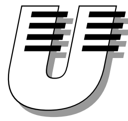


Fachbereich 4: Informatik

Entwicklung einer 3D Anwendung mit erweiterter optischer und haptischer Unterstützung

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Mirko Geissler

Betreuer: Dipl.-Inform. Matthias Biedermann
Institut für Computergraphik an der Universität Koblenz

Koblenz, im September 2006

Erklärung

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung **1**

2 Stereorendering **2**
2.1 Einleitung 2
2.2 Technik 4
2.3 Die Stereoklasse 7
2.3.1 Ziel 7
2.3.2 GLSL vs. CG 7
2.3.3 Framebuffer objects 8
2.3.4 Implementierung 9
2.4 Fazit 12

3 Haptik **13**
3.1 Einleitung 13
3.2 Haptisches Empfinden 14
3.3 Technik 14
3.4 Strategien in der Datenverarbeitung 17
3.5 OpenHaptics 19
3.6 Implementierung 19
3.7 Fazit 20

4 Der Maya Exporter **22**
4.1 Einleitung 22
4.2 Maya API 23
4.3 GPExporter Framework 23
4.4 Serialisierung 23
4.5 Implementierung 24
4.6 Fazit 26

5 Der Maya Importer **27**
5.1 Einleitung 27
5.2 Implementierung 27
5.3 GLSL Shaders 30
5.4 Fazit 32

6 Zusammenfassung **33**

7 Fazit **35**

1 Einleitung

In vielen Bereichen der Computergrafik ist es ein ständiges Ziel Bilder möglichst authentisch darzustellen um einen großen Realitätsbezug herstellen zu können. Dieses Ziel versucht man mit den verschiedensten Techniken zu erreichen. So nutzt man zum Beispiel Schatten, komplexe Texturen, besondere Lichtberechnungen und hoch aufgelöste 3D Modelle um dem Nutzer einen immersiven Eindruck von dem Objekt was er gerade betrachtet oder der Welt in der er sich gerade bewegt zu vermitteln.

Der Wunsch nach einem immersiven Eindruck kann praktische Gründe haben, wenn es zum Beispiel darum geht einen Prototyp vor seinem eigentlichen Bau so detailliert wie möglich in der Simulation zu analysieren. Nicht darstellbare Eigenschaften des Prototyps entziehen sich somit der Analyse und können dann erst nach dem Bau untersucht werden, was hohe Kosten mit sich bringen kann. Ein weiterer wichtiger Grund für den Wunsch nach hoher Immersivität ist Entertainment. Es herrscht seit einigen Jahren die Annahme, dass Unterhaltungssoftware mit möglichst detaillierter Grafik automatisch einen höheren Unterhaltungswert hat. Ob diese Annahme stimmt oder nicht, ist aber nicht Teil dieser Studienarbeit. Vielmehr wird sich diese Arbeit mit zwei anderen Möglichkeiten der Immersion auseinandersetzen. Das ist zum einen ein aktueller 3D Bildschirm der Firma Seereal [See] und zum anderen der Roboterarm PHANTOM® Omni™von Seereal [Sen].

Der C-i 3D Bildschirm ist sehr interessant, weil er einerseits die Dreidimensionalität, wohl einer der immersivsten Eigenschaften die man sich vorstellen kann, von Applikationen möglich macht und andererseits die Akzeptanz des Benutzers erreicht. Schließlich unterscheidet den Bildschirm nur wenig von einem handelsüblichen Bildschirm, vom Preis einmal abgesehen.

Mit der Akzeptanz und der Einsatzmöglichkeit des PHANTOM® verhält es sich etwas anders. Dieses Eingabegerät ist wegen der Schnittstelle „Haptik“ so interessant.

Im Entertainmentbereich wird haptisches Feedback seit einigen Jahren von herkömmlichen Eingabegeräten unterstützt, jedoch beschränken sich diese meistens auf eine vereinfachte Kraft-Gegensteuerung oder sogar nur einfaches Rütteln. Anders verhält es sich beim PHANTOM®. Er erlaubt eine gezielte Navigation und Feedbackwahrnehmung.

Im Rahmen dieser Studienarbeit wird eine OpenGL Anwendung entwickelt, die den 3D Bildschirm korrekt ansteuert, sodass ein beliebiges Modell in 3D dargestellt wird. Zusätzlich wird der PHANTOM® integriert um das Zusammenspiel dieser beiden Möglichkeiten zu testen.

Da sich abzeichnet, dass der Import von 3D Modellen aus Maya, zum Beispiel mithilfe des VRML Exporters, äußerst umständlich sein wird wird diese Studienarbeit genutzt um einen Maya Exporter und einen Importer für eine beliebige OpenGL Anwendung zu entwickeln.

Die Arbeit ist chronologisch aufgebaut und beschreibt die einzelnen Teilaufgaben in einer Einleitung, einer Zusammenfassung der Aufgabe sowie den entstandenen Ergebnissen und einem Fazit. In Kapitel 6 werden die einzelnen Teile zusammengefügt und die Arbeitsschritte an einem Beispiel gezeigt.

Im Fazit werden schließlich die Erfahrungen und Ergebnisse der gesamten Arbeit beschrieben.

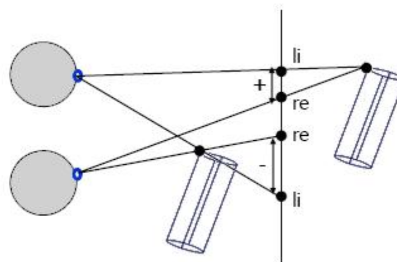


Abbildung 1: Negative und positive Parallax

2 Stereorendering

2.1 Einleitung

Das Prinzip der Stereoskopie wurde bereits 1838 von Sir Charles Wheatstone am Kings´College in London erforscht. Er begründete auf der Tatsache, dass der Mensch durch seine zwei Augen stets zwei verschiedene Bilder seiner Umwelt wahrnimmt und daraus die Entfernungen abschätzen kann, die Theorie, dass man diesen 3D Eindruck auch simulieren kann. Dafür baute er eine Apparatur, die durch Spiegel den Blick der beiden Augen auf zwei getrennte Halbbilder umlenkte. Dieses Gerät nannte er schließlich Stereoskop.

Am Prinzip der Stereoskopie hat sich bis heute nicht viel geändert. Es werden zwei Bilder aus der jeweiligen Perspektive des linken und rechten Auges gemacht, und schließlich getrennt dem Zuschauer präsentiert. Das Problem wie die beiden Bilder beim Betrachter für das linke und rechte Auge getrennt werden ohne ihn dabei zu sehr einzuschränken wurde inzwischen auf viele Weisen gelöst. Einige Möglichkeiten werden in Abschnitt 2.2 näher erläutert.

Parallaxe Als Parallaxe bezeichnet man den Abstand zwischen zwei korrespondierenden Punkten im linken und im rechten Bild. Dabei wird zusätzlich zwischen einer positiven und einer negativen Parallaxe unterschieden. Der horizontale Abstand zwischen zwei korrespondierenden Punkten gibt an, in welcher Distanz sich das Raumelement zum Betrachter befindet zu dem die beiden Punkte gehören. Je größer die Parallaxe, desto größer ist auch die Distanz zum Betrachter. Der Sonderfall, dass der Abstand gleich null ist bedeutet, dass die beiden korrespondierenden Punkte auf ein gemeinsames Pixel im Bild abgebildet werden. Das führt zu dem Eindruck, dass sich das entsprechende Raumelement auf Höhe der Betrachtungs- bzw. Bildschirmebene befindet. Für den negativen Fall hat man hingegen das Gefühl, das Raumelement würde sich vor der Betrachtungsebene näher am Betrachter befinden.

Eine positive oder negative vertikale Parallaxe ist in stereoskopischen Bildern dringend zu vermeiden. Das liegt daran, dass die Augen auf der gleichen Höhe liegen und ein vertikaler Abstand zwischen zwei korrespondierenden Punkten im alltäglichen Sehen so nicht vorkommt. Das Gehirn kann diese Information nicht sinnvoll verarbeiten, was zu einem Verlust des 3D Eindrucks führt und manchmal auch zu Kopfschmerzen.

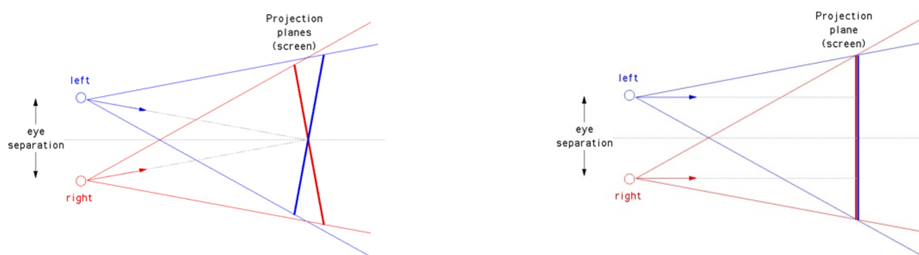


Abbildung 2: Die fehlerhafte *toe-in* und die korrekte *off-axis* Methode

Sereoskopisches Rendern Was bei einer Zwei-Objektiv-Kamera implizit geschieht muss beim stereoskopischen Rendern von virtuellen Szenen besonders vorberechnet werden. Diese Berechnung nennt man *off-axis* Methode. Mit ihr berechnet man die beiden Frusta der Kameras, sodass die Parallaxen von korrespondierenden Punkten konsistent sind und das menschliche Gehirn die beiden gerenderten Bilder automatisch zu einem Bild zusammenfügen kann. Dabei ist, gegenüber der zu ungenauen *toe-in* Methode darauf zu achten, dass die Projektionsfläche der Frusta aufeinander liegen. Sie simulieren in diesem Zusammenhang die Bildschirmoberfläche.

Der Abstand zwischen den beiden Kameras entspricht idealer Weise dem Augenabstand des Betrachters und der Abstand zur Projektionsebene idealer Weise dem Abstand des Betrachters zum Bildschirm. Diese Werte müssen hierfür in Weltkoordinaten der virtuellen Szene umgerechnet werden. Als mittlerer Augenabstand wird häufig 65mm verwendet.

Anders als bei stereoskopischen Fotografien ist es beim rendern möglich, Objekte vor dem Bildschirm treten zu lassen indem man ein Objekt vor der Projektionsebene platziert. Das kann allerdings zu einer so genannten *stereo violation* führen. Da ein Objekt, umso mehr es sich scheinbar dem Betrachter nähert, ebenfalls dem Projektionsrand nähert, kann es passieren, dass Teile aus der Sicht eines Auges bereits nicht mehr zu sehen sind. Wenn das passiert geht schließlich auch der dreidimensionale Eindruck verloren, weshalb es das zu vermeiden gilt.

Ein weiteres Problem stellt die diskrete Rasterung des Bildschirms dar die zum so genannten *depth aliasing* führen kann. Denn je weiter ein Objekt hinter der Projektionsfläche liegt desto größer werden die so genannten stereoskopischen Voxel, ein Raumabschnitt der komplett auf ein Pixelpaar in den beiden Bildern fällt. Das führt dazu, dass zwei Punkte gleicher Tiefe sehr unterschiedliche Parallaxen produzieren können und Punkte unterschiedlicher Tiefe die gleiche Parallaxe. Das Problem wird besonders durch die Tatsache verstärkt, dass viele stereoskopische Geräte heute noch eine relativ niedrige Auflösung haben, worauf in Abschnitt 2.2 näher eingegangen wird.

Die bisher beschriebenen Problemen beim Stereorendering sind theoretisch lösbar. *stereo violation* lässt sich durch eine geschickte Positionierung der Objekte auf dem Bildschirm vermeiden. *depth aliasing* Effekte kann man mit einer hohen Auflösung minimieren.

Ein bisher ungenanntes Problem lässt sich jedoch nicht lösen. Dabei handelt es sich um das Problem der Fokussierung. Bei der Betrachtung eines Stereobildes

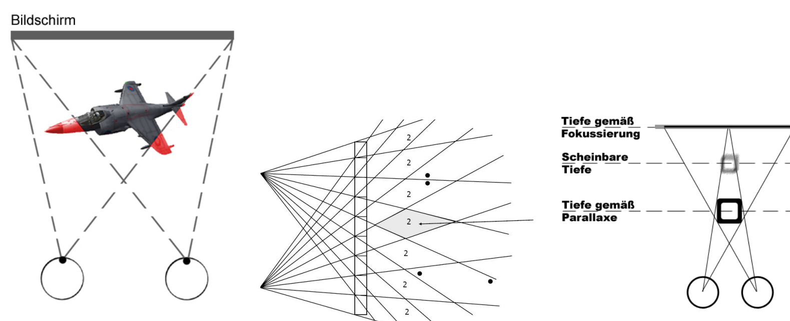


Abbildung 3: stereo-violation, depth-aliasing und das Fokussierungsproblem

fokussieren die Augen des Betrachters die Distanz zum Bildschirm. Erst dann ist das Stereobild bzw. dessen Oberfläche scharf zu erkennen. Wenn nun aber virtuelle Objekte sich scheinbar vor oder hinter dieser Ebene befinden gerät die menschliche Optik in einen Konflikt. So erkennt man zum Beispiel aufgrund der Parallaxe, das ein Objekt dreißig Zentimeter entfernt ist muss aber bei der Fokussierung stattdessen auf eine Distanz von sechzig Zentimetern blicken, damit das scheinbar nähere Objekt scharf ist. Damit löst man bei der Assoziation des Gehirns einen Widerspruch zu allen bisherigen Seherfahrungen aus. In der Konsequenz fällt es dem Beobachter in der Regel schwer die korrekte Größe eines virtuellen Objekts zu messen. Außerdem kann die ungewohnte Wahrnehmung zu Übelkeit führen.

