UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Universitätsklinikum Hamburg-Eppendorf

Computergestützte Rekonstruktion und Modellierung von Skelettstrukturen aus Knochenfragmenten mit haptischen Ein-/Ausgabegeräten

Diplomarbeit

zur Erlangung des Grades einer Diplom-Informatikerin
im Studiengang Computervisualistik

vorgelegt von
Kristina Gans-Eichler

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Prof. Dr. Heinz Handels (Institut für medizinische Informatik, UKE)

Koblenz, im Januar 2007

Erklärung

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

	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

In diesem Kapitel werden die Motivation und die daraus resultierenden Ziele für die vorgestellte Arbeit präsentiert.
Das darauffolgende Kapitel bietet einen Überblick über die Grundlagen der menschlichen haptischen Wahrnehmung, der zeitlichen Entwicklung der haptischen Geräte, sowie der Computerhaptik, in deren Zusammenhang das haptische Rendering erläutert wird.
Im dritten Kapitel werden die für die vorgestellte Anwendung verwendeten Bibliotheken beschrieben und deren Grundlagen erklärt. Zusätzlich wird in diesem Kapitel näher auf das genutzte haptische Gerät (PHANTOM Omni Haptic Device) eingegangen.
Das vierte Kapitel enthält eine Übersicht einer bereits bestehenden Arbeit ([4]), welche in die vorgestellte Anwendung integriert werden soll. Desweiteren wird auf die Änderungen und Optimierungen des bestehenden Ansatzes eingegangen und die Erweiterungen erläutert.
Im fünften Kapitel folgt die Beschreibung eines Anwendungsszenarios für die hier vorgestellte Arbeit. Das letzte Kapitel schließt mit einer Zusammenfassung der entwickelten Komponenten und geht auf mögliche Erweiterungen ein.

1.1 Motivation

Mit der Entdeckung der Röntgenstrahlen (1895) wurde es erstmals möglich, innere Organe und das Skelett eines Menschen sichtbar zu machen. Im Laufe der Zeit wurden bildgebende Verfahren, wie die Sonographie, die Magnetresonanztomographie (MRT) oder die Computertomographie (CT) entwickelt. Mit Hilfe der so gewonnenen Bilddaten können virtuelle, dreidimensionale Oberflächenmodelle erstellt werden.
Diese Objekte können in vielfacher Weise genutzt werden. Eine Anwendung ergibt sich in der computergestützten Planung von chirurgischen Eingriffen im Vorfeld der Operation anhand der Patientendaten.
Studenten wird die Möglichkeit gegeben, ihre Kenntnisse über die Anatomie und Beschaffenheit des menschlichen Körpers zu vertiefen. Auch in der Rechtsmedizin und Archäologie werden zunehmend bildgebende Verfahren zur Untersuchung von Knochenfragmenten eingesetzt. Durch zusätzliche Oberflächengenerierung können zum Beispiel historische Funde am Computer dreidimensional dargestellt und untersucht werden.
Der Einsatz von haptischen Geräten lässt die Interaktion des Benutzers mit den Oberflächenmodellen zu. Neben der visuellen Darstellung kann so unter anderem die Beschaffenheit von verschiedenen Gewebearten und Knochenteilen simuliert und vom Anwender erfahren werden. Das „Anfassen“ von Oberflächen wird ermöglicht, was bei der Arbeit mit einer 2D-Maus nicht der Fall ist. Der rein visuelle Eindruck wird durch ein *force feedback*

erweitert.
Mit der Integration von haptischen Geräten ergeben sich vielfältige Anwendungen; zum Beispiel können Trainingsumgebungen mit haptischer und visueller Unterstützung für komplizierte oder gefährliche medizinische Eingriffe erstellt werden. Desweiteren lassen sich beispielsweise historische Funde wiederholt anfassen und untersuchen, ohne die porösen oder zerbrechlichen Strukturen zu zerstören.
Am Institut für medizinische Informatik des Universitätsklinikums Hamburg-Eppendorf wurden bereits Trainingssimulationen (z.B. Simulation einer Lumbalpunktion) mit haptischer Unterstützung entwickelt. Auch an der Integration von Kraftrückkopplung in Operationsplanungssysteme wurde schon gearbeitet. Hier bietet besonders die Positionierung von Oberflächenmodellen mit dem haptischen Gerät eine Erleichterung. Das Anfassen und Bewegen von Modellen durch das Gerät ermöglicht ein intuitiveres Arbeiten als mit der 2D-Maus.
Ein Teil der am Institut für medizinsche Informatik entwickelten Anwendungen nutzen die institutseigene Bibliothek *mihLib*. Die Einbindung der haptischen Komponente in diese Bibliothek bietet den Vorteil ohne viel Zeitaufwand ein haptisches Gerät in eine Anwendung einbetten und die Vorzüge der Kraftrückkopplung nutzen zu können. Weitere Anwendungsfelder mit haptischer Unterstützung können entwickelt werden.

1.2 Ziel

Das Ziel der in diesem Rahmen vorgestellten Arbeit besteht darin, eine Anwendung zu entwickeln, die eine Repositionierung von Knochenfragmenten zu einer zusammenhängenden Skelettstruktur erlaubt. Die Unterstützung soll durch haptische Ein-/Ausgabegeräte erfolgen. Eine bestehende Implementierung, die im Rahmen der Diplomarbeit [4] entwickelt wurde, soll erweitert, verändert und optimiert werden.
Um die Möglichkeit bereitzustellen, die haptische Komponente in Anwendungen zu integrieren, die mit der institutseigenen Bibliothek *mihLib* hergestellt wurden, soll der schon vorhandene Ansatz, sowie die Optimierungen und Veränderungen in diese Bibliothek eingegliedert werden. Es soll eine einfache Möglichkeit entwickelt werden, das haptische Gerät in eine mit der *mihLib* erstellte Anwendung zu integrieren, um die sichtbaren Oberflächenmodelle anfassen und bewegen zu können.
Bestehende Funktionen zum Bewegen der dargestellten Modelle, sowie zum Steuern der Kamera mit dem haptischen Gerät sollen optimiert werden. Die Rekonstruktion von Knochenstrukuren soll durch Funktionen zum Schneiden von Oberflächenmodellen unterstützt werden. Die Schneidefunktionen sollen die Möglichkeit geben, zerstörte oder verformte Knochenteile zu entfernen. Vorhandene Strukturen sollen gespiegelt werden können, um die entfernten Knochenteile zu ersetzen. Um die korrekte Positonierung

von Skelettstrukturen zueinander zu vereinfachen, soll die Möglichkeit geboten werden, diese anhand eines Referenzmodelles auszurichten.

2 Haptik

Der Begriff Haptik leitet sich von dem griechischen Wort *haptikos* für *greifbar* (den Tastsinn betreffend) ab. Ursprünglich bezieht sich das Wort Haptik auf das Erfühlen, Abtasten und Manipulieren der Umwelt durch menschliche Berührung. Im Laufe der Zeit hat sich der Begriff um die Berührung und Manipulation der Umwelt durch von Menschen gesteuerte Maschinen, sowie den Austausch von haptischen Informationen zwischen Mensch und Maschine erweitert [24]. M. Srinivasan führt in [24] zusätzlich zur menschlichen Haptik die Begriffe der Maschinenhaptik und der Computerhaptik ein.

2.1 Menschliche Haptik

Die menschliche Haptik bezieht sich auf die Wahrnehmung der Umwelt durch den Berührungssinn. Eine Vielzahl von Informationen wird an das Gehirn übermittelt und ermöglicht so die haptische Wahrnehmung.

2.1.1 Haptische Wahrnehmung

Menschen sind in der Lage bestimmte Reize wahrzunehmen. Diese Reize werden über die sensorischen Nerven zum Gehirn weitergeleitet und dort verarbeitet. Über die motorischen Nerven werden Befehle an die Muskeln gesendet, die eine motorische Aktivität ausführen. Es wird unterschieden zwischen taktilen und kinästhetischen Reizen. Beide zusammengenommen bilden ein System, das umgangssprachlich als Tastsinn bezeichnet wird.
Die haptische Wahrnehmung wird in zwei Bereiche unterteilt:

- Taktile Wahrnehmung (Oberflächensensibiltät)
- Kinästethische Wahrnehmung (Tiefensensibilität)

Taktile Wahrnehmung Taktile Reize, die zur taktilen Wahrnehmung führen, werden durch bestimmte Rezeptoren in der Haut an das Gehirn übermittelt. Es findet eine Information über Druck, Temperatur, Schmerz oder Vibration statt. Die vier wichtigsten Rezeptoren, die diese Information weiterleiten, können anhand der Größe ihres rezeptiven Feldes (Bereich von dem aus ein Rezeptor erregt werden kann) und ihrer Adaptionsgeschwindigkeit unterschieden werden. Ein schnell adaptierender Rezeptor reagiert sofort auf eine Reizänderung. Hält der Reiz an, sendet der Rezeptor kaum noch Signale. Adaptiert ein Rezeptor dagegen langsam, meldet er einen Reiz so lange bis er verschwindet.
In der unbehaarten Haut befinden sich Merkel-Zellen, Meissner-Körperchen,

Pacini-Körperchen und Ruffini-Körperchen. Merkel-Zellen besitzen ein kleines rezeptives Feld und sind langsam adaptierend. Sie melden Druck, Vibration und Berührung auf einer kleinen Fläche, solange der Reiz anhält. Die Meissner-Körperchen spüren Berührung und Vibration auf kleiner Fläche und sind schnell adaptierend, das heißt sie reagieren sofort auf den Reiz, schalten sich jedoch ab, wenn der Reiz andauert. Pacini-Körperchen melden Berührung und Vibration auf einer großen Fläche und adaptieren schnell. Sie reagieren wie die Meissner-Körperchen nur auf eine Reizänderung. Ruffini-Körperchen sind langsam adaptierend, reagieren auf einen Reiz solange dieser anhält und registrieren Bewegungen der Gelenke und Dehnungen der Haut, sowie Druck und Berührung. Alle Rezeptoren sind äußerst sensitiv und haben eine niedrige Auslöseschwelle. Sie werden schon bei einem sehr schwachen Reiz aktiv.
In der behaarten Haut des Menschen sind ebenfalls Pacini- und Ruffini-Körperchen vorhanden, jedoch keine Merkel-Zellen oder Meissner-Körperchen. Stattdessen befinden sich dort Haarfolikelsensoren und Tastscheiben. Die Haarfolikelrezeptoren haben die gleiche Funktion wie die Meissner-Körperchen in der unbehaarten Haut; die Tastscheiben übernehmen die Aufgabe der Merkel-Zellen [5].

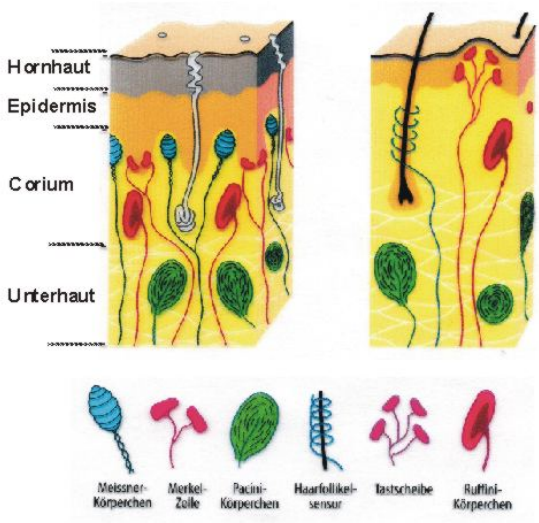

Abbildung 1: Die Rezeptoren in der Haut (Quelle: [8])

Kinästhetische Wahrnehmung Der Mensch ist fähig, die Stellung und Position seiner Körperteile, sowie ihre Lage zueinander, zu erkennen. Rezeptoren für die kinästhetische Wahrnehmung befinden sich in Gelenken, Muskeln und Sehnen. Zu den Gelenkrezeptoren gehören die Golgi-Rezeptoren, die Ruffini- und Pacini-Körperchen. Sie registrieren den Winkel und

die Winkeländerung zwischen den Extremitätenabschnitten. In den Muskeln befinden sich zwei Arten von Rezeptoren: die Golgi-Sehnenorgane, welche sich am Übergang der Muskelfasern in die Sehne befinden, und die Muskelspindeln. Golgi-Sehnenorgane geben Informationen über Spannungszustände der Muskeln an das zentrale Nervensystem weiter. Die Muskelspindeln messen die Muskeldehnung [25][23].
Abbildung 2 zeigt die besondere Rolle der Hände für die haptische Wahrnehmung. Entlang der Großhirnrinde sind verschiedene Körperabschnitte

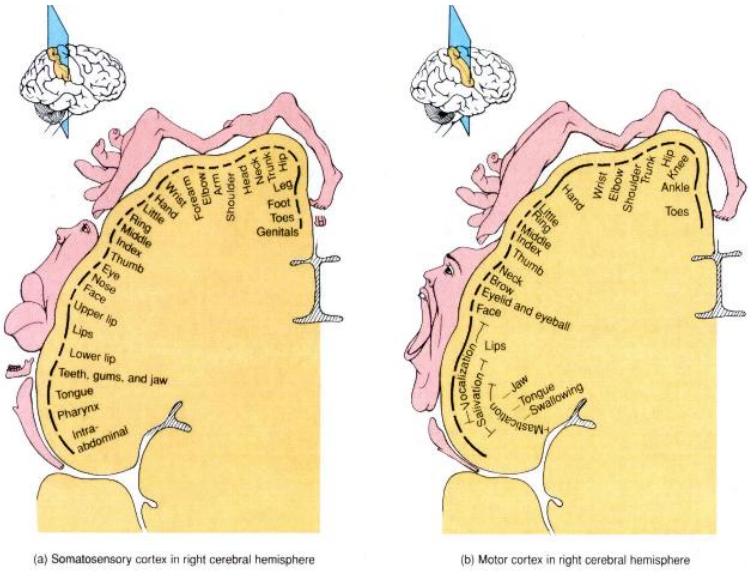

Abbildung 2: Sensibler und motorischer Homunculus (Quelle: [12])

eingezeichnet. Große Bereiche der Hirnrinde werden den Händen zugeteilt. Sie sind besonders feinsensibel und feinmotorisch. Die menschliche Hand besteht aus 29 Knochen, die das passive Element des Bewegungssystems darstellen. Aktive Elemente bilden die Muskeln, die über Sehnen mit den Knochen verbunden sind. So wird die Aktivität der Muskeln auf die Knochen übertragen. Zusätzlich besitzt die Hand eine große Zahl an Gelenken. Als wichtiges Gelenk ist das Sattelgelenk zu nennen, das sich zwischen Handwurzel und Mittelhandknochen des Daumens befindet. Aufgrund dieses Gelenks ist der Mensch in der Lage den Daumen den anderen Fingern gegenüberstellen, um so wichtige Greiffunktionen auszuführen. Ein weiterer wichtiger Aspekt ist die Verteilung der oben angesprochenen Rezeptoren in der Haut der menschlichen Hand. Die Rezeptoren, die hauptsächlich für die Empfindung von Druck und Berührung (Merkel-Zellen und Meissner-Körperchen) zuständig sind, befinden sich in großer Anzahl in der Fingerbeere (circa 23 Rezeptoren pro mm^2 [2]).

2.2 Maschinenhaptik

„Machine haptics refers to the design, construction, and use of machines to replace or augment human touch; although such machines include autonomous or teleoperated robots,...“[24]
Haptische Schnittstellen oder haptische Geräte stehen mit dem Menschen in direktem körperlichen Kontakt. Entweder wird das Gerät angefasst oder es in anderer Art und Weise am Körper befestigt. Dadurch kann Information zwischen dem Gerät und dem Benutzer ausgetauscht werden. Das Gerät ist somit bidirektional. Es nimmt Input vom Benutzer auf und gibt ihm gleichzeitig einen solchen zurück. Diese Bidirektionalität ermöglicht es, eine virtuelle oder entfernte Umgebung zu erfühlen und zu manipulieren. Das haptische Gerät bildet die Schnittstelle zwischen dem Benutzer und der virtuellen oder entfernten Umgebung, indem es *haptisches feedback* an den Benutzer zurückgibt.
Es kann zwischen *force feedback*-Geräten und Geräten, die *tactile feedback* generieren, unterschieden werden. „Force feedback integrated in a VR simulation provides data on a virtual object hardness, weight, and inertia. Tactile feedback is used to give the user a feel of the virtual object surface contact geometry, smoothness, slippage, and temperature. Finally, proprioceptive feedback in the sensing of the user's body position, or posture.“[3] M. Srinivasan gibt in [24] drei Anforderungen für kraftrückkoppelnde Geräte an, die für ein bestmögliches Arbeiten mit dem Gerät wünschenswert sind:

1. Das Gerät soll keine Kräfte auf den Benutzer ausüben, wenn dieser sich durch freien Raum bewegt. Die mechanischen Komponenten des Gerätes sollten nicht schwerfällig sein, wenig Reibung erzeugen und die Bewegung des Benutzers nicht hemmen.

2. Dem Benutzer soll es nicht möglich sein, feste Objekte zu durchdringen, indem er die Belastungsgrenze des Gerätes überschreitet. Das Gerät muss in der Lage sein, größere Kräfte zu generieren als der Benutzer aufwenden kann. Desweiteren sollte der Benutzer keine Vibrationen spüren, die zum Beispiel aufgrund einer zu niedrigen Servo-Rate entstehen können. Feste Objekte sollten sich nicht, wegen mangelnder technischer Umsetzung, weich anfühlen.

3. Das Gerät soll ergonomisch gebaut und komfortabel zu nutzen sein. Empfindet der Benutzer Unbehagen oder Schmerz beim Tragen oder Bewegen des Gerätes würde dieses Empfinden die positiven Seiten verdrängen.

2.2.1 Geschichtlicher Überblick

Mittlerweile existiert eine große Zahl an Geräten die *haptisches feedback* geben und für die unterschiedlichsten Anwendungen genutzt werden kön-

nen. Bereits in den 50er Jahren wurde das erste *force feedback*-Gerät entwickelt: der *„Argonne Remote Manipulator"* (*ARM*), welcher vom Argonne National Laboratory entwickelt wurde (Abbildung 3). Die Idee hinter dem Gerät war, es für Anwendungen in Umgebungen einzusetzen, die vom Menschen nicht unmittelbar erreicht werden können, wie beispielsweise der Weltraum oder die Tiefsee. Eine weitere Anwendung findet sich in der Arbeit mit gefährlichen Stoffen, zum Beispiel radioaktiven Materialien. Durch das Gerät wird es möglich, mit diesen Stoffen arbeiten zu können ohne unmittelbar damit in Berührung zu kommen. Das Gerät gehört zu den bilateralen Master-Slave-Manipulatoren. Ein Master-Kontrollarm wird von dem Benutzer bewegt und steuert hierbei den häufig baugleichen Slave in der entfernten Umgebung. Hier bestand bereits die Möglichkeit, Kräfte an den Benutzer des Masters zurückzugeben. Ende der 60er Jahre wurde von ei-

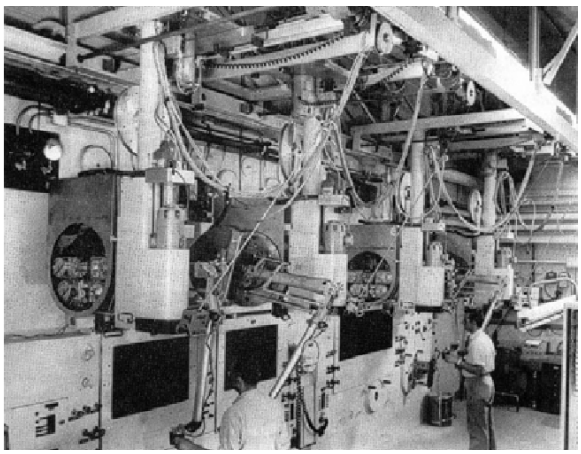

Abbildung 3: Argonne Remote Manipulator (Quelle: [20])

ner Gruppe um Prof. Brooks der Universität Chapel Hill/USA das Projekt mit dem Namen GROPE gestartet. Das Ziel dieses Projekts bestand darin, das Zusammenfügen und Modellieren von Molekülgruppen mit Hilfe von Kraftrückkopplung zu realisieren. Der Benutzer kann atomare Bindungskräfte wahrnehmen, wodurch sich das Zusammenfügen von simulierten Molekülgruppen (molecular docking) erheblich vereinfacht. Da zu diesem Zeitpunkt kein Kraftrückkopplungsgerät zur Verfügung stand, begann die Entwicklung eines solchen. Das Erste war Grope-I, das sich in den folgenden Jahren immer weiter entwickelte (Abbildung 4).

