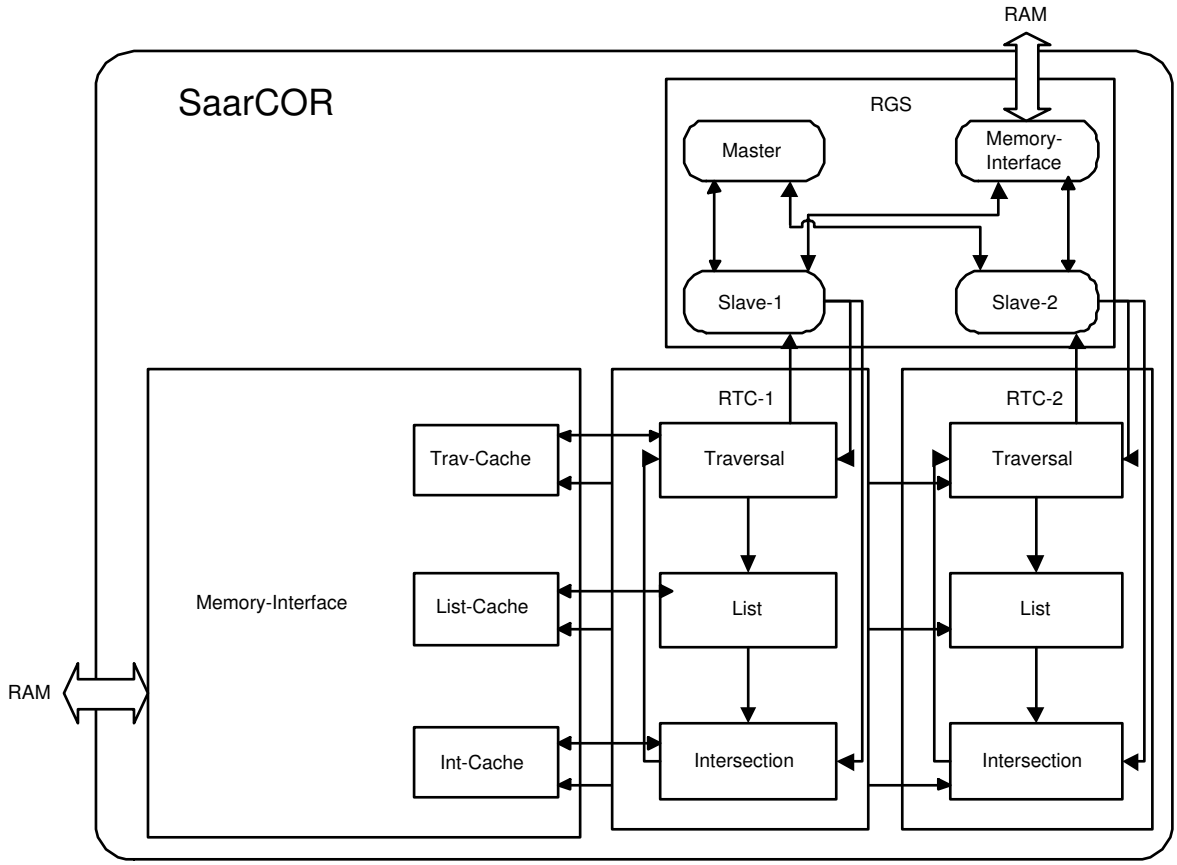


Abbildung 9.17: SaarCOR-Architektur

Die **RGS**-Einheit ist in drei Typen von Untereinheiten aufgeteilt. Es gibt eine *Master*-Einheit, die für die Verwaltung zuständig ist und den nächsten zu verfolgenden Strahl bestimmt. Für jeden im System vorhandenen RTC existiert innerhalb der RGS-Einheit ein Slave, der ein Pixel vom Master zugeordnet bekommt und die dazugehörigen Strahlen mit Hilfe seiner angeschlossenen RTC-Einheit solange verfolgt, bis die Farbe des Pixels bestimmt ist. Die Slave-Einheiten enthalten einen frei programmierbaren Mini-Prozessor, der neben der Steuerung der assoziierten RTC-Einheit auch das Shading der Oberfläche übernimmt, sobald die Strahlverfolgung einen Schnitt mit einem Dreieck ermittelt hat. Dieser Prozessor schickt dann je nach Oberflächenbeschaffenheit auch weitere Strahlen zur Verfolgung zum RTC. Neben diesen Einheiten verfügt der RGS noch über ein eigenes Speicherinterface.
Die Aufgabe der **RTC**s ist die schnelle Berechnung von Schnitten des Strahls mit Szenedreiecken. Der RTC besteht wiederum aus drei Einheiten, dem *Traverser*, der den Strahl von der RGS entgegennimmt und nun den BSP-Baum durchläuft, um eine Unterszene zu finden, die vom Strahl geschnitten wird. Ist dies der Fall, so werden von der *list unit* die Adressen der Dreiecke dieser Unterszene geholt und an die *Intersector*-Einheit weitergereicht, die nun die eigentlichen Schnittberechnungen durchführt. Die Resultate dieser Schnittberechung werden zurückgereicht an die Traverser-Einheit, die nun entweder den Baum weiter durchlaufen muß, oder die Ergebnisse an die Slave-Einheit im RGS zurückgibt. Die dritte Einheit innerhalb der SaarCOR-Architektur ist das Raytracing-Speicherinterface, dessen Aufgabe es ist, die für die RTCs benötigten Daten aus dem Speicher zu holen und auch in Caches vorzuhalten.
Die Architektur ist skalierbar ausgelegt, d.h. die Anzahl der RTC-Einheit kann der Anwendung

angepasst werden und bisherige Experimente zeigen, dass auch die erzielbaren Frameraten sehr gut mit der Anzahl der RTCs skalieren.
Die bisher mit diesem System erzielte Leistung ist ermutigend. Bei etwa gleichem Hardwareaufwand wie eine Geforce3 werden z.B. bei *Quake3*-Szenen auch mit dieser vergleichbare Frameraten erzielt, allerdings in Raytracing-Qualität. Die SaarCor-Engine benötigt dafür etwa 3/4 der Fließkommaleistung einer Geforce3 und nur ein Viertel der Speicherbandbreite. Es können daher günstigere SD-RAMs zum Einsatz kommen im Gegensatz zu DDR-RAM.

9.9 Ausblick

Die Entwicklung der Grafikkarten, vor allem in den letzten Jahren, ist beeindruckend. Die Einführung der programmierbaren Einheiten stellte einen Evolutionssprung dar. Jetzt ist es möglich, auch kompliziertere und beim Design der Grafikkarte nicht vorgesehene Effekte in Echtzeit zu berechnen und darzustellen. Vor allem die neueren Grafikkarten, deren Vertex- und Pixel-Shader DirectX9 unterstützen geben dem Programmierer viel Freiraum und so werden in nächster Zeit, wenn diese Grafikkarten zur Standardausrüstung gehören, beeindruckende Bilder in Echtzeit zum Alltag gehören. Die Grenzen hierbei liegen hauptsächlich in der Kreativität der Programmierer, die ihnen an die Hand gegebenen Möglichkeiten auszureizen.
Filme wie Monster AG, Shrek, Final Fantasy und andere haben eindrucksvoll die Möglichkeiten der computererzeugten Bilder demonstriert. Bis solche Filme allerdings in Echtzeit berechnet werden können ist es noch ein weiter Weg. Jedes der Filmbilder benötigte im Durchschnitt mehrere Stunden Rechenzeit. Die Datenmenge, die für solche Bilder benötigt wird, liegt bei etwa einen Gigabyte, jeweils für Geometrie- und Texturdaten. Im Vergleich zu dem was heutige Grafikkarten schaffen liegen da noch Welten dazwischen und selbst mit der Extrapolation des Wachstums der Grafikkarten mit dem Mooreschen Gesetz werden noch einige Jahre ins Land gehen, bis wir solche Bilder in Echtzeit berechnet sehen werden.
Erste Ansätze sind jedoch schon sichtbar, so zeigte NVidia schon eine Demo, in der Teile des Films Final Fantasy in quasi-Echtzeit (10 Bilder pro Sekunde) in einfacherer Qualität berechnet werden. Gerade auch die Firma NVidia hat sich das Motto *Cinematic Rendering* auf die Flagge geschrieben und wirbt damit für Ihre neuen Grafikkarten. Die Entwicklung wird sich in naher Zukunft wohl auf das Verbessern der programmierbaren Einheiten konzentrieren und der Grafikprozessor wird dem Hauptprozessor an Leistungsfähigkeit in nichts mehr nachstehen und ihn sogar übertreffen. Schon jetzt verfügen Grafikprozessoren um ein Vielfaches an Transistoren der Hauptprozessoren.
Die Entwicklungsgeschwindigkeit bei Grafikprozessoren übertrifft die bei Hauptprozessoren und damit auch das Mooresche Gesetz zur Zeit um ein Vielfaches und man darf gespannt sein, was sich hier in den nächsten Jahren noch alles tun wird.

Literaturverzeichnis

[ATI '03] ATI. ATI developer web site. 2003. http://mirror.ati.com/developer/index.html (gesehen 05/2003).

[Ecker '03] Martin Ecker. Programmable Graphics Pipeline. 2003. http://xengine.sourceforge.net (gesehen 05/2002).

[Eyles et al. '97] John Eyles, Steven Molnar, John Poulton, Trey Geer, Anselmo Lastra, Nick England und Lee Westover. PixelFlow: The Realization. 1997. http://www.cs.unc.edu/~pxfl/papers/PxFl-hwws97.pdf (gesehen 05/2003).

[Harris et al. '02] Mark J. Harris, Greg Coombe, Thorsten Scheuermann und Anselmo Lastra. Physically-Based Visual Simulation on Graphics Hardware. 2002. http://www.cs.unc.edu/~harrism/cml/ (gesehen 05/2003).

[Microsoft '02] Microsoft. Microsoft DirectX 9.0 Documentation. 2002. http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000410 (gesehen 05/2003).

[NVidia '03] NVidia. NVidia developer web site. 2003. http://developer.nvidia.com (gesehen 05/2003).

[Proudfoot et al. '01] K. Proudfoot, W. R. Mark, S. Tzvetkov und P. Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. 2001. http://graphics.stanford.edu/projects/shading/pubs/sig2001/ (gesehen 05/2002).

[Purcell et al. '02] T. Purcell, I. Buck, W. Mark und P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. 2002. http://graphics.stanford.edu/papers/rtongfx/ (gesehen 05/2002).

[Schmittler et al. '03] Joerg Schmittler, Ingo Wald und Philipp Slusallek. SaarCOR - A Hardware Achitecture for Ray Tracing. 2003. http://www.saarcor.de (gesehen 05/2003).

10 Verteilte Virtuelle Umgebungen

Zhixing Xue

10.1 Einleitung

10.1.1 Motivation

Eine verteilte virtuelle Umgebung, kurz DVE[1] ist eine gemeinsame synthetische Umgebung, die auf mehreren Rechnern läuft und durch ein Netzwerk verbunden ist. Sie soll viele selbstständige Benutzer unterstützen. D.h, die normale allein stehende virtuelle Umgebung, wie es heutzutage die schon weit entwickelten Spielesysteme sind, ist keine DVE. Kommunikationstechniken wie Chate oder Videokonferenz zählen auch nicht zu DVEs, da dort nur einfacher Texte bzw. Videosequenzen verteilt werden und keine Objekte gemeinsam benutzt werden. Andere Medien wie WWW oder Email sind auch keine DVE, weil alle Interaktionen in DVEs in Echtzeit ablaufen müssen.

Einsatzgebiet

Die Einsatzgebiete von DVEs schließen folgende Anwendungsbereiche ein: virtuelle Gemeinden, Multiplayer-Spiele, virtuelle Einkaufszentren, Fernstudium, kollaboratives Entwerfen und Konstruieren, militärische und industrielle Teamausbildungen. Die Abb. 10.1 stammt aus „Americas army“, in welchem die Teamarbeit der Soldaten trainiert werden soll.

Eigenschaften von DVEs

Unter anderem hat die DVE folgende fünf typische Eigenschaften:

- *Eine gemeinsame Wahrnehmung von Raum.* Der selbe Ort in der DVE sollte für alle Terminals, die mit dieser DVE verbunden sind, gleich sein, obwohl auf solchen Terminals unterschiedliche Applikationen laufen oder sie in verschiedenen Orten rund um die Welt liegen könnten.
- *Eine gemeinsame Wahrnehmung von Anwesenheit*, so genannter „Avatar“ von Teilnehmern. Wenn zwei oder mehr Benutzer zum gleichen Zeitpunkt am selben Ort sind, müssen sie die Änderungen von anderen Benutzern wahrnehmen können.
- *Eine gemeinsame Wahrnehmung von Zeit.* Alle Interaktionen in der DVE müssen in Echtzeit ablaufen.
- Die DVE muss *verschiedene Methoden der Kommunikation* haben. Das umschließt vor allem Texte, Audio und Video. Die extra Eingabegeräten bieten auch viele andere Möglichkeiten.

[1] Distributed Virtual Environment

Abbildung 10.1: Americas Army, militärische Teamausbildungssystem

- Die DVE muss *Verteilt* sein. D.h, es ist eine dynamische, durch Benutzer beeinflussbare, dynamische Umgebung.

Probleme bei DVE

In der Implementierung von DVEs gibt es viele Probleme. Da eine DVE häufig in einem Netz wie dem Internet verteilt ist, ist die *geringe Bandbreite* ein großes Problem. Wegen *Latenz* kann der Computer nur wissen, was in der Vergangenheit passierte. Das ist wiederum ein großes Problem für große Netze wie dem Internet, aber auch ein kleines Problem in lokalen Netzen. Für die Mess- und Kontrolldaten ist die *Fehlertoleranz* sehr wichtig. Für Audio- oder Videodaten sind aber kleinere Datenverluste erlaubt. Um *heterogene Netzwerke* zu implementieren, stehen die systemunabhängigen Sprachen wie Java und Protokolle wie VRML zur Verfügung. Je mehr Benutzer es in der DVE gibt, desto mehr muss der Server rechnen. So wird die Leistung der DVE niedriger. Ein Beispiel für dieses *Erweiterbarkeitsproblem* ist wie folgt: Wenn ein Benutzer 30 mal pro Sekunde seine Position mit 50 Byte für die Beschreibung der Aktualisierung und zusätzliche 42 Byte zum Server sendet, werden also insgesamt $(42Byte/Aktualisierung + 42Byte/Overhead) * 30/Sekunde = 22,080\ Bit\ pro\ Sekunde$ für nur einen Benutzer benötigt. D.h, ein Server mit 1Mbit/Sek kann nur 40-50 Benutzer unterstützen. Um die Erweiterbarkeit zu vergrößern kann man einerseits die unbenötigten Informationen herausfiltern. Andererseits kann eine andere Kommunikationsarchitektur entworfen werden.

10.1.2 Kommunikationsarchitektur

Client/Server Architektur

Es gibt zwei extreme Fälle von Kommunikationsarchitekturen eine von ihnen ist die Client/Server-Architektur, wie in Abb. 10.2 gezeigt. In dieser Architektur gibt es einen Computer, auch Server genannt, der mit allen anderen Computern verbunden ist. Die Clients können nicht untereinander kommunizieren, sondern nur über den Server. Es ist einfach zu implementieren und bei kommerziellen Spielen sehr verbreitet. Es bietet hohe Sicherheit, da es nur einen

Autorisierungspunkt beim Server gibt. Weil alle Berechnung im Server allein erledigt werden, kann es kaum erweitert werden. Das bedeutet, es kann nicht gleichzeitig sehr viele Benutzer unterstützen. Es hat auch schlechte Fehlertoleranz und der Server ist hierbei der Flaschenhals. Zur Verbesserungen kann man „Time out“ und „Caching“-Mechanismen verwenden. Wenn der Client selbst die Aktualisierungen der Objekte berechnet, kann die Leistung dadurch noch verbessert werden.

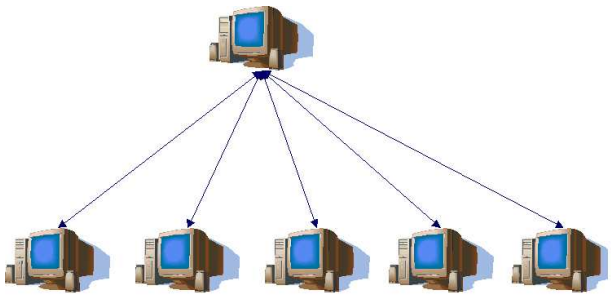


Abbildung 10.2: Client/Server Architektur

Peer/Peer Architektur

Eine andere Architektur ist die so genannte Peer/Peer Architektur(siehe Abb. 10.3), in welcher es kein Server gibt und alle Computer miteinander kommunizieren können. Diese Architektur wird hauptsächlich in Forschung und Armee angewendet. Da die Berechnung auf alle Computer verteilt ist, wird ein Flaschenhals vermieden. Auch die dynamische Einstellung ist möglich, d.h, wenn ein Computer zu viele Rechenaufgaben hat, kann der andere Computer die restliche Arbeit übernehmen. Aber trotz solcher Vorteile ist diese Architektur schwieriger zu implementieren. Auch die Synchronisierung und Sicherheit stellen bei dieser Architektur große Probleme dar. Man muss sich um alle Computer im Netz kümmern. Da jeder Computer seinen eigenen Datenbestand hat, geht dieser bei Ausfall des Computers für die anderen verloren.