Eine Lösung dieses Problems ist praktisch nicht möglich, weil der Stereobildschirm (oder die Stereoleinwand) ein Bild erzeugen müsste, dass dem Beobachter nur scharf erscheint, wenn er die Distanz zum Objekt, aber nicht die Distanz zur Bildschirmoberfläche fokussiert.

2.2 Technik

Als besondere technische Herausforderung galt, die beiden Halbbilder so zu projezieren, dass die Bilder getrennt nur im linken bzw. im rechten Auge wahrgenommen werden. Der erste Ansatz ist das Stereoskop von Wheatstone. Ein Doppelspiegel auf den man frontal schaut damit die beiden links und rechts montierten Bilder zu einer Übereinstimmung finden. Zugegeben ist hier der technische Anspruch nicht besonders groß gewesen, jedoch war man mit diesem Gerät auch an eine feste Kopfposition gebunden und die Bilder waren unbewegt. Der Schotte David Brewster entwickelte aber bereits sechs Jahre später ein Stereoskop in einer Fernglas ähnlichen Form.

Etwa 1860 entwickelte der amerikanische Arzt Oliver Wendell Holmes dann ein neues, leichteres Stereoskop. Es beruhte auf dem Linsensystem Brewsters, war aber handlicher, weil das Stereobild auf einem Stab vor den Linsen an einem beweglicher Schlitten befestigt war.

In den folgenden Jahrzehnten änderte sich die Technik dann nur noch wenig. Das lag zum Beispiel an der Entwicklung der Fotografie, die die Stereoskopie vorerst in den Hintergrund rückte.

(a) Brewster

(b) Holmes

Abbildung 4: Die Entwicklung des Teleskops

Lediglich das View-Master Stereoskop fand noch eine große Verbreitung, weil man mit einem Hebel sieben Stereobildpaare nacheinander betrachten konnte und es im Laufe der Zeit tausende Bildpaare dafür gab.

Es entwickelte sich parallel noch eine andere Technik die beiden Halbbilder voneinander zu trennen und zwar die anaglyphen Bilder, die man mit einer rot-grün Brille betrachte. Hier zeigte sich zum ersten Mal der Einsatz einer Brille, die ein Bild in zwei Halbbilder trennen konnte.

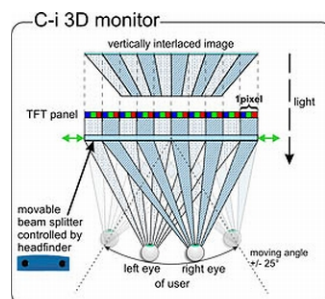
Der Computer weckte mit steigender Leistung die Hoffnung bewegte Bilder in Stereo betrachten zu können. Das war aber Anfang der achtziger Jahre nur mit den schnellsten Computern beschränkt möglich. Mit der Entwicklung leistungsfähiger 3D Grafikkarten wurden die Leistungsbarrieren schließlich durchbrochen die nötig waren um räumliche Welten plastisch, in Stereo und vor allem in Echtzeit darzustellen. Das führte schließlich zu mehreren Generationen von stereoskopischen Brillen, wie zum Beispiel der eyeSCREAM Voodoo2-3D-Brille, VFX-1 und VFX3D. Sie stellen für das linke und rechte Auge jeweils ein Bild dar. Durch die extreme Nähe der Bildebenen zu den Augen und die geringe Auflösung der Bilder blieb der Eindruck allerdings hinter den Erwartungen zurück.

Eine etwas andere Strategie verfolgten hingegen Shutterbrillen. Sie sind prinzipiell lichtdurchlässig, während der Bildschirm das linke und rechte Bild immer abwechselnd anzeigt. Die Shutterbrille schaltet dann synchron immer ein Auge auf lichtdurchlässig und das Andere auf Schwarz. Der Effekt ist, dass man einen dreidimensionalen Eindruck vom Bildschirmbild bekommt. Diese Technik hat aber den gravierenden Nachteil, dass die Bilder sehr stark flimmern, weil sich die Bildwiederholfrequenz des Bildschirmes für jedes Auge halbiert. Erst Bildschirmfrequenzen von 140 bis 160 Herz, eine extrem hohe Bildwiederholfrequenz, ist somit ausreichend für ein entspanntes 3D Bild.

Ein wesentlicher Vorteil dieses Systems war aber, dass zum handelsüblichen und akzeptierten Bildschirm nur eine Brille hinzukam, durch die man hindurch sehen konnte. Man war mit dieser Brille jederzeit in der Lage die Umgebung klar zu erkennen und zu reagieren, was mit HMD Stereobrillen nicht möglich war.

Weitere und vor allem detailliertere Informationen zur historischen Entwicklung von Stereogeräten findet man hier [Ste05].

(a) Der C-i Bildschirm



(b) Funktionsprinzip

Eine noch recht aktuelle Weiterentwicklung im Bereich der Stereobildschirme ist der C-i Bildschirm [See], der in der Lage ist ein dreidimensionales Bild ohne eine zusätzliche Brille darzustellen. Dabei werden die beiden Bilder spaltenweise voneinander getrennt auf dem Bildschirm dargestellt. Ein spezielles Blendensystem sorgt dafür, dass beide Augen jeweils nur jede zweite Spalte wahrnehmen um so beide Bilder voneinander zu trennen. Die Blende ist zusätzlich in der Lage sich an die Position der Augen anzupassen indem zwei Kameras im Rahmen des Bildschirms permanent die Position der Augen verfolgen.

Diese Technik besitzt den Nachteil, dass immer genau ein Benutzer den dreidimensionalen Effekt wahrnehmen kann, während zum Beispiel ein daneben sitzender Zuschauer beide Bilder leicht verschwommen wahrnimmt. Wenn sich der Benutzer in einem bestimmten Rahmen mit seinem Kopf bewegt richten sich die Blenden neu aus, was in nahezu Echtzeit geschieht. Bei der Betrachtung von dreidimensionalen Objekten stellt sich dann der Effekt ein, dass man sie nicht von der Seite betrachten kann, sondern sich stattdessen scheinbar nach dem Benutzer ausrichten. Dieser Effekt trägt die Tatsache Rechnung, dass immer nur genau zwei Bilder angezeigt werden und die Betrachterposition in der Applikation fix ist, solange keine konkrete Eingabe dies ändert.

Eine Weiterentwicklung dieses Bildschirms bietet inzwischen zum Beispiel die Firma Newsight [New]. Der große Fortschritt dieses Bildschirms ist ein spezielles Prismensystem auf der Bildebene, dass automatisch die Bilder für das linke und rechte Auge trennt. Eine Positionsverfolgung durch Kameras entfällt. Damit können auch mehrere Betrachter gleichzeitig den 3D Eindruck wahrnehmen. Darüber hinaus berechnet der Treiber pro Frame direkt acht Bilder, die je nach Winkel zum Bildschirm zu sehen sind. Damit gewinnt der Betrachter den Eindruck er könne durch die Bewegung seines Kopfes zum Beispiel um eine Mauerecke schauen. Ebenfalls bemerkenswert ist, dass diese Bildschirme von vornherein zu einem bezahlbaren Preis vertrieben werden, was den Einstieg in den Massenmarkt begünstigen könnte.

Ein aktuelles Problem ist und bleibt aber bei allen Bildschirmen die geringe Auflösung. Da sich zwei (oder sogar bis zu acht) Bilder die Auflösung des Bildschirms teilen müssen ist deren Zielauflösung höchstens halb so groß wie die Auflösung des Bildschirms. Es gibt aber bereits einen Ansatz um das Problem in der neuesten Generation zu verringern. Hier teilen sich die Pixel mehrerer Bilder die Farbkanäle des Bildschirms.

Einen etwas anderen Weg auf dem Gebiet der 3D Bildschirme verfolgt zur Zeit die Firma Lightspace Technologies [Lig]. Ihr erster Prototyp ist ein Bildschirm, bestehend aus zwanzig hintereinander angeordneten TFT Oberflächen und einem dahinter liegenden Beamer. Während der Beamer in einem bestimmten Takt die Ebenen projeziert schalten die TFT Oberflächen jeweils auf „durchlässig“ oder „nicht durchlässig“. Damit wird ein realer Tiefeneindruck erreicht. Problem bei diesem Entwurf ist aber zur Zeit noch das hohe Gewicht von 100 kg für einen Bildschirm normaler Größe, sowie der hohe Preis, der vom Hersteller mit 12.500 Euro angegeben wird.

2.3 Die Stereoklasse

2.3.1 Ziel

Die Klasse „StereoMaker“ hatte von Anfang an drei wichtige Ziele:

- die korrekte Darstellung der Bilder für das linke und rechte Auge auf dem C-i 3D Bildschirm
- möglichst hohe Performanz bei der Berechnung
- einfache Integration in bereits bestehende OpenGL Programme

Die korrekte Darstellung der beiden Bilder entspricht, nach aktuellem Stand, der Berechnung mittels der Off-Axis Methode (siehe Abschnitt 2.1). Schließlich werden beide Bilder spaltenweise kombiniert, wofür man einen Fragmentshader benötigt.

Um eine hohe Performanz zu gewährleisten sollten nur so wenig wie möglich „Renderpasses“ durchgeführt, und die entstandenen Bilder möglichst wenig kopiert werden.

Performanz ist besonders wichtig, weil der Berechnungsaufwand von Stereobildern bereits mindestens doppelt so groß ist, weil die Szene zweimal gerendert werden muss. Angesichts der Tatsache das die meisten 3D Programme bereits durch sehr komplexe Szenen und Shaderberechnungen viel Rechenzeit verbrauchen ist bei einer theoretischen Verdoppelung des Aufwandes darauf zu achten, dass sich die Rechenzeit nicht vielmehr als verdoppelt. Das muss deshalb so deutlich erwähnt werden, weil das hin- und herkopieren von Pufferinhalten sowie das spaltenweise Trennen der beiden Bilder mittel eines Shaders noch verzögernd zur Berechnungszeit hinzukommt.

2.3.2 GLSL vs. CG

OpenGL besitzt in seiner ursprünglichen Implementierung ein starres Pipelinekonzept zum Rendern von Dreiecken. Durch die großen Leistungsfortschritte bei den Grafikkarten mussten deshalb neue Erweiterungen (Extensions) entwickelt werden um mit der standard Pipeline neue Techniken für spezielle Effekte einsetzen zu können. In der Konsequenz führte das jedoch zu einer reduzierten Portabilität, da viele Erweiterungen herstellerspezifisch waren.

Als Alternative entwickelte man schließlich das Konzept programmierbarer Pipelinestufen, was zuerst in Assembler umgesetzt wurde. Die Programmierung in Assembler erwies sich aber in vielen Situationen als unpraktisch, sodass die Entwicklung von Hochsprachen, sogenannten *shading languages*, vorangetrieben

wurde. Eine Hardwarenahe Programmiersprache wie C kam dabei aber nur bedingt für die Semantik und die Syntax in Frage, da GPU´s zur Zeit keine indirekte Addressierung (Pointer) unterstützen.

Es entwickelten sich schließlich drei zur Zeit häufig verwendete Sprachen. Cg[1], HLSL[2] und GLSL[3]. Während HLSL für den Einsatz in DirectX Programmen entwickelt wurde ist GLSL für OpenGL Programme entwickelt worden. CG läuft dagegen auf beiden Plattformen.

GLSL gehört seit Februar 2003 offziell zum OpenGL Standard und wird ab OpenGL 1.5 unterstützt. Im Gegensatz zu CG besitzt GLSL nicht nur die typische „C“ Syntax sondern auch einen ähnlichen Präprozessor, was besondere Strategien in der Shaderentwicklung ermöglicht, auf die in Kapitel 5.3 am Beispiel des Importers eingegangen wird. Cg ermöglicht Spracherweiterungen durch ein Profilkonzept, welches je nach Hardware gewählt wird. GLSL hingegen bietet Spracherweiterungen als OpenGL Erweiterungen an, sodass man sich als Programmierer nicht direkt um ein unterstütztes Profil kümmern muss.

Entscheidung Im Verlauf der Implementierung wurde zuerst CG als Shadersprache verwendet um die beiden Bilder im letzten Schritt wieder zu kombinieren, weil damit bereits Erfahrungen gemacht wurde. Jedoch traten nach einigen Tagen Grafikprobleme bei der Einarbeitung auf, die je nach Komplexität des Shaders scheinbar willkürliche Bilder zur Folge hatten. Wie sich erst nach längerem Debugging herausgestellt hat, war der Grund für die Probleme die Inkompatibilität mit dem verwendeten Notebook, das mit einer Grafikkarte von ATI arbeitete. Es war zwar bekannt, dass CG besser mit den NVidia Grafikkarten arbeitete, jedoch nicht, dass solche Grafikprobleme mit aktuellen Grafikkarten von ATI auftreten würden.

In der Konsequenz wurde dann schließlich GLSL getestet und verwendet. Dafür gab es zwei Gründe. Zum Einen konnte somit die aktuelle Entwicklungsumgebung weiter verwendet werden, was GLSL anscheinend unterstützte, und zum Anderen widersprach eine systemabhängige ShaderImplementierung dem Ziel, die Integration der Stereoklasse für den Benutzer so einfach wie möglich zu machen. Das wurde durch die Tatsache verstärkt, dass GLSL eine Erweiterung von OpenGL ist und somit keinerlei externe Bibliotheken benötigt, im Gegensatz zu CG.