In Richtung der Exoskelettgeräte wurde ebenfalls bereits in den 60er Jahren experimentiert. Diese Geräte sollen wie ein zweites Skelett getragen werden. Der „Hardiman", entwickelt von General Electrics ab den 60er Jahren, ist ein frühes Beispiel für ein Ganzkörper-Exoskelett (Abbildung 5). Während die ersten Geräte zur Unterstützung des Menschen beim Bewegen oder Tragen von schweren Lasten entwickelt wurden, dienen heutige

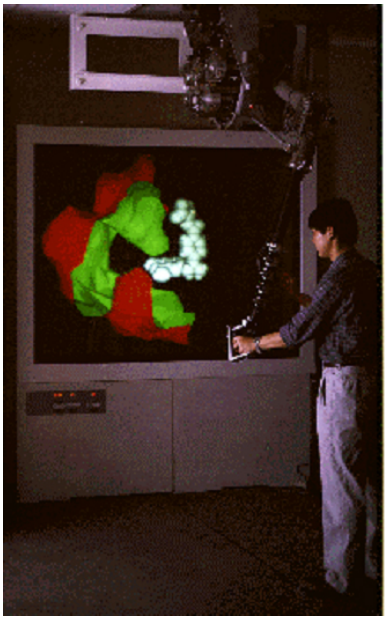

Abbildung 4: GROPE-System (Quelle: [9])

Abbildung 5: Hardiman General Electrics (Quelle: [11])

Geräte zur Interaktion mit virtuellen Umgebungen. So ist der heute erhältliche *CyberGrasp* von Immersion ein kraftrückkoppelndes Exoskelett für die Hand. Zusammen mit dem *CyberGlove*, einem Datenhandschuh, der über 18 oder 22 Sensoren verfügt, um die Stellung der Hand des Trägers zu bestimmen und in einer virtuellen Umgebung abzubilden, kann *CyberGrasp* (Abbildung 6) benutzt werden, um Objekte in einer virtuellen Umgebung

zu erfühlen und zu ergreifen.
Ende der 80er Jahre wurde mit der Entwicklung der Rutgers Master-Reihe

Abbildung 6: CyberGrasp (Quelle: [15]

begonnen. Der Rutgers Master (Abbildung 7) gehört zu den pneumatischen Geräten. In der Hand angebracht, wird mittels Drucklufteinheiten

Abbildung 7: Rutgers Master II (Quelle: [3])

Druck auf die Haut ausgeübt. So kann das Gefühl eines Kontakts mit einem Objekt erzeugt und seine Beschaffenheit bei Bewegung auf der Objektoberfläche gespürt werden.
Ein frühes System für *taktiles force feedback* war ein taktiler Joystick, mit dem die Oberfläche von Sandpapier simuliert werden konnte. Er wurde Ende der 80er/Beginn der 90er von dem MIT Media Laboratory entwickelt [3]. Ein heute erhältliches System mit *taktilem feedback* ist *CyberTouch* (Abbildung 8) von Immersion. Wie *CyberGrasp* kann *CyberTouch* mit dem *Cyber-*

Glove verwendet werden. *CyberGlove* generiert kein haptisches Feedback, sondern wird benutzt, um die Stellung der Hand zu erfassen und darzustellen. *CyberTouch* ist ein System, das mit vibrotaktilen Stimulatoren arbeitet, die an jedem Finger und an der Handfläche angebracht sind."Each stimulator can be individually programmed to vary the strength of touch sensation. The array of stimulators can generate simple sensations such as pulses or sustained vibration, and they can be used in combination to produce complex tactile feedback patterns."[15]

Auch Mäuse mit haptischem Feedback wurden entwickelt. Ein Beispiel ist

Abbildung 8: CyberTouch (Quelle: [15])

die FEELit Mouse von Immersion (Abbildung 9). Mit nur zwei Freiheitsgraden konnte man rauhe Texturen, gummiartige Materialien und harte Oberflächen erfühlen [3].

Die heute bekannntesten und am weitesten verbreiteten haptischen Ge-

Abbildung 9: FEELit Mouse (Quelle: [3])

räte, sind die der PHANTOM-Serie, die von SensAble Technologies hergestellt werden. Bereits 1993 entwickelte Thomas Massie am MIT Artificial Intelligence Laboratory das erste PHANTOM. Heute werden von SensAble vier Serien des PHANTOM angeboten: das PHANTOM Omni Haptic

Device, das PHANTOM Desktop Device, die PHANTOM Premium Devices und die PHANTOM Premium 6DOF Devices. Sie unterscheiden sich zum Beispiel in der Größe des Bereiches, in dem Kraftrückgabe möglich ist, in der Anzahl der Freiheitsgrade der Kraftrückgabe und der maximalen Rückgabekraft.

2.3 Computerhaptik

Die Computerhaptik befasst sich mit den Prozessen und Techniken, die in Zusammenhang mit der Darstellung und Erzeugung von haptischem Feedback stehen. Nicht alleine das Gerät, das das Feedback an den Benutzer gibt, ist beteiligt an der Erzeugung eines haptischen Eindruck von virtuellen oder entfernten Gegenständen/Szenerien. Analog zur Computergrafik befasst sich die Computerhaptik mit Modellen und dem Verhalten von virtuellen Objekten, zusammen mit Rendering-Algorithmen für die Darstellung in Echtzeit [24].

2.3.1 Haptisches Rendering

Um Objekte haptisch darstellen zu können, müssen sie, analog zum grafischen Rendern, haptisch gerendert werden. Allgemein gesagt müssen Kräfte vom Computer berechnet werden, die über das haptische Gerät an den Benutzer gegeben werden. Dieses Zusammenspiel eröffnet die Möglichkeit, virtuelle Objekte zu erfühlen und zu manipulieren. Ein Algorithmus für das haptische Rendern besteht aus zwei großen Teilen: der Kollisionserkennung und der Reaktion auf diese Kollision. Für die Kollisionserkennung wird zunächst innerhalb eines haptischen Renderzyklus die Gerätepposition bestimmt und auf Koordinaten in der virtuellen Szene umgerechnet. Dann werden die aktuellen Koordinaten auf Kollision mit den sich in der Szene befindenden Objekten getestet. Findet eine Kollision statt, wird auf diese reagiert, indem eine Rückgabekraft berechnet und an das Gerät gegeben wird. Ein Renderzyklus muss aufgrund der Sensitivität der menschlichen taktilen Wahrnehmung eine Update-Rate von mindestens 1kHz haben, damit der Anwender keine ungewollten Vibrationen spürt und Objekte so hart dargestellt werden können, wie es gewollt ist. Die Berechnung der Kraft findet erst statt, nachdem sich die Gerätespitze schon in dem haptisch zu rendernden Objekt befindet. Diese Position ist notwendig, um die Kraftstärke und Kraftrichtung zu berechnen.
Es können zwei Methoden für das haptische Rendering unterschieden werden: Zum einen die *vector field based methods*, die nur die aktuelle Position der Gerätespitze zur Kraftberechnung hinzuziehen und zum anderen die *constraint based methods*, die außer der aktuellen Gerätespitzenposition einen zweiten Punkt betrachten, welcher das Objekt nicht durchdringen kann.

Vector Field Based Methods Die *vector field based methods* gehören zu den ersten Ansätzen des haptischen Renderings. Ausgehend von der Eindringtiefe der Gerätespitze in das Objekt wird die Kraft berechnet. Je tiefer die Gerätespitze eindringt, desto größer wird die Rückgabekraft. Durch diese Kraftrückgabe entsteht beim Benutzer der Eindruck der Berührung einer Oberfläche.
Zur Kraftberechnung dient das in Abbildung 10 zu sehende Modell. Zwi-

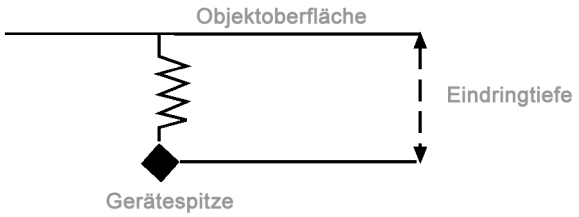

Abbildung 10: Federmodell

schen der Oberfläche des Objektes und der Position der Gerätespitze wird eine Feder gespannt. Die Kraft F wird wie folgt berechnet:

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

Neben dem Parameter d, der Eindringtiefe, existiert zusätzlich der Parameter k, die Federkonstante. Durch die Federkonstante lassen sich verschiedene Härtegrade von Federn darstellen und so verschiedene Härtegrade von Objekten simulieren. Die Kraft F steigt proportional zu Federkonstante k und Eindringtiefe d.
Da bei den *vector field based methods* nur die aktuelle Position der Gerätespitze zur Kraftberechnung betrachtet wird, können Probleme auftreten. Das erste Problem verdeutlicht Abbildung 11: Ohne die vorherigen Positionen der Gerätespitze zu kennen, ist nicht eindeutig, von welcher Seite der Benutzer in das Objekt eingedrungen ist und in welche Richtung die Rückstellkraft zu geben ist. Bei tiefem Eindringen in das Objekt kann es vorkommen, dass der Benutzer in die Richtung der anderen, näher zur Gerätespitze liegenden, Oberfläche aus dem Objekt herausgedrückt wird.
Auch bei sehr dünnen Objekten kann es zu Problemen kommen. Um eine Kraft zu generieren, die den Eindruck einer soliden Oberfläche erweckt, muss die Gerätespitze ein Stück weit in das Objekt eingedrungen sein. Bei einem sehr dünnen Objekt kann es vorkommen, dass die Gerätespitze sich zwar innerhalb des Objekte befindet, aber aufgrund der geringen Eindringtiefe nur eine unzureichende Kraft berechnet wird (Abbildung 12a)). Drückt der Benutzer daraufhin fester gegen die Oberfläche, bewegt sich die Gerätespitze weiter in das Objekt hinein. Eine Verstärkung oder Erhaltung der Kraft sollte die Folge sein. Wegen der geringen Objektdicke wird der

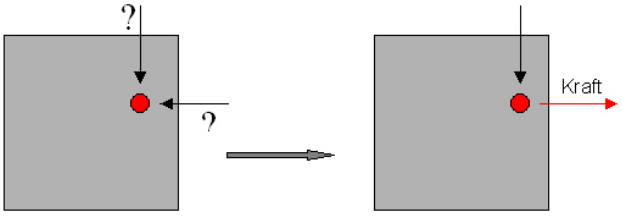

Abbildung 11: Ist nicht bekannt von welcher Seite der Benutzer in das Objekt eingedrungen ist, kann es vorkommen, dass die Ausgabe der Rückstellkraft in die falsche Richtung erfolgt.

Benutzer jedoch zur anderen Seite des Objektes hinausgedrückt und die Kraft erlischt (Abbildung 12b)). Das Objekt wurde durchstoßen.
Um komplexe Objekte mit einer *vector field based method* zu rendern, muss

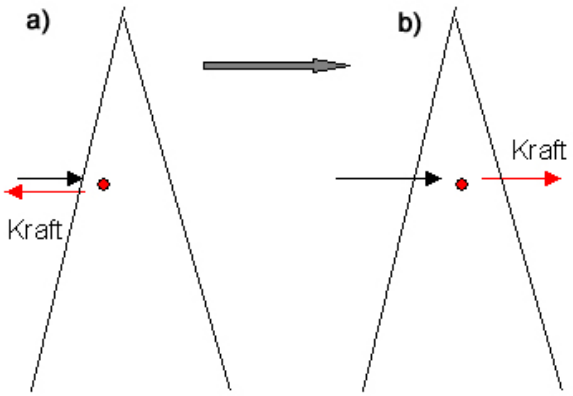

Abbildung 12: Dünne Objekte können durchstoßen werden

dieses in mehrere einfache (sich überlappende) Objekte zerlegt werden. An Stellen, an denen sich Objekte überlappen, soll die Rückstellkraft durch Summieren der Kräfte der durchdrungenen Teilobjekte berechnet werden. Stehen die Flächen senkrecht aufeinander, wird die richtige Kraft berechnet. Je stumpfer der Winkel ist, desto größer wird die aufsummierte Kraft. Die zurückgegebene Kraft ist demnach viel höher, als sie eigentlich sein müsste. Überlappen sich die Oberflächen so, dass sie fast parallel zueinander liegen, ist die Rückstellkraft fast zwei Mal zu groß. Liegen die Oberflächen so, dass sich ein spitzer Winkel bildet, wird zu wenig Kraft generiert und die Objektoberfläche nicht fest genug dargestellt.

Da die Kraft bei den frühen Verfahren entgegen der Richtung der Oberflächennormalen generiert wird, können bei Übergängen harte Kanten erfühlt werden. Dieser Effekt tritt nicht nur bei Übergängen auf, bei denen eine Kante erfühlt werden soll, sondern bei allen Übergängen. Folglich auch dort, wo sich keine Kante befindet, sondern lediglich zwei Polygonflächen aufeinanderstoßen. An diesen Stellen fühlt der Benutzer nicht vorhandene Übergänge.

Constraint Based Methods Um die Nachteile der *vector field based methods* zu minimieren wurden von Zilles et al. [32] und Ruspini et al. [21] die *constraint based methods* entwickelt.
Das Prinzip der beiden Algorithmen ist gleich: Da es nicht möglich ist, die Gerätespitze am Eindringen in ein Objekt zu hindern und das Eindringen zur Kraftberechnung benötigt wird, wird ein zweiter Punkt hinzugezogen, der *virtual proxy* [21] oder das *God-object* [32]. Diesem zweiten Punkt ist nicht erlaubt, eine Oberfläche zu durchdringen; er ist durch die in der Szene befindlichen Objekte in seiner Bewegungsfreiheit eingeschränkt (*constraint*).
In diesem Abschnitt soll auf das von Ruspini et al. [21] entwickelte Verfahren näher eingegangen werden, da es in der in dieser Arbeit verwendeten haptischen Bibliothek des OpenHaptics Toolkit Verwendung findet.
Der *virtual proxy* wird durch eine masselose Kugel repräsentiert, die sich zwischen den Objekten der Szene bewegen kann. Da es aufgrund numerischer Fehler zu kleinen Spalten zwischen benachbarten Polygonen kommen kann, muss der Radius der Kugel ausreichend groß gewählt sein, um zu verhindern, dass der Proxy durch eine dieser Spalten „fällt“. Das Ziel des Proxy besteht darin, den Abstand zu der aktuellen Position des Geräte-Endpunktes zu minimieren, indem er sich auf direktem linearen Weg zum Geräte-Endpunkt bewegt. Befindet sich der Geräte-Endpunkt in freiem Raum, kann sich der Proxy ungehindert zu diesem Punkt bewegen, da ihn keine Objekte in seiner Bewegung einschränken. Damit fallen Proxy und Geräte-Endpunkt zusammen. Um zu überprüfen, ob ein Hindernis zwischen der Gerätespitze und dem Proxy liegt oder ob sich der Proxy ungehindert zu seinem Ziel bewegen kann, wird ein Schnitttest zwischen dem Liniensegment Proxy-Gerätespitze und allen Objekten der Szene durchgeführt. Ist der Schnitttest erfolgreich, kann sich der Proxy nur so lange auf direktem Weg zum Geräte-Endpunkt entlangbewegen, bis er auf das erste Hindernis stößt. Im Anschluss wird der *configuration space* des Proxy betrachtet. In diesem Raum wird der Proxy nicht mehr als Kugel angesehen, sondern als Punkt. Daher dürfen die Objekte keine Löcher aufweisen, da der Proxy durch diese hindurchrutschen könnte. Objekte, die sich im *configuration space* des Proxy befinden, sind deswegen von einer zusätzlichen, kontinuierlich definierten Hülle in der Dicke des Proxyradius umgeben.

Die Objekte werden als *configuration space obstacles* bezeichnet (Abbildung 13). Die Ebenen, die die Bewegung des Proxy beschränken und ihn am Ein-

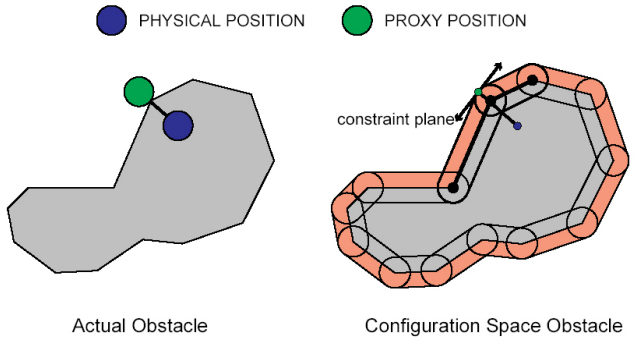

Abbildung 13: Configuration Space Obstacles (Quelle: [21])

dringen in das Objekt hindern, werden *constraint planes* genannt. Nur auf diesen Ebenen oder innerhalb des Halbraums über den Ebenen kann sich der Proxy frei bewegen. Die Ebene auf der sich der Proxy befindet wird als aktiv bezeichnet. Die zu einem Zeitpunkt aktiven Ebenen werden iterativ bestimmt. Da maximal drei Ebenen gleichzeitig aktiv sein können, wird diese Iteration drei Mal ausgeführt. Die erste Ebene, die von dem Liniensegment Proxy-Position zum Zeitpunkt $t-1$ - Geräteposition zum Zeitpunkt t geschnitten wird, wird als aktiv markiert (Abbildung 14). Danach

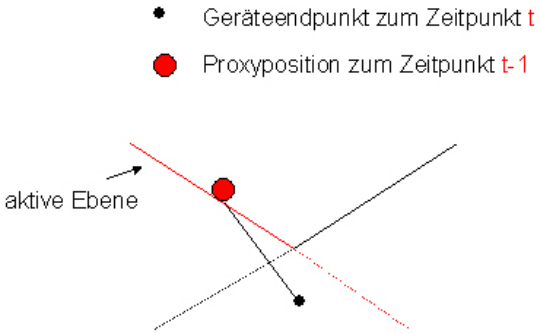

Abbildung 14: 1. Iteration Schritt 1

wird die neue Position des Proxy zum Zeitpunkt t bestimmt, die die Distanz zur Geräteposition minimiert (Abbildung 15). Anschließend wird die neue Proxy-Position als Geräteposition angenommen und nach weiteren aktiven Ebenen gesucht (Abbildung 16). Zur Verdeutlichung wird in Abbildung 17 der zweite Iterationsschritt dargestellt. Liegt der Proxy auf einer Ebene, wird nur diese Ebene als aktive *constraint plane* betrachtet und die

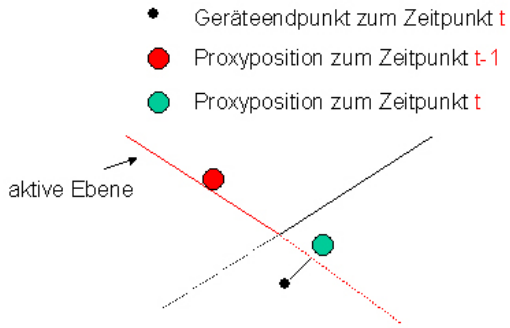

Abbildung 15: 1. Iteration Schritt 2

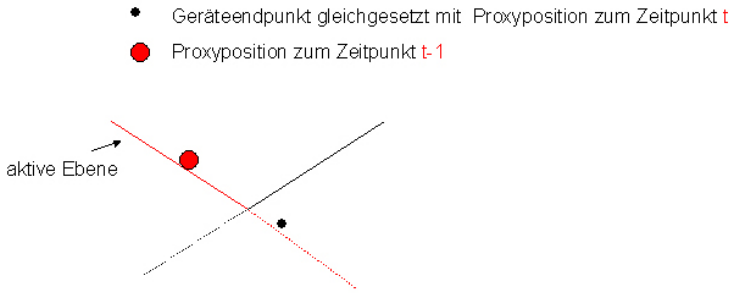

Abbildung 16: 1. Iteration Schritt 3

Proxy-Position ist auf dieser Ebene zu finden. Wenn der Proxy dagegen auf einer Kante liegt, sind die beiden an der Kante beteiligten Ebenen aktiv und die Proxy-Position, die dem Geräte-Endpunkt am nächsten liegt, befindet sich auf der Schnittgeraden der beiden Ebenen. Befindet sich der Proxy auf einer Ecke im Polygonnetz sind drei Ebenen aktiv und der Punkt mit der minimalen Distanz ist der Schnittpunkt der drei Ebenen.
Ausgehend von den aktiven *constraint planes* kann die endgültige Position des Proxy mit der geringsten Distanz zum Geräte-Endpunkt berechnet werden. Ruspini verweist in [21] zur Berechnung dieser Position auf [32]. Dieser Lösungsansatz wird im Folgenden kurz dargestellt. Die *constraint planes* seien als implizite Gleichungen gegeben:

$$Ax + By + Cz - D = 0 \tag{2}$$

Die folgende Gleichung repräsentiert die Energie einer virtuellen Feder mit Federkonstante 1. x, y und z sind die Koordinaten des virtuellen Proxy und x_p, y_p und z_p stehen für die Koordinaten der Gerätespitze.

$$Q = \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2 \tag{3}$$

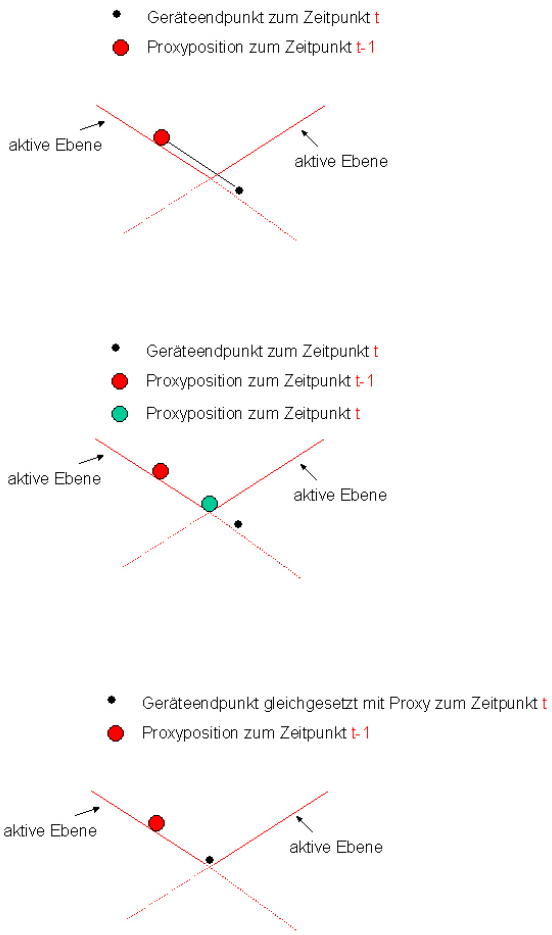

Abbildung 17: 2. Iteration

Um die neuen Koordinaten des Proxy zu berechnen, werden La Grange-Multiplikatoren verwendet. L muss in folgender Gleichung minimiert werden:

$$\begin{aligned} L = & \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2 \\ & + l_1 (A_1 x + B_1 y + C_1 z - D_1) \\ & + l_2 (A_2 x + B_2 y + C_2 z - D_2) \\ & + l_3 (A_3 x + B_3 y + C_3 z - D_3) \end{aligned}$$

Die Minimierung geschieht durch Nullsetzen der sechs partiellen Ableitungen nach den Variablen (x, y, z, l_1, l_2, l_3):

Ableitung nach x:

$$x^{'} = x - x_p + l_1 A_1 + l_2 A_2 + l_3 A_3 = 0 \tag{4}$$

Ableitung nach y:

$$y^{'} = y - y_p + l_1 B_1 + l_2 B_2 + l_3 B_3 = 0 \tag{5}$$

Ableitung nach z:

$$z^{'} = z - z_p + l_1 C_1 + l_2 C_2 + l_3 C_3 = 0 \tag{6}$$

Ableitung nach l_1

$$l_1^{'} = A_1 x + B_1 y + C_1 z - D_1 = 0 \tag{7}$$

Ableitung nach l_2

$$l_2^{'} = A_2 x + B_2 y + C_2 z - D_2 = 0 \tag{8}$$

Ableitung nach l_3

$$l_3^{'} = A_3 x + B_3 y + C_3 z - D_3 = 0 \tag{9}$$

Daraus ergibt sich folgendes Gleichungssystem in Matrixschreibweise:

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 & A_3 \\ 0 & 1 & 0 & B_1 & B_2 & B_3 \\ 0 & 0 & 1 & C_1 & C_2 & C_3 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & 0 & 0 \\ A_3 & B_3 & C_3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ l_1 \\ l_2 \\ l_3 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} \tag{10}$$

Diese Matrixgleichung entsteht für den Fall, dass drei *constraint planes* aktiv sind. Bei weniger aktiven Ebenen werden Nullen an die entsprechenden Stellen gesetzt und es ensteht für eine bzw. zwei aktive Ebenen eine Matrixgleichung mit einer 4x4-Matrix bzw. 5x5-Matrix.
Auch bei den *constraint based methods* besteht das Problem, dass die Übergänge zwischen den einzelnen Polygonen sehr abrupt sind und der Benutzer harte Übergänge fühlt. Das resultiert aus der Berechnung der Kraftrichtung mit den Normalen der Polygonen. Der nächste Abschnitt geht kurz auf das *Force Shading* ein, welches das Problem des Fühlens von harten Übergängen löst.