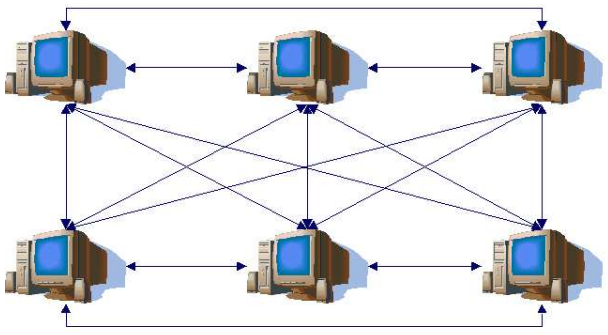


Abbildung 10.3: Peer/Peer Architektur

Hybride Architektur

In der Praxis kombiniert man häufig oben vorgestellte Architekturen. In Abb. 10.4, wird die hybride Architektur gezeigt. Die wichtigsten Daten können beim Server aufbewahrt werden. Mit

Hilfe des Servers ist die Synchronisierung auch einfacher zu implementieren. Um die Erweiterbarkeit zu erzielen, kann hier der Multicasting-Mechanismus eingesetzt werden, welche die gleichzeitige Datensendung an mehrere Computer ermöglicht. Dabei werden die so genannten „Multicasting Router“ (kurz MRouter) eingesetzt, denen eine Gruppe von Computern zugewiesen ist. Wenn ein MRouter eine Nachricht bekommt, schickt er sie an andere benachbarte MRouter, sowie an alle Computer in seiner Gruppe. Wenn eine Nachricht einmal abgeschickt ist, kann sie von mehreren Computern empfangen werden. Dadurch ist die Kommunikation effizienter.

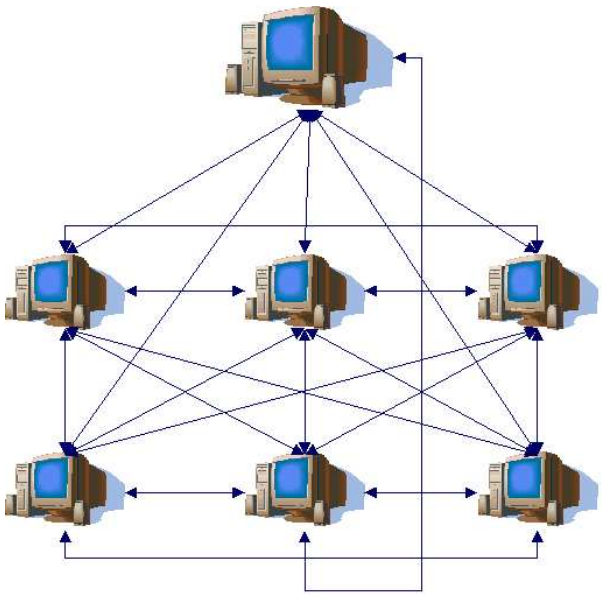


Abbildung 10.4: Hybride Architektur

10.2 Protokolle

10.2.1 Internet Protokolle

Fast alle Protokolle, die für DVEs entworfen wurden, basieren auf TCP/IP bzw. UDP. Mit TCP wird eine Verbindung zwischen zwei Computer aufgebaut, wobei die Datenübertragung gesichert wird, indem die Daten immer wieder auf Richtigkeit überprüft werden. UDP dagegen baut keine Verbindung auf. Es ist kein sicheres Protokoll. Aber da es schneller als TCP ist, passt es ganz ideal zu den Audio- oder Videodaten, welche einen kleinen Datenverlust erlauben.

10.2.2 VRML

Die Idee von VRML (Virtual Reality Modelling Language)
[Web3DC ’03b] wurde im Frühling 1994 bei der ersten WWW-Konferenz in Genf diskutiert. 1995 wurde die Version 1.0 und einige VRML-Browser veröffentlicht, welche nur statische Szene boten. Nur eineinhalb Jahre später wurde Version 2.0 auf der SIGGRAPH’96 präsentiert. Es fügte Verhalten und Benutzerinteraktion hinzu und wurde als offizieller ISO-Standard in VRML97 umbenannt. Es wurde anfänglich nicht im Hinblick auf DVEs entworfen. Bis Version 2.0 passt es noch nicht perfekt zu DVEs. Daher wurde es erweitert.

Eine Erweiterung ist VSPLUS, welche durch die Event-Verteilung die Interaktion zwischen Benutzern ermöglicht. In dem von SONY entwickelten „Living Worlds“, [Web3DC ’03a] kann jedes Objekt in dieser DVE einen Besitzer haben. Wenn sich ein Benutzer einloggt, kontrolliert er ein Menschobjekt, welches von ihm selbst gesteuert wird. Wenn er sich ausloggt, bleibt das Objekt weiter in der DVE, nur wird es nun vom Computer weiter gesteuert. Auch wenn der Benutzer in ein Auto ein- oder aussteigt, wird der Besitzer des Autos gewechselt. Auf diese Weise kann die DVE lebendig aussehen.

10.2.3 VRTP

Ein noch laufendes Projekt ist VRTP (Virtual Reality Transfer Protocol) [Brutzman ’03]. Es ist eine Erweiterung von VRML und ist ein heterogenes Protokoll. Dabei können verschiedene Typen von Daten gesendet werden. Der Monitoring-Mechanismus wird in VRTP integriert. Dank automatischen Auswählens des Protokolls kann ein Computer im Netz Client, Server oder Peer sein. Das große Ziel dieses Protokolls ist es, die gleiche Rolle für DVE wie HTTP für Webseiten zu spielen. D.h, in der Zukunft kann man direkt im Browser die DVE-Adressen wie vrtp://..... eingeben und ohne zusätzliche Applikation die DVE benutzen.

10.2.4 DIS

DIS [Locke ’03] (Distributed Interactive Simulation) wurde 1985 von der US-Armee für die Simulation eines Krieges mit Panzern, Flugzeugen usw. entworfen. Die Kriegssimulation soll die Soldaten von Panzerführer bis Kommandeur ausbilden. Es beginnt 1989 und wurde 1993 ein IEEE-Standard und beeinflusste viele andere Protokolle.

In DIS werden 27 Typen von PDU (protocol data units) benutzt. Die PDU sind kurze Nachrichten, die den Zustand eines Objektes beschreiben, inklusive der Position, Richtung und Geschwindigkeit. Es ist heterogen, verschiedene Typen von Benutzern und Maschinen werden unterstützt. DIS ist ein zustandloses Protokoll, d.h, wenn ein Computer eine PDU bekommt, benötigt er nicht die vorherige PDU. HTML ist beispielsweise auch ein zustandloses Protokoll. In DIS müssen alle Objekte ihren Zustand an alle Computer schicken. Diese Nachrichten gelten als Puls, also so genannter „Heartbeat “. DIS benutzt die Peer/Peer Architektur. Aber wegen des Heartbeats ist es schlecht erweiterbar. Es wird geschätzt, dass für 100.000 Clients eine Verbindung von bis zu 375Mbit/s Bandbreite zu jedem Client benötigt wird.

Unter anderem wird die Methode „Dead Reckoning“ eingesetzt, um die Größe der Aktualisierungen zu minimieren. Dazu braucht man zwei Algorithmen, die exakte Simulation und eine Extrapolation-basierte Simulation. Es muss gelten, dass die Extrapolation-basierte Simulation weniger Rechenzeit als die exakte Simulation benötigt. Im Server, der das Verhalten von Objekten simuliert, laufen die exakte Simulation und die Extrapolation-basierte Simulation, während in den anderen Clients nur die Extrapolation-basierte Simulation läuft. Basierend auf die exakte Simulation extrahiert der Server die Parameter des Objekts, und schickt den Zustand des Objekts zu diesem Zeitpunkt und die späteren Parameter als Teil von PDU an alle anderen Computer. Das Objekt wird von den anderen Computern auf die Extrapolation-basierte Art simuliert.

Diese Methode wird in Abb. 10.5, Abb. 10.6 und Abb. 10.7 gezeigt. Dabei werden der Anfangspunkt, sowie die Geschwindigkeiten übermittelt. In Abb. 10.7 werden die Bewegungsbahnen kontinuierlich gemacht.

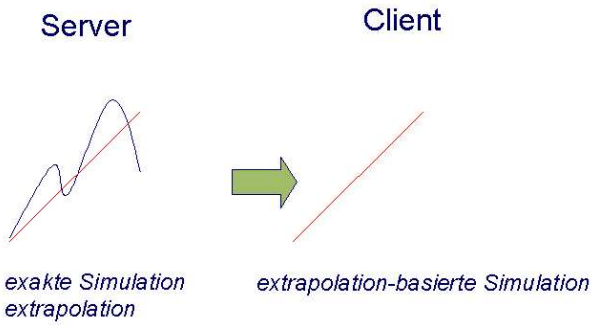


Abbildung 10.5: Bewegungsbahn eines Objekts im ersten Heartbeat

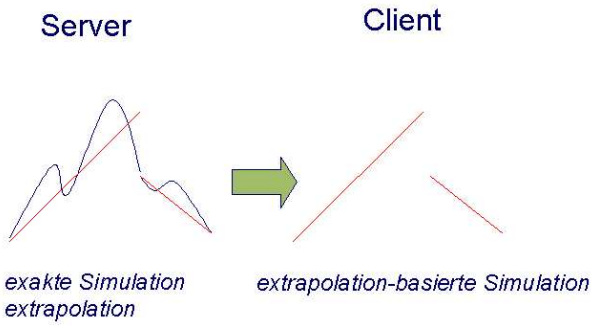


Abbildung 10.6: Bewegungsbahn eines Objekts im zweiten Heartbeat

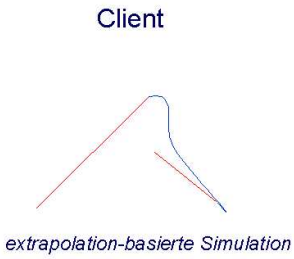


Abbildung 10.7: Die Bewegungsbahn dieses Objekts wird bei Client kontinuierlich gemacht

10.2.5 DWTP

DWTP (Distributed Worlds Transfer and Communication Protocol) wurde von der Gesellschaft für Mathematik und Datenverarbeitung in Deutschland entworfen. Das wichtige Ziel ist die Erweiterbarkeit. Es soll applikations-unabhängig und heterogen sein. Dabei werden verschiedene Typen von Daten gesendet.
DWTP benutzt IP-Multicasting mit UDP. Um die DWTP besser kennenzulernen ist es sinnvoll, dass wir zuerst das Problem bei IP-Multicasting mit UDP betrachten, weil es in vielen anderen Protokollen wie ISTP[Laboratories '03], RAMP[Koifman & Zabele '96] und RMP[Team '03] auch gefunden werden kann. Solche Protokolle benutzen so genannte NACKs (Negative acknowledge messages). Die Clients schicken die Nachrichten also die NACK zu dem Server, und fordern die Daten erneut an, wenn die erwarteten Daten nicht empfangen werden. Bei einem überlasteten MRouter kann es passieren, dass viele Computer die Daten nicht bekommen. Wenn diese alle gleichzeitig die NACKs schicken, bekommt der MRouter sehr viel überflüssige Informationen. Das kostet natürlich auch Bandbreite. Um solche Fälle zu vermeiden, lassen manche Protokolle die Clients eine zufällige kurze Zeit warten, damit diese nicht gleichzeitig die NACKs schicken.
in DWTP gibt es kein NACK, nur eine oder einige Empfänger, die so genannten „Demons ". Diese schicken ACKs (acknowledgement message) zu den Clients, was bedeutet, dass der Client bestimmte Daten bekommt haben soll. In DWTP werden u.a. folgende drei Demons eingeführt:

- *Ein World Demon* transportiert die kompletten Daten zu neuen Clients.
- *Ein Recovery Demon* kümmert sich um die sichere Übertragung zum Client, der die Daten nicht richtig bekommt hat.
- *Ein Reliability Demon* kontrolliert auf Fehler bei der Übertragung.

Der Client entdeckt einen Fehler, wenn er nicht die Daten bekommt, aber die entsprechende ACK. Daraufhin schickt er eine Anforderung zum Recovery-Demon und bekommt von ihm per sicherem Protokoll wie z.B. TCP die richtigen Daten. Wenn der Recovery-Demon selbst die Daten nicht bekommt, müssen die Daten vom World-Demon neu aufgefordert werden.

10.2.6 Mu3D

Mu3D (Multi-User 3D Protokoll) ist von Galli und Luo in der Universität von Balearic Islands als Teil des kollaborativen architektonischen Design [Galli & Luo '99] entworfen worden. Er ist ein auf VRML basierten Editor, der das Online-CAD-Zeichnen von Designteams ermöglicht. Im Gegensatz zu DWTP wird in Mu3D nur Aktualisierungen von Objekten gesendet. Das dabei entstehende Problem wird in Abb. 10.8 gezeigt. Wenn die Veränderung von Client 2 vor der von Client 1 an Client 3 gelangt, dann gibt es bei Client 3 eine Kollision.
Um ein solches Problem zu vermeiden, muss der Client, der eine Veränderung an einem Objekt vornehmen will, zuerst die Kontrolle dieses Objektes bekommen, indem er die Erlaubnis von allen anderen Clients anfordert. Nur wenn alle anderen Clients zustimmen, kann der erste Client das Objekt verändern und danach die Aktualisierung des Objekts an alle Clients schicken. Schließlich gibt es die Kontrolle über dieses Objekt ab, und erst dann können es andere Clients auf gleiche Weise verändern.

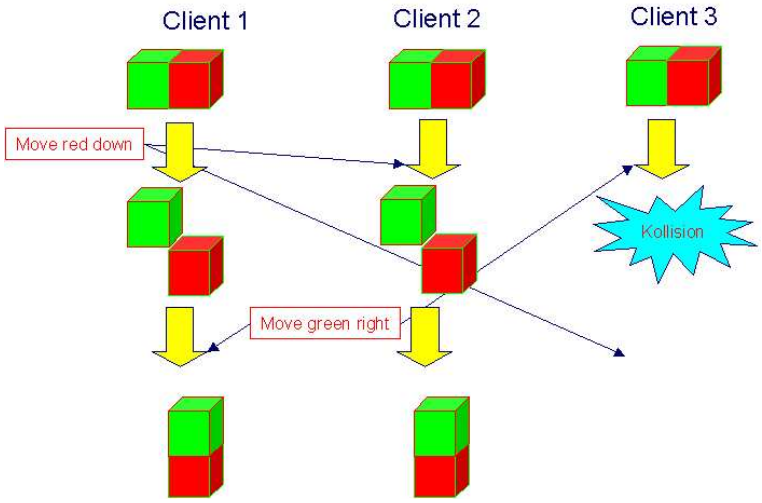


Abbildung 10.8: Kollision entsteht, wenn die Aktualisierung ohne Kontrolle gesendet wird

10.3 Unterteilung der DVE

Da die Bandbreite begrenzt ist, wird das System effizienter, wenn nur die tatsächlich benötigten Daten an andere Computer geschickt werden. Ein Beispiel hierfür ist ein Benutzer in einem Zimmer. Dieser braucht nur die Aktualisierung aller Benutzer in demselben Zimmer, und kann die Aktualisierung von anderen Benutzern ignorieren. Manche Techniken werden auch in anderen Bereichen der Computergrafik eingesetzt und werden deshalb hier nur kurz vorgestellt.

10.3.1 Zellraster

Bei dieser Methode wird der Raum in viele kleinere Zellen unterteilt. Nur die Aktualisierungen von benachbarten Zellen werden betrachtet. Ein Kollisionstest wird durch ein Zellraster beschleunigt, weil nur zwischen den in derselben Zelle liegenden Objekte getestet werden muss. In Abb. 10.9 werden nur die rundumliegende 5x5 Raster für die Zelle in der Mitte berücksichtigt.

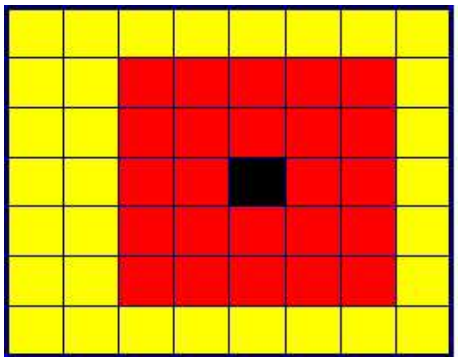


Abbildung 10.9: Nur die rundumliegende 5x5 Raster werden für die Zelle in der Mitte berücksichtigt

10.3.2 Bounding-Boxes

Eine Bounding-Box erhält immer nur den kleinsten Raum, der alle Objekte innerhalb der Bounding-Box umschließt. In Abb. 10.10 wird gezeigt, dass zwei minimale Bounding-Boxes zwei Objekte umschließen. Der Kollisionstest kann mit Hilfe dieser Methode beschleunigt werden, weil nur die Objekte, die in derselben Bounding-Box liegen, getestet werden müssen.

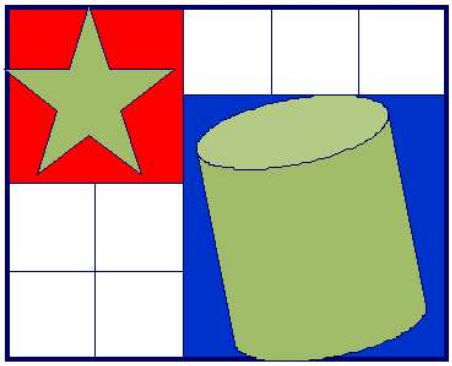


Abbildung 10.10: Boundingbox ist der kleinste Block, der ein Objekt völlig einschließt

10.3.3 BSP-Trees

BSP steht für „Binary Space Partitioning“. Der Raum wird bei BSP-Trees immer in zwei kleinere Räume unterteilt. Durch die Unterteilung baut sich ein Baum auf. In Abb. 10.11 wird der ganze Raum mit Linien f, g und h in die Räume von A bis E unterteilt. Es entsteht ein Baum wie in Abb. 10.12. So kann der gewünschte Raum schnell, in $\log n$, aufgefunden werden.

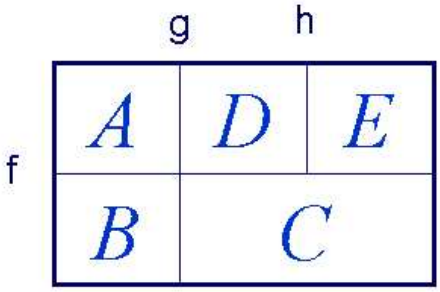


Abbildung 10.11: Ein Raum wird durch Linien f, g, h in kleine Räume A, D, E, B, C unterteilt

10.3.4 Zellen und Portale

Mit einem BSP-Tree kann man den Raum in einer Baum-Struktur speichern. Mit einer Technik wie „Zellen und Portale“ kann man die Sichtbarkeit eines Punktes in dem Raum bestimmen. Es ist geeignet für Szenen mit den Räumen, die von anderen Räumen nur durch die Löcher gesehen werden können. Solche Räume sind z.B. Gebäude, Fahrzeuge und Höhlen. Die Löcher in solchen Räumen, hier „Portale“ genannt, sind Fenster, Tore oder kleine Durchgänge. Die Räume, hier

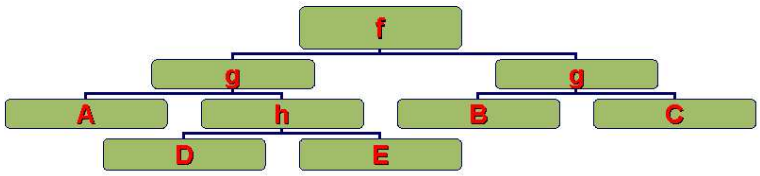


Abbildung 10.12: Dabei entstehnde Baumstruktur

„Zellen“ genannt, sind als Polygone dargestellt. Durch die geometrische Berechnung kann man das Bild von einem Aussichtspunkt bestimmen, wie es in Abb. 10.13 gezeigt wird.