Um GLSL nutzen zu können benötigt man lediglich einen aktuellen Grafiktreiber, welcher OpenGL 1.5 unterstützt und um die Registrierung der neuen Shaderfunktionen zu vereinfachen die *glew* Bibliothek.

2.3.3 Framebuffer objects

Framebufferobjects stellen eine relativ neue Erweiterung in OpenGL dar. Sie bieten ein einfaches Interface um den Zielpuffer eines Renderprozesses zu ändern. Diese neu definierten Zielpuffer werden allgemein als *framebuffer-attachable images* bezeichnet und bieten die Möglichkeit die standard Puffer von OpenGL[4] zu erset-

[1] *„C for Graphics“ wurde von Nvidia entwickelt*
[2] *„High Level Shading Language“*
[3] *„OpenGL Shading Language“*
[4] Color-, Depth-, Accumulation und Stencilbuffer

zen. Wird solch ein Bild an ein FBO gebunden wird es von Fragmentoperationen ab sofort als Quelle und Ziel verwendet.

Der Hauptvorteil liegt schließlich in der Verwendung der *framebuffer-attachable images* als Texturen. Es ist mit Framebufferobjects möglich direkt in eine Textur zu rendern ohne wie bisher üblich zuerst in den aktiven OpenGL Framebuffer zu rendern und anschließend das Bild mit *glCopyTexSubImage* in die Textur zu kopieren.

Hierbei wird in zwei neuen Objektarten unterschieden. Das *renderbuffer object* beinhaltet ein einzelnes 2D Bild. Es unterstützt Farb- und Tiefenpuffer und darüber hinaus Textur-inkompatible Puffertypen wie den Stencilbuffer und den Accumulationbuffer.

Das *framebuffer object* subsummiert eine Reihe von verknüpften Bildern und enthält Informationen über den gesamten Frambuffer, den man letztlich an den Kontext als Zielpuffer binden kann. Damit kann man den Framebuffer wechseln, ohne den Kontext, wie beim *pbuffer*, wechseln zu müssen.

2.3.4 Implementierung

Die korrekte Berechnung der beiden Bilder wird mittels der in Abschnitt 2.1 beschriebenen Off-Axis Methode erreicht. Die Kamera wird mit den vorberechneten Frusta initialisiert und die Szene aus beiden Perspektiven gerendert. Die beiden Bilder werden in einem *framebuffer object* (FBO siehe Kapitel2.3.3) zwischengespeichert und schließlich mit einem GLSL Shader (siehe Kapitel2.3.2) verrechnet. Da ein großer Wert auf die einfache Integration in bestehenden OpenGL Code gelegt wurde, wurde die Stereoklasse so kontextunabhängig wie möglich implementiert. Das bedeutet im konkreten Fall, dass die Klasse für die Berechnung der Projektionsmatrix und der Modelviewmatrix mit gegebenen Parametern verwendet wird. Die Renderfunktion wird dabei als Parameter an die Klasse übergeben und stattdessen deren Rendermethode *SMRender()* aufgerufen. Diese sorgt für das Rendern der Szene aus beiden Augenpositionen und dem anschließendem Zusammenfügen der Bilder. Nachdem die Funktion *SMRender()* ausgeführt wurde wird das Stereobild auf dem Bildschirm angezeigt.

Die Übergabe der Modelviewmatrixparameter geschieht in der externen Renderfunktion mit der Methode *SMLookAt()*. Da die Renderfunktion nur von der Stereoklasse aus aufgerufen wird befindet sich die Stereoklasse zu dem Zeitpunkt entweder im Renderprozess des linken oder des rechten Bildes. Entsprechend wird die Kamera nach links oder rechts verschoben.

Um den Ablauf des Rendervorgangs und den Shader selbst möglichst performant zu halten wurde für das zeichnen und kombinieren beider Bilder folgende Strategie gewählt.

Der Programmierer ruft die Funktion *SMRender()* auf. Dadurch wird die Szene zweimal nebeneinander in eine Textur (FBO) jeweils halb so breit gerendert. Anschließend wird ein Polygon mit der Texture gerendert. Dabei wird der Shader zum verrechnen der Bilder aktiviert.

Der Aufbau der Hauptfunktion *SMRender()* wird in Listing 1 skizziert.

Listing 1: SMRender

```
mSceneBuffer.Bind();     // bind buffer
...
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//// left eye
// set viewport to left half of screen
glViewport(0,0, width/2, height);

...                      // load left projection matrix
(*mRenderFunc)();        // execute render function

//// right eye
// set viewport to right half of screen
glViewport(width/2,0, hidth/2, height);

...                      // load right projection matrix
(*mRenderFunc)();        // execute render function

FramebufferObject::Disable();

...                      // bind colorbuffer as texture

// set viewport to full screen
glViewport(0,0, width, height);

mStereoShader->turnOn();
glCallList(mRectangleList);   // draw our rectangle
mStereoShader->turnOff();
```

Die Entscheidung die beiden Bilder in ein gemeinsames *FBO* zu rendern, anstatt zwei halb so große *FBOs* zu verwenden wurde getroffen, weil damit das Binden der *FBOs* minimal ist (ein *bind* und ein *disable* pro Frame) und der Shader ohne konditionale Operationen auskommt um zu entscheiden, von welcher Seite er das Pixel wählen muss. Listing 2 zeigt den kompletten Vertex- und Fragmentshader, der dafür benötigt wurde. Die Entscheidung wird mit einer Modulooperation getroffen, ob der Pixel aus dem linken oder rechten Bild gewählt wird. Eine Trennung der beiden Bilder hätte eine if-Abfrage erforderlich gemacht, welche der beiden Texturen gewählt werden soll. Diese Funktion ist aber bisher noch nicht in jedem Fragmentshader verfügbar und wurde daher vermieden.

Ein interessanter Nebeneffekt dieser Vorgehensweise wird deutlich, wenn man auf den folgenden Schritt verzichtet und die beiden Bilder nebeneinander ausgibt. Das Ergebnis ist ein (horizontal gestauchtes) *cross-eye* Bild, also ein Bild, dass man bei frontaler Betrachtung mit den Augen übereinanderlegen kann. Der dabei entstehende Effekt entspricht bis zu einem gewissen Grad dem dreidimensionalen Bild des Objektes.

GLSL bietet im Fragmentshader die vordefinierte Variable *gl_FragCoord*, die die Pixelposition im Bild mit x und y angibt. Die z Komponente enthält den Tiefenwert an der Position. Es hatte sich angeboten die Modulooperation mit der x-Komponente durchzuführen, jedoch führte das zu einer Framerate von weniger als einem Frame, bei einer korrekten Anzeige des Bildes. Es ist davon auszugehen, dass die verwendete Grafikkarte bzw. deren Treiber diese GLSL Funktion nicht

unterstützt und daher die gesuchten Werte auf einem sehr inperformanten Weg abgefragt werden müssen. Der gewählte „work-around“ berechnet die aktuelle Pixelposition aus der Texturkoordinate und der Fensterbreite.

Der Vertexshader übergibt aus Performanzgründen lediglich die Position und die Texturkoordinaten des Rechtecks.

Listing 2: Vertex- und FragmentShader

```
//// VertexShader
void main()
{
  gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
  gl_Position = ftransform();
}


//// Fragmentshader
uniform sampler2D sceneColorTex;
uniform float fullTexWidth;

void main ()
{
  vec2 texcoord = gl_TexCoord[0].st;

  // this line needs 1500 ms per frame
  //float column = mod(gl_FragCoord.x, 2.0);

  float column = mod(texcoord.s * fullTexWidth, 2.0);

  texcoord.s /= 2.0 + floor(column) * 0.5;

  gl_FragColor = texture2D(sceneColorTex, texcoord.st);
}
```

2.4 Fazit

Die Anforderungen, die zu Beginn ausgearbeitet wurden, konnten alle erfüllt werden. Die Entscheidung für GLSL hat sich in diesem Kontext als richtig erwiesen. Eine Umsetzung mit CG auf einem System mit einer kompatiblen Grafikkarte ist aber ohne weiteres denkbar. Die verwendete Strategie hat sich als performant erwiesen. So stieg die Berechnungsdauer für ein Stereobild nur um ca. 77 % gegenüber dem einfachen Bild. Das der Anstieg unter 100 % liegt lässt sich durch die Tatsache erklären, dass die Gesamtgröße des gezeichneten Bildes der Szene identisch ist (die Halbbilder sind auch wirklich nur halb so groß) und das zeichnen des Rechtecks lediglich den Textur-Lookup für die beiden Halbbilder beinhaltet.

Ein sicherlich noch performanterer Weg wäre es die gesamte Szene nur einmal zu zeichnen und dabei direkt die Dreiecke in die beiden Koordinatensysteme zu transformieren und schließlich die Bilder direkt im Fragmentshader zu verrechnen, was zur Zeit mit gängigen Grafikkarten noch nicht möglich ist. Mit dieser Vorgehensweise kann man im Idealfall direkt ohne Umwege in den Framebuffer rendern, jedoch bedeutet diese Strategie auch, dass die Objekte der Szene direkt vom Stereo-Shader abhängig sind. Sollte ein Objekt mit einem eigenen Shader gezeichnet werden, hätte man diesen in den Stereo-Shader integrieren müssen. Das widersprach allerdings deutlich der Anforderung der möglichst einfachen Integrierbarkeit von bestehenden Projekten, daher wurde dieses Konzept nicht umgesetzt.

Der 3D Eindruck auf dem C-i Bildschirm ist zufriedenstellend. Es hat sich gezeigt, dass der Eindruck sogar deutlich besser ist, wenn es sich um bewegte Objekte handelt, was mit der entwickelten Klasse nun möglich ist. Erklären lässt sich das dadurch, dass die ständigen Änderungen in den Parallaxen die beiden Augen permanent kalibrieren. Dadurch lassen sich gerade die Probleme des *depth aliasing* verringern.

3 Haptik

3.1 Einleitung

Die wohl komplexeste Schnittstelle zwischen dem menschlichen Körper und seiner Umwelt stellt der haptische (Tast-)Sinn dar. Diese hohe Komplexität rührt vor allem daher, dass die haptischen Rezeptoren in der Haut liegen die den gesamten Körper von der Außenwelt abgrenzt. Permanent wirken Wahrnehmungen auf dieses System ein und bieten damit dem Menschen nicht nur eine Unterstützung bei der Orientierung, sondern auch die Fähigkeit Formen wahrzunehmen und zu manipulieren. Die Explorationsfähigkeit (Formwahrnehmung) wird dabei vorwiegend durch die sensorischen Rezeptoren ermöglicht, während die Manipulationsfähigkeit hauptsächlich von der Muskulatur abhängt.

Die Fähigkeit die Umwelt direkt manipulieren zu können ist ein grundlegender Aspekt um sie zu verstehen. (Genau genommen unterstreicht die Fähigkeit mit Objekten interagieren zu können nicht nur das Wissen um die Präsenz der Objekte, sondern auch das Wissen um die eigene Präsenz in der wahrgenommenen Umgebung.) Deshalb ist dieser Aspekt so wichtig für ein möglichst immersives Gefühl in einer virtuellen Welt. Der Grad der Immersivität ist unter anderem davon abhängig auf welche Art und Weise man die Welt manipulieren kann. Dabei ist eine möglichst natürliche und authentische Repräsentation der Manipulation, wie man sie aus realen Situationen gewöhnt ist, sicherlich die immersivste Form. Im Idealfall (mit der höchsten Immersion) würde das bedeuten die Objekte der virtuellen Objekte genauso manipulieren zu können als wären sie real.

Die gezielte Manipulation der Umwelt erfordert ein Feedback von der Umwelt. Das ist in virtuellen Welten in erster Linie das optische Resultat (ein verschobenes Objekt wird verschoben dargestellt etc.). Dieses optische Feedback enthält aber nur einen Bruchteil der Informationen die im Vergleich zur Realität mit einer Manipulation einhergehen, das Wissen um die Masse eines Körpers und die damit verbundenen Kräfte. Dieses Wissen gewinnt man durch die haptische Erfahrung des Objekts.

Wie sich zeigen wird ist die haptische Schnittstelle zwischen einem Menschen und der virtuellen Welt, was die Hardware und Software betrifft, äußerst komplex. Es gibt mehrere Ansätze die sich hinsichtlich ihrer technischen Komplexität und ihrem haptischen Eindruck unterscheiden. In fast allen Ansätzen wird dabei nur die Kraftrückkopllung simuliert, wie es auch bei dem hier verwendeten PHANTOM® Omni™Haptic Device von SensAble der Fall ist. Auf die sensorische Wahrnehmung von Oberflächentexturen und Temperaturunterschieden wird daher in diesem Kapitel nur im Abschnitt 3.3 kurz eingegangen.

Es gibt drei grundsätzliche Anforderungen, die ein haptisches Gerät so gut wie möglich erfüllen sollte:

1. Wenn keine Gegenkraft existiert darf der Benutzer auch keine erfahren (Gewicht und Widerstand des Gerätes ohne Kraftrückkopplung).
2. Feste Oberflächen müssen sich auch fest anfühlen und dürfen nicht durchdrungen werden (Finger können kurzzeitig 50 Newton Druck erzeugen, ein Handgriff sogar bis zu 400 Newton)
3. Beim Streichen über eine Oberfläche muss sich die Gegenkraft ohne Latenz anpassen (Updateraten müssen haptische Authentizität gewährleisten)

Die speziellen Anforderungen hängen darüber hinaus besonders vom Einsatzgebiet ab. Diese sind in den vergangenen Jahren im Zuge technischer Verbesserungen immer vielfältiger geworden. So gibt es inzwischen entsprechende Forschungsprojekte bei medizinischer Operationstechnik, bei der Entschärfung von Landminen, bei der Konstruktion komplexer Geräte in der Luft- und Raumfahrt, bei der haptischen Darstellung chemischer Verbindungen und auch bei der Abtastung (Scanning) realer Objekte.