Force Shading Der Benutzer fühlt bei Übergängen von Polygonen scharfe Kanten, wenn die Rückstellkraft in jedem Punkt, den der Benutzer auf dem Polygon berührt, in Richtung der Oberflächennormale des Polygons berechnet wird. Um diesen Effekt abzumildern werden beim *Force Shading*

die Normalen zu den Oberflächenpunkten interpoliert. Dies geschieht ähnlich der Interpolation von Normalen, die bei Farbinterpolationsverfahren (zum Beispiel *Gouraud* oder *Phong Shading*) angewendet wird, wie sie aus der Computergrafik bekannt sind.
Um die Normale in dem Punkt zu berechnen, in dem der Benutzer das Polygon berührt hat (Berührungspunkt), werden die Normalen der Eckpunkte des Polygons benötigt. Diese erhält man durch eine Mittelung der Oberflächennormalen der an die Eckpunkte grenzenden Polygone. Anschließend wird aus den Eckpunktnormalen des Polygons mittels Interpolation die Normale zu dem Berührungspunkt berechnet. Für ein Polygon mit k Eckpunkten und somit N_k Eckpunktnormalen geschieht dies nach folgender Formel:

$$\frac{\sum_{i=1}^{k}\left(x_i\left(\left(\sum_{j=1}^{k}N_j\right)-N_i\right)\right)}{\sum_{i=1}^{k}x_i} \tag{11}$$

x_i bezeichnet den Abstand zwischen dem Berührungspunkt und der Eckpunktnormalen N_i (Abbildung 18).
Auch Ruspini beschreibt in [21] das *Force Shading* in Verbindung mit sei-

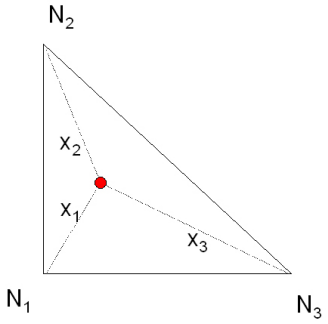

Abbildung 18: Berechnung der interpolierten Normalen in einem Oberflächenpunkt

nem Proxy-Algorithmus: Die Berechnung der neuen Proxy-Position in Verbindung mit *Force Shading* erfolgt in zwei Schritten (Abbildung 19). Der erste Schritt besteht darin, die neue Zwischenposition des Proxy auf Basis der Ebenen zu berechnen, die orthogonal zu den interpolierten Normalen (*force shading planes*) stehen, anstatt auf Basis der *constraint planes*. Diese Zwischenposition wird dann als Geräte-Endpunkt angesehen. Im zweiten Schritt wird die neue Proxy-Position für diese Iteration berechnet. Dies geschieht unter Verwendung der vorher berechneten Zwischenposition und der nicht-interpolierten *constraint plane*.
Im linken Teil der Abbildung 20 ist zu sehen, dass starke Sprünge zwischen den Normalen an Kanten der Oberfläche auftreten. Da die Rückstellkraft in Richtung der Normalen generiert wird, führt das zu, für den Benutzer

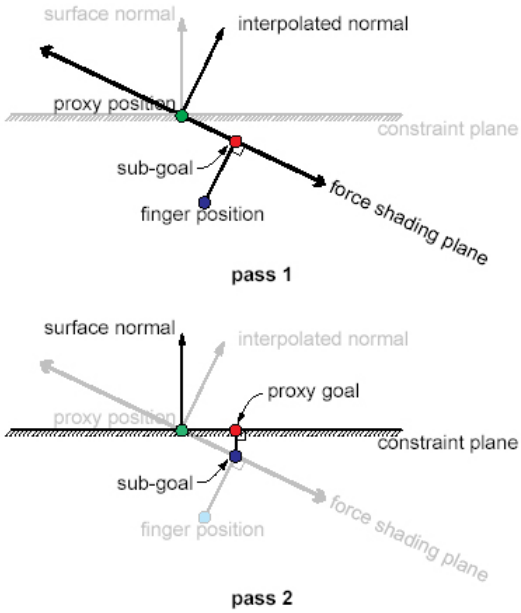

Abbildung 19: Force Shading (Quelle: [21])

spürbaren, harten Übergängen. Im rechten Teil ist zu erkennen, dass die Übergänge der Normalen an Kanten keine große Sprünge mehr aufweisen. Die Bewegung des Proxy auf der Oberfläche wird kontinuierlicher und der Benutzer nimmt die Kantenübergänge durch die Rückgabe der Kraft in Richtung der interpolierten Normalen als weicher wahr [21].

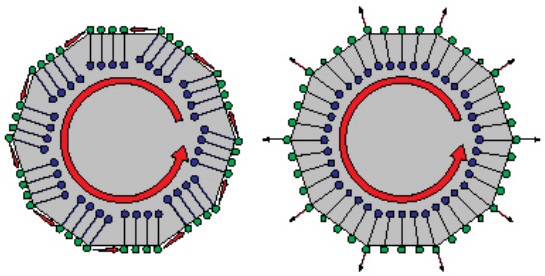

Abbildung 20: **Links:** ohne Force Shading **Rechts:** mit Force Shading (Quelle: [21])

3 Verwendete Geräte und Bibliotheken

In den nächsten Abschnitten wird auf Geräte und Bibliotheken eingegangen, die für die vorliegende Arbeit verwendet wurden.

3.1 PHANTOM Omni Haptic Device

Das PHANTOM Omni Haptic Device gehört zur PHANTOM-Serie von SensAble Technologies. Außer dem „Omni“ werden drei weitere Geräteserien angeboten: das PHANTOM Desktop Device, die PHANTOM Premium Devices und die PHANTOM Premium 6DOF Devices. Alle Geräte gehören zu den stiftbasierten haptischen Geräten.
Das haptische Gerät kann als Nachbildung des menschlichen Armes beschrieben werden. An der Basis des Gerätes sitzt ein Gelenk, das ähnlich dem Schultergelenk, eine Rotation und Kippung um das Zentrum des Gelenkes zulässt. Es folgt ein Gelenk, das nur eine Kippbewegung ermöglicht und somit dem Ellbogengelenk nachempfunden ist. Das letzte Gelenk ist mit dem Handgelenk vergleichbar. Es lässt sich drehen und kippen. An ihm angebracht, befindet sich der stiftähnliche Griff (Endeffektor) mit dem der Benutzer interagieren kann. Dieser lässt sich um den eigenen Schaft drehen. Um die Position des Endeffektors zu berechnen, befinden sich in den Gelenken Sensoren, die die Stellung der jeweiligen Gelenke auslesen. Abbildung 21 zeigt das Gerät mit den möglichen Bewegungen.
Die Position des Endeffektors kann auf etwa 0.005 mm genau bestimmt

Abbildung 21: PHANTOM Omni Haptic Device (Quelle: [27] Original ohne Pfeile)

werden. Der haptische Arbeitsbereich des Gerätes beträgt 160 mm in der

Breite, 120 mm in der Höhe und 70 mm in der Tiefe. Das Gerät kann eine maximale Rückgabekraft von 3.3 Newton erzeugen. In Richtung der x-Achse des Gerätekoordinatensystems kann eine Steifigkeit von 1.26 N/mm, in Richtung der y-Achse eine Steifigkeit von 2.31 N/mm und in Richtung der z-Achse eine Steifigkeit von 1.02 N/mm erzeugt werden. Im Gegensatz zu dem PHANTOM Premium 1.5 6DOF kann das „Omni“ keine Kraft auf die Drehung des Endeffektors ausüben. Um eine komfortable Nutzung des Gerätes zu gewährleisten, beträgt die maximale Reibung bei der Bewegung im freien Raum 0.26 Newton.
SensAble bietet zwei Bibliotheken, die in Verbindung mit den Geräten der PHANTOM-Serie genutzt werden können: das OpenHaptics ToolKit und das GHOST SDK. Das Omni Haptic Device ist lediglich mit dem OpenHaptics ToolKit kompatibel.

3.2 OpenHaptics ToolKit

Das OpenHaptics ToolKit (Abbildung 22) besteht neben Gerätetreibern, Source Code-Beispielen und Benutzerhandbüchern aus zwei weiteren wichtigen Teilen: der HDAPI (haptic device API) und der HLAPI (haptic library API). Die HLAPI setzt auf der HDAPI auf und bietet dem Benutzer die

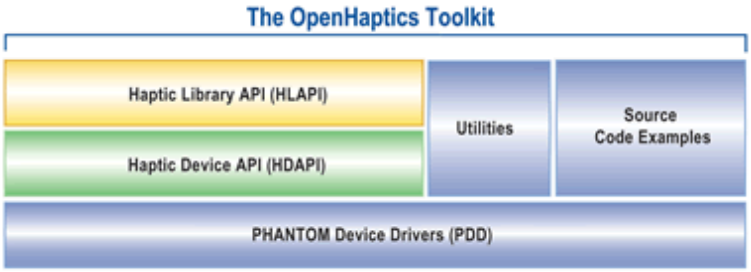

Abbildung 22: Komponenten des OpenHaptics ToolKit (Quelle: [27])

Möglichkeit auf einfachem Weg haptische Komponenten in ein grafikbasiertes Programm zu integrieren. Während die HLAPI dem Benutzer zum Beispiel die Aufgabe der Berechnung von Kräften und der Wahrung von Thread-Sicherheit abnimmt, müssen diese Elemente bei Nutzung der HDAPI vom Benutzer selbst gesteuert werden. Bei Bedarf können Elemente beider APIs miteinander kombiniert werden.
Da in der angefertigten Arbeit lediglich die HLAPI Verwendung findet, wird im folgenden Abschnitt näher auf diese eingegangen.

3.2.1 HLAPI

Die Abbildung 23 bietet einen groben Überblick der Komponenten, die in der HLAPI integriert sind. Eine rein grafische Anwendung benötigt ei-

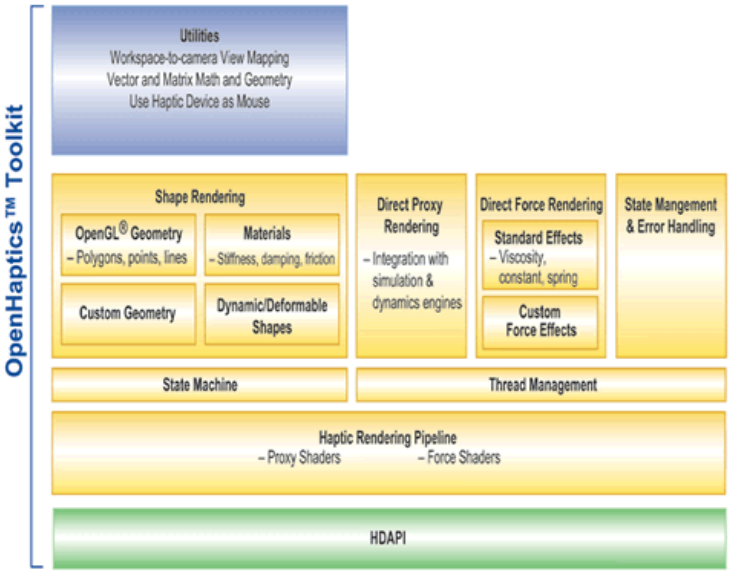

Abbildung 23: HLAPI

ne Update-Rate von 30-60 Hz, um dem menschlichen Auge den Eindruck zu geben, dass bewegte Darstellungen auf dem Bildschirm kontinuierlich sind. Für eine Anwendung, die haptisches Feedback geben soll, muss aufgrund der Sensibilität der menschlichen Haptik die Kraftberechnung jede Millisekunde neu erfolgen, um dem Benutzer den Eindruck fester Oberflächen zu vermitteln. Die HLAPI erlaubt es, die Geometrie für das haptische und grafische Rendering im gleichen Thread festzulegen. Der Benutzer muss nicht für eine Update-Rate der Haptik von 1000 Hz sorgen und keine Zustandssynchronisierung zwischen haptischem und grafischem Thread implementieren.
Die HLAPI legt drei verschiedene Threads an: den *Client Thread*, den *Collision Thread* und den *Servo Thread*.

Client Thread Der Thread der Hauptanwendung in einem typischen HLAPI-Programm wird als *Client Thread* bezeichnet. In diesem Thread kann der Benutzer die Geometrie für das haptische und das grafische Rendering festlegen.

Listing 1: Initialisierung der HLAPI

```
//Initialisieren des Gerätes
hHD=hdInitDevice(HD_DEFAULT_DEVICE);

//Erstellen und Zuweisen des
//haptischen Rendering-Kontextes
hHLRC=hlCreateContext(hHD);

//der aktuelle Rendering-Kontext
hlMakeCurrent(hHLRC);
```

Collision Thread Der *Collision Thread* führt die Kollisionsberechnung des Proxy mit den Objekten in der Szene durch. Es wird festgestellt, welches Objekt, das im *Client Thread* definiert wurde, mit dem Proxy kollidiert. Von diesem Objekt wird eine Approximation berechnet und an den *Servo Thread* weitergeleitet, der auf Grundlage dessen die Kraftberechnung durchführt. Die Update-Rate des *Collision Threads* beträgt ungefähr 100 Hz.

Servo Thread Im *Servo Thread* wird die Geräteposition und Orientierung ausgelesen und die Kraft berechnet, die an das Gerät weitergegeben wird. Die Update-Rate dieses Threads beträgt 1000 Hz. Bei der Programmierung mit der HLAPI bleibt der Thread weitestgehend vor dem Anwender verborgen.

Die Einfachheit der Integration von haptischen Komponenten mittels der HLAPI besteht darin, dass der Benutzer nur wenige Zeilen Code in eine bestehende OpenGL-Anwendung einfügen muss, um die von OpenGL erzeugten Primitive haptisch zu rendern. Zusätzlich ist es möglich, den haptisch darzustellenden Objekten Materialparameter wie zum Beispiel Festigkeit und Reibung zuzuweisen.
Abbildung 24 zeigt den typischen Aufbau einer OpenGL-Anwendung mit, durch die HLAPI eingebundener, haptischer Komponente.

Initialisierung der HLAPI Nachdem OpenGL initialisiert wurde, muss das haptische Gerät initialisiert und diesem ein Rendering-Kontext zugewiesen werden (siehe Listing 1). Dieser beinhaltet den aktuellen Rendering-Status; alle Kommandos der HLAPI werden an den aktuellen Kontext weitergeleitet. Es ist möglich mehr als einen Kontext zu erstellen, um zwischen den einzelnen Kontexten zu wechseln. Dabei darf immer nur ein Kontext akiv sein [29].

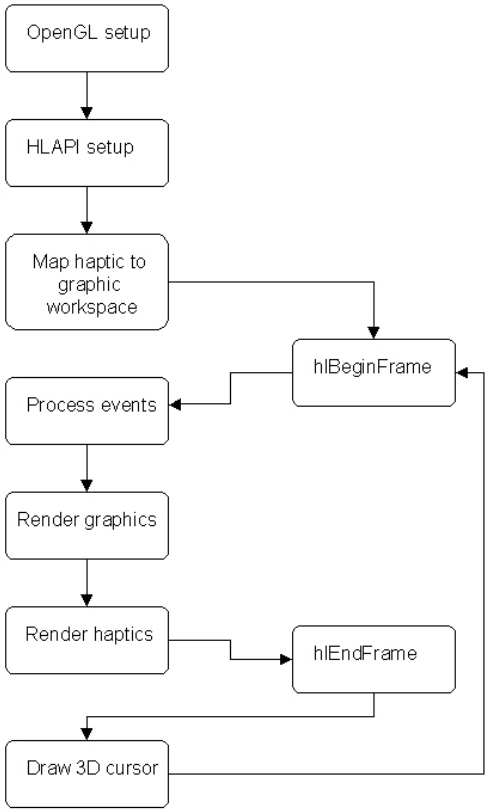

Abbildung 24: Ablauf einer Anwendung mit der HLAPI (Quelle: [29])

Mapping Nachdem das Gerät initialisiert und der Rendering-Kontext zugewiesen wurde, erfolgt die Anpassung des haptischen Arbeitsbereiches an die grafische Szene. Der haptische Arbeitsbereich beschreibt den Bereich, den das haptische Gerät erreichen kann. Je nachdem welcher Gerätetyp verwendet wird, variiert die Größe des Bereiches. Die HLAPI beinhaltet Funktionen, die es ermöglichen, die Größe dieses Bereiches abzufragen. Soll eine Anwendung erstellt werden, in der nicht der volle Arbeitsbereich des Gerätes genutzt wird, kann er eingeschränkt werden. Beispielsweise beschränkt `hlWorkspace(-80, -80, -80, 80, 80, 20)` den Bereich auf eine Box mit den Ausmaßen 160 mm x 160 mm x 100 mm.
Um das *Mapping* zwischen dem lokalen Koordinatensystem des haptischen Gerätes (*workspace coordinates*) und den grafischen Sichtkoordinaten (*view coordinates*) zu bestimmen, multipliziert die HLAPI zwei Transformationen miteinander. Die erste Transformation bilden die *touch coordinates* (die Koordinaten liegen im Koordinatensystem des haptischen Arbeitsbereiches vor) auf die *workspace coordinates* (lokale Koordinaten des haptische Gerätes) ab, während die zweite Transformation die *view coordinates* (lokale Ko-

ordinaten der Kamera) auf die *touch coordinates* abbildet. Die HLAPI besitzt einen Matritzenstapel (Stack) für jede der beiden Transformationen. Die Matrix, die obenauf liegt wird für die aktuelle Transformation benutzt. Die beiden Transformationsmatritzen heißen `HL_TOUCHWORKSPACE_MATRIX` und `HL_VIEWTOUCH_MATRIX`. In den meisten Anwendungen wird die *View-Touch Matrix* die Einheitsmatrix sein. Ein Gegenbeispiel ist das Abbilden (*Mappen*) der x-Achse des haptischen Arbeitsbereiches auf die z-Achse der Szene.
Der haptische Arbeitsbereich des PHANTOM Omni beträgt 160 $mm\ x$ 120 $mm\ x$ 70 mm. Ein direktes *Mapping* dieses Arbeitsbereiches auf die Szene, führt zu einer ungleichmäßigen Bewegung des Proxy in Breite, Höhe und Tiefe. Um dennoch eine gleichmäßige Bewegung zu gewährleisten, stellt HLAPI verschiedene Methoden bereit. Eine Übergabe der Projektionsmatrix von OpenGL an den Aufruf `hluFitWorkspace(matrix)`, definiert ein uniformes *Mapping* des haptischen Arbeitsbereiches auf das Sichtvolumen. Es ist möglich das *Mapping* auf einen Teil der Szene zu beschränken. So kann der Arbeitsbereich zum Beispiel auf eine vom Benutzer definierte Box abgebildet werden. Der Nachteil bei einem gleichmäßigen *Mapping* besteht darin, dass nicht der gesamte haptische Arbeitsbereich ausgenutzt wird. Ist dies jedoch für eine Anwendung unerlässlich und die gleichförmige Bewegung des Proxy zweitrangig oder unerwünscht, können Funktionen der HLAPI genutzt werden, die ein nicht-uniformes *Mapping* durchführen [29].