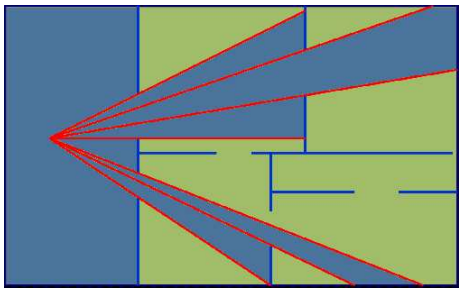


Abbildung 10.13: Dadurch ergibt sich die Sichtbarkeit des links liegende Punktes

10.3.5 Level Of Detail

In VRML können einige alternative Szenengraphen für dasselbe Objekt in einem LOD-Knoten definiert werden. Die Szenengraphen enthalten jeweils unterschiedliche Detailstufen eines Objektes. Je näher der Benutzer an das Objekt herankommt, desto detaillierter wird das Objekt dargestellt. In dem Feld *range* des LOD-Knotens ist eine Liste von zunehmenden Distanzen spezifiziert, welche das Intervall von Distanzen beschreibt. In dem folgenden Beispiel wird der Szenengraph SG1 für das Intervall 0 bis 10 Meter ausgewählt, der Szenengraph SG2 für 10 bis 100 Meter, SG3 für 100 bis 1000 Meter. Schließlich ist SG4 für die Distanz über 1000 Meter zuständig.

```
Beispiel LOD {
  range [ 10, 100, 1000]
  level [
    USE SG1,
    USE SG2,
    USE SG3,
    USE SG4] }
```

10.3.6 Aura

In Systemen wie MASSIVE und DIVE wurde eine Technik vorgestellt, welche dadurch motiviert ist, dass man nur diejenigen Dinge in seinem Sichtfeld sehen kann, während man Geräusch auch hinter einem hören kann. Mit anderen Worten, man nimmt Sinnesstimuli in räumlichen

Bereichen wahr, welche zueinander nicht identisch sind. Die Folgenden werden mit einem Mensch-Objekt verbunden:

- Aura: 3D-Raum, in dem der Benutzer mit anderen Objekten kommunizieren kann. Z.B. können zwei Benutzer nur kommunizieren, wenn ihre Auren zusammen sind.
- Fokus: 3D-Raum, in dem der Benutzer Nachrichten wahrnehmen kann.
- Nimbus: 3D-Raum, in dem die von einem Benutzer ausgegebene Nachrichten empfangen werden können.

Der Nimbus bzw. Fokus eines Objekts kann für verschiedene Medien unterschiedlich sein. Zum Beispiel ist der visuelle Fokus einer Person direkt vor der Person, während der akustische Fokus um die Person herum liegt.

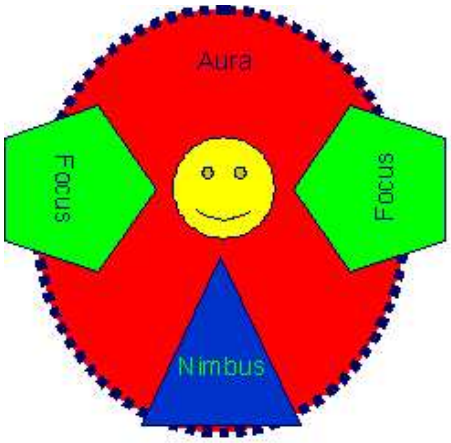


Abbildung 10.14: Fokus, Nimbus sowie Aura

10.4 Zusammenfassung und Ausblick

In dieser Ausarbeitung wurde der Begriff der „verteilten “ virtuellen Umgebung, sowie zugehörige Protokolle und Techniken vorgestellt. Die Technik ist relativ jung. Obwohl einige internationale Standards schon entstanden sind, ist die DVE-Technik immer noch nicht reif. Andere für große Datenströme geeignete Protokolle, wie RSVP, RTSP und RTP, sind in dieser Ausarbeitung nicht eingeschlossen.
Zum Schluss werfen wir ein Blick auf die Herausforderungen, die von zukünftigen DVEs bewältigt werden müssen:

- Spezifikationsprachen und Modellierwerkzeuge: Sprachen wie VRML sind aufwändiger zu erlernen als HTML. Mit modernen Modellierwerkzeugen soll das Erzeugen von Objekten für eine in DVE nicht schwieriger sein als die traditionelle Textverarbeitung. Um den Massenmarkt zu erreichen, müssen bedienungsfreundliche gebietabhängige Modellierungswerkzeuge entwickelt und integriert werden.
- Komponenten: Das Design der interaktive 3D-Modelle ist teuer. Amortisation dieser Kosten ist nur durch mehrfache Wiederverwendung möglich. Ein standardisiertes System für anpassungsfähige 3D-Bestandteile bietet diese technische Grundlage.

- Anpassungsfähigkeit: DVEs erfordern eine Anpassungsfähigkeit an die Graphikleistung eines Rechners zum 3D-Modelle und Anpassungsfähigkeit an die vorhandene Netzwerkstruktur mit Rücksicht zum Netzwerkanschluss. Um die Rechnervielfalt des Internets zu berücksichtigen, müssen Algorithmen und Protokolle so entwickelt werden, das es sich den vorhandenen Betriebssystemen anpasst.
- Immersion: Realismus ist nicht genug. Nicht lineare erzählende Elemente sowie Echtzeitkommunikation lassen Benutzer erfahren, dass sie ein Teil der DVE sind. Mehr Experimente von Entwicklern und Geschichtenerzählern sowie systematische Auswertungen der Welten ist erforderlich

Literaturverzeichnis

[Brutzman '03] Don Brutzman. Virtual Reality Transfer Protocol Homepage, 2003. http://www.stl.nps.navy.mil/~brutzman/vrtp/ (gesehen 07/2003).

[Diehl '00] Stephan Diehl. *Distributed Virtual Worlds: Foundations And Implementation Techniques Using VRML, Java, And Corb*. Springer-Verlag Berlin Heidelberg, Heidelberg, 2000.

[Galli & Luo '99] R. Galli und Y. Luo. In *Mu3D: A Causal Consistency Protocol for a Collaborative VRML Editor*. Symposium on the Virtual Reality Modeling Language VRML99, ACM SIGGRAPH, 1999.

[Koifman & Zabele '96] Alex Koifman und Stephen Zabele. Ramp: A Reliable Adaptive Multicast Protocol. In *INFOCOM (3)*, S. 1442–1451, 1996.

[Laboratories '03] Mitsubishi Electric Research Laboratories. The Interactive Sharing Transfer Protocol, 2003. http://www.merl.com/projects/opencom/WWW/istp.html (gesehen 07/2003).

[Lee et al. '02] Dongman Lee, Mingyu Lim und Seunghyun Han. ATLAS: a scalable network framework for distributed virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments*, S. 47–54. ACM Press, 2002.

[Locke '03] John Locke. An Introduction to the Internet Networking Environment and SIMNET/DIS, 2003.

[Team '03] IDDS Development Team. Reliable Multicast Protocol, 2003. http://salem.cs.depaul.edu/~ehab/Docs/rmp_ps.html (gesehen 09/2003).

[Web3DC '03a] The Web3D Consortium Web3DC. Living Worlds Working Group, 2003. http://www.web3d.org/WorkingGroups/living-worlds/ (gesehen 07/2003).

[Web3DC '03b] The Web3D Consortium Web3DC. Virtual Reality Modelling Language, 2003. http://www.web3d.org/vrml/vrml.htm (gesehen 07/2003).

11 Visuelle Modellierung

Bert Völker

11.1 Einführung

Digitalisierte Bilder werden seit Beginn der Computergrafik verwendet, um visuelle Informationen zu speichern. Da ein Bild - speziell ein Foto - oder aber auch Video nur begrenzt Informationen über eine aufgenommene Szene enthält, besteht oft der Wunsch mehr Informationen aus diesen zu extrahieren. An dieser Stelle tritt erstmals der Begriff der visuellen Modellierung auf. Diese befasst sich mit der Rekonstruktion von 3 dimensionalen Objekten aus einer Vielzahl von digitalisierten Bildern bzw. Videoaufnahmen.
Ein menschlicher Betrachter hat die Fähigkeit intuitiv aus einem Foto 3 dimensionale Objekte zu extrahieren. Da einem Computer diese Intuitionen fehlen, sind für die rechnergestützte Rekonstruktion mehrere Aufnahmen des zu rekonstruierenden Objektes aus unterschiedlichen Blickwinkeln notwendig. Je mehr verschiedenen Ansichten des zu rekonstruierenden Objektes zur Verfügung stehen, desto bessere Resultate liefern die in dieser Arbeit vorgestellten Rekonstruktionsverfahren.
Im Folgenden werden einige Verfahren zur Rekonstruktion von 3 dimensionalen Daten aus 2 dimensionalen Foto- / Videoaufnahmen vorgestellt sowie Problem bzw. Grenzen dieser Verfahren analysiert. Weiterhin wird die Echtzeitfähigkeit einiger Verfahren diskutiert und Ausblicke für einige Anwendung gegeben.

11.2 Grundlagen

Bevor die verschiedenen Rekonstruktionsverfahren, um aus 2D Daten 3D Daten zu extrahieren, vorgestellt werden, soll erst einmal die Bedeutung einiger grundlegender Begriffe definiert werden.

11.2.1 Was ist ein Voxel?

Ein Voxel[1] ist im Vergleich zum Pixel[2] die kleinste beschreibbare Einheit im 3 dimensionalen Raum. Wie bei einem Pixel enthält ein Voxel einige Parameter um es im Raum zu charakterisieren. Zu ihnen zählen:

- Position im 3 dimensionalen Raum, bestehend aus x-y-z-Koordinaten,
- Farbeigenschaft, je nach verwendeter Farbkodierung zum Beispiel: RGB,
- Sichtbarkeit, gibt an ob ein Voxel sichtbar (gefüllt) oder unsichtbar (leer) ist.

[1] Abkürzung für Volumeelement
[2] Abkürzung für Pictureelement

Es gibt zwei verschiedene Definitionen eines Voxels: zum einen wird es als *ein* Punkt im Raum beschrieben, zum anderen als ein Würfel mit 8 Stützpunkten. Im weiteren Verlauf wird ein Voxel mit der letzteren Definition - als ein Würfel mit 8 Stützpunkten aufgefaßt. Eventuell können auch noch mehr Parameter, wie zum Beispiel die Lichtcharakteristika des im Voxel befindlichen Materials, gespeichert werden:

- Reflektion (stark oder schwach),
- Glanz (matt oder glänzend),
- bei Lichtdurchlässigen Material (klar oder diffus).

Diese zusätzlichen Parameter sind für photorealistische Darstellungen nötig, wenn die Szene gerendert werden soll, nachdem Veränderungen vorgenommen wurden, zum Beispiel wenn die Lichtverhältnisse geändert werden.

11.2.2 Polygon vs. Voxel

Nachdem die Definition des Voxels in Kapitel 11.2.1 gegeben wurde, stellt sich die Frage wie ein Objekt durch Voxel dargestellt werden kann und ob es Vorteile bzw. Nachteile gegenüber einer Polygondarstellung gibt.

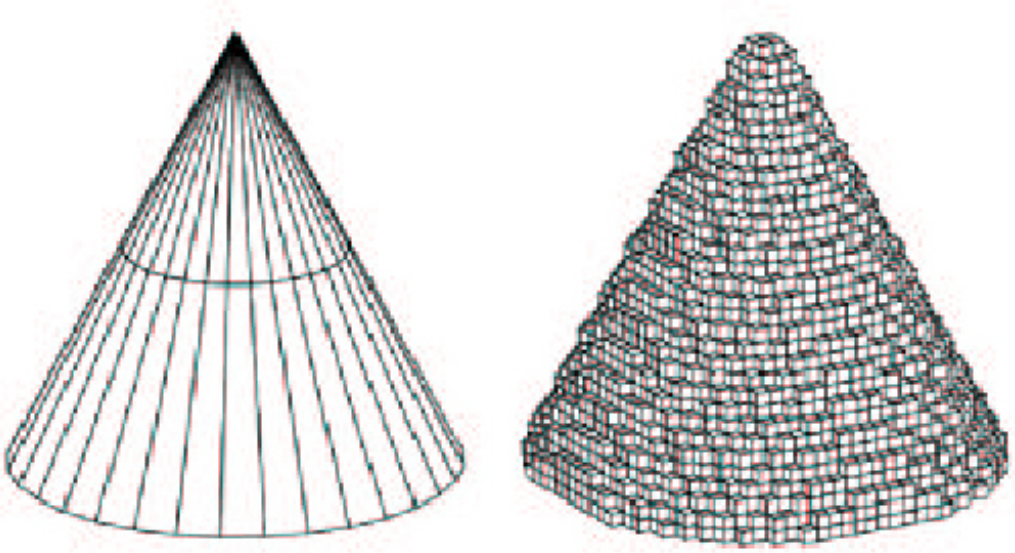


Abbildung 11.1: Ein Kegel als Polygon (links) und durch Voxel (rechts) dargestellt. Um bei dem Voxelkegel eine glattere Oberfläche zu generieren, muß das Voxelraster hinreichend klein gewählt werden.

In Abbildung 11.1 ist ein Polygonkelgel (links) und ein Voxelkegel (rechts) abgebildet. Soll ein Körper durch die Voxeltechnik beschrieben werden, müssen zunächst die zu erfassenden Ausmaße festgelegt werden. Es wird schnell klar das die Oberfläche des Voxelkegels stark von der Größe der Voxelrasterung abhängt. Je kleiner die Voxelgröße gewählt wird, desto glatter wird seine Oberfläche. Im abgebildeten Kegelbeispiel ist von einer Auflösung von 32x32x32

Voxel ausgegangen. Durch die Vorgabe der Voxelauflösung kann die Zahl der gesamten Voxel und des erforderlichen Speichers berechnet werden:

$$erfoderlicher\ Speicher = \frac{Kantenlänge\ der\ Szene^3 * Speicher\ pro\ Voxel}{Kantenläng\ eines\ Voxels^3}$$

Der erforderliche Speicher nimmt also mit der dritten Potenz zu, wenn die Genauigkeit zunimmt. Halbiert man die Kantenlänge eines Voxels, steigt die erforderliche Speichermenge auf das achtfache an.
Wird nur der Speicher betrachtet, so sind Polygonkörper Voxelkörpern vorzuziehen. In der weiteren Ausarbeitung werden jedoch auch Verfahren beschrieben, die mit der Voxeltechnik arbeiten. Anhand eines später beschriebenen Octrees kann der Speicherplatz der Voxeltechnik effizient verkleinert werden. Dazu werden die Voxel in einer Baumstruktur abgespeichert.

11.3 3D Rekonstruktionsverfahren

11.3.1 Volumenschnittverfahren

Wird ein von einer Kamera aufgenommenes Bild unter geometrischen Aspekten betrachtet, so kann von einem Sichtkegel gesprochen werden, dessen Spitze von der Kamera ausgehend die gesamte aufgenommene Szene aufspannt. Die Szene wird dabei auf die, nahe der Sichtkegelspitze befindlichen, Projektionsfläche abgebildet. Im weiteren Verlauf wird der Sichtkegel weiter eingeschränkt auf den Sichtkegel der Silhouette des zu rekonstruierenden Objektes. Liegen verschiedene Ansichten eines Objektes vor, so können aus jeder dieser Ansichten die Silhouetten-Sichtkegel bestimmt werden und anschließend deren Schnitt bestimmt werden. Das so beschriebene Volumen wird als *visuelle Hülle* bezeichnet, siehe
[Laurentini '94].

11.3.2 Voxel-Verfahren

Vorgehensweise

Zuerst wird im Rechner ein leeres kubisches Voxelfeld, dessen Größe so gewählt wird, dass das zu rekonstruierende Objekt mit Sicherheit hinein passt. Jedes aufgenommene Bild des Objektes wird analysiert und die zum Objekt gehörende Bildfläche entsprechend markiert. Das Voxelfeld projiziert man nun in jedes Bild und legt es über das zu rekonstruierende Objekt . das ist nur möglich, wenn der genau Standpunkt der Kamera des jeweiligen Bildes bekannt ist. Alle Voxel, die sich innerhalb der Silhouette des zu rekonstruierenden Objektes befinden, werden entsprechend, als dem Objekt zugehörig, markiert.
Das beschriebene Vorgehen wird für alle Bildaufnahmen wiederholt. Das Verfahren unterscheidet zwischen dem zu analysierenden Objekt und dem Hintergrund. Es gibt zwei Möglichkeiten dies zu realisieren:

1. Der Hintergrund ist in einem konstanten, nicht im Objekt auftretenden, Farbwert. Ist diese Szeneneigenschaft gegeben, so kann der Algorithmus durch einen Farbvergleich einfach unterscheiden ob ein projizierter Voxel in der Silhouette liegt und somit zum Objekt gehört oder außerhalb liegt. Da bei Kameraaufnahmen die Farben in der Regel nicht exakt wiedergegeben werden, sondern durch Rauschen beeinflusst sein können, muss bei der Zuordnung eines Pixels zum Objekt bzw. zum Hintergrund ein Farbspektrum für die

Klassifikation zugelassen werden. Die Farben für den Hintergrund sollten also nicht zu ähnlich mit denen des Objektes sein.