Im Abschnitt 3.2 wird ein Überblick über die haptische Wahrnehmungsfähigkeit des Menschen gegeben. Im Abschnitt 3.3 wird die technische Entwicklung haptischer Geräte skizziert und im zweiten Teil gezeigt, welche Anforderungen haptische Geräte an die Software stellen.

3.2 Haptisches Empfinden

Die Haut des Menschen besteht aus drei Schichten:

1. Die Oberhaut: schützt vor Verletzungen, Hitze, Kälte und Krankheitserregern.
2. Die Lederhaut: enthält neben den Talg- und Schweißdrüsen die Nervenzellen, die Tast-, Schmerz- und Temperaturempfindungen aufnehmen.
3. Die Unterhaut: besteht aus Bindegewebe mit eingelagerten Fettzellen

Der Großteil der Rezeptoren für das haptische Empfinden befindet sich demnach in der Lederhaut.

Es gibt verschiedene Arten von Rezeptoren, die unterschiedliche Bereiche des Empfindens (besser) wahrnehmen können. Dazu zählen die so genannten *Merkelscheiben*, *Ruffini-Körperchen*, *Meissner-Körperchen* und *Pacini-Körperchen*. Sie decken gemeinsam ein Empfindungsspektrum ab, um Geschwindigkeiten, Vibrationen (0-300Hz), Beschleunigungen, statische Kräfte, Haut-Dehnungen und Temperaturen wahrzunehmen.

Die Stimulation eines oder mehrerer dieser Rezeptoren führt zur Weiterleitung des Reizes über drei Neuronen in den sensorischen Bereich des Neocortex. Dort findet schließlich die Auswertung des Reizes zur Wahrnehmung statt.

Die haptische Wahrnehmung ermöglicht schließlich auch das Konzept der *Propriozeption*. Dabei handelt es sich um die Selbstwahrnehmung des Körpers. Die Muskel-, Sehnen-, Knochen- und natürlich die Hautrezeptoren geben in ihrer Gesamtheit die Positionen der einzelnen Gliedmaßen in Relation zueinander und zur Umwelt wieder. Damit ist es dem Menschen möglich die Position seiner Hände zu bestimmen, ohne sie sehen zu müssen.

3.3 Technik

Es gab in den letzten fünfzig Jahren viele technische Ansätze für haptische Eingabegeräte, die sich in verschiedene Gruppen zusammenfassen lassen.

Zu den ersten technischen Geräten mit einer Kraftrückkopplung zählt der „Argonne Remote Manipulator“ (ARM) aus dem Jahr 1954. Er war Teil eines Master-Slave Systems in dem der Benutzer einen mechanischen Arm mit der Bewegung seiner Hände steuerte. Das Gerät wurde damals aber nicht für den Einsatz in einer

virtuellen Umgebung konzipiert, sondern für die direkte Manipulation von entfernten oder unzugänglichen Orten, wie zum Beispiel bei Tiefseeeinsätzen oder bei der Arbeit mit gefährlichen Stoffen. Bemerkenswert ist, dass dieser ARM bereits damals über Mechanismen zur Kraftrückkopplung verfügte.

1971 entwarf Hardiman ein Ganzkörper-Exoskelett. Dabei wurden die Kräfte von Armen und Beinen direkt auf das Exoskelett übertragen und bei Widerstand ebenso umgekehrt. Das Exoskelett konnte aber letztlich nur einen Arm seiner Konstruktion bewegen. Hob damit aber zum Beispiel einen Kühlschrank mit Leichtigkeit auf. Versuche seine beiden Beine gleichzeitig zu bewegen, resultierten schließlich nach eigener Aussage „in gewalttätigen und unkontrollierbaren Bewegungen".

Da das Ziel eines „haptischen Ganzkörperanzugs" eine so große Zahl an Problemen mit sich brachte versuchten einige Forschungsprojekte sich auf einzelne Körperteile zu beschränken.

Ein kompakterer Ansatz war der *Dextrous Master Manipulator* von Jones und Thousand, der ab 1981 entwickelt wurde. Dabei handelte es sich um ein Exoskelett nur für die Hand, das mit Hilfe von kleinen Zylindern arbeitete. Die Kontraktion der Zylinder konnte gemessen (um die Bewegung und Position der Finger zu ermitteln), oder zum Luftdruckaufbau genutzt werden (um so eine Kraft auszugeben).

Entwicklungen in diesem Bereich verfolgen allerdings nicht nur das Ziel Kräfte virtueller Objekte zu simulieren wie das *BLEEX* Projekt aus dem Jahr 2004 zeigt [Ber]. In diesem Projekt wurde ein Exoskelett der Beine entwickelt mit dem Ziel schwere Lasten über große Strecken ohne Ermüdung tragen zu können. Dieses Projekt wird vom Berkely Robotics Laboratory in Zusammenarbeit mit dem amerikanischen Verteidigungsministerium entwickelt und soll amerikanischen Soldaten im Einsatz helfen. Die Übertragung haptischer Informationen zwischen dem Exoskellet und dem Benutzer ist äußerst komplex, da sich der Benutzer im Idealfall vollkommen frei mit dem Gerät bewegen können soll. Die transportierte Last soll dabei völlig vom Exoskellet ausbalanciert und getragen werden.

Das erste PHANTOM® wurde 1993 von Thomas Massie am Artificial Intelligence Laboratory des Massachusetts Institute of Technology entwickelt. Massie hat daraufhin die Firma SensAble Technologies gegründet und vertreibt inzwischen weiterentwickelte Versionen des PHANTOM®.

Die Konstruktion ist einfach gehalten und funktional. Sie entspricht einem Roboterarm, der vom Benutzer geführt wird. Anstelle einer Roboterhand besitzt sie einen Stift, der vom Benutzer in der Hand gehalten und zur Navigation verwendet wird.

PHANTOM® Omni™ Das Gerät besitzt sechs passive Freiheitsgrade (drei translatorisch und drei rotatorisch) zur Navigation. Der Benutzer kann den Stift dreidimensional durch den Raum bewegen, wobei das Gerät gleichzeitig über die verschiedenen Gelenke die aktuelle Position der Hand exakt bestimmt. Die Position im lokalen Koordinatensystem des PHANTOM® lässt über den Treiber auslesen und verarbeiten.

Es besitzt darüber hinaus drei aktive Freiheitsgrade in denen mittels des Stifts haptisches Feedback gewirkt werden kann. Dazu erzeugt der „Roboterarm" des PHANTOM® die Kräfte durch eingebaute Motoren, die über Seilzüge auf die Gelenke des Arms wirken. Intern besitzt es sechs Freiheitsgrade, was der Anzahl

Abbildung 5: Haptische Eingabegeräte

Abbildung 6: Der PHANTOM® Omni™von SensAble

an Gelenken der kinematischen Kette entspricht.

Ein Vorteil dieser Konstruktion ist die exakte Messung der aktuellen Handposition und die schnelle und einfache Erzeugung der Kräfte. Laut Spezifikation beträgt die Positionsauflösung 450dpi (0.055 mm) und das Gegenkraftmaximum 0.75 lbf. (3.3 N).

Das PHANTOM® ist unabhängig von der benutzenden Person und kann im Gegensatz zu vielen anderen haptischen Eingabegeräten sofort benutzt werden.

Ein Nachteil der Konstruktion ist der indirekte Interaktionsmechanismus, welcher in etwa einer dreidimensionalen Maus entspricht. Diese indirekte Mensch-Maschine-Interaktion widerspricht smit dem direkten Umgang mit virtuellen Objekten.

Ein weiteres Problem in diesem Zusammenhang ist der eingeschränkte Aktionsradius des Gerätes der einem Volumen von 160x120x70 Millimetern (WxHxD) entspricht. Dieser Umstand ist aber bei anderen Modellen von SensAble nicht so gravierend, da diese über längere Arme verfügen.

3.4 Strategien in der Datenverarbeitung

Man unterscheidet zwei dynamische Modelle beim Einsatz von haptischen Geräten.

- Impedanz-Ansatz: Das haptische Gerät liefert die aktuelle Position und die Software berechnet daraus die gegenwärtig wirkenden Kräfte.
- Admittanz-Ansatz: Das haptische Gerät liefert die gemessenen Kräfte und die Software berechnet daraus die aktuelle Position.

Bei beiden Modellen wird die Kollisionserkennung von der Software durchgeführt.

Kraftrückkopplung Wenn eine Kollision zwischen dem haptischen Gerät und einem virtuellen Objekt festgestellt wird, muss berechnet werden welche Gegenkraft daraus folgt. Das Problem dabei ist, dass eine Kollision nur dann festgestellt wird, wenn das haptische Gerät bereits in das virtuelle Objekt eingedrungen ist. Wie weit es bereits eingedrungen ist bis die Kollision festgestellt wird ist wiederum von der Geschwindigkeit des haptischen Geräts, von der Geschwindigkeit des Objekts und vor allem von der Frequenz der Kollisionstests abhängig.

Diesem Umstand wird im so genannte „Penalty-Ansatz" Rechnung getragen. Dabei wird eine resultierende Gegenkraft F proportional zur Eindringtiefe d berechnet (siehe Formel 1). Die Steifigkeit der Feder (die gefühlte Härte der Oberfläche) wird mit k definiert.

$$F = k * d \tag{1}$$

Je weiter sich das haptische Gerät im Objekt befindet, desto größer ist die gegenläufige Kraft. Diese Berechnungsmethode bietet damit eine Lösung, wenn man sich bereits im Objekt befindet. Die diskrete Berechnung der Gegenkraft führt aber zu einem neuen Problem. Die berechnete Gegenkraft ist immer verzögert und hält zu lange an. Für den haptischen Eindruck bedeutet das, dass man mit dem haptischen Gerät nicht auf der Oberfläche verweilen kann, da man permanent eindringt und verzögert wieder hinausgedrückt wird (und beim Versuch auf der Oberfläche zu bleiben erneut eindringt etc.).

Dieses Problem kann man lösen indem man die Berechnung der Gegenkraft um eine Dämpfung b ergänzt, die von der aktuellen Geschwindigkeit des haptischen Gerätes bzw. der relativen Geschwindigkeit v zwischen dem haptischen Gerät und dem Kollisionsobjekt abhängt. Die Auslenkung der Feder wird hier mit x beschrieben.

$$F = k * x - b * v \tag{2}$$

Kraftrichtung (SCP) Die Berechnung der Kraftrichtung scheint im ersten Moment trivial, da man bei einer Kollision theoretisch die Normale der kollidierten Fläche wählen könnte. Es zeigt sich aber bei genauerer Betrachtung, dass diese Wahl nicht immer sinnvoll ist. Wenn man zum Beispiel nicht genau senkrecht zur Oberfläche in das Objekt eindringt und sich mit dem haptischen Gerät gegen die Gegenkraft im Objekt fortbewegt ist es sinnvoller den Verlaufspfad (Historie) zu berücksichtigen und die Gegenkraftrichtung daran anzupassen. Dafür bietet sich die Verfolgung des so genannten *Surface Contact Point* an. Das ist der

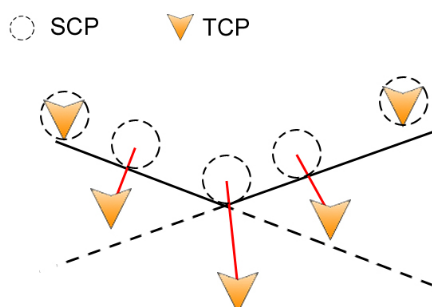


Abbildung 7: Bestimmung der SCPs beim durchstoßen einer Oberfläche.

nächstgelegene Punkt außerhalb des Objekts in Abhängigkeit des bisherigen Bewegungspfades. Abbildung stellt einen möglichen Verlauf und das entsprechende Resultat dar.

Die Berechnung eines SCP in Abhängigkeit des aktuellen *Tool Center Points* (die Position des haptischen Eingabegerätes) ergibt sich aus folgenden Schritten:

- Bestimme das Polygon, dass zwischen dem SCP^{t-1} und dem TCP^t liegt.
- Bestimme den Punkt auf dem Polygon mit dem kürzesten Abstand zum TCP^t als neuen SCP^t.
- Wiederhole die ersten beiden Schritte (max. dreimal) mit $TCP^t := SCP^t$.

Der Einsatz dieses Verfahrens führt allerdings Nachteile mit sich. So ist eine Kollision in jedem Fall lediglich auf einen Punkt und somit nur auf einen translatorischen Aspekt beschränkt. Rotationsmomente lassen sich dadurch nicht berechnen. Darüber hinaus ist der Strahltest relativ aufwändig und benötigt viel Rechenzeit. Es macht Sinn aus diesem Grund die Nachbarschaftsstrukturen von Dreiecken zu kennen, da sonst mit steigender Komplexität des Objekts auch der Aufwand steigt. Beschleunigungsmechanismen wie k-D Trees können zwar auch die Rechenzeit verringern haben aber keine konstante Updaterate.