Das haptische Frame Nachdem das *Mapping* durchgeführt wurde, wird das haptische Frame mit dem Aufruf `hlBeginFrame` gestartet. Beendet wird es mit `hlEndFrame`. Innerhalb des haptischen Frames müssen alle Aufrufe stehen, die das haptische Rendering betreffen. Zu Beginn des haptischen Frames wird der aktuelle Rendering-Zustand von dem haptischen Rendering-Thread (*servo thread*) abgefragt. Desweiteren wird mit dem Aufruf von `hlBeginFrame` die aktuelle Modelview-Matrix von OpenGL abgefragt, um ein Weltkoordinatensystem für das haptische Frame bereitzustellen. Alle Positionen, Transformationen oder Vektoren, die während eines haptischen Frames im Client- oder Collision-Thread abgefragt werden, werden in dieses Weltkoordinatensystem transformiert. Die Häufigkeit der Positionsabfrage des Proxy während eines Frames spielt keine Rolle; es wird immer die Position ausgegeben, die zu Beginn des haptischen Frames vorlag, selbst wenn sich die Position mittlerweile geändert hat. Am Ende des haptischen Frames werden alle Änderungen, die vorgenommen wurden, an die haptische Rendering-Engine weitergeleitet und die Position des Proxy wird aktualisiert.
Innerhalb eines haptischen Frames muss die durch OpenGL-Kommandos definierte Geometrie der haptisch zu rendernden Objekte aufgerufen und

Listing 2: Haptisches Rendern einer Fläche

```
//beginne mit der Definition
//der haptisch darzustellenden Geometrie
hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId)

glBegin(GL_POLYGON);
glVertex3f(0,0,0);
glVertex3f(1,0,0);
glVertex3f(1,2,0);
glVertex3f(0,2,0);

//Ende der Definition
hlEndShape()
```

Listing 3: Erstellen eines eindeutigen Bezeichners für ein Objekt

```
HLuint myShapeId;
myShapeId = hlGenShapes(1);
hlGenShapes(1)
```

von den Kommandos `hlBeginShape` und `hlEndShape` umklammert werden. Listing 2 zeigt den Code zum haptischen Rendern einer 1x2-Fläche in der xy-Ebene (entnommen aus [29]). Der Funktion `hlBeginShape` aus Listing 2 werden zwei Parameter übergeben: `myShapeId` und `HL_SHAPE_DEPTH_BUFFER`. Jedes Objekt innerhalb eines haptischen Frames benötigt eine eindeutige Identifizierungsnummer. Anhand dieser Nummer erkennt die haptische Rendering-Engine die einzelnen Objekte und kann feststellen, ob das Objekt im Vergleich zum letzten haptischen Frame verändert wurde. So können zu jedem Zeitpunkt die richtigen Kräfte generiert werden.
Um einen eindeutigen Bezeichner zu erstellen werden die Funktionen aus Listing 3 benötigt. Soll das Objekt nicht mehr haptisch gerendert werden, kann der erstellte Bezeichner mit `hlDeleteShapes` freigegeben werden.
Der zweite Parameter, der `hlBeginShape` übergeben wird, gibt an, woher die HLAPI die zu rendernden Daten von OpenGL bekommt. Es gibt zwei Möglichkeiten: `HL_SHAPE_DEPTH_BUFFER` und `HL_FEEDBACK_BUFFER`. Wird die erste der beiden Möglichkeiten übergeben, erfasst die HLAPI die Daten aus dem *depth buffer* von OpenGL. Bei dem Aufruf `hlEndShape` liest die HLAPI das Bild aus, das im *depth buffer* von OpenGl vorliegt und übergibt dieses an den Collision-Thread. Dort wird dieses Bild benutzt, um Kollisionen des Objektes mit dem Proxy zu berechnen. Nehmen zusätzliche OpenGL-Kommandos innerhalb eines haptischen Frames Einfluss auf den *depth buffer*, werden diese als Teil der *shape* angenommen und ebenfalls

haptisch gerendert. Da im *depth buffer* lediglich ein Bild der dreidimensionalen Geometrie vorliegt, wird einzig der Teil des Objektes haptisch dargestellt, der vom Blickpunkt, welcher zum Rendern des Bildes benutzt wird, sichtbar ist. Somit können die Rückseiten der Objekte nicht haptisch erfühlt werden.
Der *feedback buffer* dagegen ermöglicht es, alle Teile des Objektes zu erfühlen. Der Aufruf von `HL_FEEDBACK_BUFFER` bewirkt, dass der Rendering-Modus von OpenGL zum Feedback-Modus wechselt. Alle geometrischen Primitive werden in dem *feedback buffer* gespeichert und beim Aufruf von `hlEndShape` von der haptischen Rendering-Engine gesichert und zur Kraftberechnung benutzt. Da nicht wie beim *depth buffer* ein Bild zum haptischen Rendering benutzt wird, sondern die geometrischen Primitive, können auch nicht-sichtbare Teile der Objekte haptisch dargestellt werden [29].

Events Mit Hilfe von Callback-Funktionen bietet die HLAPI die Möglichkeit über *Events* zu informieren, die während des haptischen Renderings auftreten. Zu diesem Zweck muss der Benutzer ein Callback in sein Programm integrieren, das ausgeführt wird, wenn das Event auftritt, für das das Callback registriert ist. Um ein Callback hinzuzufügen, wird der Aufruf `hlAddEventCallback` benötigt. Dieser Funktion muss die Art von Event, auf das die Reaktion erfolgen soll, mitgeteilt werden. Die HLAPI erlaubt, Callbacks für folgende *Events* zu registrieren:

- *Touch Events*
  Ein *Touch Event* wird ausgelöst wenn ein Kontakt des Proxy mit einem haptischen Objekt festgestellt wird. Das *Event* wird nur beim ersten Kontakt mit dem Objekt ausgelöst und nicht kontinuierlich gemeldet. In gleicher Weise verhält es sich mit dem *Untouch Event*, das beim Verlassen des Objektes ausgelöst wird.

- *Button Events*
  Diese Art von *Events* werden ausgelöst, wenn die Buttons, die sich an dem haptischen Gerät befinden, gedrückt oder wieder losgelassen werden.

- *Motion Events*
  Motion Events werden ausgelöst, wenn sich die Position oder Orientierung des Proxy ändert. Das *Event* wird nur dann ausgelöst, wenn die Distanz zwischen aktueller und vorheriger Proxy-Position einen festgelegten Schwellwert überschreitet.

Für das Auslösen von Events existieren zwei Möglichkeiten:

- Ein Auslösen des Events findet immer statt

- Das Event wird nur bei der Berührung mit einem bestimmten Objekt ausgelöst

Listing 4: Beim Drücken des ersten Geräteknopfes, wird die Position des Proxy zu diesem Zeitpunkt ausgegeben

```
void HLCALLBACK button1DownCallback
(HLenum e, HLuint o, HLenum t, HLcache *c, void *userdata)
{
hduVector3D proxyPos;

//liest die Position des Proxy aus dem Cache
//und speichert sie in der Variablen proxyPos
hlCacheGetDoublev(cache, HL_PROXY_POSITION, proxyPos);

//Ausgabe der Position
cout<<"'Proxyposition bei Button1Down: "'<<proxyPos<<endl;
}
```

Eine der beiden aufgeführten Möglichkeiten muss beim Hinzufügen des Callbacks bestimmt werden. Die HLAPI bietet die Auswahl, ein Callback im *Client Thread* oder *Collision Thread* aufzurufen. Der Unterschied besteht darin, dass der *Client Thread* in regelmäßigen Zeitabständen vom Programm aufgerufen werden muss, während der *Collision Thread* von der HLAPI selbst aktualisiert wird.
Ein kleines Beispiel für ein Callback das beim Drücken des ersten Geräteknopfes ausgelöst wird, findet sich in Listing 4 [29].

Materialeigenschaften Die HLAPI erlaubt es, Objekten auf einfache Art und Weise Materialeigenschaften zuzuweisen. Zu diesem Zweck muss der Aufruf `hlMaterialf(HLenum f, HLenum pname, HLfloat param)` erfolgen. Der erste Parameter gibt an welcher Teil des Objektes die Materialeigenschaft annehmen soll. Es besteht die Auswahl zwischen `HL_FRONT`, `HL_BACK` und `HL_FRONT_AND_BACK`. Der zweite Parameter gibt die Materialeigenschaft an, die gesetzt werden soll. Als dritter Parameter folgt der Wert für diese Materialeigenschaft. Die HLAPI unterstützt folgende Eigenschaften:

- Festigkeit
  Die Festigkeit gibt an wie hart sich das Objekt bei einer Berührung durch das haptische Gerät anfühlt. In der Kraftberechnung nach dem Hook'schen Gesetz $F = k * d$ definiert der Parameter k die Festigkeit. Je höher der für die Festigkeit angegebene Wert, desto stärker ist die Rückstellkraft.

- Dämpfung
  Mit einem Wert für die Dämpfung wird einem Objekt eine geschwindigkeitsabhängige Eigenschaft hinzugefügt. Je schneller das Gerät be-

wegt wird, desto größer ist die erzeugte Gegenkraft. In der HLAPI können Werte für die Dämpfung festgelegt werden, die zwischen 0 und 1 liegen; 0 bedeutet dass keine Dämpung erzeugt wird, wohingegen der Wert 1 die größte Dämpfung erzeugt, die mit dem Gerät darstellbar ist.

- Reibung
  Die HLAPI bietet die Möglichkeit eine Haftreibung oder eine Gleitreibung zu definieren. Als Haftreibung wird der Widerstand bezeichnet, der gespürt wird, wenn der Proxy mit der Bewegung über eine Oberfläche beginnt. Je größer der Wert, desto mehr Kraft muss vom Benutzer aufgeboten werden, um den Widerstand zu überwinden. Daraufhin ist es möglich, den Proxy über eine Oberfläche zu bewegen. Während dieser Bewegung kann eine Reibung erzeugt werden, die als Gleitreibung bezeichnet wird.

- Durchdringung
  Normalerweise ist es dem Proxy nicht möglich, eine Oberfläche zu durchdringen. Um dies dennoch zu erlauben, kann ein Wert für die Materialeigenschaft `HL_POPTHROUGH` festgelegt werden. Ein Wert größer 0 definiert wieviel Kraft der Benutzer aufwenden muss, um die Oberfläche zu durchstoßen.

3.3 Visualization ToolKit (VTK)

„The Visualization ToolKit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world.“[30]
VTK ist eine objektorientierte in C++ implementierte Bibliothek mit der Anwendungen in den Sprachen C++, TCL, Python oder Java erstellt werden können.
VTK lässt sich in zwei Modelle einteilen, das *graphics model* und das *visualization model*. Das erste besteht aus einer *pipeline*, die grafische Daten in Bilder umsetzt, während das *visualization model* eine Visualisierungspipeline beinhaltet, welche Informationen in grafische Daten umwandelt. Die beiden Modelle werden in den nachfolgenden Abschnitten näher betrachtet.

3.3.1 Graphics Model

Das *graphics model* beschreibt die Elemente einer Szene, die benötigt werden, um ein Bild aus grafischen Daten der Szene zu erzeugen. Bei der Benennung der Elemente einer Szene orientiert sich VTK an der Filmindustrie. Eine Filmszene ist aufgebaut aus Kameras, Lichtern und Requisiten. Analog dazu benötigt eine Szene, die mit dem Visualization ToolKit

dargestellt werden soll, eine `vtkCamera`, ein `vtkLight` und einen `vtkProp` (englisch *props* = Requisiten). Props sind die sichtbaren Elemente einer Szene. Dabei stellt `vtkProp` eine abstrakte Oberklasse dar. Spezialisierungen dieser Klasse sind zum Beispiel `vtkProp3D` oder `vtkActor2D`. `vtkProp3D` wiederum stellt eine abstrakte Klasse für Props dar, die im dreidimensionalen Raum positioniert und manipuliert werden (alle Props besitzen eine 4x4 Transformationsmatrix). `vtkActor2D` stellt eine konkrete Unterklasse für Props dar, die zweidimensionale Daten repräsentieren. Ein `vtkActor2D` kann in der Szene positioniert werden, besitzt jedoch keine Transformationsmatrix wie die Elemente vom Typ `vtkProp3D`. Konkrete Unterklassen für einen `vtkProp3D` stellen beispielsweise `vtkActor` und `vtkVolume` dar. Alle Props haben eine Verbindung zu einem *Mapper* und einer *Property*. Der *Mapper* dient dazu, die Daten in geometrische Primitive umzusetzen, während `vtkProperty` zusätzliche Rendering-Parameter beinhaltet, wie Farb- und Materialeigenschaften. Im Gegensatz zu `vtkProperty`, für die bei der Erstellung eines Prop ein „default“ Property-Objekt generiert wird, muss der Mapper für einen Prop explizit erstellt und mit dem Aufruf `SetMapper` gesetzt werden.
Weitere wichtige Elemente stellen `vtkRenderWindow` und `vtkRenderer` dar. Ein Objekt der Klasse `vtkRenderWindow` ist verantwortlich für die Handhabung der Darstellung des Ausgabefensters. Dazu gehört zum Beispiel das Speichern der Größe, Position und Titels dieses Fensters. Objekte der Klasse `vtkRenderer` verwalten Lichtquellen, Kameras und Aktoren während des Rendering-Prozesses. Ist weder Lichtquelle noch Kamera definiert, werden diese vom Renderer bereitgestellt. Im Gegensatz dazu ist der Benutzer dafür verantwortlich mindestens einen Aktor zu generieren. Weiterhin muss ein Objekt des `vtkRenderer` mit einer Instanz eines `vtkRenderWindow` verbunden sein, da `vtkRenderWindow` das Fenster bereitstellt, in welches der Renderer zeichnet. Es besteht die Möglichkeit mehrere Renderer zu erstellen und mit dem RenderWindow zu verbinden. Eine zusätzliche Aufgabe von `vtkRenderWindow` stellt somit die Verwaltung der ihm zugewiesenen Renderer dar. Um mit der Szene interagieren zu können, stellt VTK die Klasse `vtkRenderWindowInteractor` bereit. Dazu muss eine Instanz des `vtkRenderWindowInteractor` dem `vtkRenderWindow` zugewiesen werden. Interaktionen mit der Maus (zum Beispiel Mausklick) oder Tastatureingaben werden in VTK-spezifische Events (Signale) umgesetzt und Reaktionen auf diese Events ausgelöst. VTK stellt mit der Klasse `vtkInteractorStyle` verschiedene *interactor styles* zur Verfügung, mit denen unterschiedliche Reaktionen auf Events ausgelöst werden können. Zusätzlich besteht für den Benutzer die Möglichkeit, Reaktionen auf Events selbst zu definieren. Das geschieht auf Basis des Command/Observer-Modelles von VTK. Jedes Objekt, das von `vtkObject` abgeleitet ist, besitzt die Methode `AddObserver`, mit der Ereignisse beobachtet werden können. Tritt das Ereignis ein, wird ein dem

Listing 5: benutzerdefiniertes Callback

```
#include vtkCommand.h

class vtkMyCallback : public vtkCommand
{
public:
static vtkMyCallback *New()
{ return new vtkMyCallbackCommand; }
   virtual void Execute(vtkObject *caller,
   unsigned long, void*)
   {
      vtkRenderer *ren =
      reinterpret_cast<vtkRenderer*>(caller*);
      cout<<ren->GetActiveCamera()->GetPosition()[0]<<"' "'
      <<ren->GetActiveCamera()->GetPosition()[1]<<"' "'
      <<ren->GetActiveCamera()->GetPosition()[2]<<"'\n "';
   }
};
```

Ereignis zugeordnetes Callback ausgeführt. Dieses Callback kann vom Benutzer definiert werden, indem er eine Klasse von `vtkCommand` ableitet und dessen `Execute`-Methode überschreibt (siehe Listing 5, entommen aus [22]). Das Callback wird in der `AddObserver`-Methode angegeben und ausgeführt, sobald das Ereignis eintritt, welches der Observer beobachtet. Das Beispiel aus Listing 5 gibt bei Eintreten des Events die aktuelle Kameraposition aus. Listing 6 (entnommen aus [22]) zeigt das Zusammenspiel von *Renderer*, *RenderWindow*, *Actor*, *Mapper* und dem selbst erstellten Callback.

3.3.2 Visualization Model

Um Informationen in grafische Daten zu übersetzen, benutzt VTK einen Datenfluss in den zwei elementare Objekte eingebunden sind: *Data Objects* und *Process Objects*.
Data Objects repräsentieren Daten verschiedenen Typs. VTK unterstützt unterschiedliche Typen, wie zum Beispiel Bilddaten, Punktmengen und Polygonmodelle. *Data Objects* stellen Methoden bereit, um Daten zu erstellen, zu löschen und abzurufen. Eine direkte Änderung der Informationen ist jedoch nicht möglich. Das ist den Methoden der *Process Objects* vorbehalten. *Process Objects* können *source objects*, *filter objects* oder *mapper objects* sein. *Source objects* generieren oder lesen externe Daten ein (*reader objects*), während *filter objects* auf Daten operieren. Sie nehmen Daten über einen oder mehrere Eingänge auf; verändern, verknüpfen oder verarbeiten diese und geben die neuen Daten über einen oder mehrere Ausgänge aus. Der Aus-

Listing 6: Tritt das *StartEvent* des Renderers ein, wird das benutzerdefinierte Callback aufgerufen und die aktuelle Kameraposition ausgegeben

```
int main(int argc, char *argv[])
{
    vtkConeSource *cone = vtkConeSource::New();
    cone->SetHeight(3.0);
    cone->SetRadius(1.0);
    cone->SetResolution(10);

    //erstellen eines Mappers
    vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    coneMapper->SetInput(cone->GetOutput());
    vtkActor *coneActor = vtkActor::New();
    //übergeben des Mappers an den vtkActor
    coneActor->SetMapper(coneMapper);
    vtkRenderer * ren1 = vtkRenderer::New();
    //hinzufügen des Actors zum Renderer
    ren1->AddActor(coneActor);
    ren1->SetBackground(0.1,0.2,0.4);

    //erstellen des vtkRenderWindow
    vtkRenderWindow * renWin = vtkRenderWindow::New();
    //hinzufügen des Renderers zum RenderWindow
    renWin->AddRenderer(coneActor);
    renWin->SetSize(300,300);

    vtkMyCallback *mo1 = vtkMyCallback::New();
    //das Callback wird ausgeführt, sobald das StartEvent
    //des Renderers eintritt
    ren1->AddObserver(vtkCommand::StartEvent, mo1);
    mo1->Delete();

    int i;
    for(i = 0; i<360; ++i)
    {
        //render the image
        renWin->Render();
        //rotate the acitive camera by one degree
        ren1->GetActiveCamera()->Azimuth(1.0);
    }
    cone->Delete();
    coneMapper->Delete();
    coneActor->Delete();
    ren1->Delete();
    renWin->Delete();
}
```

gang eines Filters kann gleichzeitig der Eingang eines anderen Filters sein. *Mapper objects* werden als *sinks* (engl. für Senke) bezeichnet, da sie Daten aufnehmen, jedoch keine ausgeben. Sie beenden den Datenfluss der *Visualization Pipeline*.

Visualization Pipeline Um die Informationen in grafische Daten zu transformieren, wird eine *Pipeline* aufgebaut, in der die oben beschriebenen Objekte verknüpft sind. Um die Ausführung der *Pipeline* zu erwirken, bedient sich VTK einer impliziten Kontrolle. Dazu sind zwei Methoden implementiert: `Execute()` und `Update()`. Gibt der Benutzer den Befehl zum Rendern der Szene, wird der Update-Prozess angestoßen. Die Aktoren senden den Befehl zum Rendern an ihre Mapper. Diese wiederum senden ein `Update()` an ihre Inputs. Der Prozess wird so lange weiterverfolgt, bis ein *source object* erreicht wird. Wenn dieses nach dem letzten Aufruf verändert wurde, löst es eine Wiederausführung durch den Befehl `Execute()` aus. Daraufhin überprüfen die nachfolgenden Filter, ob eine Wiederausführung notwendig ist. Ist dies der Fall senden sie ebenfalls ein `Execute()`. Dieser Prozess dauert fort, bis ein *mapper object* erreicht wird. Dieses konvertiert schließlich die Daten für die grafische Ausgabe. Abbildung 25 zeigt den Aufbau der *Visualization Pipeline* mit den zugehörigen Elementen und den Ausführungsrichtungen der Methoden `Update()` und `Execute()` [22].