2. Der Hintergrund ist statisch und bereits bekannt bevor sich das zu rekonstruierende Objekt in der Szene befindet. Die Szene muss zuerst ohne das Objekt aufgenommen werden und anschließend von exakt der gleichen Position aus mit dem Objekt. Die so entstandenen Bilder können voneinander Subtrahiert werden. Da sich der Hintergrund nicht ändert, ergeben sich bei der Subtraktion schwarze Bereich, in denen sich das Objekt nicht befindet. Wichtig ist auch hier, dass sich das Objekt *stark* vom Hintergrund unterscheidet. Ein Pixel eines Bildes wird genau dann (abgesehen von einem Rauschwert) als Hintergrund klassifiziert, wenn es schwarz ist.

Bevor näher auf den Markierungsalgorithmus eingegangen wird, wird ersteinmal der Algorithmus vorgestellt:

1. *Initialisiere das Voxelvolumen V*
2. *für alle Silhouetten S_i*
   markiere alle Voxel
3. *lösche alle weiß gefärbten Voxel*

Markierungsalgorithmus (Einfärbung)

Der Markierungsalgorithmus soll anhand einer Tabelle dargestellt werden.

	schwarz	grau	weiß
innerhalb	schwarz	grau	weiß
ungewiss	grau	grau	weiß
außerhalb	weiß	weiß	weiß

Tabelle 11.1: Markierungsvorschriften: in der ersten Zeile stehen die bereits vergebenen Farben, die je nach Lage des Voxels in einer anderen Ansicht umgefärbt werden kann.

Im ersten Schritt sind alle Voxel leer. Wird das Voxelfeld auf das erste Bild projiziert, werden alle Voxel eingefärbt:

- Voxel, die komplett in der Silhouette liegen werden schwarz gefärbt;
- Voxel, die direkt auf der Silhouette liegen und weder direkt innerhalb noch außerhalb liegen werden grau markiert;
- Voxel, die außerhalb der Silhouette liegen, werden weiß markiert.

Nach dem ersten Schritt sind alle Voxel eingefärbt. Als nächstes werden alle Voxel in das nächste Bild projiziert und eventuell gemäß der Tabelle 11.1 umgefärbt. Der Algorithmus terminiert, wenn alle Ansichten verarbeitet wurden. Das resultierende Voxelvolumen entspricht

der approximierten[3] visuellen Hülle. Das Voxelvolumen ist stark von der Auflösung des Voxels abhängig. Wird das Voxelfeld mit einer immer kleineren Voxelrasterung initialisiert, so steigt sein Speicheraufwand in das unermeßliche.
Aus diesem Grund wurde der Octree entwickelt.

Octree

Der Octree ist eine baumartige Speicherstruktur, die unterschiedlich große bzw. kleine Voxelwürfel zulässt. Wie im Algorithmus in Kapitel 11.3.2 wird das zu rekonstruiernde Objekt durch Voxel initialisiert. Anders als im Algorithmus wird das Objekt aber nicht mit einem Voxelfeld initialisiert sondern mit nur **einem** Voxel. Dieses erste Voxel ist mindesten so groß wie das zu rekonstruierende Objekt und wird somit grau oder schwarz gefärbt. Da jedes schwarz gefärbte Voxel komplett in der Silhouette des Objektes liegt, wird dieses nicht weiter betrachtet. Jedes graue Voxel wird in 8 Subwürfel - gleicher Größe - geteilt. Für jeden neuen Subwürfel wird der Markierungs-Algorithmus erneut durchgeführt. Abbildung 11.2 zeigt ein Beispiel eines Octrees.

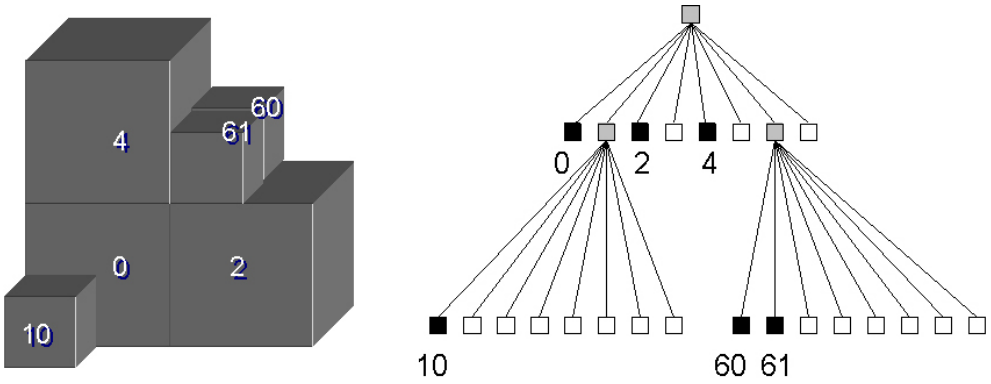


Abbildung 11.2: Beispiel eines Octrees (rechts) und sein Würfelabbild (links)

Würde in diesem Beispiel von Begin an mit der Voxelgröße des Voxels 10 gearbeitet, so währen 64 Projektionstests notwendig. Bei Verwendung der Octree-Methode werden im obigen Beispiel jedoch nur 24 Würfel betrachtet. Auf diese Weise müssen $64-24 = 40$ Projektionstests weniger berechnet werden. Allgemein ist die Knotenzahl und damit der benötigte Speicherplatz und Rechenaufwand bei der Octree-Methode proportional zur Oberfläche des zu rekonstruierenden Objektes.

11.3.3 Polygon-Verfahren (Volumenschnitt)

Um die Nachteile des Voxel-Verfahrens zu umgehen, versucht der Polygon-Ansatz die Visuelle Hülle durch Polygone direkt zu ermitteln [Matusik et al. ’].

Vorgehensweise

Aus jedem Bild muss zuerst einmal die Silhouette des zu rekonstruierenden Objektes ermittelt werden. Als nächstes werden Polygonzüge, die die Silhouette repräsentieren erzeugt. Ausgehend von jedem Sichtpunkt mit zugehöriger Silhouette wird der Silhouetten-Sichtkegel erzeugt.

[3] Abhängig von der Voxelauflösung ist das Voxelvolumen immer noch größer als die visuelle Hülle

Anschließend werden alle Silhouetten-Sichtkegel miteinander geschnitten. Das so entstehende Schnitt-Volumen repräsentiert dann die Visuelle Hülle. Im Gegensatz zum Voxel-Verfahren ist die Berechnung aufwendiger, da im schlechtesten Fall bei n Photoaufnahmen n^2 Schnitte der Silhouetten-Sichtkegel berechnet werden müssen. Diese Silhouetten können beliebig komplex sein.
Abbildung 11.3 zeigt ein Beispiel, dass aus zwei Kameraansichten V_1 und V_2 und zugehörigen Silhouetten S_1 und S_2 die Visuelle Hülle R erzeugen.

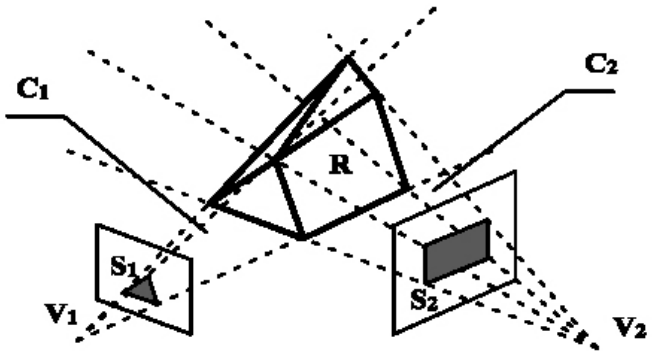


Abbildung 11.3: Visuelle Hülle eines durch zwei Ansichten (V_1und V_2) betrachteten Körpers. Die Silhouetten S_1 und S_2 stellen dabei die projizierten Abbilder des Körpers dar.

Dabei werden die Silhouetten-Sichtkegel C_1 (entstanden aus der Silhouetten S_1) und C_2 (entstanden aus S_2) geschnitten. Der entstandene Schnittkörper wird mit R bezeichnet.
Da die Schnittbestimmung im 3-dimensionalen sehr rechenaufwendig sein kann, wird im Folgenden ein Algorithmus zur vereinfachenden Bestimmung der Silhouetten-Sichtkegel-Schnitte vorgestellt.

Polygonschnittverfahren

Bei dem hier vorgestellten Polygonschnittverfahren, wird ausgenutzt, dass der Polygonschnitt eines Silhouetten-Sichtkegels auf jede weiter Ansicht abgebildet werden kaann. Abbildung 11.4 zeigt einen Würfel, der aus 3 Ansichten rekonstruiert werden soll.
Der Silhouetten-Sichtkegel der mittleren Kamera (grün dargestellt) ist in den verbleibenden Ansichten als konischer Strahl projektiv abgebildet. Für jede Strecke des Polygonzuges wird ein eigener Bereich klassifiziert (Bin-Bereich), siehe Abbildung 11.5.
Zu jeder Kante eines Bin-Bereiches werden nun die Schnittpunkte mit einer der anderen Silhouetten bestimmt. Auf diese Weise entstehen kleine Facetten. Jeder Bin-Bereich muß mit jeder Silhouette geschnitten werden. Wird das Verfahren für jede Ansicht durchgeführt, so ergeben sich eine Vielzahl von Facetten, die die Visuelle Hülle beschreiben.

11.3.4 Space Carving

Das Space Carving Verfahren ist eine Abwandlung des Voxel-Verfahrens. Es basiert auf der Erzeugung eines Voxelvolumens mit Hilfe der Photokonsistenzeigenschaft, siehe
[Kutulakos & Seitz '98].

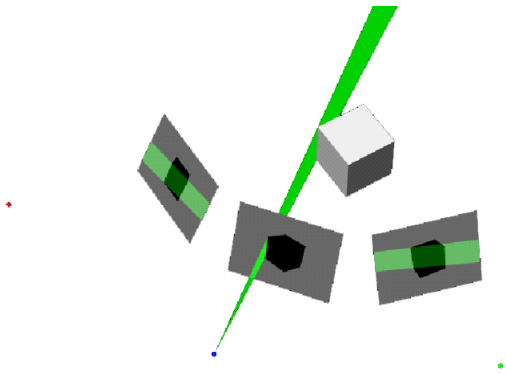


Abbildung 11.4: Ein Bin-Bereich eines Silhouettenpolygonzuges wird abgebildet auf zwei benachbarte Ansichten.

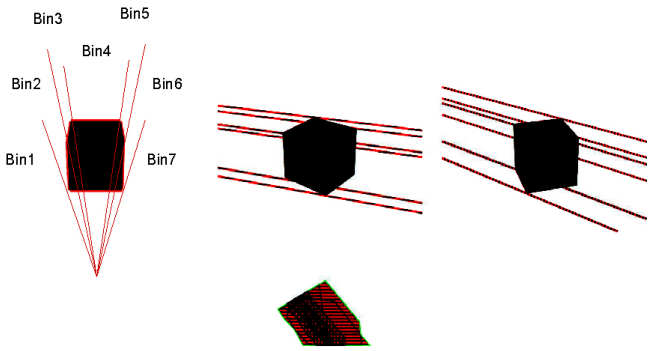


Abbildung 11.5: Einteilung der Bin-Bereiche und deren Kantenabbildung auf zwei weitere Ansichten. Daraus kann die Schnittberechnung der Silhouetten-Sichtkegel vereinfacht werden.

Vorgehensweise

Wie bei dem Voxel-Verfahren muss zuerst einemal ein Voxelraster, das mindestens so groß wie der zu rekonstruierende Körper sein muss, initialisiert. Als nächstes wird für jedes Voxel geprüft ob es sich im Objekt befindet oder außerhalb. Dies geschieht, im Gegensatz zum Voxelverfahren, nicht nur anhand der Silhouette, sondern auch noch mit Hilfe der Photokonsistenz. Alle außerhalb liegenden Voxel werden gelöscht.
Somit ergibt sich der folgende Algorithmus:

1. *Initialisiere das Voxelvolumen V*
2. *für alle Voxel* $v_i \in V$
   if *Photokonsist*(v_i)==*false*
   then *lösche*(v_i)
3. **repeat until** *Photokonsistenz*(v_i)

Der Algorithmus terminiert, wenn alle photo*in*konsistenten Voxel gelöscht sind. Zu klären ist im weiteren nur noch die Frage was die Photokonsistenz bedeuted.

Was ist Photokonsistenz

Ähnlich der Silhouetten Zugehörigkeit muss festgestellt werden ob ein Voxel in das zu rekonstruierende Objekt abgebildet wird. Weiterhin werden die Farbinformationen im Bild verwendet, um die Voxel, die innerhalb einer Silhouette liegen, zu klassifizieren. Somit ergibt sich: Ein Voxel $v_i \in V$ ist genau dann photokonsistent, wenn

- v_i wird nicht auf den Hintergrund abgebildet,
- die Farbe des projizierten Voxels v_i ist in mehr als einer Ansicht gleich.

Wird der letzte Punkt des Photokonsistenztestes vernachlässigt, so ergibt sich das Voxelverahren. Der Algorithmus 11.3.4 konvergiert gegen das photokonsistente Objekt - die Photohülle. Abbildung 11.6 zeigt drei Querschnitte eines Würfels.

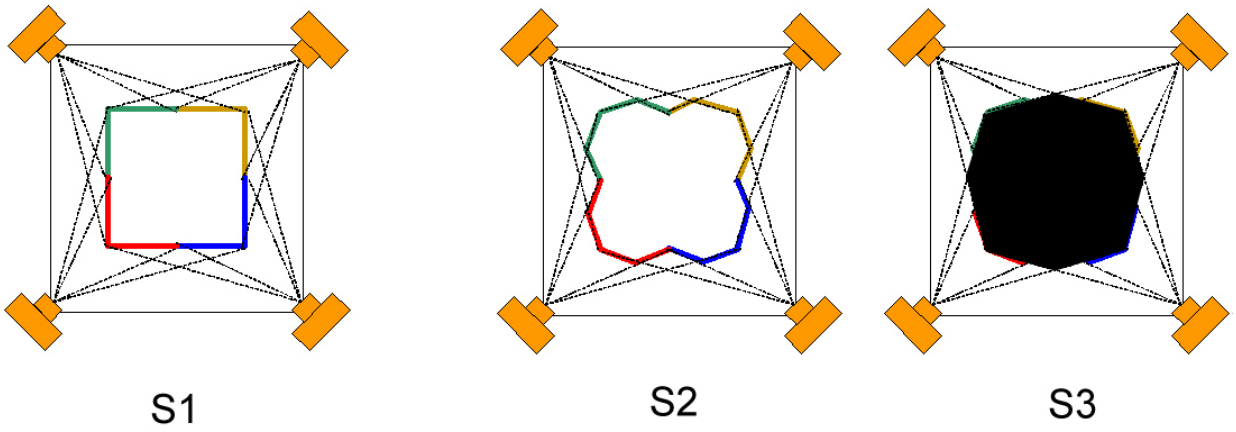


Abbildung 11.6: Vergleich der Photohülle (mitte) und der Visuellen Hülle (rechts) ausgehend von dem wahren Objekt (links). Die Photohülle ist kleiner als die Visuelle Hülle und somit stellt sie das Objekt genauer dar.

Der Würfel S1 stellt die wahre Szene dar. S3 ist die durch die Silhouettentechnik erzeugte Visuelle Hülle. Die Photohülle nutzt die Informationen nicht nur zwischen Hintergrund und Objekt, sondern auch die Farbgebung im Objekt selbst. Somit ist die Photohülle (S2) genauer als die bereits vorgestellten Silhouettenverfahren mit vergleichbar vielen aufgenommenen Ansichten.
Abbbildung 11.7 zeigt eine rekonstruierte Hand, die mit Hilfe der Photohülle erzeugt wurde. Teilbild (a) zeigt eine von ca 100 Aufnahmen. Die Teilbilder (b) bis (f) zeigen die rekonstruierte Hand aus verschidenen Ansichten. Die silber-farbenen Bereiche zeigen noch nicht eindeutig klassifizierte Voxel - der Grund dafür liegt in den verschiedene Ansichten, durch ungünstige Blickwinkel können einige Bereiche immer noch nicht richtig rekonstruiert werden.

11.4 Probleme der Silhouettentechniken

Die Silhouettentechniken benutzen die Sichtstrahlen die von einer Ansicht ausgehen. Da sich diese nur geradlinig ausbreiten, stellt sich schnell die Frage wo die Grenzen solcher Verfahren liegen. Abbildung 11.8 stellt zwei grundlegende Probleme dar.

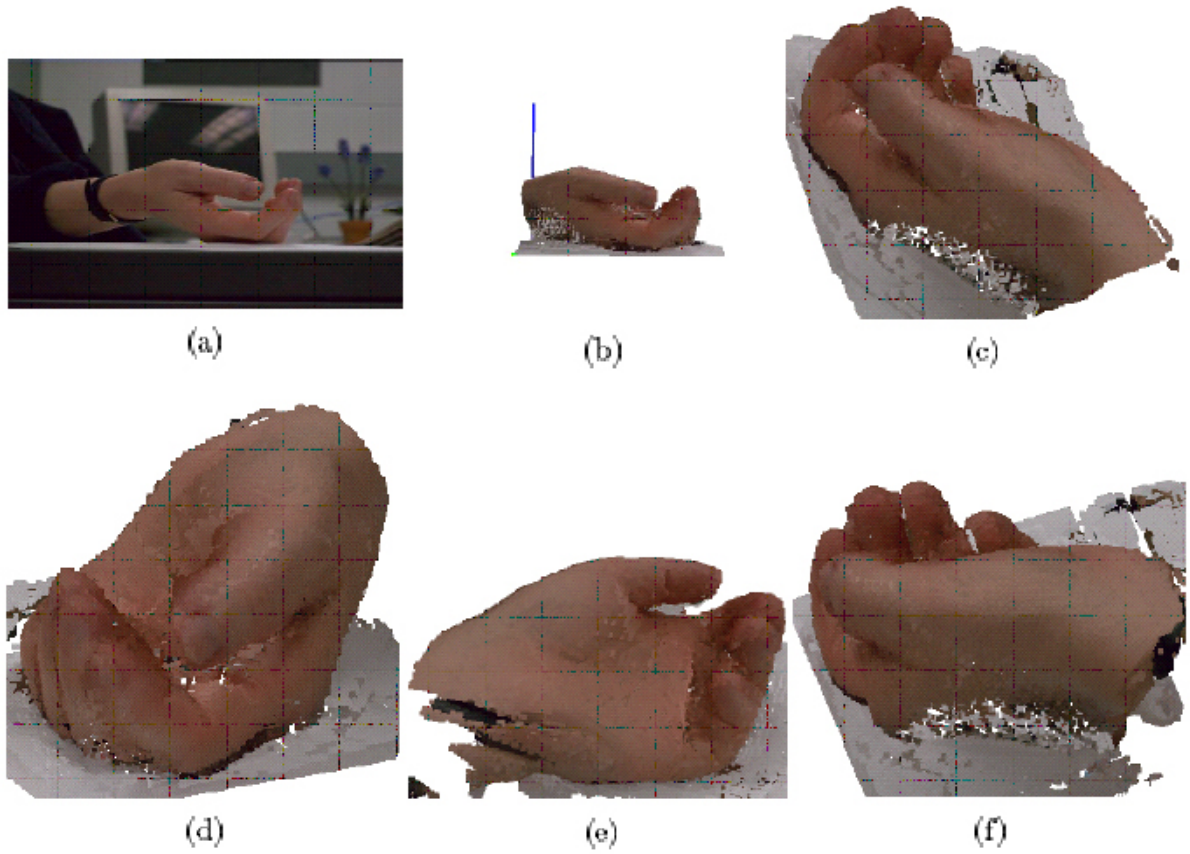


Abbildung 11.7: Space Carving Verfahren angewand um eine Hand zu rekonstruieren.