Kraftrichtung (Voxmap/Pointshell) Einen anderen Ansatz um die Gegenkraft zu berechnen bietet das *Voxmap Pointshell* Verfahren. Hierbei werden die Kollisionsobjekte als Volumen betrachtet die von Bounding Boxes umrahmt sind, welche wiederum in einzelne Voxel unterteilt sind. Die einzelnen Voxel enthalten Informationen darüber, ob sie sich außerhalb, innerhalb oder auf dem Objekt befinden. Für den Fall, dass ein Voxel auf der Hülle liegt (also die Hülle das Voxel schneidet) wird ein Punkt (*pointshell*) auf der Hülle gespeichert und dessen Normale ermittelt. Im Falle einer Kollision wird dann die Position zur Berechnung der Kraft und die Normale zur Berechung der Kraftrichtung verwendet. Die Anzahl und die Position der Points liefern schließlich Informationen darüber wie stark und in welche Richtung die gesamte Kollisionskraft auf das Objekt wirkt. Dabei wird die Kraft in sechs Freiheitsgraden berechnet. Drei Richtungen für die Translation und drei für die Rotation. Damit bietet dieses Verfahren eine umfassendere Interaktionsmöglichkeit als das *SCP* Verfahren.

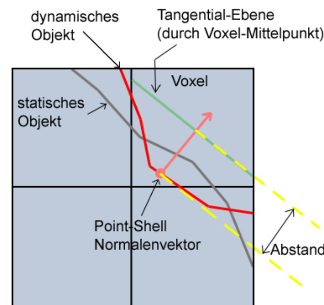


Abbildung 8: Voxmap-PointShell Kraftberechnung

3.5 OpenHaptics

OpenHaptics ist ein Toolkit, welches es dem Entwickler ermöglicht in der OpenGL Umgebung bestehende Geometriedaten um haptische Materialeigenschaften wie zum Beispiel Steifigkeit und Reibung zu erweitern. Darüber hinaus bietet es eine Schnittstelle für externe dynamische Physiksimulationen und Kollisionsmechanismen. Als haptisches Eingabegerät wird die PHANTOM® Reihe von SensAble unterstützt. Im Umfang von OpenHaptics sind neben dem DeviceDriver des haptischen Gerätes, sowie einiger Werkzeuge und Beispiele die HLAPI und die HDAPI enthalten. Dabei handelt es sich um zwei unterschiedliche Schnittstellen für den Entwickler.

Während HDAPI dem Entwickler ein Interface zur Interaktion mit dem haptischen Threads bietet (Scheduler), mit dem er Kräfte manuell berechnen und setzen kann bietet die HLAPI ein OpenGL nahes Interface zum einfachen zeichnen der Objekte in eine haptische Umgebung. Diese berechnet dann nach Angabe der haptischen Materialeigenschaften intern die Gegenkräfte und übergibt sie an das haptische Gerät.

Die hohe Konsistenz zwischen der HLAPI und OpenGL und die einfache Handhabung der haptischen Materialeigenschaften vereinfachen den Einstieg und die Integration in bestehende Programme.

Die HLAPI ist ebenso wie OpenGL als eine State-Machine konzipiert. Zustandsänderungen werden zum Beispiel mit hlBeginFrame() und hlEndFrame() gekapselt. Ebenso ist ein Matrix Stack implementiert, der die Transformationen vom OpenGL Koordinatensystem in das Koordinatensystem des Gerätes durchführt. Bekannte Matrixoperationen, wie hlPushMatrix(), hlPopMatrix(), hlLoadMatrix() und hlMultMatrix() sind implementiert.

3.6 Implementierung

Bei der Implementierung wurde das HLAPI Interface von OpenHaptics verwendet, da es die benötigten Funktionen bereitstellte. Dafür war lediglich die Anpassung der Initialisierung, die Übergabe der aktuellen Matrizen für die Projektion an das OpenHaptics Framework und das zusätzliche Zeichnen der haptischen Modelle nötig. Zusätzlich wurden Fehlerabfragen in der *idle()* Funktion implementiert, die Fehler des haptischen Gerätes zur Laufzeit anzeigen. Im Falle ei-

ner Größenänderung des OpenGL Fensters wird standardmäßig auch die Projektionsmatrix angepasst. Diese Aktualisierung muss nun auch mit der integrierten HL_TOUCHWORKSPACE_MATRIX von OpenHaptics vorgenommen werden. Auf Grund der hohen Konsistenz zwischen OpenGL und OpenHaptics werden die Matrizen mit dem gleichen Interface aktiviert.

Die Zeichnung des haptischen Objekts, was im hier verwendeten Fall dem gleichen Objekt entspricht, das auch in den Farbpuffer gezeichnet wird, ist in der Funktion *drawSceneHaptics()* implementiert (siehe Listing 3). Entscheidende Voraussetzung für die richtige Position der Objekte ist eine konsistente Transformationskette, die der Reihenfolge des eigentlichen Rendervorgangs enstprechen muss. Der Aufruf der Zeichenfunktion ist im Regelfall nicht ausreichend, da die Materialeigenschaften pro Objekt sowie der Beginn eines neuen *haptic frame* nur beim haptischen Zeichnen ausgeführt werden sollten. Schließlich wird jedes Obbjekt, wenn es in den haptischen Puffer (HL_SHAPE_FEEDBACK_BUFFER) gerendert wird nicht gleichzeitig in den Farbpuffer gezeichnet.

Listing 3: Haptisches Zeichnen mit der HLAPI

```
void drawSceneHaptics()
{
    glLoadMatrixd(gModelview);

    // Start haptic frame.
    hlBeginFrame();

    // Set material properties for the shapes to be drawn.
    hlMaterialf(HL_FRONT_AND_BACK, HL_STIFFNESS, 0.7f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_DAMPING, 0.1f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_STATIC_FRICTION, 0.2f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_DYNAMIC_FRICTION, 0.3f);

    // Start a new haptic shape.  Use the feedback buffer to
    // capture OpenGL geometry for haptic rendering.
    hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, gSceneShapeId);


    drawObject();    // draw object in OpenGL manner

    hlEndShape();    // End the shape.
    hlEndFrame();    // End the haptic frame.
}
```

3.7 Fazit

Die Integration des PHANTOM® in eine OpenGL Applikation mit OpenHaptics hat sich als sehr ökonomisch erwiesen. Dank der mitgelieferten Tutorials die die für diese Arbeit notwendigen Konzepte und Funktionen bereits enthalten war der Transfer einfach. Die Kommunikation mit dem Treiber des Endgerätes geschieht vollständig im OpenHaptics Framework, sodass man bei der Programmierung nur mit dessen Interface kommunizieren muss.

Trotz der intuitiven Handhabung wäre es für den Entwickler wahrscheinlich noch einfacher, wenn man mit dem Zeichnen des Objekts, auch gleichzeitig die

haptischen Daten rendern könnte. Das würde dann zwar intern immer noch zum gleichen Rechenaufwand führen, würde aber den Entwickler die Arbeit mit OpenHaptics und den Umgang mit dynamischen und komplexen Szenen mit vielen Objekten wesentlich vereinfachen. Schließlich muss für die haptischen Informationen immer eine angepasste zweite Renderfunktion implementiert werden. Da ansonsten so viel Wert auf die Nähe zur OpenGL Syntax und Semantik gelegt wurde ist zu vermuten, dass dieses System aus taktischen oder technischen Gründen nicht realisierbar war.

4 Der Maya Exporter

4.1 Einleitung

Maya[5] ist ein 3D Modellierungsprogramm. Es besitzt eine Fülle von Funktionen um 3D Objekte zu erstellen, zu verändern, zu animieren und mit diversen Effekten zu editieren. Die Modelle lassen sich schließlich mit einem integrierten Raytracer rendern und ganze Filme damit erstellen.

Es bietet sich an, 3D Modelle die man in seinem eigenen Programmen verwenden möchte mit Maya oder einem ähnlichen Programm zu modellieren. Jedoch stellt sich schnell die Frage wie man die erstellten Modelle aus Maya exportieren kann.

Maya bietet zwei Möglichkeiten eine Szene standardmäßig zu speichern. Das ist einmal das Binärformat mit der Endung „mb“ und das ASCII Format mit der Endung „ma“. Da das binäre Format nicht dokumentiert ist blieb schließlich nur das ASCII Format als mögliche Quelle. Es zeigte sich aber bei einem Blick in eine ASCII Datei, dass Maya eine sehr abstrakte Datenstruktur zur Speicherung der 3D Modelle, der Materialien und der Historie verwendet. Schließlich enthält eine solche Datei die kompletten Knoten des Hypergraphs der Szene sowie alle teilweise sehr spezifischen Manipulationen. Da die meisten Informationen sehr Maya spezifisch und für andere Programme irrelevant sind fiel die Wahl schließlich auf die Implementierung eines eigenen Exporters. Das Maya API stellt dafür eine spezielle Schnittstelle zur Registration von Dateiexportern bereit.

Die ersten Überlegungen führten zu speziellen Anforderungen an den Exporter. Die wohl wichtigste war von Beginn an die Benutzerfreundlichkeit des Dialogs. Das hieß im Klartext, dass der gesamte Export/Import Prozess so einfach und konsistent wie möglich verlaufen sollte, damit sich der Benutzer nicht weiter darum kümmern musste. Genau das war der Ausschlag gewesen dieses Programm zu schreiben, denn der bereits verfügbare VRML Exporter in Maya erfüllte diese Anforderung nicht[6].

Im ersten Schritt wurde die Eigenschaften, die man aus Maya exportieren können sollte zusammengestellt. Es ergab sich schließlich folgende Liste:

- Geometriedaten von Mesh Objekten, sowie Normalen und Texturkoordinaten
- grundlegende Materialeigenschaften
- Mesh Animationen, die sich auf eine Transformationsmatrix zurückführen lassen
- animiertes skinning
- Texturen mit Animationen, die sich auf die Texturmatrix zurückführen lassen
- optional der Export von Textur Rohdaten
- Nurbs Parameter
- Lichter

[5] Es handelt sich in der vorliegenden Version um Maya 6.5 Complete

[6] Es war nötig einen eigenen Importer zu schreiben um die Daten, die im ASCII Format gespeichert sind, verwenden zu können.

4.2 Maya API

Das Maya API ist eine mächtige Schnittstelle um mit externen PlugIns auf die Szene von Maya zugreifen zu können. Es bietet eine Vielzahl von Klassen um auf die Knotenelemente des Maya Hypergraphen zuzugreifen. Diese Schnittstelle stellt sich nach einiger Praxis jedoch teilweise als sehr kompliziert und verwirrend heraus. Gerade der Zugriff auf Maya Objekte und deren Attributen ist in manchen Fällen unnötig umständlich. Vermutlich wurde das Interface deshalb so gewählt, weil man keinen direkten Einblick in die Datenstrukturen von Maya bekommen soll. Es existieren inzwischen einige Tutorials im Internet, die im Umgang mit dem Maya API Hilfe versprechen (siehe [Bat06]) und halfen.

4.3 GPExporter Framework

Florian Loitsch [Loi04] hat 2004 ein Exporter Framework für Maya entwickelt und zur freien Verfügung bereitgestellt. Darüber hinaus stehen auf seiner Internetseite Auszüge die er hinsichtlich der Maya API Funktionalität dokumentiert hat. Als Beispiel wurde von ihm außerdem ein Milkshape Exporter auf Basis seines Frameworks geschrieben um die Schnittstellen zu verdeutlichen. Damit gibt er unerfahrenen Nutzern die Chance einen eigenen Exporter vom Framework abzuleiten ohne sich mit dem eigentlichen Maya API beschäftigen zu müssen. Ich habe mich nach einigen Tests schließlich dazu entschlossen dieses Framework zu verwenden und an meine Bedürfnisse anzupassen.

Da inzwischen konkrete Anforderungen an den Exporter bestanden und diese zum Teil noch nicht vom Framework erfüllt wurden musste es in vielen Teilen angepasst werden, was dazu führte, dass über achtzig Prozent des Codes umgeschrieben bzw. neu geschrieben wurden. Es erwies sich dennoch als die richtige Entscheidung dieses Framework als Basis zu verwenden, da es an vielen Stellen hilfreiche Kommentare beinhaltete, die Probleme und Inkonsistenzen in dem Maya API beschrieben. Besonders erwähnt sei an dieser Stelle auch die Struktur des Frameworks, die nur wenig verändert wurde. Sie zeichnet sich durch geschickte Vererbung von zum Teil abstrakten Klassen aus und verwendet Mehrfachvererbung. Es wurden darüberhinaus virtuelle Methoden verwendet, die bestimmte Exportoptionen lieferten und erst durch Vererbung sinnvoll überschrieben wurden.

Durch die intensive Arbeit mit dem Framework konnten schließlich wichtige Erfahrungen in diesen Programmiertechniken gemacht werden.

4.4 Serialisierung

In diesem Abschnitt möchte ich auf das programmiertechnisches Verfahren „Serialisierung" eingehen. Man versteht darunter eine sequenzielle Abbildung von Objekten auf eine externe Darstellungsform. Im hier behandelten Fall war das eine Binärdatei in der alle nötigen Informationen gespeichert wurden.
Die Konzept wurde für den Exporter folgendermaßen umgesetzt:

Eine Reihe von Klassen deren Objekte serialisierbar sind, also in der Lage sind ihre Daten i.d.R. vollständig auf einen Datenstrom zu schreiben, leiten sich von einer abstrakten Klasse ab, die alle Funktionen liefert um Member von Objekten auf einem Datenstrom zu schreiben oder von einem Datenstrom einzulesen. Die abstrakte Klasse fordert von seinen abgeleiteten Klassen die Definition einer „read"

und einer „write“ Funktion, in der jedes abgeleitete Objekt explizit die geerbten Funktionen für jedes seiner Member aufruft.

Der praktische Nutzen der Serialisierung von Daten wird besonders dann deutlich, wenn ein Member eines serialisierbaren Objektes ebenfalls ein serialisierbares Objekt ist. In diesem Fall wird auch für den Member die Prozedur durchgeführt. Dieser Prozess entspricht einer rekursiven Traversierung aller Member einer Klasse.