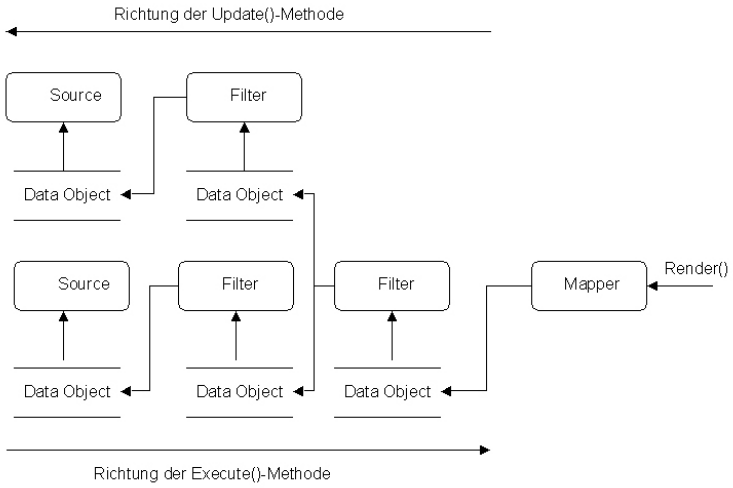

Abbildung 25: Visualization-Pipeline [22]

3.4 mihLib

Die Bibliothek *mihLib* wird am Institut für medizinische Informatik der Universitätsklinik Hamburg-Eppendorf entwickelt. Sie dient zur Unterstüt-

zung beim Einlesen, Abspeichern und Visualisieren von medizinischen Daten. Desweiteren bildet sie eine Verknüpfung verschiedener Bibliotheken, wie zum Beispiel *VTK*, *ITK* und *Qt*. Die *mihLib* vereinfacht den Visualisierungsprozess von Objekten durch VTK.
Wichtige Bestandteile der *mihLib* sind `mihScene`, `mihSceneObject` und `mihQtSceneWidget`. Die Oberklasse (`mihSceneBase`) von `mihScene`, implementiert die Verwaltung der Objekte, die in der Szene dargestellt werden. Zusätzlich wird in der Klasse ein Renderer (`vtkRenderer`) und eine Kamera (`vtkCamera`) bereitgestellt und es stehen Methoden zur Verfügung, die der Kamera- und Objektmanipulation dienen. Zu einer Szene werden mittels `AddSceneObject` Szeneobjekte hinzugefügt, die von der Klasse `mihSceneObject` abgeleitet sind. Dazu gehört `mihSurfaceObject`, welche wiederum die Oberklasse für `mihSmoothedSurfaceObject`, `mihSurfaceSliceObject`, `mihVectorFieldObject` und `mihVolumeObject` bildet.
Wesentliche Bestandteile von `mihQtSceneWidget` sind die VTK-Klassen `vtkMapper`, `vtkActor` und `vtkDataSet`. Mit der Integration dieser VTK-Klassen entfällt für den Benutzer ein Teil des in Kapitel 3.3.1 beschriebenen Pipeline-Aufbaus. Eine .vtk-Datei kann beispielsweise mittels des Aufrufs `LoadObject` geladen werden. Durch ein darauffolgendes `AddSceneObject` wird das Objekt zur Szene hinzugefügt und zur Darstellung durch den Renderer gebracht. Intern bewirkt der Aufruf `LoadObject` den Aufbau der VTK-Pipeline: Zuerst werden dem *Mapper* die Poly-Daten zugewiesen; daraufhin erfolgt die Zuordung des *Mappers* zum *Actor*. Durch `AddSceneObject` wird das Objekt an den Renderer übergeben und ein Rendern der Szene erwirkt.
Um ein `vtkRenderWindow` in eine Qt-Oberfläche integrieren zu können, wurde `mihQtSceneWidget` von `QVTKWidget` abgeleitet. Mit dem Aufruf `SetScene` wird dem `mihQtSceneWidget` eine Szene übergeben, deren Renderer zum *RenderWindow* hinzugefügt wird. Damit können die Objekte einer Szene in dem Ausgabefenster dargestellt werden. Die Möglichkeit der Interaktion mit der Szene wird durch den `QVTKInteractor`, der in `QVTKWidget` eingebunden ist, bereitgestellt.
In der vorgestellten Anwendung wurden von den Klassen `mihScene`, `mihSurfaceObject`, `QVTKWidget` und `QVTKInteractor` abgeleitete Klassen erstellt, die der Integration von haptischen Objekten in eine Szene dienen und die Interaktion mit diesen Objekten ermöglichen.

3.5 Qt

Qt wurde von der Firma Trolltech entwickelt und ist eine Klassenbibliothek, um grafische Benutzeroberflächen unter C++ zu erstellen.
Der Qt-Designer stellt ein wichtiges Hilfsmittel dar, um grafische Oberflächen mittels *drag and drop* zusammenzustellen. Dazu wählt der Benutzer

Listing 7: Qt-Befehl connect()

```
//Verknüpfung des click()-Signals des Buttons
//mit dem slot buttonClick(), der in der
//Anwendung mApp definiert ist
connect(button, SIGNAL(click()),mApp, SLOT(buttonClick()));
```

aus der Toolbox das Element aus, das er verwenden möchte und platziert es auf dem *widget* (Komponente einer Benutzeroberfläche). Die Eigenschaften für die gewählten *widgets* können zu jeder Zeit in dem *Property Editor* festgelegt werden. Größe und Position der *widgets* können mit den von Qt bereitgestellten *layouts* bestimmt werden.
Grafische Benutzeroberflächen dienen zur Interaktion zwischen Benutzer und Programm. Wird ein Button auf der Oberfläche benutzt, führt das Programm eine Aktion aus. Um diese Kommunikation (Drücken des Buttons-Ausführen der Aktion) zwischen den einzelnen Objekten zu realisieren, implementiert Qt *signals* und *slots*. *Widgets* in Qt senden ein *signal*, wenn sie ein Event registrieren. Dieses *signal* kann mit einem *slot* verknüpft werden, der eine vom Benutzer definierte Aktion ausführt. Die Verknüpfung eines *signals* mit einem *slot*, geschieht über den Befehl `connect()`. Der Funktion `connect()` müssen vier Parameter übergeben werden: der Sender, das *signal*, der Empfänger und der *slot* (siehe Listing 7). Um eine Klasse mit *signals* und *slots* verwendbar zu machen, muss sie von `QObject` abgeleitet sein und das `Q_OBJECT`-Makro in der Definition beinhalten. Hinter dem Schlüsselwort `signals` werden *Signals* und hinter den Schlüsselworten `public slots`, `protected slots` oder `private slots` werden *slots* in der Klasse deklariert. Es können beliebig viele *signals* mit beliebig vielen *slots* verknüpft werden (siehe Abbildung 26).

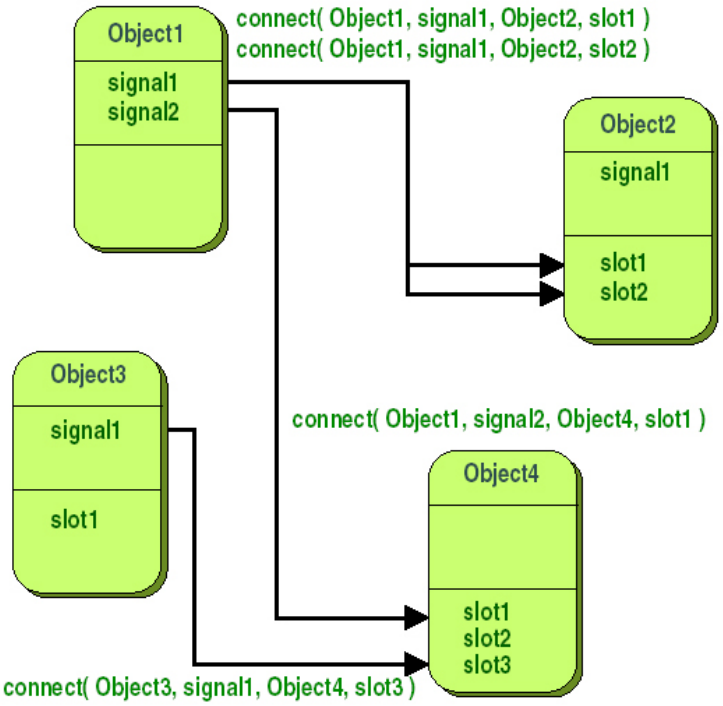

Abbildung 26: *signals* und *slots*

4 Realisierung und Implementierung

In der vorliegenden Arbeit wird ein bestehender Ansatz zur haptischen Interaktion mit 3D-Modellen (siehe [4]) erweitert und in die Software integriert.
Die folgenden Abschnitte beschreiben die Funktionalitäten des bestehenden Ansatzes und gehen auf Probleme ein, die im Verlauf der vorliegenden Diplomarbeit in verschiedenen Komponenten der Vorgängersoftware aufgedeckt wurden. Eingefügte Erweiterungen werden erläutert.

4.1 Vorarbeit

Die folgenden Abschnitte gehen genauer auf die zu integrierenden Funktionen ein und beschreiben auftretende Probleme.

4.1.1 Funktionalität der bestehenden Software

Die bereits bestehende Software beinhaltet folgende Funktionen:

1. **Integration des haptischen Renderings in VTK**
   Da das *Visualization Toolkit* kein haptisches Rendering der dargestellten Modelle beinhaltet, wurde das *OpenHaptics Toolkit* integriert. Zu diesem Zweck wurden die Klassen in VTK identifiziert, die an dem Rendering-Prozess beteiligt sind und Funktionen eingegliedert, die für ein haptisches Rendering der Objekte sorgen.

2. **Haptisches Rendering mehrerer VTK-Modelle und Unterscheidung zwischen fühlbaren und nicht-fühlbaren Objekten**
   Sind mehrere Modelle in einer Szene zu sehen, ist es möglich, jedes einzelne der haptischen Objekte zu erfühlen. Zusätzlich können nicht-haptische Objekte vorhanden sein.
   Ein Timer, der alle 10 Millisekunden aufgerufen wird, bewirkt ein regelmäßiges Rendern der Szene. Er stößt den Rendering-Prozess an, wenn sich die Position des Proxy seit dem letzten Timer-Aufruf geändert hat.
   Wie in Kapitel 3.2.1 erläutert, werden Objekte, die während eines haptischen Frames den Tiefenpuffer verändern, als Teil einer haptischen *Shape* angenommen. Um zu gewährleisten, dass während eines Frames lediglich die haptischen Objekte in den Tiefenpuffer geschrieben werden, wurde eine Sortier-Funktion erstellt. Diese wird bei Veränderung der Objektanzahl in der Szene aufgerufen. Die Sortier-Funktion erzeugt zwei Listen: In die erste Liste werden die Objekte eingefügt, die einen haptischen Mapper besitzen und haptisch gerendert werden; die zweite Liste beinhaltet alle Objekte, die einen normalen Mapper besitzen und nicht haptisch gerendert werden. Anschließend

werden zuerst die haptischen und danach die nicht-haptischen Objekte an den Renderer übergeben. Vor dem Rendern des ersten Objektes wird das haptische Frame geöffnet (`hlBeginFrame`) und mit dem Rendern des letzten haptischen Objektes geschlossen (`hlEndFrame`). Ein weiteres Hinzufügen von Objekten verändert zwar weiterhin den Tiefenpuffer, hat jedoch keinen Einfluss mehr auf das haptische Rendering, da bereits der Inhalt des Tiefenpuffers zur Weiterverarbeitung an den *Collision Thread* gegeben wurde. Dieser Prozess gewährleistet, dass ausschließlich Objekte mit haptischem Mapper haptisch dargestellt werden.

3. **Visualisierung eines haptischen Mauszeigers**
   Die Position und Orientierung des Proxy wird durch einen Zeiger visualisiert. Die Bewegungen des Gerätes werden auf den haptischen Mauszeiger übertragen.

4. **Bewegung von Modellen mit dem haptischen Gerät**
   Die haptischen Objekte können mit dem Gerät bewegt werden. Zu diesem Zweck werden vier Schritte durchgeführt:

   (a) **Ermittelung des aktuellen Objektes**
   Das Callback für ein *Touch Event* wurde bei der Initialisierung der Haptik hinzugefügt. Eine Berührung zwischen Objekt und Proxy führt zum Aufruf dieses *Callbacks*. Innerhalb der Funktion wird das berührte Objekt anhand seiner *shapeId* identifiziert.

   (b) **Erfassen des Objektes**
   Mit dem Drücken des ersten Knopfes des haptischen Gerätes wird das im ersten Schritt bestimmte Objekt erfasst. Die aktuelle Rotation und Position des Gerätes wird abgespeichert. Zudem wird eine Kopie des Objektes angefertigt.

   (c) **Bewegung des Objektes**
   Die Bewegung des Gerätes wird auf das Objekt übertragen. Dabei wird nicht das eigentliche Objekt bewegt, sondern die zuvor erstellte Kopie. Das ursprüngliche Objekt bleibt an der originären Position sichtbar.

   (d) **Absetzen des Objektes**
   Mit dem Loslassen des Geräteknopfes wird die Bewegung des Objektes beendet. Die Kopie verschwindet und das ursprüngliche Objekt wird an die aktuelle Position gezeichnet.

5. **Anwendung unterschiedlicher haptischer Eigenschaften**
   Verschiedene Parameter wie Reibung, Dämpfung und Festigkeit können den einzelnen Modellen zugewiesen werden.

6. **Kamerasteuerung mit dem haptischen Gerät**
   Um die Anzahl der Wechsel zwischen Maus und haptischem Gerät zu minimieren, wurde die Möglichkeit eingebunden, die Kamera der Szene mit dem haptischen Gerät steuern zu können. Dazu wird zunächst der Mittelpunkt des Geräte-Arbeitsbereiches berechnet. Eine simulierte Federkraft bewegt den Gerätearm an den berechneten Mittelpunkt, um ihn dort zu halten. Eine Bewegung des Gerätes hat eine Berechnung der Bewegungsrichtung zur Folge.
   VTK bietet Funktionen zur Steuerung der Kamera mit der Maus: Über Events (`LeftButtonPressEvent`, `MiddleButtonPressEvent`, `RightButtonPressEvent`) werden Aktionen zur Kamerasteuerung ausgelöst. Ein Halten der mittleren Maustaste und die gleichzeitige Bewegung der Maus beispielsweise bewirkt eine Verschiebung der Kamera in die Bewegungsrichtung der Maus. Dieser Prozess wurde auf die Kamerasteuerung mit dem haptischen Gerät übertragen.
   Zuerst wird festgestellt, wie sich der Proxy in Bezug auf den berechneten Mittelpunkt verändert hat und daraus auf ein Zoomen, eine Verschiebung oder eine Rotation der Kamera geschlossen. Um eine Rotation der Kamera zu erhalten, muss der Stift des Gerätes gedreht werden. Für eine Verschiebung der Kamera ist eine Bewegung des Proxy in der xy-Ebene notwendig, wohingegen ein Heran- oder Herauszoomen der Szene mit einer Bewegungsänderung des Proxy entlang der z-Achse erreicht werden kann. Anschließend werden die entsprechenden VTK-*Events* ausgelöst.

7. **Zeichnen auf Objekten mit dem haptischen Gerät**
   Das Zeichnen auf einem Objekt ist nur möglich, wenn der Proxy mit einem haptischen Objekt in Kontakt steht und der erste Knopf des Gerätes gedrückt gehalten wird. Bei Erfüllung dieser Kriterien werden Schnittmarken, die als rote Punkte dargestellt werden, auf der Oberfläche des Objektes eingezeichnet.

8. **Schneiden anhand von eingezeichneten Schnittmarken**
   Sind auf einer Objektoberfläche Schnittmarken eingezeichnet, ist es möglich, das Objekt in zwei Teilmodelle zu zerschneiden, die wiederum einzeln erfühlt und bewegt werden können. Das Schneiden eines Objektes erfolgt mit Hilfe einer Ebene. Diese wird durch die erste und letzte Schnittmarke sowie einen weiteren Punkt auf der *Bounding Box* des Objektes bestimmt.

4.1.2 Architektur der bestehenden Software

Die oben aufgeführten Anforderungen wurden durch die im Folgenden aufgeführten Klassen realisiert. Eine Übersicht bietet die Abbildung 27.

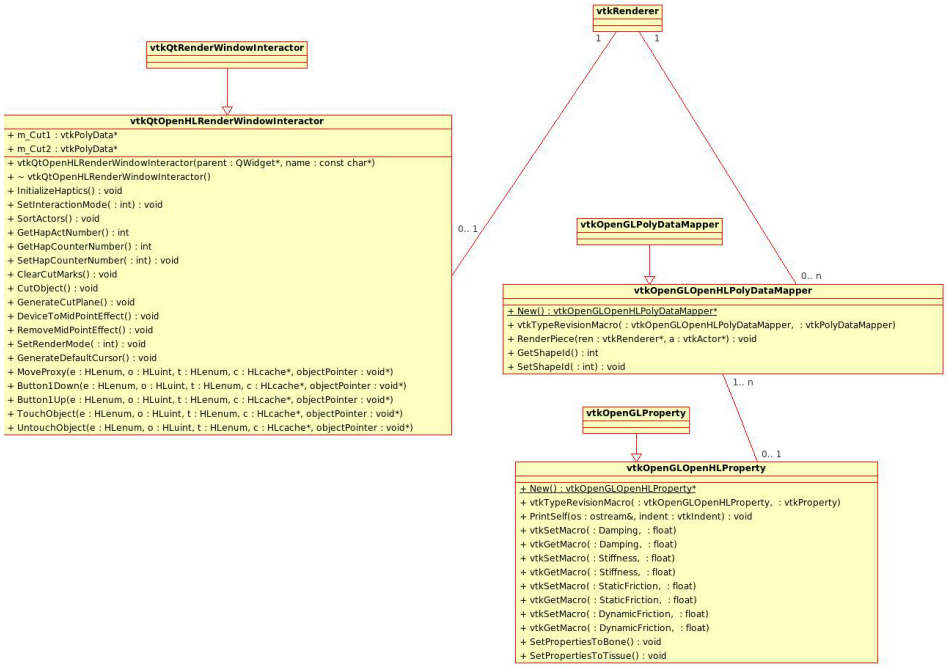

Abbildung 27: Klassendiagramm [4]

`vtkQtOpenHLRenderWindowInteractor` Zusammen mit der Klasse `vtkOpenGLPolyDataMapper` bildet diese Klasse die Komponente für die Integration des haptischen Renderings in VTK. Sie wurde von `vtkQtRenderWindowInteractor` abgeleitet. Die Klasse `vtkQtOpenHLRenderWindowInteractor` bietet Methoden für die Initialisierung des Gerätes und stellt *Callbacks* für die Verarbeitung von Eingaben bereit, die über das haptische Gerät erfolgen. Es werden zum Beispiel *Callback*-Funktionen für die Verarbeitung von *Touch* oder *Button1_Down Events* bereitgestellt. Alle implementierten Callback-Funktionen laufen im *Collision Thread*.
Zusätzlich werden Methoden zur Erstellung des Mauszeigers, Setzen von Schnittmarken, Schneiden der Objekte sowie zur Steuerung der Kamera durch das haptische Gerät zur Verfügung gestellt.

`vtkOpenGLOpenHLPolyDataMapper` In der Klasse `vtkOpenGLPolyDataMapper` werden die 3D-Modelle in OpenGL-Code übersetzt. Da die *OpenHaptics*-Bibliothek auf OpenGL-Befehle angewiesen ist, um die Modelle haptisch darzustellen, wurde diese als Basisklasse für den `vtkOpenGLOpenHLPolyDataMapper` ausgewählt. Die Funktion `RenderPiece` des `vtkOpenGLPolyDataMapper` wird für jeden in der Szene befindlichen *Actor* aufgerufen. Die Funktion erzeugt für das Objekt eine *Display*-Liste, die alle OpenGL-Kommandos zur Generierung der OpenGL-Primitive spei-

Listing 8: *Display*-Liste, umschlossen von den Aufrufen für ein haptisches Frame

```
hlBeginFrame();
hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);
glCallList(this->ListId);
hlEndShape();
hlEndFrame();
```

chert. Diese wird anschließend durch OpenGL gezeichnet. Um das haptische Rendering der *Display*-Liste zu gewährleisten, wird die Funktion `RenderPiece` in der Klasse `vtkOpenGLPolyDataMapper` überschrieben. Die Aufrufe für ein haptisches Frame und ein haptisches Objekt umschließen die *Display*-Liste (siehe Kapitel 3.2.1 und Listing 8). Demnach muss einem Objekt, das haptisch gerendert werden soll, ein haptischer Mapper (`vtkOpenGLOpenHLPolyDataMapper`) zugewiesen werden. Zusätzlich wurde die `SetShapeId` Funktion integriert. Sie erlaubt es, einem Objekt eine eindeutige *shapeId* zuzuweisen (siehe Kapitel 3.2.1).

`vtkOpenGLOpenHLProperty` Die Klasse `vtkOpenGLOpenHLProperty` wurde von `vtkOpenGLProperty` abgeleitet. `vtkOpenGLOpenHLProperty` bietet Funktionen zur Veränderung der Materialeigenschaften der haptischen Modelle. Es können Werte für die Parameter Steifigkeit, Dämpfung, statische und dynamische Reibung gesetzt werden [4].

4.1.3 Probleme und Schwächen der bestehenden Software

In den weiteren Abschnitt werden die Schwächen der Vorgängersoftware näher beschrieben und Ursachen von auftretenden Fehlern erläutert.