Im Teilbild (a) sind drei Kameras abgebildet. Durch die zwei äußersten Kameras werden die dunkelblauen Phantomvolumen mit erzeugt. Dies wird durch die mittlere Kamera ansatzweise aufgehoben bis auf die schwarz eingefärbten Restphantomvolumen.
Phantomvolumen können also durch Hinzunahme von mehreren Ansichten eingegrenzt werden.
Ein weiteres Problem ist in Teilbild (b) dargestellt. Die Sichtstrahlen liegen bi-tangential an zwei Körpern an (hier zwei Kreise). Dadurch kommt es zu einer approximativ angenäherten Visuellen Hülle, die auch nicht durch Hinzunahme von beliebig vielen Ansichten korrigiert werden kann.

Eine weitere für die Silhouettentechnik unüberwindbare Grenze stellen konkave Objekt dar. Beim Abbilden eines Objektes in eine Silhouette gehen Informationen über den Inhalt dieses Objektes verloren. Beispielsweise haben die beiden Objekte (a) und (b) der Abbildung 11.9 exakt die selben Silhouetten.

Dennoch sind diese beiden Objekte für einen menschlichen Betrachter verschieden. Die bereits vorgestellten Verfahren eignen sich nicht dazu solche - konvexen Objeke zu rekonstruieren. Das Objekt (b) kann bereits vollständig rekonstruiert werden. Wird versucht, das Objekt (a)

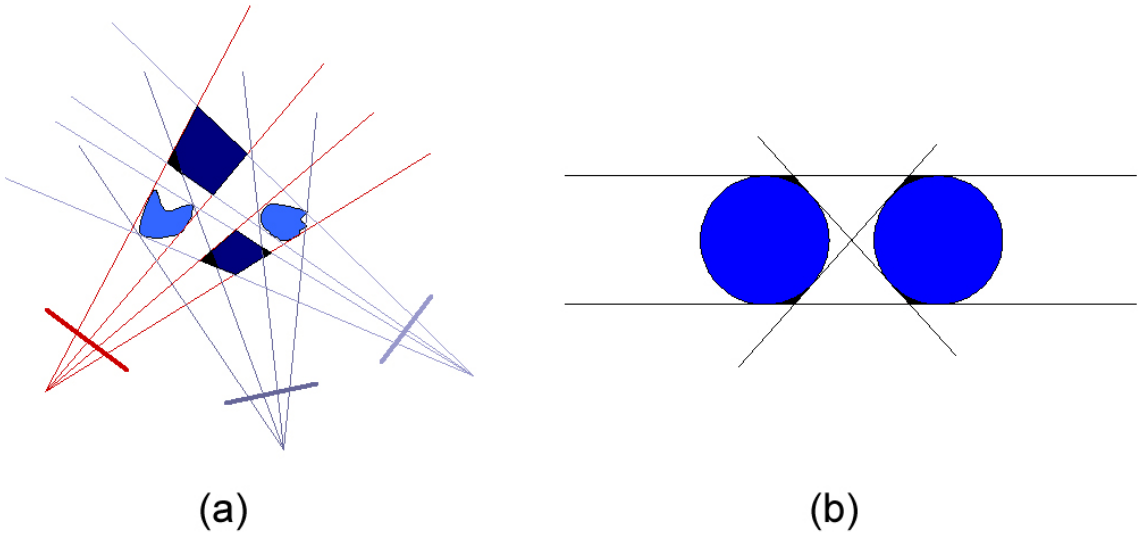


Abbildung 11.8: Probleme der Silhouettenverfahren: Phantomvolumen (links) und bitangentiale Sichtstrahlen (rechts).

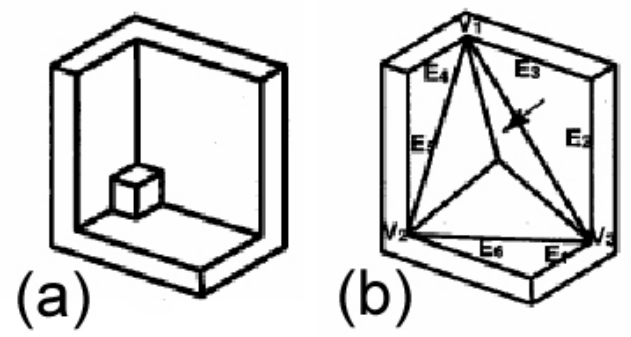


Abbildung 11.9: Die Silhouetten dieser beiden Objekte sind identisch, egal von welcher Seite sie betrachtet werden.

zu rekonstruieren, so ergibt sich, auf grund der gleichen Silhouetten, wie des Objektes (b), die identische Visuelle Hülle.

11.5 Texturierung

Nachdem der 3-dimensionale Körper mit Hilfe, der in Kapitel 11.3 vorgestellten Verfahren erzeugt wurde, stellt sich die Frage wie der Geometrie eine vergleichbare[4] Textur zugewiesen werden kann. Es liegt nahe die Texturen aus den Photos selbst zu gewinnen. Dazu wird wieder zwischen Voxel und Polygontexturierungen unterschieden.

[4] den Photoansichten vergleichbar

11.5.1 Voxeltexturierung

Ob das konstruierte Voxelvolumen durch das Voxel-Verfahren, Octree-Vefahren bzw. duch die Photohülle erzeugt wurde, spielt in diesem Fall keine Rolle. Das hier beschriebene Verfahren funktioniert bei allen Voxelverfahren gleich.
Um einem Voxel einen Farbwert zuzuweisen, muss dieser erst einmal in die verschiedenen Ansichten projiziert werden. Anschließend wird dem Voxel die Pixelfarbe zugewiesen, die ihn am besten repräsentiert, d.h. es muß die Ansicht gefunden werden, in der der Voxel sichtbar ist und nicht von anderen Voxeln verdeckt wird. Je nach dem wie der Voxel modelliert ist, können alle sechs Seiten des Voxels einzeln koloriert werden oder der Voxel kann nur einen Farbwert bekommen.
Diese Einfärbetechnik muß für alle sichtbaren Voxel das Voxelvolumens durchgeführt werden. Das Resultat ist das komplett eingefärbte Voxelvolumen.

11.5.2 Polygontexturierung

Anders als beim Voxelverfahren kann nich einfach jedes Polygon in eine Ansicht projiziert werden und dessen Farbwert bestimmt werden, da die Polygone sehr groß sein können und sich so über mehrere unterschiedliche Pixel erstrecken können.
Wie in der Computergrafik wird versucht eine Textur aus den einzelnen Ansichten zu extrahieren. Dazu gibts es zwei Grundlegende Verfahren:

1. Die zu rendernde Ansicht wird über ein Winkelmaß mit allen verfügbaren Photoaufnahmen verglichen. Die Ansicht, die der neuen Ansicht am nächsten ist, wird verwendet um deren Objektoberfläche als Textur auf die zu rendernde Objektansicht zu projizieren. Darstellungsfehler werden gegenüber einer schnellen Texturextraktion in Kauf genommen.

2. Über das bereits Eingeführte Winkelmaß werden 2 oder 3 Photoansichten ausgewählt, die der zu rendernden Ansicht am nächsten sind. Jetzt werden in Abhängigkeit ein Gewichtsfunktion die verschiedenen Texturen übereinander geblendet, siehe Abbildung 11.10. Die so berechnete Textur wird dann auf das Objekt projiziert und kann gerendert werden.

11.5.3 Texturierungsprobleme

Die beiden vorgestellten Texturierungsverfahren (Texturextrusion und Texturprojektion) haben beide das Problem, dass wenn sie zu stark vergrößert werden der im 2-dimensionalen bekante Pixeleffekt auftritt. Speziell bei der Voxeltechnik führt dies zu sehr unschönen Resultaten.
Die vorgestellten Texturierungsmethoden gehören zu den „forward mapped“ Texturen, d.h. unabhängig was mit der weiteren Geometrie geschenen wird, werden diese texturiert.
Um diesen Problemen entgegen zu treten, kann an dieser Stelle auch eine „backward mapped“ Texturierung vorgenomen werden [Buehler et al. ’99]. Dazu wird die erzeugte Geometrie erst verändert - zum Beispiel vergrößert - und anschließend wird am veränderten Objekt die Texturierung durchgeführt, siehe Abbildung 11.11

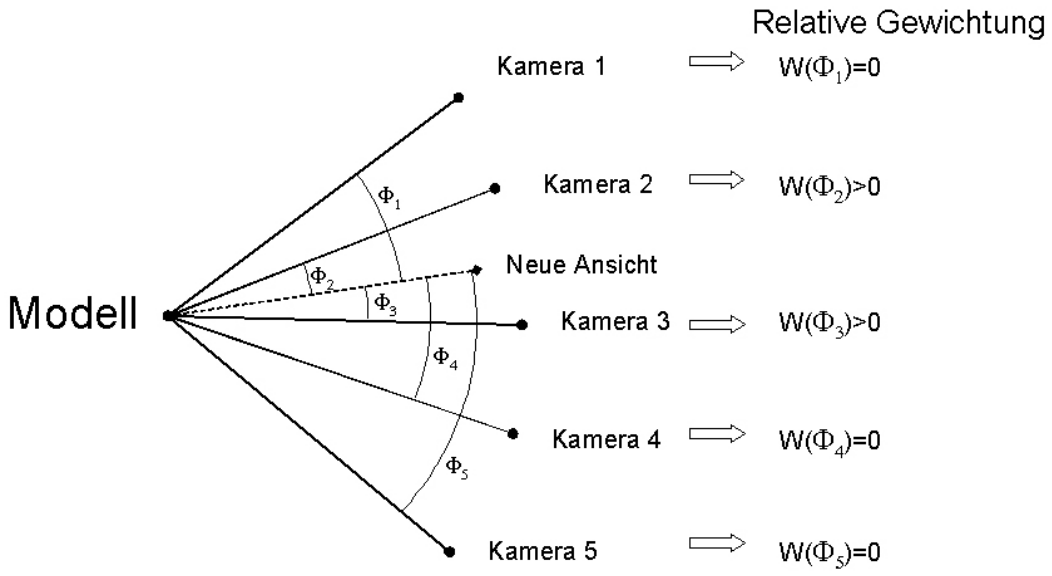


Abbildung 11.10: Für die neue zu rendernde Ansicht wird der Abstand zu allen Kameraansichten durch ein Winkelmaß (ϕ) bestimmt. durch die relative Gewichtungsfunktion können die Informationen über die Texturblendoptionen betsimmt werden.

11.6 Ausblick

Die vorgestellten Verfahren zur visuellen Modellierung und Texturierung lassen es zu aus einigen wenigen Aufnahmen 3-dimensional Objekte zu generieren. Auch wenn diese nicht exakt den original Objekten entsprechen, so lassen sie jedoch Einblicke in ihre Gestallt geben.
Durch die in der vorliegenden Arbeit diskutierten optiemierungsansätzte (Octree, Polygonschnitt im 2D) sind zwei Verfahren hervorgegangen, die der Echtzeitfähigkeit sehr nahe kommen. Es sollen im folgenden einige Ausblicke über deren Anwendungsbereiche gegeben werden:

Interaktive Visualisiertung: mit Hilfe von Kameras könnten zu verkaufende Immobilien von zu Hause aus besichtigt werden und sogar begangen werden. Dazu wird die Immobilie mit mehreren Kameras ausgestattet und somit ein virtuelles Abbild generiert.

Telefonkonferenz: Das Bildtelefon hat sich bis Heute nicht richtig durchsetzten können. Um diesen Aspekt interessanter gestalten zu können, kann ein Telefongespräch in zukunft zu einem Besuch in einem virtuellen Meetingroom werden. Statt den Benutzer im Detail zumodellieren, wird ein Avatar an seine Stelle treten. Es werden lediglich die Gesten des Benutzers an seinen Avatar weitergeleitet mit dessen Hilfe lebt das Konzept des Gestikulierens in einem Telefongespräch neu auf.

Augmentet Reality: bezeichnet die Verschmelzung von virtual Reality mit der Wirklichkeit. Ein KFZ Mechaniker ausgestattet mit einem Headup-Display und einer Stereokamera kann von einem computergestützten System unterstützt werden. Dazu nehmen die Kameras das auf was der Mechaniker sieht, erzeugen ein virtuelles Abbild und der Mechaniker kann sich bevor er das Getriebe zerlegt genau anschauen was passiert, wenn er eine Schraube löst.

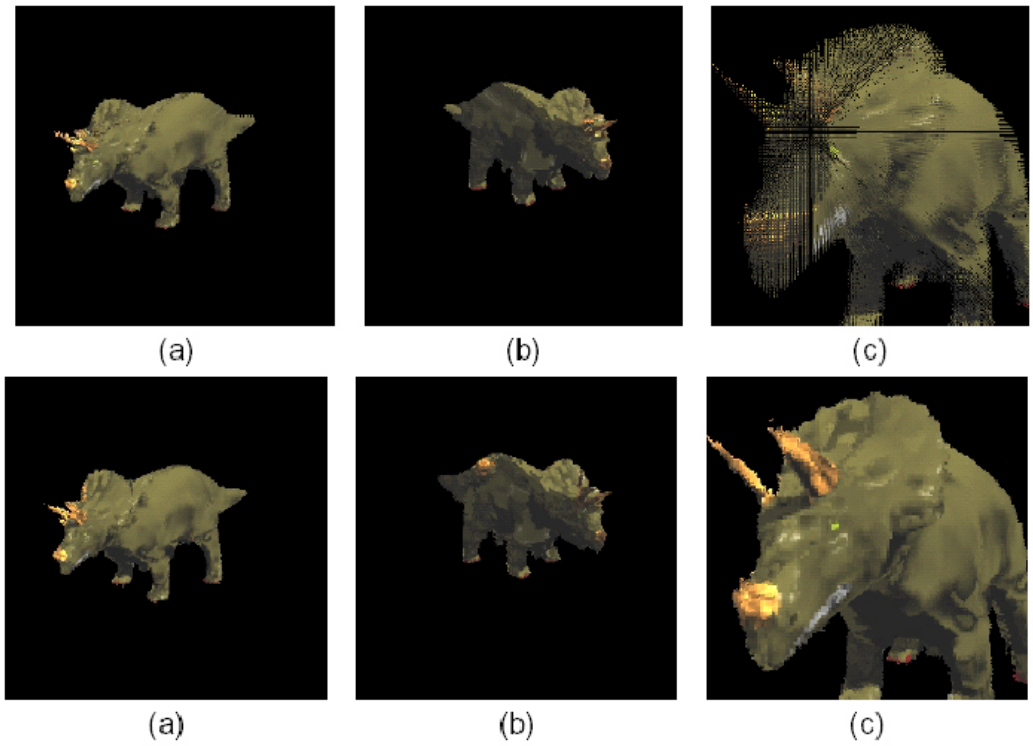


Abbildung 11.11: Die obere Bildfolge zeigt eine rekonstruierte Abbild eines Dinosaurier, der in der letzten Aufnahme vergrößert wurde. Es fehlen hier einige Informationen im Bild. Unten das gleiche Modell. Die Aufnahmen wurden mit der „backward mapped“ Texturierung erstellt. Speziell das in (c)-unten vergrößerte Abbild zeigt die Vorteile der „backward mapped“ Texturierung.

Objekterfassung: In der Archeologie werden immer wieder neue Ojeke gefunden. Statt diese wie bisher zu kartographieren, können diese sofort 3-dimensional im computer rekonstruiert werden.

Literaturverzeichnis

[Bottino & Laurentini '01] Andrea Bottino und Aldo Laurentini. Experimenting with nonintrusive motion capture in a virtual environment. In *The Visual Computer*, Bd. 17(1), S. 14–29. Springer, 2001.

[Bottino et al. '97] A. Bottino, A. Laurentini und P. Zuccone. Toward Non-intrusive Motion Capture. *Lecture Notes in Computer Science*, 1352, S. 416–??, 1997.

[Bottino et al. '01] A. Bottino, L. Cavallero und A. Laurentini. Interactive Reconstruction of 3-D Objects from Silhouettes. In V. Skala (Hrsg.), *WSCG 2001 Conference Proceedings*, 2001.

[Buehler et al. '99] C. Buehler, W. Matusik, L. McMillan und S. Gortler. Creating and Rendering Image-Based Visual Hulls. Technical Memo MIT/LCS/TR-780, Massachusetts Institute of Technology, Laboratory for Computer Science, Mai 1999.

[Kutulakos & Seitz '98] Kiriakos N. Kutulakos und Steven M. Seitz. A Theory of Shape by Space Carving. Technical Report TR692, 1998.

[Laurentini '94] Andrea Laurentini. The Visual Hull Concept for Silhouette-Based Image Understanding. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (Vol 16, No. 2), S. 150–162, February 1994.