Während der Schreibvorgang alle Member eines Objektes auf einen Datentrom schreibt arbeitet, der Lesevorgang von einem Datenstrom konsistenter Weise genau umgekehrt, indem die Daten in der gleichen Reihenfolge wie beim Schreiben gelesen werden.

Man erreicht mit dieser Technik, dass Objekte zur Laufzeit persistent werden können, was beispielsweise bei verteilten Softwaresystemen eine wichtige Rolle spielt. Denn nach der Serialisierung liegt ein Objekt zweimal vor: als externe Darstellung (z.B. als Datei) und im Arbeitsspeicher. Wird dann nach der Serialisierung eine Änderung am Objekt im Arbeitsspeicher vorgenommen, hat das keine Auswirkung auf das serialisierte Objekt in der externen Darstellung. Sobald dieses System einmal implementiert wird und bei neu hinzukommenden Membern auf den konsistenten Aufruf obiger Methoden geachtet wird, braucht man sich in der Folge nicht mehr um das korrekte Lesen und Schreiben der Daten in eine Datei Gedanken machen.

Im Fall des Exporters bedeutete das, dass sich die gesamten zu extrahierenden Daten in einem Szeneobjekt befanden und beim Speichervorgang lediglich die Methode „write“ mit Angabe des Zieldatenstroms aufgerufen werden musste. Damit wurden alle Member der Szene (Meshes, Vertexarrays etc.) automatisch auf den Datenstrom geschrieben bzw. haben sie sich selbst auf den Strom geschrieben.

Der Import erfolgte analog.

4.5 Implementierung

Die Implementierung orientiert sich strukturell am GPExporter Framework (siehe Abschnitt 4.3) und umfasst zur Zeit einschließlich der serialisierbaren Datenstrukturen 41 Klassen. Abbildung 9 skizziert die wichtigsten Abhängigkeiten in einem UML-Diagramm.

Es wurden Extraktoren für verschiedene Aspekte implementiert, die sich von einer Basisklasse *DAGExtractor* ableiten, die allgemeine Prüf- und Traversierungsmethoden für den Maya DAG Graph enthält. Die Klasse *SceneExtractor* leitet sich schließlich mittels multipler Vererbung von verschiedenen Extraktoren ab. Sie fasst damit die Extraktion einzelner Aspekte zur logischen Extraktion der gesammten Szene zusammen.

Die Klasse *MOExporter* leitet sich vom *SceneExtractor* ab und ergänzt damit die Funktionalität um das Interface des Maya Plugins. Da der *MOExporter* von der Pluginschnittstelle die Optionsparameter für den Export der Szene erhält, diese aber bereits bei den verschiedenen Extraktoren bekannt sein müssen, wurde die abstrakte Basisklasse *Configurable* implementiert, die eine abstrakte Methode besitzt, die erst vom *MOExporter* definiert wird. Mithilfe dieser virtuellen Methode können schließlich die Optionsparameter abgefragt werden, ohne diese an jeden einzelnen Extraktor übergeben zu müssen.

Die einzelnen Extraktoren (*MaterialExtractor*, *MeshExtractor*, *NurbsCurveExtrac-*

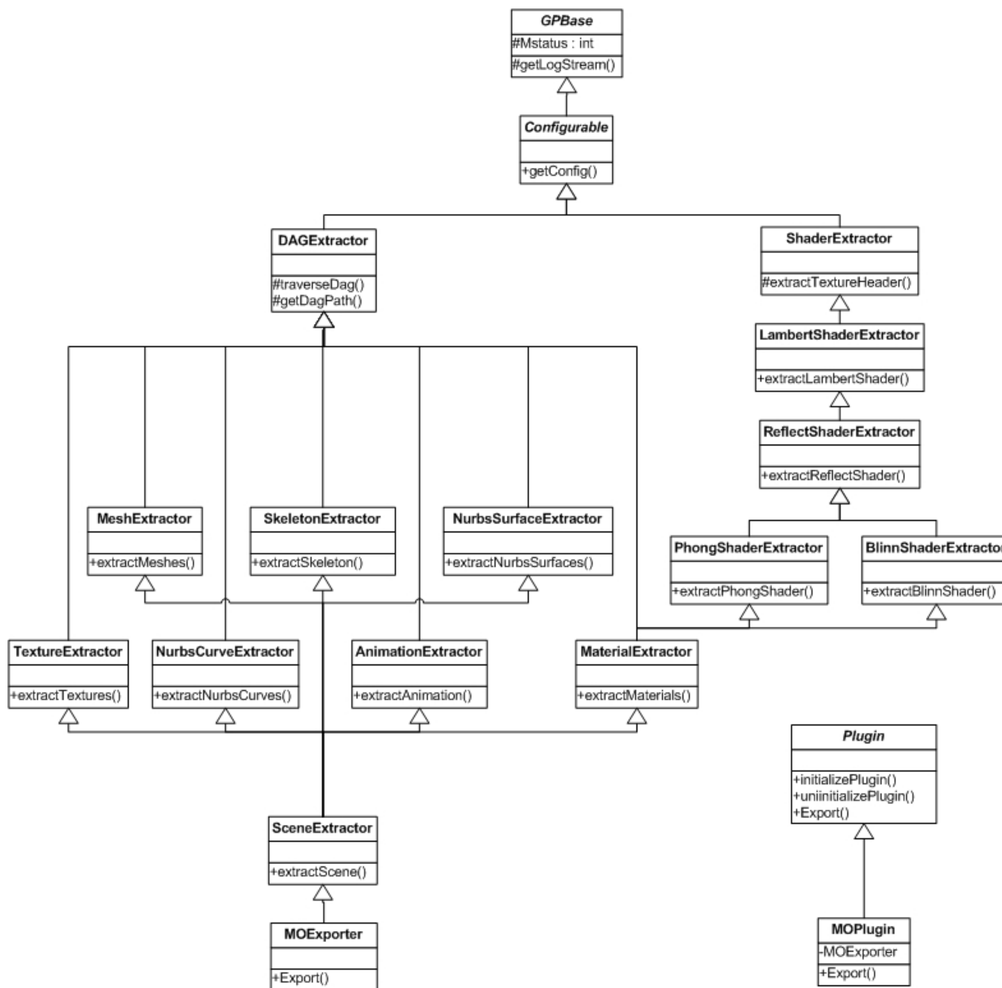


Abbildung 9: UML Diagramm des Maya Exporters

tor usw.) arbeiten unabhängig voneinander alle mit dem selben *Scene* Objekt in dem sie ihre jeweilig extrahierten Daten speichern.

Während zum Beispiel der *MeshExtractor* lediglich die Meshobjekte findet und speichert, nimmt der *MaterialExtractor* die bereits gespeicherten Meshdatensätze als Grundlage um zu entscheiden, welche Materialien exportiert werden müssen. Dabei werden auch die Meshdatensätze um die entsprechenden Materialreferenzen ergänzt. Ähnlich verhält es sich mit der Extraktion der Knochen (Joints) für das *Vertex Skinning*. Eine Sonderstellung in diesem Konzept nimmt die Klasse *SceneOptimizer* ein. Sie wurde implementiert, weil Maya das Bemalen von Vertices mit Farben (Vertexcolors) sehr inperformant verwaltet. Die einzelnen Farben werden in einem Array abgelegt das, selbst wenn das ganze Objekt nur mit einer einzigen Farbe bemalt wird, je nach Meshkomplexität mehrere tausend Einträge groß werden kann. Um diese redundanten Daten nicht mit zu exportieren wurde deshalb die Klasse *SceneOptimizer* implementiert die gleiche Farbeinträge zusammenfasst und die Indices aktualisiert. Nach dem Extraktionsprozess wird ein Objekt der

Klasse erzeugt und die Optimierungsmethode aufgerufen.

Der Exporter extrahiert neben den bekannten Meshmodellen und deren verknüpften Daten auch Nurbskurven und Flächen. Bei den extrahierten Daten handelt es sich um die verschiedenen Parameter der Funktionen. Diese Daten werden bei einem Import in ein OpenGL Programm nicht angezeigt und wurden nur der Vollständigkeit halber implementiert, da eine spätere Triangulierung von Nurbsoberflächen nicht dem Ziel des Export/Import Prozesses entspricht. Schließlich bietet Maya Funktionen zur Triangulierung von Nurbsobjekten, die im Zweifelsfall vor dem Export ausgeführt werden können.

4.6 Fazit

Die Implementierung und die damit einhergehende Auseinandersetzung mit dem Maya API zählte zu den umfangreichsten Teilen dieser Studienarbeit. Dabei wurde das Ziel, einen funktionsfähigen Exporter zu schreiben, erreicht.

Die Anforderungen konnten bis auf Eine umgesetzt werden. Dabei handelte es sich um den Export von Lichtquellen aus Maya. Der Grund dafür liegt in der Tatsache, dass beim Import von Lichtquellen die Behandlung als OpenGL Lichtquelle nicht eindeutig wäre. Es müssten erst weitere Funktionen implementiert werden, die eine konsistente Organisation der OpenGL Lichtquellen ermöglichen würden, was im Rahmen dieser Studienarbeit nicht mehr umgesetzt werden konnte. Als Aussicht wäre eine Implementierung aber denkbar. Ein wichtiger Aspekt beim Export ist die Datei in der die Daten gespeichert werden. Durch die Serialisierung werden alle Daten binär und bei einer durchdachten Implementierung ohne redundante Daten gespeichert. Dennoch lag bei einem Objekt mit bis zu neunzigtausend Polygonen die Dateigröße bei ca. 5 MB. Zudem wird bei der Speicherung der Texturrohdaten weiterer Speicherplatz benötigt.

Eine mögliche Lösung dieses Problems stellt die nachträgliche Kompression der Daten dar. Dafür käme zum Beispiel eine frei verfügbare Bibliothek wie zLib [RA] in Frage. Da die Daten bevor sie in die Datei geschrieben werden als *stringstream* vorliegen, wäre die Implementierung eines Kompressionsverfahrens für zukünftige Versionen ohne weiteres möglich.

Eine weitere Möglichkeit zur Speicherminimierung kann durch das intelligente Speichern von Matrizen gewährleistet werden. Wenn ein Objekt eine Animation über einhundert Frames besitzt, werden auch einhundert Matrizen gespeichert. Das könnte man möglicherweise dadurch verhindern, dass man lineare Abhängigkeiten zwischen den Keyframes findet und nur diese Information einschließlich der ersten und letzten Matrix speichert. Das wäre zum Beispiel für einfache Translationen von Position A nach Position B denkbar. Jedoch müsste die funktionsabhängige Geschwindigkeit beachtet werden. Ein animierter Pfad in Maya besitzt bereits standardmäßig ein *smooth-in* und *smooth-out*.

5 Der Maya Importer

5.1 Einleitung

An den implementierten Importer mussten völlig andere Anforderungen gestellt werden als an den Exporter. Er steht in direktem Zusammenhang mit einer OpenGL Applikation und ist für das korrekte Darstellen der Modelle verantwortlich.

Daher wurden zu Beginn folgende Anforderungen festgelegt:

- Alle Daten werden mittels Serialisierung aus einer Datei importiert
- Die Benutzerschnittstelle ist simpel und konsistent
- Die Darstellung der Szenenelemente ist möglichst performant
- Die Dokumentation der Klassenvorlagen ist vollständig, um die Integration zu vereinfachen
- Exportierte Texturrohdaten werden in OpenGL geladen
- Die Klassen sind unabhängig von Maya spezifischen Bibliotheken

Der Importer sollte so konzipiert sein, dass der Benutzer nur mit einer Klasse kommunizieren muss, die als Schnittstelle für die gesamte importierte Szene steht. Die Szene sollte mittels einem Dateinamen geladen und schließlich auch gezeichnet werden. Dieser Ablauf scheint der einfachste Weg für einen Entwickler zu sein Modelle in seinen OpenGL Kontext zu importieren und wurde daher weiterverfolgt.

Nach Beginn der Implementierungsphase wurden zwei weitere Anforderungen definiert.

Zum einen sollte der Importer die Darstellung von Materialien mittels GLSL Funktionalitäten zur Verfügung stellen und zum anderen eine Verwendung als Material und/oder Texturbibliothek ermöglichen. Dabei war die erste Anforderung durch die bisher gemachte Erfahrung mit GLSL und der Hoffnung auf eine höhere Qualität der exportierten Modelle motiviert.

Die Verwendung als Materialbibliothek ergab sich während der Implementierungsphase und konnte mit geringem Aufwand integriert werden.

5.2 Implementierung

Mesh Objekte Da die Serialisierung beim Import eine entscheidende Rolle spielt ist der Importer an die entsprechenden Datenstrukturen gebunden. Das bedeutet, dass alle Mesh Objekte aus verschiedenen Feldern bestehen in die aus einer Dreiecksindexliste indexiert wird. Diese Strukturierung ist für das effiziente Zeichnen von 3D Objekten in OpenGL nicht optimal, da die Indexierung von Positionen, Normalen und Texturkoordinaten viel Rechenzeit beansprucht. Daher wurden zunächst, nach dem Laden der Datenstruktur, Felder für die einzelnen Datenkomponenten und für alle Dreiecke des Objekts erstellt, um die performanteren OpenGL Feldfunktionen nutzen zu können[7].