Bewegen von Objekten Der Wechsel zwischen dem Bewegen eines Objektes mit dem haptischen Gerät und dem Bewegen des Objektes mit der 2D-Maus läuft nicht fehlerfrei ab. Es ist beispielsweise nicht möglich, ein beliebiges Objekt mit der 2D-Maus im Raum zu positionieren und es anschließend mit dem haptischen Gerät weiterzubewegen. Die Bewegungen des haptischen Mauszeigers stimmen nicht mehr mit den Bewegungen des Gerätes überein, nachdem das Objekt mit der 2D-Maus positioniert wurde. Es existieren zwei Möglichkeiten, um Objekte mit dem *Visualization Toolkit* zu transformieren. Die erste Möglichkeit besteht darin, die Methoden zur Transformation (Rotation, Translation, Skalierung, Anwenden einer benutzerdefinierten Matrix) eines *Actors* der Klasse `vtkActor` zu verwenden. Eine Transformation des *Actors* bewirkt lediglich eine Veränderung der grafische Repräsentation des Objektes. Die interne Matrix des *Actors*

wird modifiziert; die polygonalen Daten werden nicht transformiert. Die zweite Möglichkeit ist, den `vtkTransformPolyDataFilter` anzuwenden. Durch diesen werden die Polygondaten des Objektes transformiert und die Transformationsmatrix des *Actors* bleibt unberührt.
Wird ein Objekt mit der 2D-Maus transformiert, findet die erste Möglichkeit Verwendung. In dem zu optimierenden Ansatz werden bei der Bewegung der Objekte mit dem haptischen Gerät beide Möglichkeiten vermischt. Während der Bewegung des Objektes wird eine Transformation auf den *Actor* der Kopie (*Dummy*) des Original-Objektes ausgeführt. Beim Loslassen des Objektes hingegen wird die Transformation durch den `vtkTransformPolyDataFilter` auf die polygonalen Daten des Originalobjektes ausgeführt. Eine *Actor*-Transformation des Original-Objektes wird somit übergangen.
Wie in Kapitel 3.2.1 erläutert wird beim Aufruf von `hlBeginFrame` die aktuelle *Modelview*-Matrix von OpenGL ausgelesen. Für alle Positionen, Transformationen und Vektoren, die im Collision- oder Client-Thread mittels `hlGet` oder `hlCacheGet` abgefragt werden, findet eine Transformation in dieses Koordinatensystem statt. Beim Rendern der Szene wird für jeden *Actor* die Funktion `Render` der Klasse `vtkOpenGLActor` ausgeführt. Innerhalb dieser Funktion findet die Verknüpfung der *Modelview*-Matrix mit der internen Matrix des *Actors* statt. Zudem wird das Ausführen der Funktion `RenderPiece` des zum *Actor* gehörenden *Mappers* angestoßen. Das Rendern eines haptischen Objektes hat demnach den Aufruf des `vtkOpenGLOpenHLPolyDataMapper` zur Folge. Dieser *Mapper* ruft `hlBeginFrame` auf. Durch das Verändern des Objektes mit der 2D-Maus wurde die interne Matrix des *Actors* transformiert und beim Aufruf der Render-Funktion des *Actors* mit der *Modelview*-Matrix verknüpft. Der Aufruf von `hlBeginFrame` veranlasst das Auslesen dieser Matrix. Alle weiteren Positionen, Transformationen und Vektoren werden in das Koordinatensystem des *Actors* transformiert. Daraus resultiert die falsche Bewegung des haptischen Mauszeigers; denn zur Berechnung der Zeigerposition und Zeigerorientierung wird die aktuelle Transformation des Proxy im Collision-Thread mittels `hlCacheGet` abgefragt. Eine Berechnung dieser Transformation in das Koordinatensystem des *Actors* findet statt.
Beim Bewegen des Objektes mit dem haptischen Gerät wird nicht die interne Transformationsmatrix des *Actors*, sondern die des *Dummy* verändert. Die *Modelview*-Matrix wird mit der internen Transformationsmatrix des *Dummy* verknüpft. Da vor dem Rendern der nicht-haptischen Objekte (zu denen der *Dummy* gehört) die haptischen Objekte gerendert wurden, ist das haptische Frame bereits geschlossen. Für den *Dummy* wird ein `hlBeginFrame` nicht aufgerufen und somit hat die mit der Transformationsmatrix verknüpfte *Modelview*-Matrix keinen Einfluss. Aus diesem Grund werden korrekte Transformationen beim Bewegen des Objektes mit dem haptischen Gerät berechnet. Eine Veränderung der Transformationsmatrix

des *Actors* während der Bewegung führt, wie bei der Manipulation des Objektes mit der 2D-Maus, zu Fehlern (siehe Abbildung 28).

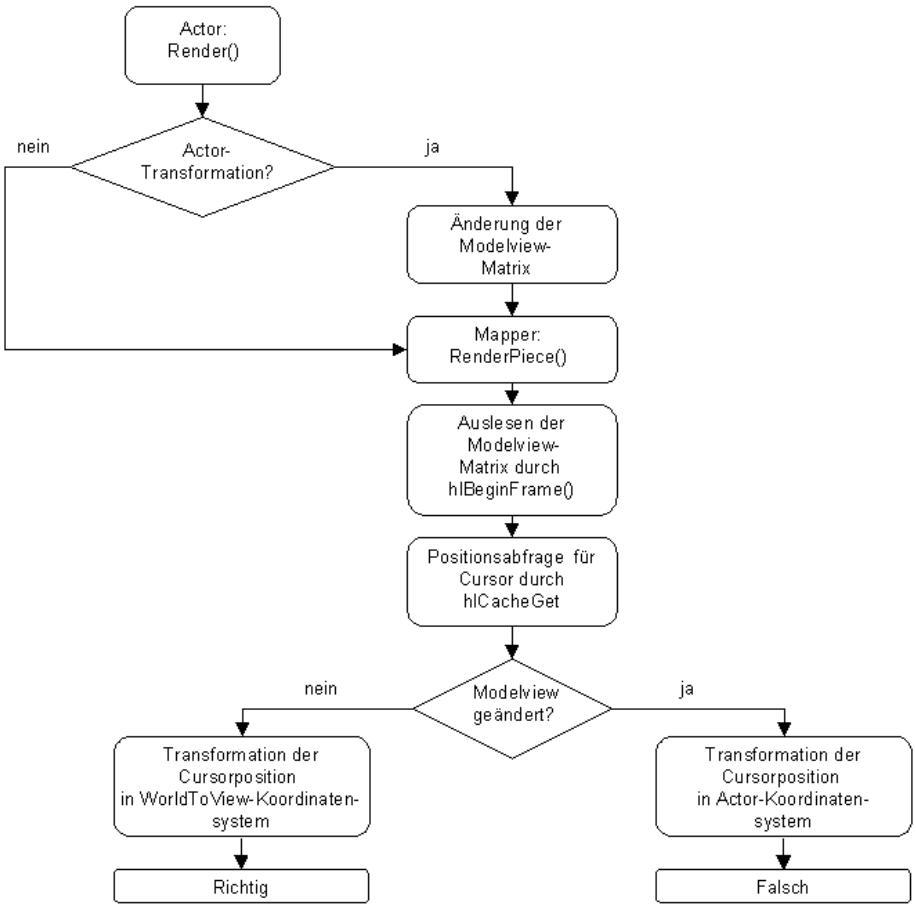

Abbildung 28: Berechnung der falschen Transformation bei Veränderung der internen Matrix des Actors

Schneidefunktion Die Schwäche in der Funktion zum Schneiden von Objekten besteht darin, dass nicht entlang der eingezeichneten Schnittpunkte geschnitten wird. Stattdessen wird eine Ebene durch den ersten und letzten Punkt gelegt und das Objekt durch diese Ebene in zwei Teilobjekte zerlegt. Die zwischen diesen beiden Punkten liegenden Schnittmarken werden ignoriert (siehe Abbildung 29). Damit fehlt die Möglichkeit, Objekte anhand von beliebig gesetzten Marken zu schneiden. Darüber hinaus ist das Benutzen der Schneidefunktion nicht intuitiv, da der Anwender erwartet, dass das Objekt anhand der von ihm definierten Schnittmarken geteilt wird.

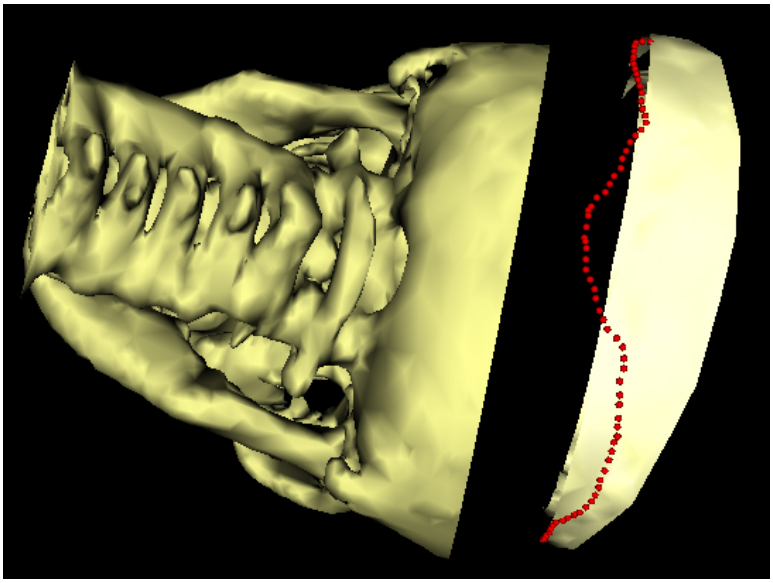

Abbildung 29: Schneiden in der Vorgängersoftware

Kamerasteuerung Der Gerätestift wird während der Kamerasteuerung durch eine simulierte Federkraft in der Mitte des haptischen Arbeitsbereiches gehalten. Dies erschwert die Steuerung der Kamera mit dem haptischen Gerät.
Darüber hinaus ist es für den Anwender nicht nachvollziehbar welcher Kameramodus (Rotation, Zoom, Verschiebung) aktuell ist. Dies resultiert daraus, dass er den Modus nicht vor der Gerätebewegung selbst bestimmt, sondern aus der Gerätebewegung auf einen Kameramodus geschlossen wird (vgl. Kapitel 4.1.1). Die Bewegungsmodi sind nicht parallel ausführbar, sondern schließen sich gegenseitig aus. Es ist für den Benutzer schwierig bei einer Art der Bewegung die Schwellwerte für die anderen Bewegungsarten nicht zu überschreiten. So kann ein gewünschtes Zoomen schnell zu einer Verschiebung der Kamera führen, da diese die höchste Priorität innerhalb der Bewegungsmodi besitzt. Für den Anwender ist daher die Kamerasteuerung mit dem haptischen Gerät nicht intuitiv zu benutzen.

Softwarearchitektur Die Klasse `vtkQtOpenHLRenderWindowInteractor` enthält einen großen Teil der beschriebenen Funktionen. Methoden, die keine gemeinsame Verbindung besitzen, wie zum Beispiel das Erzeugen einer Schnittebene und das Transformieren des haptischen Mauszeigers, sind in dieser Klasse implementiert. Dies erschwert eine Erweiterung der Software und die Wiederverwendbarkeit einzelner Komponenten.

4.2 Änderungen und Erweiterungen

Die weiteren Abschnitte gehen auf die notwendigen Änderungen ein, die die sich aus den beschriebenen Problemen und Schwächen (siehe Kapitel 4.1.3) ergeben haben. Zusätzlich wird die Erweiterung einzelner Komponenten erläutert und auf neu eingefügte Elemente eingegangen.

- **Optimierung und Erweiterung der haptischen 3D-Interaktion**
  Die Anwendung *ReModelVR* modifiziert die bestehende Implementierung so, dass die Notwendigkeit entfällt, einen *Dummy* anstelle des Original-Objektes zu bewegen. Desweiteren läuft das Zusammenspiel einer Bewegung der Objekte mit der Maus durch die von VTK bereitgestellten *interactor styles* und einer Bewegung der Objekte durch das haptische Gerät fehlerfrei ab.
  Das *OpenHaptics Toolkit* bietet die Möglichkeit durch die *haptic mouse utility library* eine 2D-Maus-Eingabe nachzuahmen. So können zum Beispiel Menüs oder Slider mit dem haptischen Gerät bedient werden. Die *haptische Maus* wird in die Anwendung integriert.

- **Softwaretechnische Neugestaltung**
  Eine Integration der HLAPI in VTK wurde in [4] entwickelt. Im vorgestellten Projekt wird dieser Ansatz in das Konzept der *mihLib* eingebunden. Darüber hinaus erfolgt eine bessere Aufteilung der verschiedenen Funktionen in Klassen, um die Erweiterbarkeit und Wiederverwendbarkeit von einzelnen Komponenten zu verbessern. Bislang sind die meisten Methoden in der Klasse `vtkQtOpenHLRenderWindowInteractor` implementiert.

- **Verbesserung der Kamerasteuerung**
  In der Vorgängersoftware wird die Steuerung der Kamera durch eine simulierte Federkraft erschwert. Eine intuitive Steuerung der Kamerabewegungen durch das haptische Gerät ist nicht möglich.
  Die vorliegende Anwendung lässt eine freie Bewegung des Gerätearmes während der Kamerasteuerung zu. Zusätzlich wird dem Benutzer die Möglichkeit gegeben, aus verschiedenen Kameramodi auszuwählen.

- **Verbesserung und Erweiterung der Schneidefunktion**
  Um zum Beispiel zerstörte Knochenteile oder bei der Segmentierung entstandene Ausreißer entfernen zu können, ist die Implementierung einer Funktion notwendig, die es ermöglicht, Modelle beliebig zu schneiden. In dem beschriebenen Ansatz (Kapitel 4.1) hat der Benutzer die Möglichkeit, Schnittmarken auf der Objektoberfläche einzuzeichnen. Geschnitten wird jedoch nicht entlang der von diesen Marken definierten Kante, sondern entlang einer Ebene. Dieser Ansatz

wird für den Benutzer intuitiver gemacht, indem entlang der eingezeichneten Schnittmarken geschnitten wird. Weitere Optionen, wie zum Beispiel die Auswahl verschiedener Schneidewerkzeuge, wurde implementiert.

- **Erweiterung um eine Spiegelfunktion**
  Die Funktion zur Spiegelung von Objekten wurde implementiert, um beispielsweise fehlende Knochenteile durch Spiegelung vorhandener, jedoch spiegelverkehrter, Knochenteile zu ersetzen. Der Vollständigkeit halber wurden alle bekannten Spiegelungsarten (Ebenen-, Achsen-, Punkt-Spiegelung) in die Anwendung eingefügt.

- **Einfügen eines Registrierungsalgorithmus**
  Eine Registrierung bedeutet das Abbilden eines Datensatzes auf einen anderen. Als Registrierungsalgorithmus wurde der in VTK implementierte *Iterative Closest Point Algorithmus* gewählt. Der Algorithmus soll den Anwender unterstützen, Knochen anhand eines Referenzmodelles korrekt zu positionieren.

4.2.1 Optimierung und Erweiterung der haptischen Interaktion

Damit sowohl das Transformieren der Objekte mit dem haptischen Gerät als auch mit der 2D-Maus einwandfrei funktioniert, wird der Ablauf des in Kapitel 4.1.3 beschriebenen Prozesses verändert. Das Öffnen und Schließen des haptischen Frames muss an einer anderen Stelle dieses Prozesses erfolgen. Zur Berechnung der korrekten Transformationen darf dem Aufruf `hlBeginFrame` nicht die mit der Transformationsmatrix eines *Actors* verknüpfte *Modelview*-Matrix vorliegen. In der Anwendung *ReModelVR* ist der Aufruf von `hlBeginFrame` und `hlEndFrame` aus dem `vtkOpenGL-OpenHLPolyDataMapper` entfernt.
Die Funktion `UpdateGeometry` der Klasse `vtkRenderer` wurde überschrieben. In diese Methode werden die Aufrufe zum Öffnen und Schließen des haptischen Frames eingefügt. Der Aufruf der Render-Funktion des *Actors* erfolgt erst, nachdem die *Modelview*-Matrix bereits von `hlBeginFrame` ausgelesen wurde. Somit hat die interne Transformationsmatrix des *Actors* keinen Einfluss mehr auf die Positionen, Transformationen und Vektoren, die im *Collision-* oder *Client*-Thread abgefragt werden. Die ausgelesene *Modelview*-Matrix beinhaltet lediglich die Transformation des Weltkoordinatensystems in das Kamerakoordinatensystem (siehe Abbildung 30).
Bislang lief im haptischen Mapper ein Zähler, der benutzt wurde um das haptische Frame mit dem ersten haptischen Objekt zu öffnen und mit dem letzten haptischen Objekt zu schließen. Die Notwendigkeit dieses Zählers sowie der Sortierung der Aktoren vor der Übergabe an den Renderer entfällt. Da weiterhin alle haptischen Objekte innerhalb eines haptischen Frames gerendert werden müssen, findet die Unterscheidung zwischen hap-

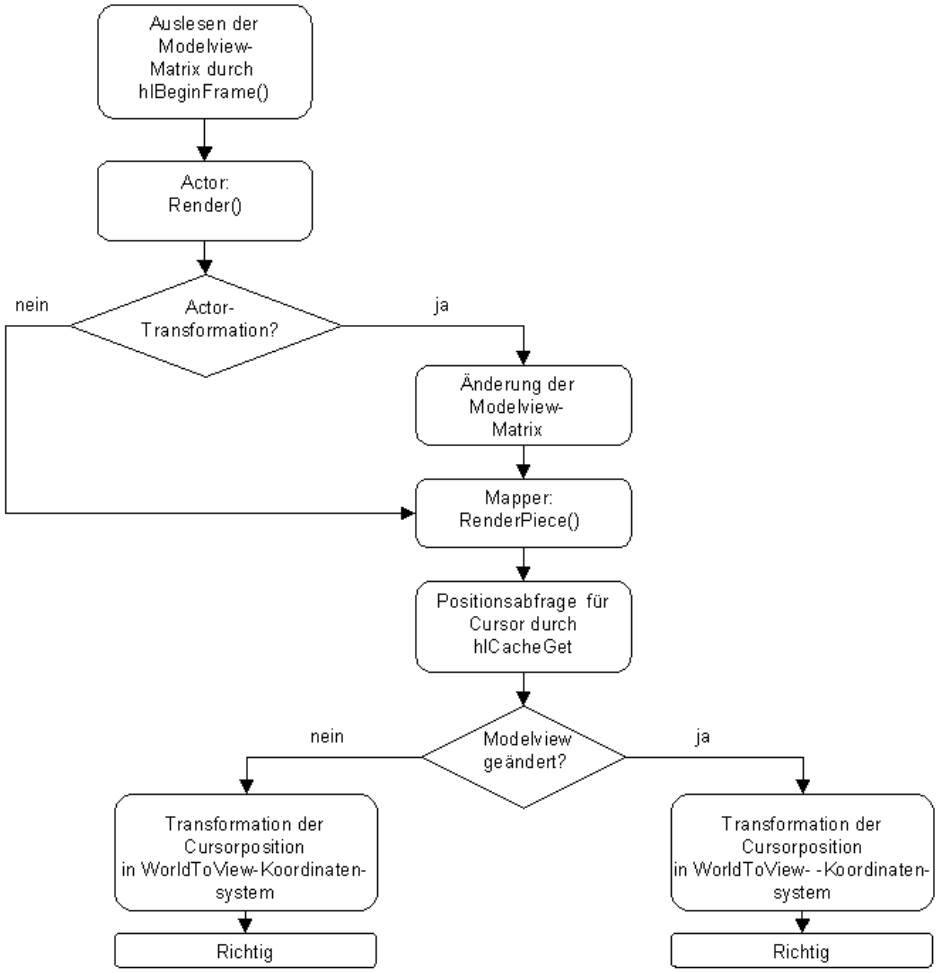

Abbildung 30: korrekte Transformationsberechnung

tischen und nicht-haptischen Objekten in der Funktion `UpdateGeometry` statt. Die haptischen Objekte werden zuerst haptisch gerendert (umklammert von `hlBeginFrame` und `hlEndFrame`) und anschließend werden alle Objekte grafisch gerendert.
Durch die Modifikation besteht keine Notwendigkeit mehr, die Änderung der internen Transformationsmatrix des *Actors* des Original-Objektes zu umgehen. Der *Dummy* wird nicht länger benötigt. Beim Wechsel zwischen dem Bewegen eines Objektes mit dem haptischen Gerät und dem Bewegen des Objektes mit der 2D-Maus werden keine fehlerhaften Transformationen mehr berechnet.
Zur Verdeutlichung der Optimierung wird der Prozess, der bei der Bewegung eines Objektes mit dem haptischen Gerät ausgeführt wird, kurz zu-

sammengefasst:

- **Erfassen des Objektes**
  Beim Hinzufügen eines Objektes zu einer Szene muss ein Callback initialisiert werden, das auf ein *Button1_Down Event* bei Berührung dieses Objektes reagiert. Beim Drücken des ersten Geräteknopfes wird der Vektor der haptischen Objekte nach dem Objekt mit der passenden *shapeId* durchsucht und als aktuelles Objekt deklariert.
  Das in der Vorgängersoftware implementierte Callback, welches auf ein *Touch Event* zur Ermittelung des aktuellen Objektes reagiert, entfällt.

- **Bewegen des Objektes**
  Durch die vorher beschriebenen Änderungen kann das Original-Objekt bewegt werden. Der zuvor benötigte *Dummy* fällt weg.

- **Loslassen des Objektes**
  Das aktuelle Objekt wird für ungültig erklärt und die Bewegung wird beendet. Ein Transformieren der Daten durch einen *TransformPolyDataFilter* ist nicht länger notwendig.

Haptische Maus Um die Elemente der grafischen Benutzeroberfläche mit dem haptischen Gerät bedienen zu können, bietet das *OpenHaptics Toolkit* die Funktion, eine 2D-Maus mit dem haptischen Gerät nachzuahmen. Die haptische Maus benötigt den aktuellen Rendering-Kontext, sowie Zugriff auf das Hauptfenster der Anwendung. Die absolute 3D-Position des haptischen Gerätes wird mittels der *viewing*-Transformationen von OpenGL auf den 2D-Bildschirm abgebildet, um einen nahtlosen Übergang zwischen der Bewegung des haptischen 3D-Zeigers und des 2D-Mauszeigers auf dem Bildschirm zu gewährleisten. Verlässt der haptische Zeiger den *Viewport*, wird er automatisch in einen 2D-Mauszeiger umgewandelt. Es findet ein Wechsel zwischen aktivem (2D-Mauszeiger) und inaktivem (haptischer Mauszeiger) Zustand der haptischen Maus statt.
Um die haptische Maus zu integrieren, muss die Anwendung in regelmäßigem Abstand `hlCheckEvents` aufrufen, da *Motion Events* des *Client*-Thread registriert werden, um den Wechsel zwischen aktivem und inaktivem Zustand der Maus zu überwachen. Desweiteren müssen mindestens vier Funktionen integriert werden:

- Das Initialisieren der Maus (`hmInitializeMouse`) mit der Übergabe des aktuellen Rendering-Kontextes und des Hauptfensters der Anwendung

- Die Übergabe der *viewing*-Transformationen von OpenGL mit dem Aufruf `hmSetMouseTransforms`

- Die Funktion `hmRenderMouseScene`, die innerhalb eines haptischen Frames aufgerufen werden, um einen Reibungseffekt bei Aktivität der haptischen Maus zu generieren

- Die Funktion zum Ausschalten der Maus (`hmShutDownMouse`), die vor dem Löschen des haptischen Rendering-Kontextes durchgeführt werden muss

Durch eine Vergrößerung des haptischen Arbeitsbereiches in x- und y-Richtung ist es möglich, die Maus außerhalb des *viewport* zu bewegen. [29]

4.2.2 Softwaretechnische Neugestaltung

In den nächsten Abschnitten werden die einzelnen Klassen, die im Zuge der softwaretechnischen Neugestaltung erstellt wurden, genauer beschrieben.