[Li et al. '02] Ming Li, Hartmut Schirmacher, Marcus Magnor und Hans-Peter Seidel. Combining Stereo and Visual Hull Information for On-line Reconstruction and Rendering of Dynamic Scenes, 2002.

[Lok '01] Benjamin Lok. Online model reconstruction for interactive virtual environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, S. 69–72. ACM Press, 2001.

[Matusik et al. '] Wojciech Matusik, Chris Buehler und Leonard McMillan. Polyhedral Visual Hulls for Real-Time Rendering. S. 115–126.

[Matusik et al. '00] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler und Leonard McMillan. Image-Based Visual Hulls. In Kurt Akeley (Hrsg.), *Siggraph 2000, Computer Graphics Proceedings*, S. 369–374. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Rusinkiewicz et al. '02] Szymon Rusinkiewicz, Olaf Hall-Holt und Marc Levoy. Real-Time 3D Model Acquisition. In Stephen Spencer (Hrsg.), *Proceedings of the 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH-02)*, Bd. 21, 3 in *ACM Transactions on Graphics*, S. 438–446, New York, Juli 21–25 2002. ACM Press.

12 X3D

Christian Wallenta

12.1 Einführung

X3D steht für “eXtensible 3D“ und ist eine plattform- und hardwareunabhängige Beschreibungssprache zur Modellierung und Darstellung von dreidimensionalen Objekten, Szenen und interaktiven Welten.

12.1.1 Die Geschichte von X3D

Anfang der 90er Jahre hatten Mark Pesce und Tony Parisi die Idee eines Datenformats, das es ermöglichen sollte, 3D-Objekte und -Welten aus dem World Wide Web (WWW) zu laden und anschließend in diesen Welten zu navigieren. Ihr erster Prototyp eines 3D-Browsers wurde im Mai 1994 auf der ersten Konferenz über das WWW in Genf vorgestellt.
Der Name der neuen Beschreibungssprache war “Virtual Reality Modeling Language“ (VRML). Die 3D-Eigenschaften von VRML basierten auf dem “Open Inventor“ Dateiformat der Firma SGI. VRML beschreibt im wesentlichen einen sogenannten Szenegraph, welcher beliebig ineinander verschachtelte Knoten (Nodes) enthält. Diese Knoten beschreiben sowohl die geometrischen Objekte der 3D-Welt, als auch ihre Eigenschaften wie Farbe oder Position im Raum.
Im April 1995 wurde die erste offizielle Version der VRML 1.0 Spezifikation herausgebracht. Diese erste Version beschrieb allerdings noch einfache, statische Welten.
Im Januar 1996 wurde von der VRML Architecture Group (VAG) dazu aufgerufen, Vorschläge für VRML 2.0 zu machen. Diesem Aufruf folgten unter anderem namhafte Firmen wie SUN, Microsoft, IBM und Apple. Bei einer Internet-Abstimmung wurde schließlich der Vorschlag von SGI und Sony unter dem Titel “Moving Worlds“ ausgewählt. Dieser beinhaltete ein Event-basiertes Ausführungsmodell. “Moving Worlds“ wurde dann auf der Siggraph Konferenz 1996 als VRML 2.0 Spezifikation vorgestellt. Gleichzeitig wurde das VRML Konsortium (VRMLC) gegründet, welches die Pflege und Weiterentwicklung von VRML fördern soll.
Im April 1997 wurde VRML 2.0 von der ISO unter dem Namen VRML 97 zum Standard erklärt.
Da aber VRML einige Schwächen aufwies, wurde eine neue Spezifikation entworfen. 2001 wurde auf der Siggraph Konferenz X3D als Nachfolger von VRML vorgestellt.

12.1.2 Modularer Ansatz, Komponenten, Profile

Ein Hauptkritikpunkt an VRML war die Komplexität und das hohe Datenvolumen. VRML verfolgt einen monolithischen Ansatz. Das heißt, eine kleine 3D-Welt mit nur geringer Anzahl von Objekten hat den gleichen Funktionalitätsumfang wie zum Beispiel eine komplexe geographische 3D-Welt.

Bei X3D hingegen hat man einen modularen, komponentenbasierten Ansatz gewählt. Ein kleiner Kern (Core) beinhaltet alle grundlegenden Objekte und Funktionen. Komplexere Objekte und Funktionen können über Komponenten und Profile, welche eine Ansammlung von Objekten und Funktionen bilden, eingebunden werden. So benötigt man für eine kleine 3D-Welt auch nur einen kleinen Player. Dieser Ansatz ermöglicht es auch, die Szene genau auf die Bedürfnisse an Objekten und Funktionen zuzuschneiden. Die benötigten Komponenten bzw. das benötigte Profil wird im Header der X3D-Datei angegeben und der Browser muss die Profile dann entweder nachladen oder eine Fehlermeldung ausgeben. Abb. 12.1 verdeutlicht den modularen Ansatz von X3D.

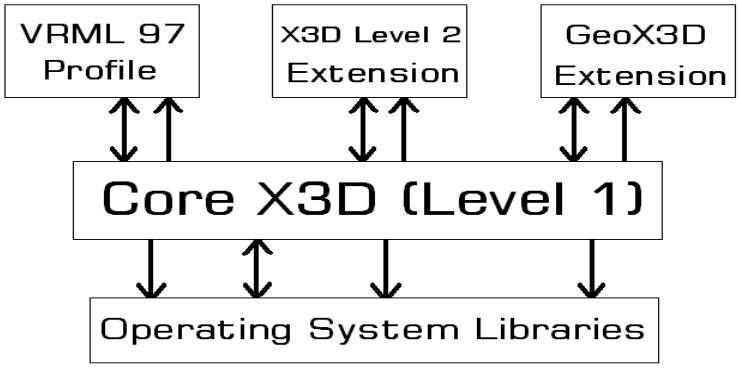


Abbildung 12.1: Das modulare Konzept von X3D

12.1.3 Ziele und Designkriterien

X3D unterstützt weitere Kodierungsformate. So können die 3D-Welten weiterhin in der klassischen VRML Kodierung geschrieben werden, aber auch die Extensible Markup Language (XML) wird unterstützt. Das erlaubt es, den Inhalt der 3D-Szene von der Darstellung zu trennen, was ein Plus an Flexibilität bringt. Des weiteren sollten neue Grafik-, Verhaltens- und interaktive Objekte hinzugefügt werden und alles unterspezifizierte Verhalten aus VRML entfernt werden. Mit der Möglichkeit Profile anzugeben, wollte man speziellen Anforderungen gerecht werden. Ein Profil soll zum Beispiel die Rückwärtskompatibilität zu VRML 97 gewährleisten.

12.1.4 X3D Features

Im Bereich der 3D-Grafik umfassen die Features polygonale Geometrie, parametrische Geometrie, verschiedene Lichtquellen und Materialien sowie Texturen.
Des weiteren sollen Animationen mit Hilfe von Timern ermöglicht werden.
Der Benutzer hat die Möglichkeit mit der 3D-Welt zum Beispiel über eine Maus oder ein Keyboard zu interagieren. X3D ist allerdings hardwareunabhängig, also kann man sich auch andere Eingabegeräte vorstellen.
Im Bereich der Navigation stellt der Browser verschiedene Bewegungsmöglichkeiten wie Fliegen oder Gehen zur Verfügung, so dass der Benutzer verschiedene Positionen in der Szene einnehmen kann.
Weitere Features sind Audioquellen in der Szene und Videos als Texturen abgebildet auf Geometrien.

In X3D gibt es die Möglichkeit, eigene Objekte aus bereits vorhandenen Objekten zu definieren und später wieder zu verwenden.
Unter Verwendung von Programmier- und Skriptsprachen kann man die Szene dynamisch verändern.
Eine X3D-Szene kann Hyperlinks zu anderen Szenen oder Dateien beinhalten. Als Beispiel stelle man sich eine Szene vor, die eine Wohnung modelliert. Hier könnten zum Beispiel die Türen Hyperlinks zu anderen Szenen enthalten, die andere Zimmer darstellen.
Ein weiteres Feature von X3D ist die Möglichkeit humanoide Animationen zu modellieren.

12.1.5 Anwendungsbereiche

Folgende Anwendungsgebiete sind für X3D denkbar:

- Wissenschaftliche Visualisierungen / Lehre
- Technische / industrielle Visualisierungen
- Multimedia-Präsentationen
- Unterhaltung / Spiele
- Architektur
- Webseiten
- Produktvisualisierungen (e-Commerce)
- Datenbankvisualisierungen

12.2 Aufbau von X3D

Abb. 12.2 zeigt ein kleines Beispiel. An diesem Beispiel werden später einige Dinge erläutert.

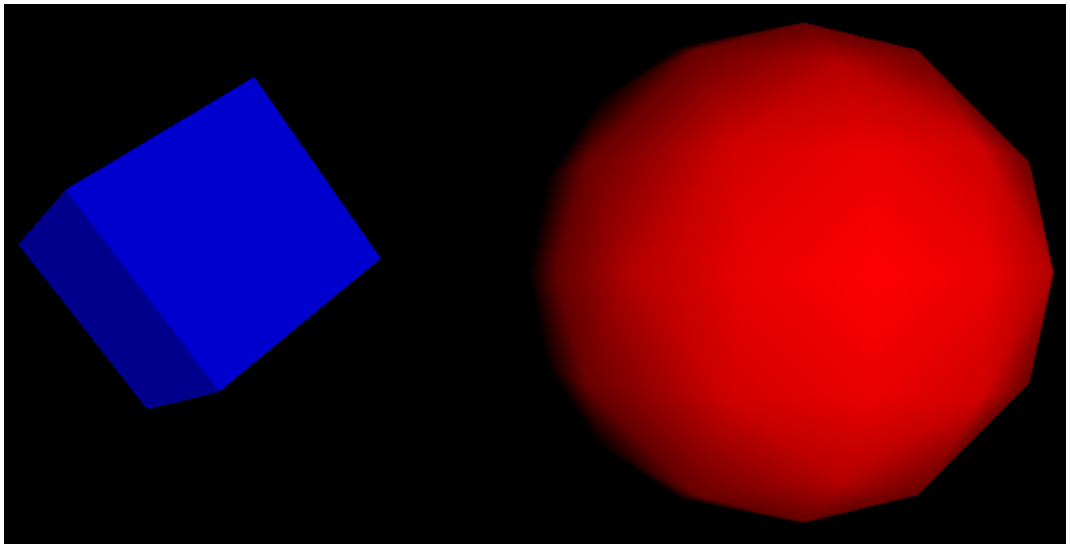


Abbildung 12.2: Kleines X3D-Beispiel

12.2.1 VRML-Kodierung vs XML-Kodierung

Wie bereits erwähnt, gibt es in X3D die Möglichkeit XML als Kodierungsformat zu benutzen. Weitere Kodierungsformate machen die Darstellung der Szene unabhängig von der Kodierung und bieten somit mehr Flexibilität. Neben XML wird meistens noch das klassische VRML Kodierungsformat benutzt.
Die beiden folgenden Codebeispiele beschreiben die rote Kugel aus dem obigen Beispiel. Die erste Version ist die klassische VRML Kodierung und die zweite XML-Kodierung.

```
Shape {
    geometry Sphere { radius 2.3
    }
    appearance Appearance {
        material Material { diffuseColor 1.0 0.0 0.0
        }
    }
}
```

```
<Shape>
    <Sphere radius='2.3'/>
    <Appearance>
        <Material diffuseColor='1.0 0.0 0.0'/>
    </Appearance>
</Shape>
```

12.2.2 Der Szenegraph

Der Szenegraph ist die Basiseinheit in der Laufzeitumgebung. Es ist ein gerichteter, azyklischer Graph, der eine baumartige Struktur besitzt. Die Knoten des Szenegraphen stehen für die Objekte in der 3D-Welt. Dies können zum Beispiel Geometrien wie Kugeln oder Kegel sein, aber auch Objekte wie Transformationen, Farben, Texturen stehen als Knoten im Graph. Die Knoten wiederum enthalten Felder, welche die Daten der Knoten oder Verweise auf Kindknoten enthalten. So hat zum Beispiel ein Knoten für eine Kugel ein Feld, welches den Radius der Kugel angibt. Eine Kante von A nach B (B ist Nachfolger von A) im Szenegraph bedeutet, dass A ein Feld mit einem Verweis auf B hat. Abb. 12.3 zeigt den Szenegraph zum Eingangsbeispiel.

12.2.3 Standardeinheiten in X3D

Lineare Abstände werden in Meter, Winkel im Bogenmaß und Zeit in Sekunden angegeben. Das Farbmodell in X3D ist RGB (Rot-Grün-Blau), wobei die drei Grundfarben immer in Werten zwischen 0 und 1 angegeben werden. RGB ([0,1.0] ; [0,1.0] ; [0,1.0]).

12.2.4 Der Shape-Knoten

Der Shape-Knoten ist einer der wichtigsten Knoten im Szenegraph. Er definiert Objekte und ihr Aussehen. Um dies zu ermöglichen hat er ein Feld mit einem Verweis auf einen "geometry"-Knoten (wie z.B. eine Kugel) und ein Feld mit einem Verweis auf ein "appearance"-Knoten (Aussehen, z.B. Farbe).
Der "appearance"-Knoten hat wiederum ein Kindknoten "material", welcher zum Beispiel die Möglichkeit bietet, diffuses Licht, Farbe oder Transpararenz für das 3D-Objekt festzulegen.

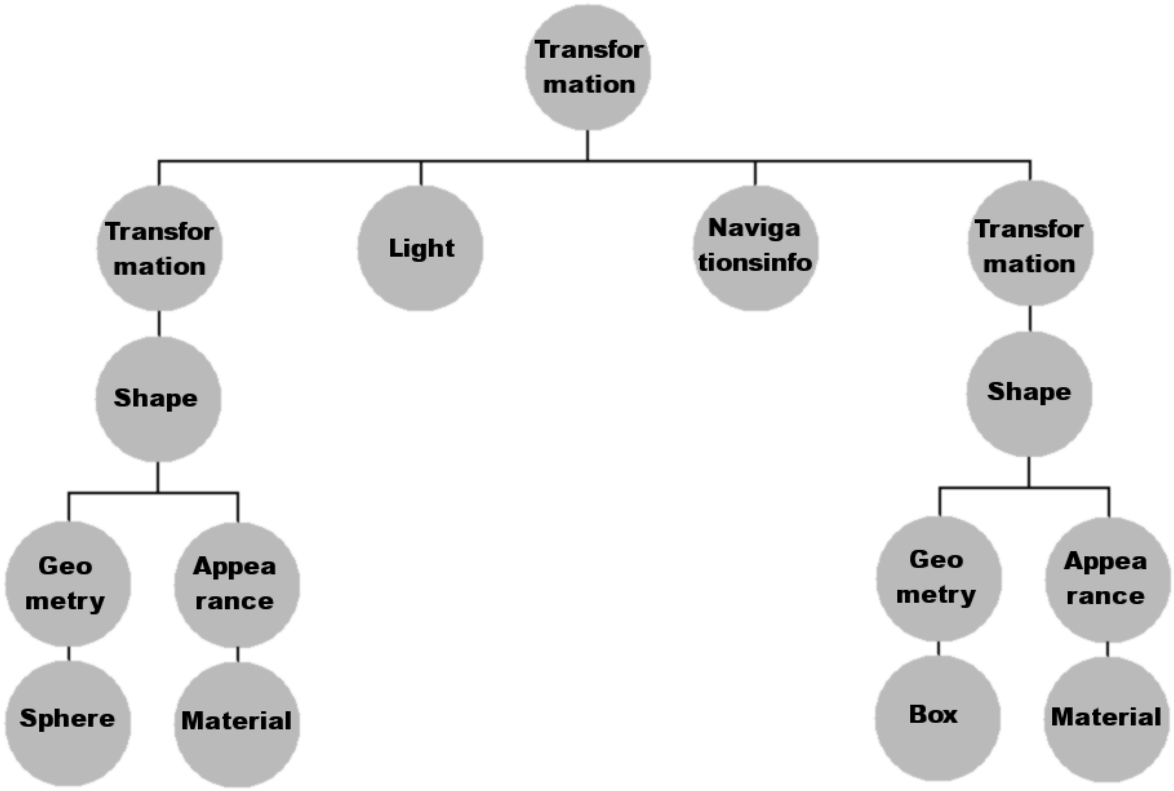


Abbildung 12.3: Der Szenegraph zum Eingangsbeispiel

Weiterhin können als Material auch Texturen auf die Geometrie aufgeklebt werden (Bilder oder sogar Video). Die Codebeispiele für die VRML Kodierung und XML Kodierung beschreiben beide einen Shape-Knoten.

12.2.5 Der Transformationsknoten

Ein weiterer wichtiger Knotentyp ist der Transformationsknoten. Er ändert die Position aller ihm untergeordneter Kindknoten. Dazu besitzt er drei Felder:

- translation x y z (x,y,z Verschiebung)
- rotation x y z a (x,y,z Rotation mit Winkel a)
- scale x y z (x,y,z Skalierung)

Der Transformationsknoten kann Verweise auf mehrere Kindknoten enthalten und die Transformationen können beliebig oft geschachtelt werden. Dabei beziehen sich der Transformationen immer auf den vorherigen Transformationsknoten.
Das Weltkoordinatensystem, also der Ursprung (0 0 0) der 3D-Welt, wird durch den obersten Knoten im Szenegraph festgelegt. Alle weiteren Transformationsknoten geben eine relative Transformation zum vorherigen Transformationsknoten an. Sie legen dann das jeweilige lokale Koordinatensystem fest.

12.2.6 Die Transformationshierarchie

Die Untermenge des Szenegraphen mit allen Objekte, welche verschiedene Positionen in der virtuellen Welt einnehmen können, wird als Transformationshierarchie bezeichnet. Sie beschreibt also alle räumlichen Beziehungen der darzustellenden Objekte.