[7] GL_VERTEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY

Was dabei zunächst außer acht gelassen wurde sind die Displaylisten[8], die zum Zeichnen von 3D Objekten verwendet wurden. Schließlich werden mit Displaylisten alle Meshdaten auf der Grafikkarte gespeichert und abgerufen, was die ursprünglich sehr inperformante Technik des Indexierens nur einmal beim Ladevorgang benötigte. Aus diesem Grund wurde schließlich die Generierung der Felder vermieden und stattdessen einmal zu Beginn eine umfangreiche Indexierung des Dreiecksobjekts durchgeführt. Damit ließ sich Speicherplatz sparen, da alle Meshinformationen bereits in der importierten Szenenstruktur vorhanden waren und nicht noch einmal reorganisiert in seperate Felder abgelegt werden mussten. Es wurden beide Möglichkeiten getestet und unter Verwendung der Displaylisten, ohne die Erstellung der Felder, eine Beschleunigung des Ladevorgangs von ca. 25% und eine geringeren Speicherbedarf von ca. 20% festgestellt. Bei einem 3D Modell mit neunzigtausend Dreiecken macht das fast zwei Sekunden[9] bzw. zehn Megabyte Speicher Unterschied aus.

Materialien Die Strategie, Displaylisten zu verwenden, ließ sich zum Teil auch für Materialien umsetzen. Hierfür wurde zunächst eine Materialmanagerklasse implementiert, die die Materialzustandsänderungen von OpenGL in Displaylisten vorkompilierte und kapselte. Da die Materialien aufgrund des vereinfachten Beleuchtungsmodells in OpenGL nur wenige Eigenschaften unterstützen (diffuse, specular, ambient, emission) sind die importierten Materialien nur Näherungen zu den ursprünglich exportierten Materialien. Auf diesen Aspekt wird in Kapitel 5.3 detaillierter eingegangen.

Texturen Texturen werden zusammen mit den Materialien ebenfalls in der Materialmanagerklasse verarbeitet. Da sie Animationen besitzen können leiten sie sich von transformierbaren Objekten ab und besitzen eine eigene Keyframeliste.

Beim Export der Texturen wird vom Benutzer entschieden, ob sie mit in der Datei gespeichert werden sollen. Ist das der Fall werden sie beim Import direkt auf die Grafikkarte geladen und der OpenGL Texturindex vermerkt. Andernfalls wird der Texturindex auf null gesetzt, was „keiner Textur“ entspricht, aber trotzdem die Transformation und gegebenfalls die Animation vermerkt. Mit dieser Strategie ist es dem Entwickler nun möglich über eine entsprechende Schnittstelle selbst die Texturindices zu setzen und zu ändern. Damit besitzt er die Möglichkeit auf Wunsch extern eingebundene Texturen für das importierte Objekt zu verwenden.

In Maya ist es möglich Texturen mit beliebigen Materialeigenschaften zu verknüpfen (z.B. eine Textur für die Transparenzeigenschaft). Diese Verknüpfungen werden beim Ex- und beim Import ebenfalls berücksichtigt. Da aber das OpenGL Beleuchtungsmodell diese Funktionen bis auf den diffusen Fall nicht unterstützt, mussten hier die Materialien vereinfacht werden. Mit GLSL Shadern konnten einige erweiterte Materialeigenschaften dennoch realisiert werden, wie in Kapitel 5.3 näher beschrieben ist.

[8]Eine OpenGL Schnittstelle zum dauerhaften speichern von Zustandsänderungen und Koordinateninformationen.

[9]Die gemessenen Zeiten beziehen sich auf den Debug Modus.

Animationen Was sich als sehr komplex herausstellte war das Animationsmanagement der Szene. Aus Konsistenzgründen werden Szenen in denen mehrere animierte Objekte vorhanden sind mit einem globalen Animationsframe gesteuert. Das bedeutet, dass alle Elemente synchron ablaufen und kein individuelles Animationsverhalten haben.

Es gibt Szenarien in denen es Sinn machen könnte ein Objekt A auf einen anderen Zeitpunkt der Animation zu setzen als ein Objekt B. Wenn das aber nötig wäre könnte man aber Objekt A und Objekt B direkt getrennt exportieren und gemeinsam in das OpenGL Programm hineinladen und individuell verwalten. Die Implementierung einer Funktion, die es dem Benutzer ermöglicht, einzelne Objekte, welche erst zur Laufzeit bekannt sind, auf einen individuellen Keyframe zu setzen, hätte die Schnittstelle für den Entwickler komplexer gemacht. Dies widerspricht einer der Anforderungen.

Als eine wichtige Erkenntnis hat sich die Strategie herausgestellt, auf welche Weise man die Animationsschritte (Keyframes) für jedes Objekt abspeichert. Die performanteste Lösung ist es, alle Transformationsmatrizen für „jeden möglichen" Keyframe zu speichern und die entsprechende Matrix zur Laufzeit auszuwählen. Diese Lösung benötigt jedoch, je nachdem im welchem Umfang Animationen verwendet werden, extrem viel Speicherplatz. Daher wurde zuerst ein anderer Ansatzverfolgt, der sich schließlich als wesentlich komplexer herausstellte.

Da Maya die entsprechende Schnittstelle bietet wurde die Translation, Rotation und Skalierung von Objekten getrennt exportiert. Als Konsequenz wurden die entsprechenden Keyframes vom Importer zur Laufzeit aus den importierten Werten interpoliert. Da die Rotation als Quaternionenrotation vorlag bedeutete das die Interpolation mit *lerp, slerp bzw. quadlerp*.

Erst nach der erfolgreichen Implementierung dieser Elemente wurde klar, dass die Interpolationsart zwischen einzelnen Keyframes in Maya eine beliebige kubische Bezierkurve sein kann und, dass diese Kurven ebenfalls extrahiert werden musste, wenn der Export nicht auf eine lineare Interpolation beschränkt sein soll. Desweiteren wurde deutlich, dass es in Maya viele Möglichkeiten und Wege gibt die Transformationseigenschaften von Objekten zu beeinflussen. Damit bestand die Gefahr, dass bestimmte Transformationsabhängigkeiten beim Export nicht korrekt erkannt wurden. Diese Probleme ergeben sich nicht, wenn man die resultierenden Transformationen[10] für jeden Frame ausliest.

Somit ergab sich ein *trade-off*[11] zwischen dem viel höheren Arbeitsaufwand, die Daten möglichst detailliert aus Maya zu extrahieren und schließlich alle Parameter pro Frame korrekt zu verrechnen und der simpleren speicheraufwändigeren Extraktion der resultierenden Tansformationsmatrizen pro Animationsframe.

Es wurde schließlich die einfachere Methode gewählt, da die Nachteile der Interpolation zur Laufzeit und der unzureichenden Extraktion der Parameter gegenüber den Vorteilen überwogen.

[10]Maya bietet eine Funktion, die die traversierte Transformationsmatrix vom „Root"-Knoten aus berechnet.

[11]Übersetzung: Zielkonflikt, Abstimmung

Knochen Die Knochen werden exportiert, wenn sie mit einem Mesh durch das sogenannte *rigid-* bzw. *smoothbinding* verknüpft sind.

Der Importer wertet die Knochen lediglich nach ihrer Transformation aus, die im *BoneManager* gespeichert werden. Dort werden die Knochentransformationen aktualisiert, falls sich der aktuelle Frame ändert. Die konkrete Verknüpfung eines Knochens mit den Vertices eines Meshobjekts wird in der Meshklasse gespeichert. Da die interpolierte Transformation einzelner Vertices eines Polygons mit der Standard OpenGL Funktionalität nicht möglich ist, höchstens in Form einer Softwarelösung, wird hierfür ein Vertexshader benötigt, der die Transformationen der Knochen entsprechend ihres Einflusses auf den Vertex interpoliert. Da es sich besonders in der Aufbereitung der Daten für den Shader um ein sehr komplexes Verfahren handelt, konnte das im Rahmen dieser Studienarbeit nicht mehr implementiert werden.

Verwendung als Textur/Material Bibliothek Im Laufe der Entwicklung zeigte sich, dass der Import von Maya Objekten und deren Eigenschaften nicht die einzige Möglichkeit ist, wie man den Importer verwenden kann. Es war gegen Ende der Implementierungsphase möglich ohne große Änderungen des Interfaces Funktionen zu implementieren, die es dem Benutzer ermöglichten auf alle im importierten Objekt vorhandenen Materialien und Texturen unabhängig zuzugreifen. Somit lässt sich der Importer auch als pure Bibliothek von Materialen verwenden, die man an und ausschalten kann. Bzw. als Bibliothek von Texturen deren OpenGL-Handle man abfragen kann. Damit kann die manuelle Organisation von Materialien und Texturen im OpenGL Programm seitens des Benutzers entfallen, da das mit Maya effektiver und übersichtlicher ist.

Zur Zeit ist die Verwendung als Materialbibliothek problematisch. Die Reihenfolge der Materialien kann noch nicht vom Anwender bestimmt werden, sondern ergibt sich aus den internen Strukturen von Maya.

5.3 GLSL Shaders

In Kapitel 2.3.2 wurde bereits die Shadersprache GLSL und deren Vorteil gegenüber dem starren Pipeline Konzept von OpenGL vorgestellt. Diese Technik wurde im Importer für die korrekte Darstellung von Materialien und Texturen implementiert. Die Verwendung ist optional und wird mit einem Präprozessorbefehl eingeschaltet.

Die Motivation GLSL zu nutzen liegt in der starken Einschränkung des Beleuchtungsmodells von OpenGL. Damit war der Unterschied zwischen dem Modell in Maya und dem gleichen Modell in OpenGL abhängig von den gewählten Materialparametern erheblich.

Die Schnittstelle zwischen dem Export und dem Import von Materialien unterstützt Informationen neben den üblichen Farbwerten Texturen für den diffusen, transparenten, specularen, emittierenden und ambienten Anteil. Diese werden bei der Aktivierung der Shader entsprechend gebunden.

Mit diesen Möglichkeiten der Materialien ergaben sich jedoch Probleme bei der Wahl der Shader. Da ein Shader immer nur zu genau einem Materialtyp passend sein kann (z.B. ein Shader, der keinerlei Texturen für die einzelnen Anteile unterstützt), hätte das bedeutet, dass für jede mögliche Kombination eines Materials ein eigener Shader existieren musste, der dann vom Materialmanager bei der

Initialisierung entschieden wird. Das hätte im obigen Fall bereits sechzehn unterschiedliche Vertex- und Fragmentshader benötigt, die sehr ähnlich sind und sich lediglich in wenigen Zeilen unterscheiden.

Es wird schnell deutlich, dass diese Strategie nicht skalierbar und äußerst inflexibel ist. Wird eine Änderung der Beleuchtungsberechnung vorgenommen, so müssten alle Shader geändert werden. Noch kritischer wird der Aufwand, wenn sich einzelne Materialattribute ändern oder hinzukommen. Die Ergänzung eines weiteren Materialattributs würde schließlich die Shaderanzahl verdoppeln, wenn man stets alle Möglichkeiten berücksichtigen will.

Die Lösung dieses Problems liegt in der Präprozessorfunktionalität von GLSL. Dabei ist das Präprozessorkonzept bei einem GLSL Shader das Gleiche wie in der Programmiersprache C++. Es können *defines* gesetzt werden die sich auf die Kompilierung des Shaders auswirken. Die Lösung unseres Problems liegt hier in der Tatsache, dass die Shader erst zur Laufzeit des Programms kompiliert werden. Zu diesem Zeitpunkt ist bereits bekannt, welche Materialien mit welchen Eigenschaften vorhanden sind. Daraus lässt sich eine Reihe von Präprozessordefinitionen für jeden Shader generieren, die einem angepassten Shader vorangestellt werden. Damit errreicht man eine bedingte Kompilierung je nachdem für welchen Materialtyp der Shader benötigt wird.

Listing 4: Konzept von Shader Praeprozessoranweisungen (VertexShader)

```
//// --- this line was added by application at runtime
#define USE_DIFFUSE_TEXTURE
//// ---

varying vec3 normal;

#ifdef USE_DIFFUSE_TEXTURE
  uniform mat4    uniform_diffuse_mat;
  varying vec2    texcoord_diffuse;
#endif


void main()
{
  normal = gl_NormalMatrix * gl_Normal;

#ifdef USE_DIFFUSE_TEXTURE
    texcoord_diffuse=(uniform_diffuse_mat*gl_MultiTexCoord0).st;
#endif

  gl_Position = ftransform();
}
```

Mit dieser Vorgehensweise besteht das Problem die erforderlichen Daten für jeden erzeugten Shader zu organisieren. Das ist kein triviales Problem, da die einzelnen Texturen in der richtigen Reihenfolge vor dem Shaderaufruf mittels einer OpenGL Erweiterung individuell gebunden werden müssen. In welcher Reihenfolge welche und wieviele Texturen schließlich angeordnet werden müssen hängt vom Shadertyp und den vorher gesetzten Texturvariablen auf dem Shader ab. Diesen Prozess skalierbar zu implementieren hat sich als einer der schwierigsten Arbeitsschritte beim Importer herausgestellt.