`QVTKHapticWidget` Die bereits vorhandene Implementierung leitet den `vtkQtOpenHLRenderWindowInteractor` von der Klasse `vtkQtRenderWindowInteractor` ab. Die neu gestaltete Implementierung nutzt stattdessen die von VTK zur Verfügung gestellte Klasse `QVTKWidget`. Diese bietet einen Weg, VTK-Daten in einem Qt-Fenster darzustellen. Die Bereitstellung eines Interactor (`QVTKInteractor`) erfolgt ebenfalls über diese Klasse. Um das `QVTKWidget` in die *mihLib* zu integrieren, wird das zuvor von `vtkQtRenderWindowInteractor` abgeleitete `mihQtSceneWidget` von der Klasse `QVTKWidget` abgeleitet. Von der Klasse `mihQtSceneWidget` wiederum wird die Klasse `QVTKHapticWidget`abgeleitet (siehe Abbildung 31).
Diese beinhaltet die Verarbeitung der auftretenden *Events* durch Callbacks, sowie die Initialisierung der Haptik. Dazu zählt die Initialisierung des haptischen Gerätes, des Cursors sowie der haptischen Maus. In der vorherigen Implementierung wurden diese Funktionen alle in die Klasse `vtkQtOpenHLRenderWindowInteractor` eingefügt.

`QVTKHapticInteractor` Diese Klasse erbt von `QVTKInteractor`. Über diese Klasse können verschiedene Modi gesetzt werden. Für eine Nutzung des haptischen Gerätes zur Exploration und Bewegung von Objekten, muss der *DragMode* ausgewählt sein. Um Schnittmarken einzuzeichnen, muss der *CutMode* aktiv sein. Für eine Steuerung der Kamera über das haptische Gerät ist eine Aktivierung des *CameraMode* notwendig. Das Setzen der Modi, sowie deren Abfrage geschieht über Funktionen der Klasse `QVTKHapticInteractor`.

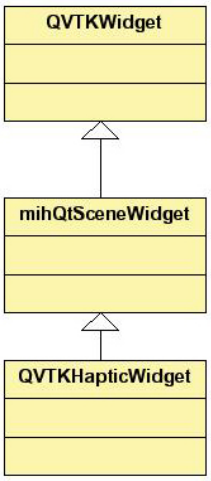

Abbildung 31: Vererbungsdiagramm für `QVTKHapticWidget`

`mihHapticScene` Um haptische Objekte in eine Szene einfügen zu können, wurde die Klasse `mihHapticScene` erstellt. Sie ist von der in der *mihLib* bestehenden Klasse `mihScene` abgeleitet. Mit den Funktionen `AddSceneObject` bzw. `AddSceneObjects` werden einer Szene eines bzw. mehrere Objekte hinzugefügt. Die beiden Funktionen werden in der Klasse `mihHapticScene` überschrieben. Sie werden dahingehend erweitert, dass anhand des *Mappers* eine Unterscheidung in haptische und nicht-haptische Objekte stattfindet. Für ein haptisches Objekt wird innerhalb der Funktion eine eindeutige Identifikatinsnummer (*shapeId*) erzeugt. Zusätzlich werden die haptischen und nicht-haptischen Objekte in zwei verschiedene *Vektoren* abgelegt. Die Funktion zum Löschen von *mihSceneObjects* wurde ebenfalls überschrieben, um dort die Funktion zum Entfernen der *shapeId* eines haptischen Objektes zu integrieren. Zusätzlich erfolgt die Darstellung von Schnittmarken über Funktionen dieser Klasse. Weiterhin ist eine Funktion integriert, die beim Bewegen eines Szene-Objektes, die Transformationen des Gerätes auf das Objekt überträgt.

`mihHapticPolyDataMapper` Um Objekte haptisch rendern zu können, muss ihnen ein haptischer Mapper zugeordnet werden. Die Klasse stellt Methoden bereit, um Objekten eine eindeutige *shapeId* zuweisen und bei Bedarf wieder freigeben zu können. Die Render-Funktion dieser Klasse wird um die Funktionen `hlBeginShape` und `hlEndShape` erweitert. Diese werden für die haptische Darstellung der Objekte benötigt.

mihHapticSurfaceObject Durch die Klasse `mihHapticSurfaceObject` wird eine einfache Möglichkeit bereitgestellt, haptische Objekte zu erzeugen. Die Klasse ist von der in der *mihLib* verfügbaren Klasse `mihSurfaceObject` abgeleitet (siehe Abbildung 32). Diese Klasse kapselt unter anderem einen `vtkPolyDataMapper`. In der neuen Klasse wird dieser Mapper durch einen `mihHapticPolyDataMapper` ersetzt.

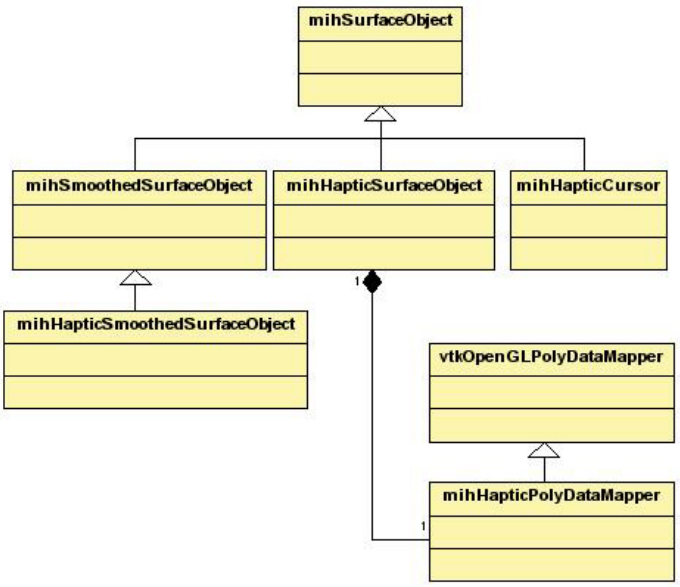

Abbildung 32: Vererbungshierarchie des `mihSurfaceObject`

mihHapticCursor Diese Klasse enthält Methoden zur Initialisierung und Generierung eines Zeigers. Zusätzlich implementiert die Klasse eine Funktion zur Berechnung der Zeigertransformation. Da der `mihHapticCursor` von der Klasse `mihSurfaceObject` abgeleitet wurde (siehe Abbildung 32), kann er wie andere *mihSceneObjects* einer Szene hinzugefügt werden.

mihHapticRenderer Die in 4.2.1 beschriebene Änderung machte das Überschreiben der Funktion `UpdateGeometry` notwendig. In dieser Funktion wird das Öffnen und Schließen eines haptischen Frames ausgeführt. Da die haptischen Objekte vor den nicht-haptischen gerendert werden müssen, wird vor dem Aufruf der Funktionen, die den Rendering-Prozess fortführen, eine Unterscheidung zwischen haptischen und nicht-haptischen

Objekten durchgeführt. Um ein Überschreiben der Funktion `UpdateGeometry` möglich zu machen, wurde die Klasse `mihHapticRenderer` erstellt, die von `vtkOpenGLRenderer` abgeleitet ist. Genau ein haptischer Renderer (`mihHapticRenderer`) wird einer haptischen Szene (`mihHapticScene`) hinzugefügt.

`mihHapticProperty` Die Klasse `vtkOpenGLOpenHLProperty` wurde lediglich in `mihHapticProperty` umbenannt und keinen weiteren Änderungen unterzogen. Da ein `mihHapticSurfaceObject` einen `vtkActor` kapselt, können über diesen die haptischen Materialeigenschaften des `mihHapticSurfaceObject` bestimmt werden.

`mihHapticDevice` Diese Klasse beinhaltet die Funktion zum Initialisieren des haptischen Gerätes, sowie Funktionen zum Ein- und Ausschalten der Kraftrückgabe. Bisher war diese Funktionalität in der Klasse `vtkQtOpenHLRenderWindowInteractor` implementiert.

`mihHapticDeviceCamera` Auch die Steuerung der Kamera findet in der Vorgängersoftware über Funktionen der Klasse `vtkQtOpenHLRenderWindowInteractor` statt. Die vorliegende Anwendung erstellt für die Kamerasteuerung die Klasse `mihHapticDeviceCamera`. Sie stellt unter anderem eine Funktion bereit, welche die Position und Drehung des haptischen Gerätes ausliest und die Kamera entsprechend bewegt. Eine tiefergehende Erläuterung der Kamerasteuerung findet sich in 4.2.3.

Eine Gesamtübersicht der oben beschriebenen Klassen zeigt Abbildung 33.

Um die Haptik in eine Anwendung, die mit der *mihLib* arbeitet, zu integrieren, reicht die Ausführung der in Listing 9 und 10 aufgeführten Funktionen. Die hinzugefügten haptischen Objekte können erfühlt und bewegt werden. Auch das Ändern ihrer Oberflächeneigenschaften ist möglich. Zusäzlich wird der haptische Mauszeiger, der den Proxy repräsentiert, angezeigt.
Eine Funktion zum Schneiden von Objekten ist in [4] bereits vorhanden. Die dazu notwendigen Funktionen wie das Setzen von Schnittmarken oder das Generieren einer Schnittebene, sind der Klasse `vtkQtOpenHLRenderWindowInteractor` hinzugefügt. Da die Schneidefunktion im Rahmen der vorliegenden Arbeit verbessert werden soll, wurden zusätzliche Klassen angelegt, die in Kapitel 4.2.4 erläutert werden.

Listing 9: Integration der Haptik

```
m_qvtk3D = new QVTKHapticWidget(m_hapticWidget,"haptic",0));
m_HapInteractor = QVTKHapticInteractor::New();
m_hapticScene = mihHapticScene::New());

//fügt die haptische Szene in
//das QVTKHapticWidget ein
m_qvtk3D->SetScene(m_hapticScene);

//setzt den haptischen Interactor
//für das RenderWindow des Widgets
m_qvtk3D->GetRenderWindow()->SetInteractor(m_HapInteractor);

//initialisiert das haptische Gerät,
//die haptische Maus und
//erstellt den Cursor
m_qvtk3D->InitHaptics(this->caption(),this->size(),
m_hapticWidget->frameGeometry());
```

Listing 10: Einfügen eines haptischen Oberflächenmodelles

```
//lädt eine .vtk-Datei
m_hapticSurfaceObject = mihHapticSurfaceObject::New();
m_hapticSurfaceObject->LoadObject(surfaceFileName);

//das Objekt wird der
//haptischen Szene hinzugefügt
m_hapticScene->AddSceneObject(m_hapticSurfaceObject);

//Callback das reagiert, wenn der
//erste Geräteknopf gedrückt wird und der
//Proxy das m_hapticSurfaceObject berührt
m_qvtk3D->AddButton1DownCallback(m_hapticSurfaceObject);
```

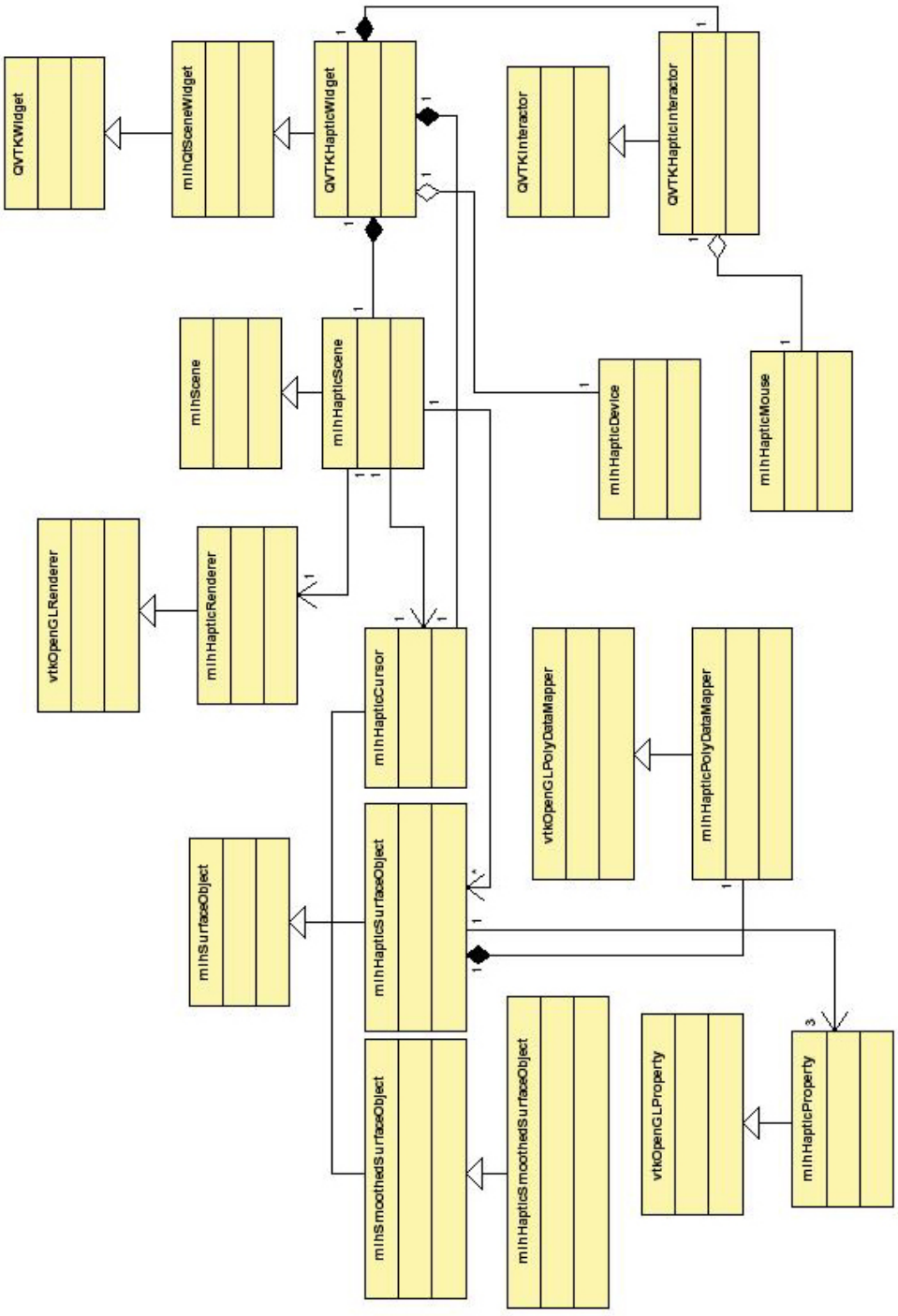

Abbildung 33: Gesamtübersicht

4.2.3 Kamerasteuerung

Die Kamera der Szene kann mit dem haptischen Gerät gesteuert werden. Folgende Kamerabewegungen sind möglich (siehe Abbildung 34):

- Bewegen der Kamera entlang der Achse der Projektionsrichtung $\vec{proj}_{dir} = p_{foc} - p_{cam}$ (`Dolly`)
- Rotation der Kamera um $\vec{proj}_{dir} = p_{foc} - p_{cam}$ (`Roll`)
- Rotation um den am Fokuspunkt liegenden *view up*-Vektor $\vec{v}_{up}$ der Kamera (`Azimuth`)
- Rotation um $\vec{e}_{orth} = \vec{v}_{up} \times \vec{proj}_{dir}$ (`Elevation`)
- Veränderung der Kameraposition (`SetPosition` und `SetFocalPoint`)

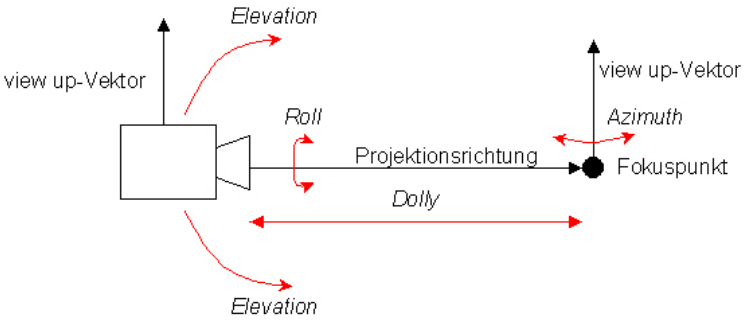

Abbildung 34: Kamerabewegungen

Beim Bewegen der Kamera hat der Benutzer die Auswahl zwischen den oben genannten Kamerabewegungen. Während der Bewegung der Kamera muss der erste Knopf des Gerätes gedrückt werden. Der zweite Knopf dient dazu, die Modi durchzuschalten und einen daraus auszuwählen:

- Zoom (`Dolly`)
- Rotate (`Azimuth` und `Elevation`)
- Spin (`Roll`)
- Pan (`SetPosition` und `SetFocalPoint`)
- Zoom + Rotate + Spin

Zur besseren Orientierung für den Benutzer wird der aktuelle Modus im Ausgabefenster angezeigt.
Der erste Modus bewirkt ein Heran- oder Heraus-Zoomen der Szene. Wird der Arm des haptischen Gerätes in diesem Modus zur Gerätebasis hin- bzw. weggeführt, bewegt sich die Kamera entlang der Projektionsrichtung. Dies bewirkt eine Annäherung oder Entfernung der Kamera zur Szene. Dazu wird die Position des Gerätestiftes abgefragt. Anhand des Vergleiches zwischen alter und neuer Position wird entschieden, in welche Richtung (zur Szene hin oder von der Szene weg) die Bewegung der Kamera erfolgt. Um die Kamera um die Szene zu rotieren, kann der Gerätearm nach oben, unten, rechts und links bewegt werden (siehe Abbildung 35). Bei einer Be-

Abbildung 35: Bewegung des Gerätearms, um eine Rotation der Kamera zu erzeugen

wegung nach oben oder unten, rotiert die Kamera um $\vec{e}_{orth}$ (`Elevation`). Eine Bewegung nach rechts oder links, rotiert die Kamera um $\vec{v}_{up}$ (`Azimuth`). Wir der Gerätearm beispielsweise nach oben rechts bewegt, werden beide Rotationsmöglichkeiten (`Elevation` und `Azimuth`) ausgeführt. Das gleiche gilt für die weiteren Kombinationen. Auch im Rotationsmodus wird anhand der Veränderung der Gerätestiftposition entschieden, in welche Richtung die Kamera rotiert werden muss. Um die Kamera um die Achse der Projektionsrichtung zu drehen, muss der Spin-Modus ausgewählt sein. Eine Drehung des Stiftschaftes des haptischen Gerätes bewirkt diese Art der Drehung.
Eine Positions- und Fokuspunktveränderung der Kamera wird durch den Pan-Modus durchgeführt. Die Kamera soll entgegen der Bewegung des Gerätes verschoben werden. Daraus resultiert eine Verschiebung der Szene in Richtung der Geräteposition. Dazu erfolgt die Berechnung eines Vektors zwischen der neuen Geräteposition und dem Fokuspunkt der Kamera. Dieser Vektor wird mit einem Faktor (hier $1/20$) skaliert und auf die Kame-

raposition und den Fokuspunkt angewendet. Um ein Zoomen zu verhindern, werden lediglich die x- und y-Koordinaten der Kameraposition und des Fokuspunktes verändert.
Um zu vermeiden, dass bei sehr kleinen Bewegungen des haptischen Gerätes (z.B. durch Zittern der Hand) die Kamera bewegt wird, müssen festgelegte Schwellwerte überschritten werden, bevor eine Bewegung der Kamera erfolgen kann.
Da nicht jedes haptische Gerät zwei Knöpfe besitzt, werden zwei zusätzliche Klassen bereitgestellt: `mihHapticInteractorStyleSwitch` und `mihHapticInteractorStyleCamera`. Die erste Klasse ist von der VTK-Klasse `vtkInteractorStyleSwitch` abgeleitet. Diese gibt dem Benutzer die Möglichkeit zwischen vier Interaktionsstilen zu wählen: *joystick actor*, *joystick camera*, *trackball actor* und *trackball camera*. Die abgeleitete Klasse wird um einen fünften Stil ergänzt: *haptic camera*. Das Drücken der Taste „H“ auf der Tastatur aktiviert den Stil und ein Callback für ein *Move Event* wird erstellt. Dieses Callback wird in der Klasse `mihHapticInteractorStyleCamera` verarbeitet. Der Benutzer kann mit den Tasten 1-5 auf der Tastatur zwischen den verschiedenen Modi wählen:

- Taste 1: Zoom
- Taste 2: Rotation
- Taste 3: Spin
- Taste 4: Pan
- Taste 5: Zoom + Rotate + Spin

Entsprechend dieser Wahl führt das Callback die gewünschte Kamerabewegung aus.
Die vorliegende Arbeit macht die Benutzung der Kamerasteuerung einfacher und intuitiver. Zum einen wird die Bewegung des haptischen Gerätes nicht weiter durch eine Federkraft erschwert, zum anderen ist durch die Auswahlmöglichkeit der Kameramodi für den Anwender zu jedem Zeitpunkt ersichtlich, welcher Modus aktuell ist. Der Benutzer entscheidet erst welche Kamerabewegung auszuführen ist und daraufhin werden die Bewegungen des haptischen Gerätes auf die gewählte Kamerabewegung übertragen.

4.2.4 Schneiden von Objekten

Um eine Möglichkeit bereitzustellen zerstörte oder verformte Knochenteile abschneiden, oder um, durch die Segmentierung entstandene, nicht zu einem Teil gehörende Strukturen, entfernen zu können, wurden Funktionen implementiert, die das Schneiden von Oberflächenmodellen durchführen.

Das Schneiden von Objekten kann mit Hilfe impliziter Funktionen umgesetzt werden.