12.2.7 Events und Routen

X3D kann eine dynamische 3D-Welt darstellen. Damit Knoten miteinander Daten austauschen können, gibt es sogenannte Events (Ereignisse). Ein Event kann man sich als Nachricht zwischen zwei Knoten vorstellen.
Damit zwei Knoten Events austauschen können, müssen sie mit sogenannten Routen verbunden werden. Routen sind keine Knoten. Sie bilden einen Eventpfad zwischen den Knoten. Man kann sie sich als eine Art Leitung zwischen den Knoten vorstellen, über die Events übertragen werden.
Verbunden werden immer die Felder zwischen den Knoten. Bei den Feldern gibt es vier verschiedene Zugriffstypen:

- **initializeOnly**: Dieses Feld wird am Anfang gesetzt und kann später nicht mehr verändert werden.
- **inputOnly**: Dieses Feld kann Events empfangen aber keine Events auslösen. Als Beispiel kann man einen Transformationsknoten betrachten, welcher durch ein Eingabefeld neue Transformationskoordinaten empfangen kann.
- **outputOnly**: Diese Zugriffsart erlaubt es nur Events zu senden aber keine zu empfangen. Als Beispiel stelle man sich einen Knoten vor, der regelmäßig Zeit-Events verschickt.
- **inputOutput**: Als letzten Typ gibt es den Feldtyp inputOutput, welcher vollen Zugriff auf das Feld erlaubt.

Wenn man zwei Felder miteinander über eine Route verbindet, muss man beachten, dass das Eingabefeld immer mit dem Ausgabefeld verbunden wird. Weiterhin müssen die Datentypen der Felder übereinstimmen. Es ist zum Beispiel nicht möglich, einen Transformationsknoten, welcher neue Koordinaten erwartet, mit einem Zeit-Knoten, welcher Zeit-Events verschickt, zu verbinden.
Events werden benutzt, um die Szene dynamisch zu verändern. Dies geschieht dadurch, dass man die Feldwerte der jeweiligen Knoten über Events verändern kann. So kann man zum Beispiel einem Transformationsknoten neue Koordinaten schicken, so dass sich ein Teil der Szene verschiebt. Oder man kann mit Hilfe von Zeit-Events Animationen ablaufen lassen. Weiterhin benötigt man Events um Benutzerbewegungen und Kollisionen zu erkennen.
Abb. 12.4 zeigt den Quellcode für eine kleine Animation. In diesem Beispiel gibt es drei Routen. Der Touchsensor (Clicker) ist mit dem Timesensor (Timesource) verbunden, der Timesensor mit dem Interpolator (Animation) und der Interpolator mit dem Transformationsknoten (XForm).
Wenn der Touchsensor einen Klick des Benutzers registriert, sendet er ein Event an den Timesensor, welcher wiederum ein Event an den Interpolator schickt. Der Interpolator berechnet die neuen Koordinaten und teilt diese dem Transformationsknoten per Event mit.
Die ROUTE Anweisungen folgen immer dem Prinzip:
ROUTE Quelle.raus TO Ziel.rein

```
DEF XForm Transform {
  children [
    Shape {
      geometry Box { }
      appearance Appearance {
        material Material { diffuseColor 1.0 0.0 0.0 }
      }
    }
    DEF Clicker TouchSensor {
    }
    DEF TimeSource TimeSensor {
      cycleInterval 2.0
    }
    DEF Animation OrientationInterpolator {
      keyValue [ 0.0 1.0 0.0 0.0, 0.0 1.0 0.0 2.1,
                 0.0 1.0 0.0 4.2, 0.0 1.0 0.0 0.0 ]
      key [ 0.0 0.33 0.66 1.0 ]
    }
  ]
}
ROUTE Clicker.touchTime TO TimeSource.startTime
ROUTE TimeSource.fraction_changed TO Animation.set_fraction
ROUTE Animation.value_changed TO XForm.rotation
```

Abbildung 12.4: Beispiel für eine Animation mit Touchsensor

12.2.8 Behaviour Graph

Der Behaviour Graph beschreibt alle Verbindungen zwischen Knoten bzw. ihren Feldern. Er zeigt also den Eventfluss im System an.

12.2.9 DEF / USE / PROTO

Mit Hilfe des DEF Statements kann man Knoten Namen geben und dann an einer anderen Stelle im Szenegraph über den Namen auf den Knoten zugreifen. Mit “DEF name Knoten“ gibt man dem Knoten einen Namen und mit “USE name“ oder “ROUTE name“ kann man wieder auf den Knoten zugreifen. Wenn man einen Knoten per Route verbinden will, muss man ihm vorher mit dem DEF Statement einen Namen gegeben haben, da die Route über “ROUTE name TO name“ benutzt wird.
Ein weiteres Statement ist das PROTO Statement. Damit kann man aus bereits definierten Knotentypen neue Objekte erzeugen. Von den neuen Knoten kann man dann an einer anderen Stelle im Szenegraph eine Instanz mit eigenen Feldwerten erzeugen. PROTO ist vergleichbar mit dem Erzeugen einer Klasse in der objektorientierten Programmierung.

1. Beispiel: Der Geometrie-Knoten “Box“ ist ein Quader. Trägt man aber in dessen Felder keine Kantenlänge ein, so hat er die Standard Kantenlänge a=2 b=2 c=2 und sieht somit aus wie ein Würfel (Cube).
   Die folgende Zeile erzeugt eine neue Geometrie “Cube“ aus der vordefinierten “Box“:

   ```
   PROTO Cube [ ] { Box { } }
   ```

Von “Cube“ kann dann später eine eigene Instanz erzeugt werden:

```
Shape { geometry Cube { } }
```

2. Beispiel: Hier wird ein Tisch aus einen Quader als Tischplatte und vier Zylindern als Tischbeine definiert. Das Ergebnis zeigt Abb. 12.5.

```
PROTO TwoColorTable [
  field SFColor legColor .8 .4 .7
  field SFColor topColor .6 .6 .1
] {
          ...
  Shape {
            appearance Appearance {
              material DEF TableTopMaterial Material {
                diffuseColor IS topColor
             }
            }
            geometry Box { size 1.2 0.2 1.2
            }
          }
          ...
          DEF Leg Shape {
            appearance Appearance {
              material DEF LegMaterial Material {
                diffuseColor 1.0 0.0 0.0
                diffuseColor IS legColor
              }
            }
            geometry Cylinder { height 1.0 radius 0.1
            }
          }
          ...
}
```

Der Prototyp kann dann folgendermaßen benutzt werden.

```
TwoColorTable {
  legColor 1 0 0
  topColor 0 1 0
}
```

12.2.10 Skriptknoten

Skriptknoten enthalten Code einer Programmier- oder Skriptsprache. Sie erlauben also das Ausführen von einem eigenem Code. Weiterhin können Skriptknoten Events empfangen und Events auslösen. Meistens enthalten sie ein Programm-Modul, das irgendeine Berechnung

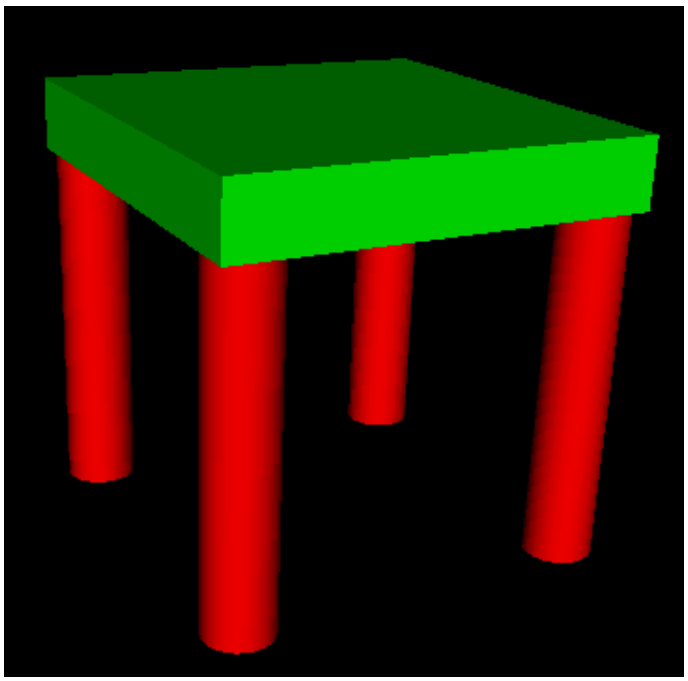


Abbildung 12.5: Beispiel für das PROTO Statement

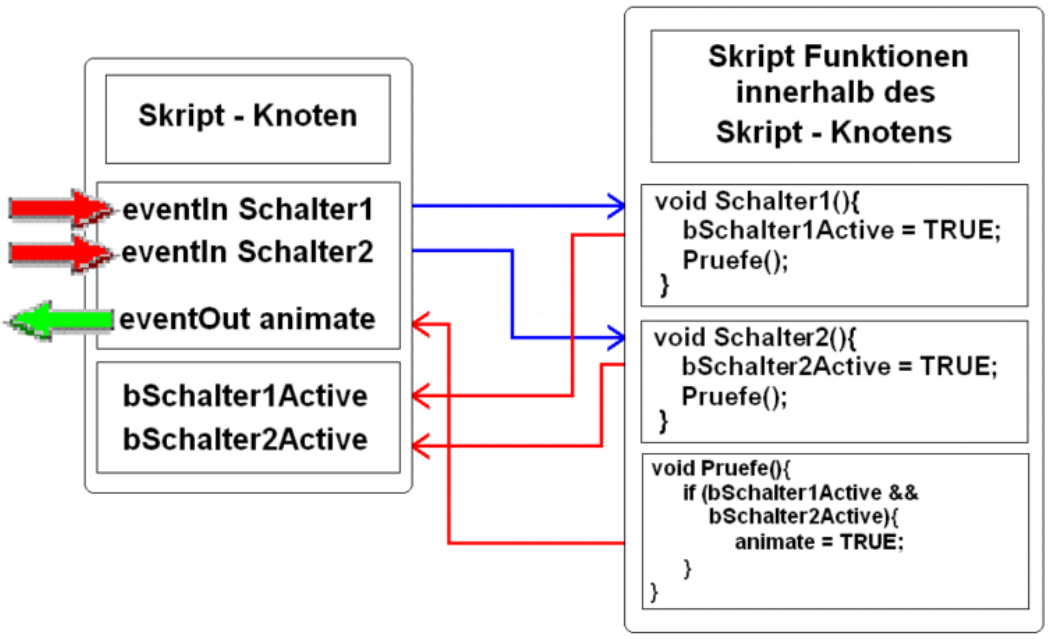


Abbildung 12.6: Beispiel für einen Skriptknoten

durchführt. Die Sprache muss vom X3D-Browser interpretiert werden können. Das Codestück wird über eine URL im Skriptknoten angegeben. Abb. 12.6 zeigt einen Beispiel-Skriptknoten. In diesem Beispiel hat der Skriptknoten zwei Eventeingänge “Schalter1“ und “Schalter2“ und ein Eventausgang “animate“, welcher zum Beispiel eine Animation startet. Weiterhin gibt es zwei Boolsche Felder “bSchalter1Active“ und “bSchalter2Active“.
Jetzt klickt der Benutzer zum Beispiel auf einen Touchsensor, welcher ein Event zum “Schal-

ter1“ schickt. Dann wird die Methode “void Schalter1()“ aufgerufen, welche wiederum Zugriff auf “bSchalter1Active“ hat und das Feld auf “TRUE“ setzt.
Danach wird die “void Pruefe()“ Methode aufgerufen, welche die beiden Boolschen Felder auf “TRUE“ überprüft. Wenn das Gleiche noch einmal für Schalter2 passiert, sind beide Boolschen Felder auf “TRUE“ gesetzt und die “Pruefe()“ Methode veranlasst, über das Ausgangsfeld “animate“ ein Event zu schicken. Daraufhin könnte irgendwo in der Szene eine Animation starten.

12.2.11 Application Programming Interface (API)

Wenn ein Benutzer mit dem X3D-Szenegraph interagieren will, hat er die Möglichkeit, einen eigenen Code zu verwenden. Entweder durch die Benutzung von den bereits erwähnten Skriptknoten oder externen Applikationen.
Die API, die benutzt werden soll heißt “Scene Authoring Interface“ (SAI). Dieses Interface ist ein Protokoll, um den Szenegraphen zu verändern, ist aber selbst nicht Teil des Szenegraphen. Die SAI erlaubt fünf Zugriffstypen in die X3D-Szene:

- Zugriff auf die Funktionalität des Browsers
- Empfangen von Browser Mitteilungen (badURL, shutdown, startup,...)
- Senden von Events zu eingabefähigen Feldern in der Szene
- Lesen des letzten gesendeten Wertes eines ausgabefähigen Feldes
- Benachrichtigung über Feldwerteänderungen von Knoten in der Szene

Es gibt hauptsächlich vier data “collections“ in einem X3D-Browser, auf die mit Hilfe der SAI Dienste zugegriffen werden kann:

- Der Browser
- Die Metadaten der geladenen Szene
- Die Knoten im Szenegraph
- Die Felder der Knoten

Ein X3D-Browser stellt verschiedene Dienste zur Verfügung, mit denen eine externe Applikation interagieren kann. So kann eine Applikation zum Beispiel neue Events generieren und über Events benachrichtigt werden. Events existieren allerdings nicht außerhalb des Browsers. Das heißt, eine Applikation kann nicht Teil einer Event-Kaskade sein.
Der Browser kann weiterhin Statusmeldungen oder Fehler-/Problemmeldungen an die Applikation übermitteln. Minimal sollten folgende Nachrichten möglich sein:

- Initialize (Der Browser hat die Szene geladen)
- Shutdown (Browser wird die Szene stoppen)
- No URL available
- Connection lost

12.2.12 Profile in X3D

Profile sollen X3D flexibel machen. Sie erlauben, es speziellen Anforderungen gerecht zu werden und im Dateiheader genau die Anforderungen an den Browser zu stellen. So kann man genau die Profile und Komponenten angeben, welche die Szene benötigt und der Browser kann erkennen, ob er alle Profile unterstützen kann.
In der X3D-Spezifikation werden folgende Profile beschrieben:

1. Core-Profil:
   Das Core Profil beschreibt absolut minimale Definitionen, welche X3D benötigt. Für minimale Szenen gibt man zusätzlich genau die Komponenten an, die man benötigt.

2. Interchange-Profil
   Dieses Profil umfasst eine kleine Menge an Komponenten, kann aber noch in einem kleinen Applet oder Browser Plug-In implementiert werden. Das Licht Modell ist zum Beispiel begrenzt worden.

3. Interactive-Profil
   Dieses Profil sollte auch noch in einer leichtgewichtigen Playback-Engine implementiert werden können, aber auch eine große Menge an Grafik und Interaktion unterstützen.

4. MPEG-4-Profil
   Das MPEG-4 Profil soll auch viel Grafik und Interaktion unterstützen. Zusätzlich soll es die Basis für die Kompabilität zum MPEG-4 Standard bilden.

5. Immersive-Profil
   Das Immersive Profil ist ein Profil für virtuelle Welten mit kompletter Navigations und Umgebungssensor Kontrolle. Weiterhin soll die Funktionalität analog zur VRML Spezifikation implementiert werden.

6. Full
   Die Angabe “Full“ bedeutet, dass alle Funktionen von X3D unterstützt werden sollen.

12.3 Vorstellung einiger Komponenten

Im folgenden Abschnitt werden einige Komponenten der X3D-Spezifikation vorgestellt.

12.3.1 Die Rendering-Komponente

Diese Komponente enthält fundamentale Darstellungsprimitive wie “TriangleSet“, “PointSet“ und Knoten, die die geometrischen Eigenschaften wie Koordinaten, Farben, Normalen und Texturkoordinaten definieren.
Folgende Knoten repräsentieren die fundamentalen visuellen Objekte:

- TriangleSet (siehe Abb. 12.8)
- TriangleFanSet (siehe Abb. 12.7)
- TriangleStripSet (siehe Abb. 12.9)
- IndexedLineSet (Linienzug)

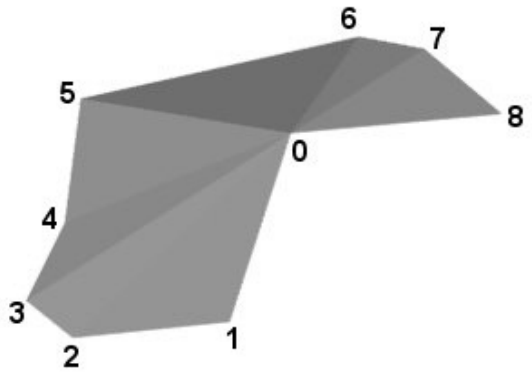


Abbildung 12.7: TriangleFanSet (Punkte werden nacheinander abgearbeitet.)

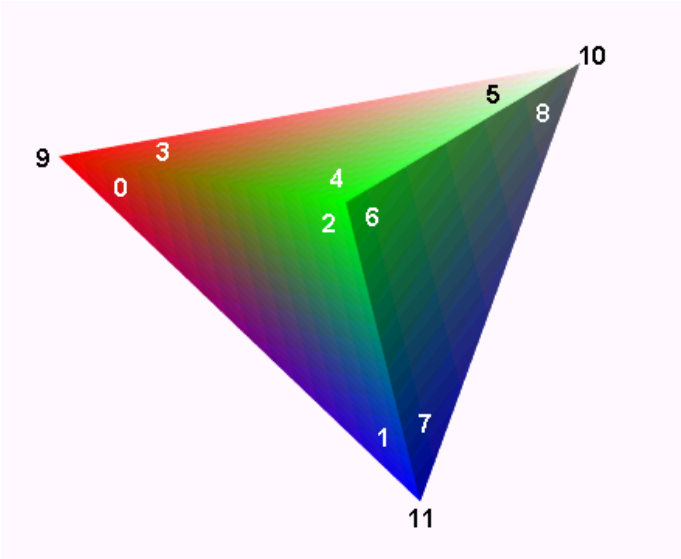


Abbildung 12.8: TriangleSet

- PointSet

Viele komplexere Geometrien können als Kombinationen von diesen Knoten implementiert werden.