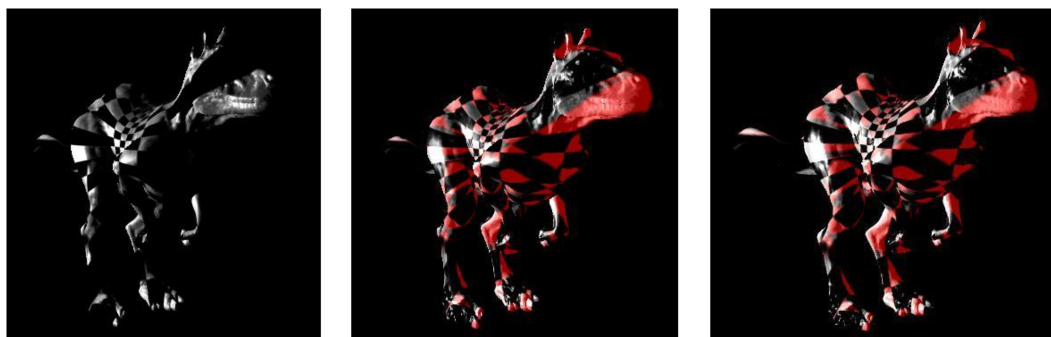


Abbildung 10: Texturemittierendes Material mit OpenGL, GLSL und Maya

5.4 Fazit

Der Importer wurde erfolgreich implementiert und alle Anforderungen bis auf eine erfüllt. Dabei handelte es sich um das *Vertex Skinning*, welches aus Zeitgründen nicht mehr umgesetzt werden konnte.

Die Integration der GLSL Shader zur Darstellung der Materialien funktioniert und bietet damit dem Benutzer weitreichende Möglichkeiten der Modellierung seiner Szenenmodelle. Abbildung 10 zeigt den Unterschied am Beispiel einer rot emittierenden Schachbretttextur.

Auf Grund der hohen Komplexität der Datenverwaltung für die einzelnen Shader ist eine Erweiterung der unterstützten Materialeigenschaften zur Zeit nur mit erheblichen Aufwand möglich. Zur Zeit wird die Verwaltung der Shaderdaten in der Klasse *Materialmanager* zusammengefasst. Eine modularere Verwaltung wäre daher eine mögliche Verbesserung.

Der Ladevorgang dauert auch bei einem 3D Modell mit neunzigtausend Polygonen nur ca. 500 ms. Lediglich Szenen mit vielen Objekten verlängern den Ladeprozess auf langsameren Computern signifikant. Wahrscheinlich ist das auf die Serialisierung (siehe Abschnitt 4.4) der Daten zurückzuführen, die bei solchen Szenen deutlich mehr Funktionsaufrufe benötigt um die Szenenstruktur rekursiv zu laden.

Die Performanz beim Darstellen der Dreiecksmodelle ist zufriedenstellend. Die OpenGL Displaylisten haben sich dabei als performantere Alternative zum Rendern von Feldern erwiesen. Als Aussicht könnte man zur Erhöhung der Performanz und zur Vereinheitlichung der Datenverwaltung eine Implementierung von *vertexbuffer objects* (VBO) vornehmen oder alternativ die entsprechende Schnittstelle der *framebuffer objects* (FBO siehe Abschnitt 2.3.3) verwenden sobald die Grafiktreiber diese Möglichkeit unterstützen.

Noch während der Implementierungsphase wurde der Importer in zwei anderen Projekten verwendet. Während dieser ersten Testphase konnten schließlich mehrere aufgetretene Fehler erkannt und behoben werden.

Bei einem der Projekte wurde auf die GLSL Funktionalität verzichtet um CG verwenden zu können. Dabei hat sich gezeigt, dass die Integration von externen Shadern ohne Probleme möglich ist, wenn sie die Materialdaten aus dem OpenGL Kontext auslesen.

6 Zusammenfassung

In den vorangegangenen Kapiteln wurde die implementierte Stereoklasse, die Anbindung von OpenHaptics, der implementierte Maya Exporter und der Importer vorgestellt. Dabei dienen der Ex- bzw. Importer dem Laden verschiedener 3D Modelle, die Stereoklasse der korrekten dreidimensionalen Darstellung auf dem C-i Bildschirm von Seereal und schließlich die Integration von OpenHaptics der haptischen Wahrnehmung durch den PHANTOM® Omni™von SensAble. Abbildung 11 und 12 illustrieren diesen Ablauf am Beispiel eines Sauriermodells, das von Maya exportiert und in ein OpenGL Programm importiert wurde. Mithilfe der beschriebenen Module konnte das Modell auf dem 3D Bildschirm mit haptischen Feedback dargestellt werden.

Koordinatentransformationen Die Applikation muss zwei Koordinatensysteme synchronisieren. Das ist zum einen das Weltkoordinatensystem des OpenGL Kontextes und zum anderen das lokale Koordinatensystem des PHANTOM® Omni™. Dabei bietet OpenHaptics bereits die Transformationsmatrizen um die Koordinaten vom PHANTOM® in ein festgelegtes „Weltkoordinatensystem“ zu transfromieren. Die Synchronisation besteht darin, die Position des haptischen Gerätes auf dem Tisch in das OpenGL Weltkoordinatensystem, welches sich in Relation zur Bildschirmoberfläche befindet, zu übertragen. Das schließt eine Skalierung, Translation und Rotation mit ein. Dieser Prozess ist theoretisch halbautomatisierbar. Dafür müsste der Bildschirm sowie mindestens drei (vom haptischen Gerät berührbare) Punkte fixiert werden. Dabei würde eine größere Anzahl von Punkten die nicht in einer Ebene liegen, den aufretenden Fehler minimieren. Da die fixen Punkte im Weltkoordinatensystem bekannt wären, könnte man die Korrespondenzen in einem linearen Gleichungssystem lösen. Der Benutzer müsste nun zu Beginn die einzelnen Punkte auf dem Tisch mit dem haptischen Gerät berühren, um beide Systeme zu kalibrieren. Als Ergebnis käme dann die Transformationsmatrix heraus um die Position des haptischen Geräts in eine Position im OpenGL Koordinatensystem zu transformieren.

Da dieser Ansatz relativ aufwändig ist wurde eine einfachere Lösung zum Kalibrieren verwendet. Dafür wurde vorausgesetzt, dass der PHANTOM® parallel zum Bildschirm ausgerichtet ist. Somit konnte man die Rotation vernachlässigen. Die Translation und Skalierung wurde dann über Testpunkte angenähert.

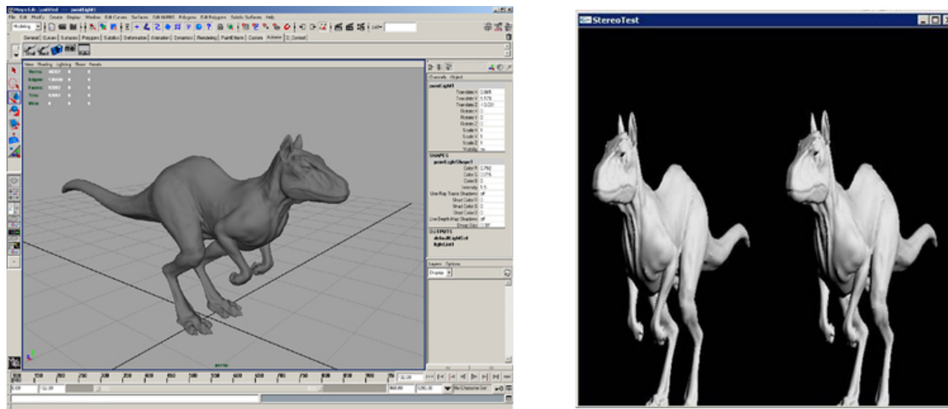


Abbildung 11: Das Modell entsteht in Maya und wird im OpenGL Programm aus zwei Blickwinkeln gezeichnet (cross-eyed-view).

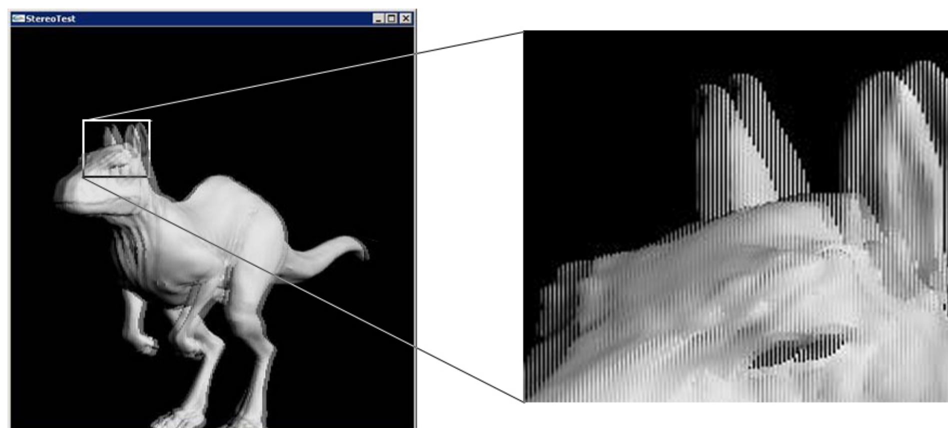


Abbildung 12: Der Stereoshader kombiniert beide Bilder spaltenweise.

Abbildung 13: Das Fokussierungsproblem

7 Fazit

Das Fazit meiner Studienarbeit ist ernüchternd. Zu Beginn dieser Arbeit bestand die Aussicht, dass die Kombination eines dreidimensionalen Bildes zusammen mit haptischen Feedback, den immersiven Eindruck deutlich erhöhen könnte. Jedoch hat die Kombination der beiden Techniken wider erwartent Probleme bereitet.

Ein großer Vorteil der Implementierung eines OpenGL Programms für den 3D Bildschirm lag darin, dass die damit möglichen Animationen den dreidimensionalen Eindruck subjektiv verbesserten (siehe Abschnitt 2.4). Einige Probleme beim Rendern von Stereobildern schienen damit minimierbar. Jedoch ist OpenHaptics nicht in der Lage animierte Objekte korrekt zu verarbeiten (siehe Abschnitt 3.7). Damit war dieser Vorteil in diesem Zusammenhang hinfällig.

Ein weiteres Problem beim Zusammenspiel beider Geräte lag im geringen Bewegungsspielraum des PHANTOM® Omni™(siehe Abschnitt 3.3). Da die Idee darin bestand mit der Spitze des haptischen Geräts scheinbar vor dem Bildschirm „schwebende“ dreidimensionale Objekte zu „berühren“, war eine gewisse Mobilität nötig. Man konnte das Objekt wegen dem kurzen Roboterarm allerdings höchstens seitlich abtasten, was dadurch erschwert wurde, dass die Objekte auf dem Bildschirm mittig platziert sein mussten, damit keine *stereo violation*eintritt.

Das letzte und entscheidende Problem ist aber auf das menschliche Auge zurück zu führen. Wie bereits in Abschnitt 2.1 beschrieben existiert, bei den derzeitigen Verfahren Stereobilder aus zwei Bildern zu generieren, das Fokussierungsproblem. Das Linsen im Auge stellen sich auf die Distanz des Bildschirms ein um das Bild mit seinen Objekten, unabhängig von der „gefühlten“ Distanz scharf wahrzunehmen. Dieser Umstand beeinträchtigt das dreidimensionale Empfinden und wird noch zusätzlich verschärft wenn man ein Objekt, wie den PHANTOM®, ins Bild bewegt. Wenn scheinbar das haptische Gerät die Oberfläche des Objektes erreicht und haptisches Feedback wahrgenommen wird entsteht ein Distanzkonflikt, den die Augen nicht lösen können.

Abbildung 13 illustriert dieses Problem. Scheinbar befindet sich der Stift und das Objekt auf einer Distanz zum Betrachter. Dennoch ist es nicht möglich beides gleichzeitig mit dem Auge zu fokussieren. Das ist ein Widerspruch den das Gehirn nicht auflösen kann. Damit ist eine erhöhte Immersion mit dieser Kombination von Techniken in diesem Anwendungsfall nicht möglich.

Literatur

[Bat06] Rob Bateman. Maya api. World Wide Web, Aug 2006. http://maya.robthebloke.org/.

[Ber] Berkely Robotics Laboratory. BLEEX Projektseite. World Wide Web. http://bleex.me.berkeley.edu/bleex.htm.

[JH06] Tomas Akenine-Möller Jon Hasselgren. An efficient multi-view rasterization architecture. In *Proceedings of EGSR 2006*. Lund University, 2006.

[JS06] Jeff Juliano and Jeremy Sandmel. Framebuffer Object Specification. World Wide Web, April 2006. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt.

[Lig] LightSpace Technologies. Webseite von LightSpace Technologies. World Wide Web. http://www.lightspacetech.com.

[Loi04] Florian Loitsch. GPExporter. World Wide Web, Aug 2004. http://florian.loitsch.com/gpExport/index-5.html.

[Mue] Stefan Mueller. Computergrafik 1 und 2 Folien der Arbeitsgruppe Computergrafik Koblenz. World Wide Web. http://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller.

[MyMS+90] Margaret Minsky, Ouh young Ming, Oliver Steele, Jr. Frederick P. Brooks, and Max Behensky. Feeling and seeing: issues in force display. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 235–241, New York, NY, USA, 1990. ACM Press.

[New] Newsight. Webseite von Newsight. World Wide Web. http://www.newsight.com.

[RA] Greg Roelofs and Mark Adler. ReleasePage of zLib 1.2.3. World Wide Web. http://www.zlib.net.

[Ros06] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[See] SeeReal Technologies. Produktseite des C-i 3D Displays. World Wide Web. http://www.seereal.com/en/products/products_ci.php.

[Sen] SensAble Technologies. Produktseite des PHANTOM®Omni™. World Wide Web. http://www.sensable.com/products/phantom_ghost/phantom-omni.asp.

[Ste05] Stefan Mueller. VRAR Materialien. World Wide Web, Aug 2005. http://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller/Teaching/WS0506/VRAR/Materialien.

[Wil78] Lance William. Casting Curved Shadows on Curved Surfaces. *Computer Graphics (Proceedings of SIGGRAPH '78)*, pages 270 – 274, 1978.