12.3.2 Die Time-Komponente

Die Komponente enthält die Beschreibung des Timesensor Knoten, welcher die X3D-Welt mit der Zeitbasis des Browsers verbindet. Der Browser veranlasst den Knoten im Laufe der Zeit Zeitevents zu generieren. Diese Zeitevents sollten angenähert in Echtzeit passieren.
Time (0.0) ist 00:00:00 GMT am 01.Januar 1970.
Wofür braucht man Zeitevents? In X3D gibt es sogenannte zeitabhängige Knoten. Beispiele sind “AudioClip“ und “MovieTexture“. Diese Knoten haben zum Beispiel eine “startTime“ und

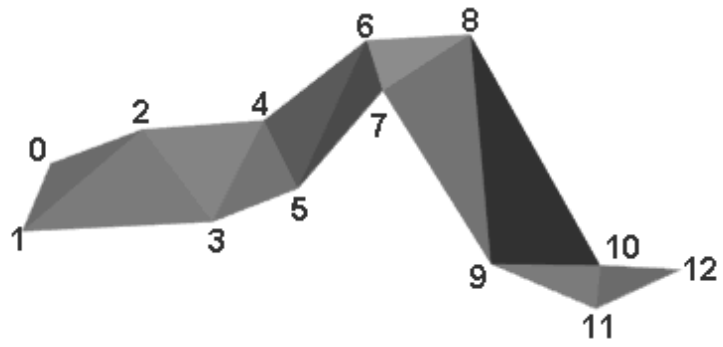


Abbildung 12.9: TriangleStripSet

eine “stopTime“ für den Clip, den sie abspielen. Diese Felder werden mit einem Timesensor Knoten verbunden, und wenn die “startTime“ erreicht ist, weiß der AudioClip Knoten, dass er den Sound abspielen muss. Wenn die “stopTime“ erreicht ist, hört er auf den Sound zu spielen.

12.3.3 Die Geometry 3D-Komponente

Die Komponente enthält vier Knotentypen: shape, geometry, geometry property und appearance. Diese Knoten werden benutzt, um die visuellen Elemente der X3D-Welt zu beschreiben. Der Shape-Knoten muss Teil der Transformationshierarchie sein um angezeigt zu werden. Genauso muss die Transformationshierarchie Shape-Knoten enthalten, damit überhaupt eine Geometrie sichtbar wird.
Der Shape-Knoten enthält jeweils genau einen Verweis auf einen Geometry Knoten und einen Appearance-Knoten. Es folgen einige der Geometrien, die in der Komponente enthalten sind.

```
Box : X3DGeometryNode {
  SFVec3f [ ] 2 2 2 (0,inf)
}
```

Die Box definiert einen Quader. Werden keine Feldwerte gesetzt, so hat der Quader die Kantenlänge 2 und sieht standardmäßig aus wie ein Würfel.

```
Cone : X3DGeometryNode {
  SFBool  [] bottom       TRUE
  SFFloat [] bottomRadius 1    (0,inf)
  SFFloat [] height       2    (0,inf)
  SFBool  [] side         TRUE
}
```

Cone definiert einen Kegel, der standardmäßig Radius 1 und Höhe 2 hat. Die beiden Boolschen Werte zeigen an, ob der Boden beziehungsweise der Mantel angezeigt wird oder nicht. Setzt man beispielsweise “bottom“ auf “FALSE“, so hat der Kegel keinen Boden.

```
Cylinder : X3DGeometryNode {
  SFBool  [] bottom TRUE
  SFFloat [] height 2    (0,inf)
  SFFloat [] radius 1    (0,inf)
```

```
  SFBool  [] side   TRUE
  SFBool  [] top    TRUE
}
```

Cylinder definiert einen Zylinder, der standardmäßig Radius 1 und Höhe 2 hat. Die Boolschen Werte verhalten sich wie beim Kegel.

```
Sphere : X3DGeometryNode {
  SFFloat radius 1 (0,inf)
}
```

Sphere definiert eine Kugel, die standardmäßig Radius 1 hat. Alle vier Geomtrien sind in Abb. 12.10 abgebildet. Weiterhin gibt es noch ein "ElevationGrid" (siehe Abb. 12.11.)
Ein sehr wichtiger Knoten ist das "IndexedFaceSet", welches ermöglicht, komplexere Figuren aus Polygonen zu erzeugen. Die Polygondaten werden dort in einem großen Array aufgelistet. Der Knoten ist wichtig, da nicht alle Objekte der 3D-Welt mit einfachen Geometrien wie Zylinder und Kugeln erzeugt werden können. Abb. 12.12 zeigt ein Objekt, was zum Beispiel nicht mit einfachen Geometrien möglich wäre.

12.3.4 Die Sound-Komponente

Die Sound-Komponente beschreibt wie man Sounds in X3D einbindet. Der Browser sollte die verschiedenen Sounds in einer Liste, welche nach Prioritäten sortiert ist, speichern. Wenn er nicht genügend Ressourcen hat, um alle Sounds abzuspielen, werden sie nach Prioritäten abgespielt.
Der Sound wird in einem Ellipsoid von der Quelle abgestrahlt. Wenn der Benutzer sich vom inneren zum äußeren Sound-Ellipsoid bewegt, wird der Sound linear abgedämpft.
Der Browser kann auch räumlichen Klang unterstützen. Der Sound klingt je nach Position des Benutzers zur Soundquelle anders. Dazu gibt es ein Feld "spatialize", welches man auf TRUE/FALSE setzen kann, wenn man diese Möglichkeit nutzen möchte.
Unterstützt werden sollte "stereo panning", je nach Winkel des Benutzers zur Soundquelle. "Stereo panning" ist in Abb. 12.13 dargestellt.

12.3.5 Die Licht-Komponente

Die Beleuchtung der Shape-Knoten entspricht der Summe aller Lichtquellen aus der X3D-Welt, welche die Shape-Knoten betreffen. Alle Lichtquellen besitzen Felder für Intensität und Farbe. Das Intensitätsfeld regelt die Helligkeit der Lichtquelle und kann von 0.0 bis 1.0 angegeben werden.
Weiterhin hat die Lichtquelle ein sogenanntes "on"-Feld, welches einem Lichtschalter entspricht. Folgende drei Knotentypen sind Lichtquellen:

- DirectionalLight
- PointLight
- SpotLight

PointLight und SpotLight beleuchten alle Objekte, welche in ihren Einflussbereich fallen, ohne Rücksicht auf die Position in der Transformationshierarchie.
PointLight ist eine Lichtquelle, die Licht in einer Kugel abgibt und den Einflussbereich über

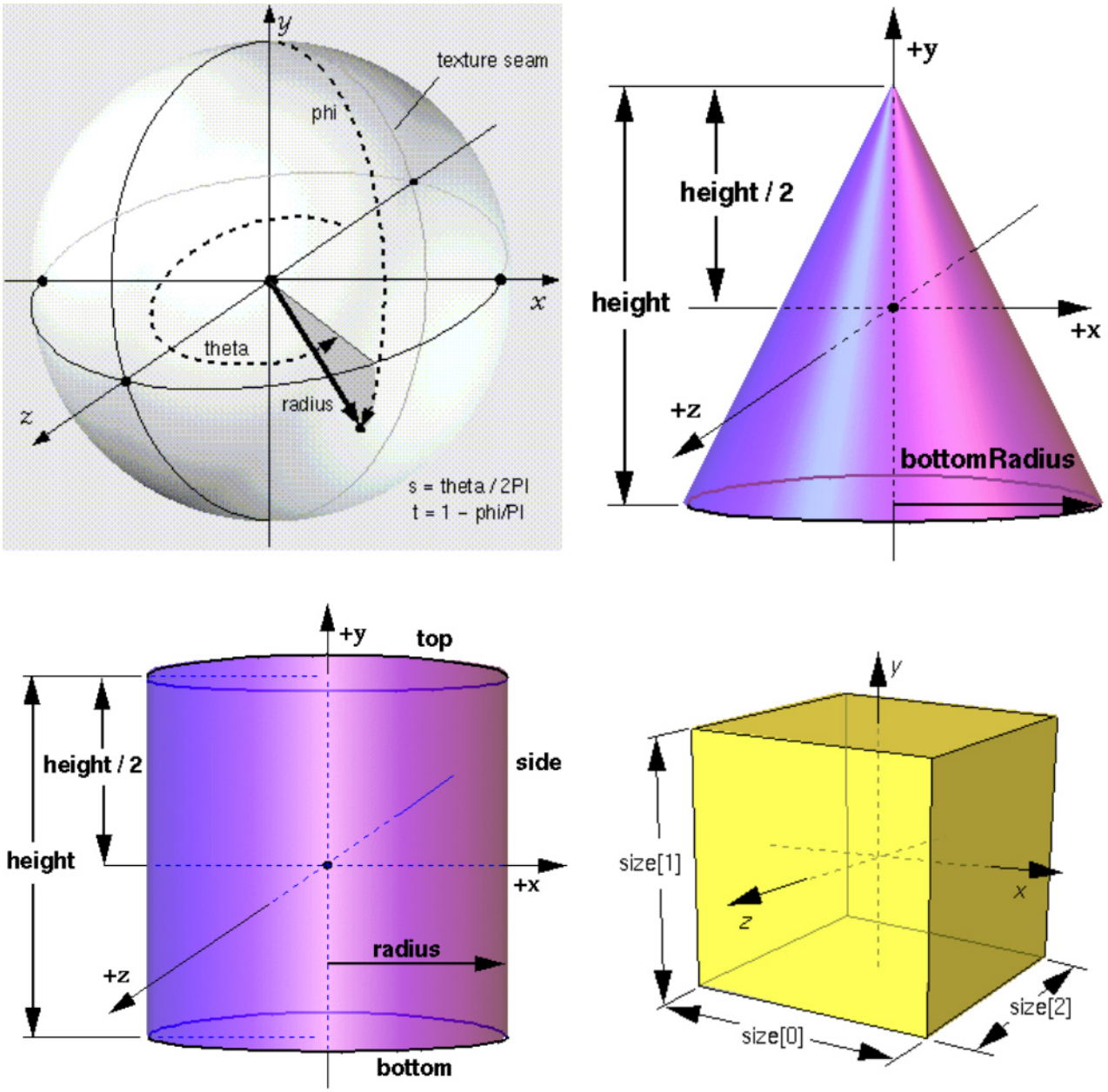


Abbildung 12.10: Geometrien Kugel, Kegel, Zylinder und Quader

einen Radius angibt. SpotLight definiert den Einflussbereich über einen Winkel. Dabei kann zum Rand des Einflussbereich die Lichtintensität abgeschwächt werden. Abb. 12.14 zeigt die Wirkungsweise von Spotlight.

DirectionalLight ist eine direkte Lichtquelle, die Strahlen parallel zu einem 3D-Vektor abgibt. Im Gegensatz zu den beiden anderen Lichtquellen beleuchtet sie nur die Objekte, welche sich im gleichen Teilgraph befinden.

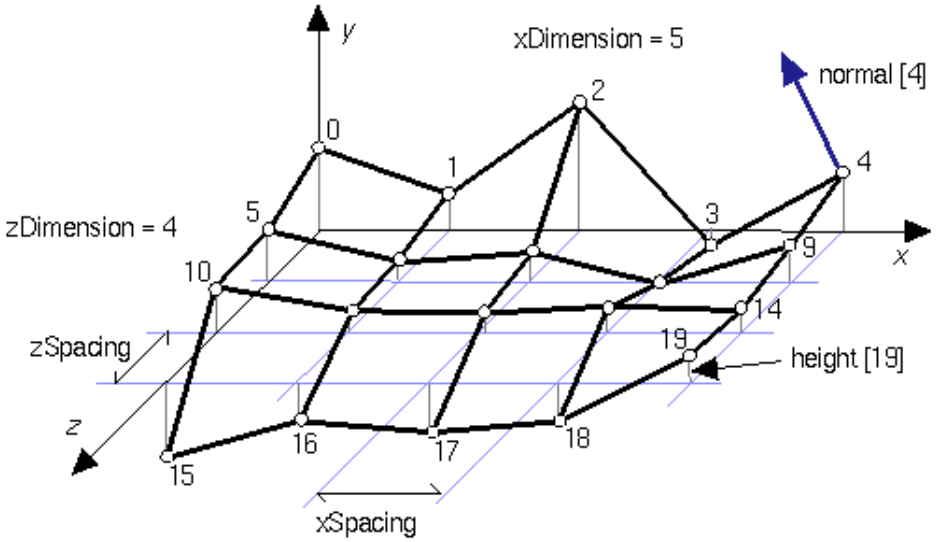


Abbildung 12.11: Das ElevationGrid

Abbildung 12.12: Beispiel für ein IndexedFaceSet

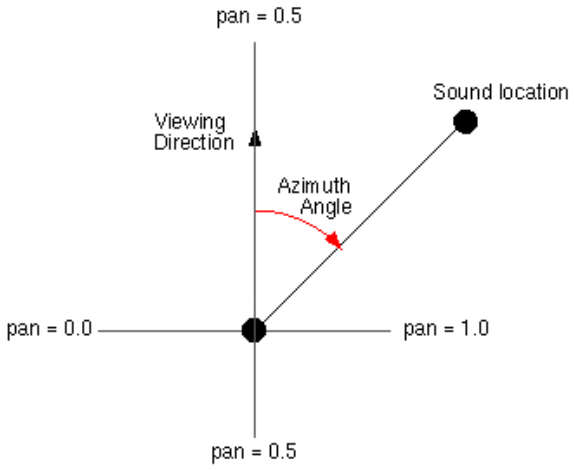


Abbildung 12.13: Funktionsweise von stereo panning

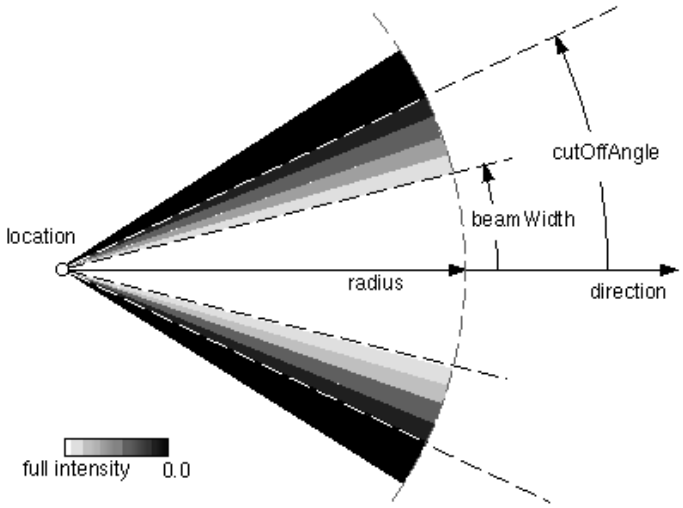


Abbildung 12.14: Wirkungsweise von SpotLight

12.3.6 Die Pointing Device Sensor Komponente

Pointing Device Sensoren erkennen Benutzerereignisse wie zum Beispiel das Klicken auf eine Geometrie (Touchsensor). Folgende Knotentypen sind Pointing Device Sensoren:

- Cylindersensor
- Planesensor
- Spheresensor
- Touchsensor
- (Anchor)

Der Anchor Knoten erkennt auch Benutzer Klicks und realisiert Hyperlinks. Er ist allerdings nicht vom “X3DPointingDeviceSensorNode“ abgeleitet. Ein Pointing Device Sensor wird akti-

viert, wenn der Benutzer das Eingabegerät über die Geometrie führt, welche von den Sensoren beeinflusst werden. Das sind alle Knoten, die Nachfolger des Vaterknotens des Sensors sind. Eine Teilmenge der Pointing Device Sensoren (Cylindersensor, Planesensor, Spheresensor) sind Dragsensoren. Diese Sensoren haben Eventausgänge wie zum Beispiel "trackPoint_changed", um Bewegungen auf ihren virtuellen Geometrien (Cylindersensor = Zylinder) zu melden.

12.3.7 Weitere Komponenten

Hier noch einige Komponenten der Spezifikation auf die nicht näher eingegangen wird.

- Humanoide Animation (H-Anim)
- NURBS (Non-Uniform Rational B-Spline)
- Distributed interactive simulation
- Geospatial (Geographische Anwendungen)
- Environment effects (Nebel, Hintergrund)

12.4 Ausblick

Der modulare Ansatz durch das Konzept der Profile und Komponenten macht X3D sehr flexibel. Der Autor der Szene kann dadurch genau angeben, was der Browser unterstützen muss. Auch die Verwendung eines weiteren Datenkodierungsformates wie XML spricht für die Flexibilität von X3D.
X3D ist offen, plattform- und hardwareunabhängig.
Es haben schon viele namhafte Firmen ihre Unterstützung zugesagt, allerdings gibt es bis jetzt noch sehr wenige Programme, welche auch nicht unbedingt voll ausgereift sind. Das Web3D Konsortium stellt einen Editor (X3DEdit) und einen Player (Xj3D) zur Verfügung, welche auch über die Webseite www.web3d.org heruntergeladen werden können.

Literaturverzeichnis

[Braitmaier '00] Michael Braitmaier. VRML, 2000. http://wwwvis.informatik.uni-stuttgart.de/img/sommer/seminar/Vortrag-VRML.pdf (gesehen 06/2003).

[Coors '00] Volker Coors. Des Cyberspace Kern. *iX 5/2000*, S. 108–113, 2000.

[Hase '97] Hans-Lothar Hase. *Dynamische virtuelle Welten mit VRML 2.0*. dpunkt.verlag, Heidelberg, 1997.

[Manuel Schiewe '00] Bozana Bokan und Manuel Schiewe. VRML / X3D und 3D-Präsentationen, 2000. http://cg.cs.tu-berlin.de/~kai/3dgraphics/X3D/x3d_ws2000/ (gesehen 06/2003).

[Renner '02] Christian Renner. 3D-Visualisierung im Internet, 2002. http://www-ra.informatik.uni-tuebingen.de/lehre/ws02/pro_internet_ausarbeitung/proseminar_renner_ws02.pdf (gesehen 06/2003).

[Schlüter '98] Oliver Schlüter. *VRML Sprachmerkmale*. O'Reilly essentials, Köln, 1998.

[Web3DC '02a] The Web3D Consortium Web3DC. Information technology - Computer graphics and image processing - Extensible 3D (X3D), 2002. http://www.web3d.org/technicalinfo/specifications/ISO_IEC_19775/index.html (gesehen 06/2003).

[Web3DC '02b] The Web3D Consortium Web3DC. Information technology - Computer graphics and image processing - Extensible (X3D) Encodings, 2002. http://www.web3d.org/technicalinfo/specifications/ISO_IEC_19775/index.html (gesehen 06/2003).