Implizite Funktionen Eine implizite Funktion hat die Form:

$$F(x, y, z) = constant \tag{12}$$

Sie teilt den euklidischen Raum in drei Bereiche: außerhalb der impliziten Funktion ($F(x, y, z) > 0$), innerhalb der impliziten Funktion ($F(x, y, z) < 0$) und auf der impliziten Funktion ($F(x, y, z) = 0$). Die implizite Funktion für eine Kugel mit Radius R beispielsweise sieht wie folgt aus:

$$F(x, y, z) = x^2 + y^2 + z^2 - R^2 \tag{13}$$

Ein Punkt kann durch einfaches Einsetzen in die Gleichung als innerhalb, außerhalb oder auf der impliziten Funktion liegend, eingeordnet werden. Eine interessante Möglichkeit von impliziten Funktionen ist, sie miteinander zu verknüpfen, um geometrische Objekte zu bilden. Zu diesem Zweck können die booleschen Operatoren Vereinigung, Durchschnitt und Differenz benutzt werden. Die Vereinigung $F \cup G$ zwischen zwei Funktionen $F(x, y, z)$ und $G(x, y, z)$ an dem Punkt (x_0, y_0, z_0) wird bezeichnet als

$$F \cup G = \min(F(x_0, y_0, z_0), G(x_0, y_0, z_0)), \tag{14}$$

während der Durchschnitt durch

$$F \cap G = \max(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \tag{15}$$

und die Differenz durch

$$F - G = \max(F(x_0, y_0, z_0), -G(x_0, y_0, z_0)) \tag{16}$$

gegeben ist. Für diese Verknüpfungen steht die VTK-Klasse `vtkImplicitBoolean` zur Verfügung.
Implizite Funktionen werden zum Schneiden benötigt, um für jeden Punkt des Objektes, das geschnitten wird, den Funktionswert der impliziten Funktion zu bestimmen.

Schneiden in VTK Um Objekte mit Hilfe des Visualization ToolKit in zwei Teilobjekte zu zerlegen, können zwei Methoden angewendet werden: Clippen oder Extrahieren.

Clippen von polygonalen Daten Für das Clippen von polygonalen Daten wird die implizite Funktion des Objektes benötigt, das als Schneidewerkzeug dient. Jeder Punkt des Objektes, das geschnitten werden soll, wird in die implizite Funktion eingesetzt und so der Funktionswert s berechnet. Daraufhin wird für jede Zelle (hier Dreiecke) eine Clip-Funktion ausgeführt. In dieser wird überprüft, ob ein Durchschneiden der Zelle erforderlich ist. Es werden die Kanten der Dreiecke betrachtet: Ein Schnitt liegt vor, wenn die vorher berechneten s der Kantenendpunkte unterschiedliche Vorzeichen haben. Ist dies der Fall, wird durch Interpolation ein neuer Punkt auf der Kante erstellt. Durch diesen Prozess entstehen beim Schneiden eines Dreieckes zwei neue Punkte, anhand derer neue Dreiecke gebildet werden. Ein Durchschneiden der Dreiecke findet statt.
Die Klasse `vtkClipPolyData` führt ein solches Clippen durch. Mit dem Aufruf `GetOutput()` des Filters werden die Dreiecke des Objektes ausgegeben, die innerhalb oder auf der impliziten Funktion ($s \leq 0$) liegen, während der Aufruf von `GetClippedOutput()` die Dreiecke ausgibt, die sich außerhalb der impliziten Funktion ($s > 0$) befinden (siehe Abbildung 36). Dies gibt die Möglichkeit mit Hilfe des Filters `vtkClipPolyData` ein Ob-

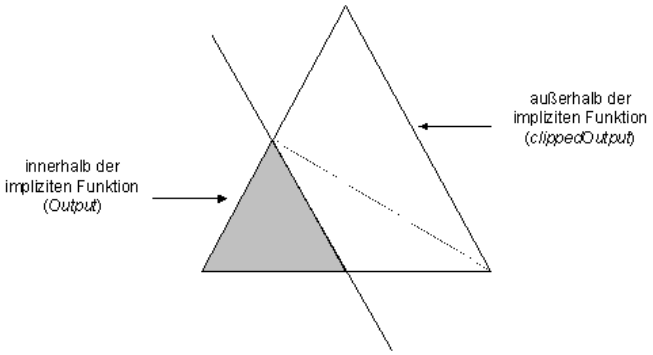

Abbildung 36: Der graue Bereich befindet sich innerhalb, der weiße Bereich außerhalb der impliziten Funktion

jekt zu schneiden und die beiden entstehenden Teilobjekte auszugeben.

Extrahieren von polygonalen Daten Zum Extrahieren von polygonalen Daten aus einem Datenset wird ebenfalls eine implizite Funktion benötigt. Für jeden Punkt des Datensets findet die Berechnung des Funktionswertes für diesen Punkt der impliziten Funktion statt. Ist dieser Wert für alle Punkte eines Dreiecks > 0, wird das Dreieck als außerhalb der impliziten Funktion liegend angenommen; sind alle Werte ≤ 0 liegt das Dreieck innerhalb der impliziten Funktion. Bei unterschiedlichen Werten kann das Dreieck als innerhalb oder außerhalb liegend eingeordnet werden. Ein

Durchschneiden der Dreiecke findet nicht statt.
Das Extrahieren von Dreiecken aus einem polygonalen Datenset kann mit dem Filter `vtkExtractPolyDataGeometry` durchgeführt werden. Hier kann entschieden werden, ob die Dreiecke deren Punkte unterschiedliche Vorzeichen besitzen, als innerhalb oder außerhalb der impliziten Funktion liegend, eingeordnet werden. Der Filter bietet ebenfalls die Möglichkeit beide Bereiche auszugeben.

Schneiden in der Anwendung Der beschriebene Ansatz (Kapitel 4.1) wurde um verschiedene Komponenten ergänzt. Der Benutzer hat die Auswahl zwischen diversen Schneidewerkzeugen: Ebene, Kugel und Box. Alternativ können Schnittmarken mit dem haptischen Gerät eingezeichnet werden, anhand derer das Objekt geschnitten wird.
In der Klasse `mihCutObject` ist die Schneidefunktion implementiert. Die Übergabe der impliziten Funktion und des zu schneidenden Objektes erfolgt über die Funktion `mihCutObject`. Ausgegeben werden die beiden durch die Schneidefunktion generierten Teilobjekte. Innerhalb der Funktion wird die implizite Funktion des Schneidewerkzeugs als Clip-Funktion an eine Instanz der oben beschriebenen Klasse `vtkClipPolyData` übergeben und so der *Output* und der *geclippte Output* erzeugt, welche die beiden Teilobjekte darstellen. Diese sind ebenfalls mit dem haptischen Gerät erfühlbar und bewegbar.
Zusätzlich hat der Benutzer vor dem Schneiden die Möglichkeit zu wählen, ob das Objekt, das geschnitten wird, beibehalten oder gelöscht werden soll.
Weiterhin kann der Benutzer entscheiden, ob Schnittflächen erzeugt werden. Dies führt zur Erstellung einer Kontur dort, wo die Dreiecke mit Hilfe der impliziten Funktion durchschnitten wurden. Anschließend wird die von der Kontur umschlossene Fläche trianguliert und an beide Teilobjekte angefügt.

Mit der 2D-Maus manipulierbare Schneidewerkzeuge Die Schneidewerkzeuge (Ebene, Kugel, Box) lassen sich durch Manipulation an bereitgestellten *Handles* skalieren, rotieren oder translatieren. Für die Implementierung dieser Funktionen wurden die von VTK bereitgestellten *Widgets* (zum Beispiel `vtkPlaneWidget` für die Schnittebene) verwendet. Da diese Klasse wiederum von `vtkInteractorObserver` abgeleitet sind, reagiert sie auf *Events*, die vom `vtkRenderWindowInteractor` ausgelöst werden. Die *Widgets* sind mit jeder Instanz von `vtkInteractor-Style` kompatibel und so kann der Benutzer, zusätzlich zur Manipulation der *Widgets* anhand der *Handles*, mit ihnen interagieren, wie mit anderen Objekten. Dem Benutzer ist damit eine komfortable Möglichkeit gegeben, die Schneidewerkzeuge mit der Maus zu manipulieren.

Ein weiterer Vorteil der *Widgets* besteht darin, dass eine Methode implementiert ist, die implizite Funktion des *Widgets* auszugeben. Diese wird zum Schneiden an die entsprechende Funktion übergeben.

Haptisch manipulierbare Schneidewerkzeuge Um Schneidewerkzeuge zu generieren, die mit dem haptischen Gerät in der Szene positioniert werden können, wurden die Klassen `mihHapticPlane`, `mihHapticBox` und `mihHapticSphere` erzeugt. Sie bieten den Vorteil, dass nicht zwischen 2D-Maus und haptischem Gerät gewechselt werden muss. Um die Schneidewerkzeuge in ihrer Größe zu verändern, können Slider auf der Benutzeroberfläche benutzt werden. Dies kann mit der haptischen Maus geschehen und so ist auch in diesem Fall ein Wechsel zwischen haptischem Gerät und 2D-Maus nicht erforderlich.
Bevor mit diesen Schneidewerkzeugen geschnitten werden kann, muss die Definition ihrer impliziten Funktion erfolgen. Dies geschieht über Funktionen der jeweiligen Klassen.

Freies Schneiden Entscheidet der Benutzer anhand von Schnittmarken zu schneiden, hat er die Möglichkeit diese auf der Objektoberfläche oder im freien Raum vor dem Objekt einzuzeichnen (siehe Abbildung 37). In beiden Fällen muss der Benutzer den ersten Knopf des Gerätes zum

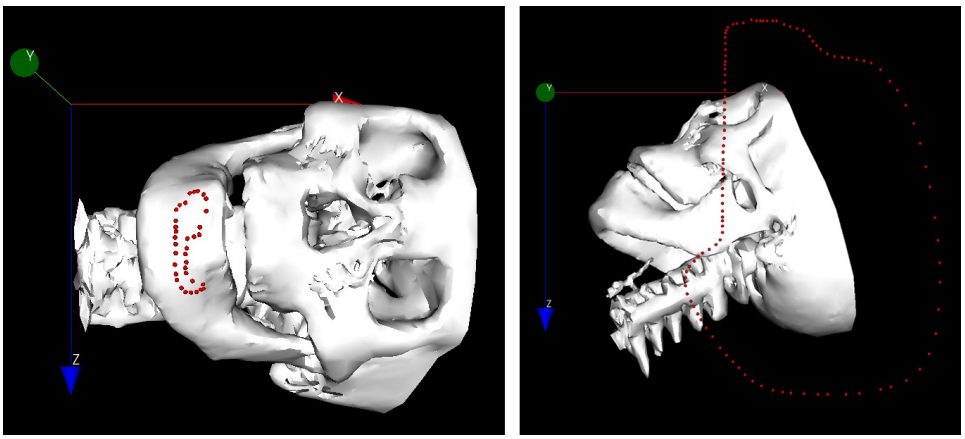

Abbildung 37: **Links:** Zeichnen auf Objekt **Rechts:** Zeichnen vor Objekt

Zeichnen gedrückt halten. In regelmäßigen Zeitabständen wird die Position des Proxy abgerufen. An diese Position wird eine Schnittmarke in Form eines Punktes gesetzt.
Die eingezeichneten Punkte werden mit Hilfe der VTK-Klasse `vtkImplicitSelectionLoop` verarbeitet: „vtkImplicitSelectionLoop computes the implicit function value and function gradient for an irregular, cylinder-like object whose cross section is defined by a set of points forming a loop.“ [30]

Die Punkte müssen weder koplanar, noch muss das Loch, welches von den Punkten geformt wird, konvex sein. Jeder Punkt des Loches wird auf eine Ebene projiziert. Das führt zur Bildung eines Polygons. Per „default“ wird die Normale der Projektionsebene automatisch berechnet. Alternativ kann diese Einstellung außer Kraft gesetzt und eine andere Ebenennormale übergeben werden. Nachdem das Polygon erstellt wurde, werden alle Punkte des Objektes, das geschnitten werden soll, auf die gleiche Ebene projiziert. Daraufhin wird für jeden Punkt überprüft, ob dieser in der *Bounding Box* des Polygons liegt. Nur wenn dies der Fall ist, wird untersucht, ob der Punkt auch innerhalb des Polygons liegt. Zu diesem Zweck wird ein Strahl generiert. Die Anzahl der Schnittpunkte des Strahls mit jeder Kante des Polygons wird berechnet. Ist diese Anzahl gerade, liegt der Punkt außerhalb, ist die Anzahl ungerade, liegt der Punkt innerhalb des Polygons. Anschließend wird für jeden Punkt des Objektes die Distanz zu allen Polygonkanten berechnet. Die kleinste Distanz ist die des zu überprüfenden Punktes zur nächstgelegenen Kante des Polygons. Abhängig davon, ob sich der Oberflächenpunkt innerhalb oder außerhalb des Polygons befindet, wird die negative (innerhalb) oder positive (außerhalb) kleinste Distanz zurückgegeben.
Durch die Projektion aller Punkte der Oberfläche auf eine Ebene kann das Schneiden mit einer *implicit selection loop* mit einem „Lochen“ verglichen werden: Das Stück, das beim „Lochen“ ausgestanzt wird, ist der innerhalb der impliziten Funktion liegende Teil; während das Stück aus dem herausgestanzt wird, der außerhalb der Funktion liegende Teil ist.

- **Schnittmarken vor der Objektoberfläche**
  Werden Schnittmarken vor der Objektoberfläche eingezeichnet, wird die automatische Normalengenerierung der Klasse `vtkImplicitSelectionLoop` außer Kraft gesetzt. Statt dessen wird die Normale der *view plane* zur Projektion benutzt. Diese liegt orthogonal zum Bildschirm. Somit wird ein unendlich langes „zylinder-ähnliches“ Objekt erzeugt, dessen Ausrichtung die Richtung der *view plane*-Normalen ist. Das Objekt wird, wie oben beschrieben, in seiner gesamten Tiefe in Richtung der Normalen geschnitten (Abbildung 38).

- **Schnittmarken auf der Objektoberfläche**
  Der Anwender hat bei dieser Option verschiedene Möglichkeiten, Teile aus dem Oberflächenmodell herauszuschneiden:

  - **Clippen ohne Beschränkung**
    Das Objekt wird in Richtung der *view Plane*-Normalen unbeschränkt durchschnitten. Ausgegeben wird der innerhalb sowie der außerhalb der impliziten Funktion liegende Teil (siehe Abbildung 39).

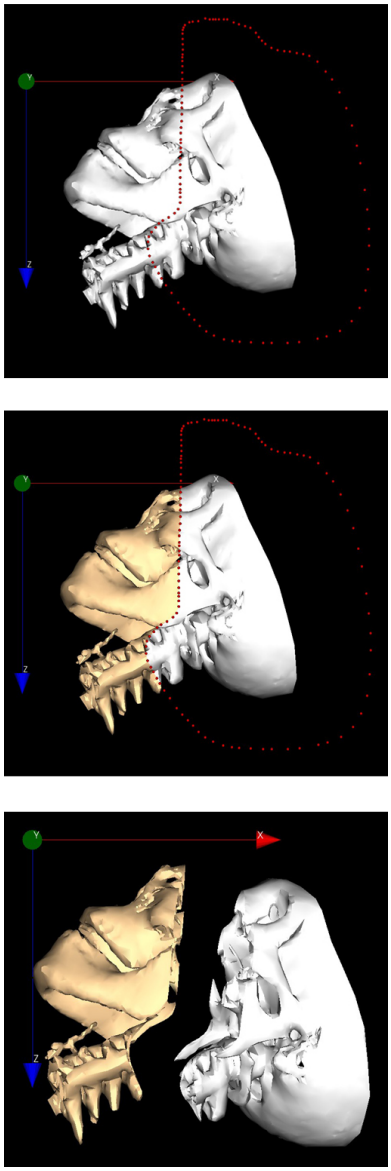

Abbildung 38: **Oben:** Schnittmarken vor dem Objekt **Mitte:** Nach der Schneideoperation **Unten:** Objekt wurde in der ganzen Tiefe durchgeschnitten

– **Clippen mit Beschränkung**
  Das Objekt wird nicht in seiner gesamten Tiefe geschnitten: Die *implicit selection loop* wird mit der impliziten Funktion einer Ebene begrenzt. Erreicht wird dies durch eine boolesche Verknüpfung (Schnittmenge) beider Funktionen (siehe Paragraph **Implizite Funktionen**). Der Ursprung der begrenzenden Ebene wird

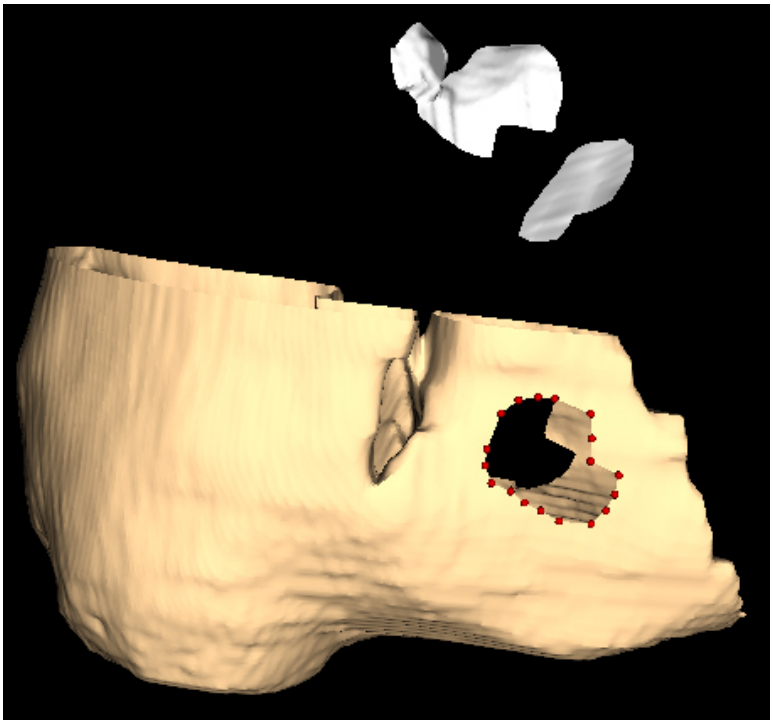

Abbildung 39: Schneiden ohne Beschränkung

von dem am weitesten von der *view plane* entfernten Punkt der Schnittmarken gebildet. Bis zu dieser Tiefe wird das Objekt geschnitten (Abbildung 40).

– **Extrahieren**
Bei dieser Option werden nur die Dreiecke ausgegeben, die innerhalb der Schnittmarken liegen.
Da jedoch bei dieser Möglichkeit ebenfalls die *implicit selection loop* als Schneidefunktion benutzt wird, wird auch hier das Objekt ganz durchschnitten. Um nur den Teil der Oberfläche auszugeben, der innerhalb der Schnittmarken liegt, wird ein weiterer Filter (`vtkPolyDataConnectivityFilter`) verwendet: Ausgehend von dem Punkt eines Dreiecks, der am nächsten zu einem definierten Punkt der *implicit selection loop* liegt, werden die mit diesem Dreieck zusammenhängenden Dreiecke gesucht und ausgegeben. Somit wird nur der herausgeschnittene Teil angezeigt. Beim Extrahieren werden keine Dreiecke durchgeschnitten.

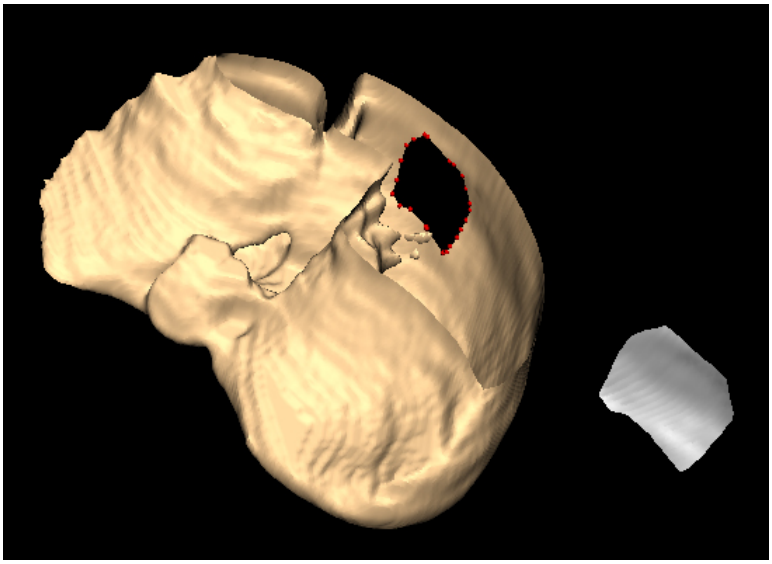

Abbildung 40: Schneiden mit Beschränkung

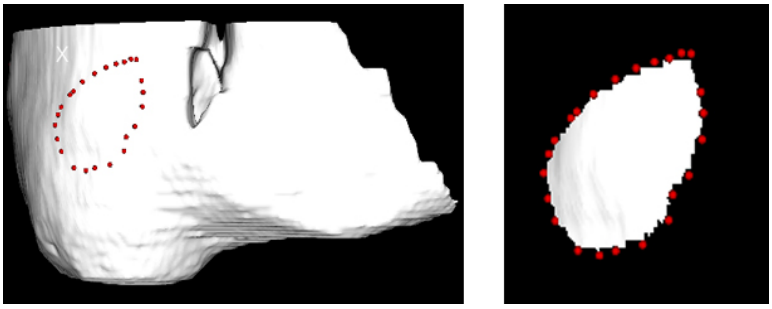

Abbildung 41: Ausgabe der Dreiecke, die innerhalb der Schnittmarken liegen

4.2.5 Spiegeln von Objekten

Die Anwendung *ReModelVR* beinhaltet die Funktion Objekte spiegeln zu können. Wurden Knochenteile eines Objektes abgeschnitten, weil sie verformt oder zerstört waren, kann die Spiegelfunktion verwendet werden, um diese Strukturen zu ersetzen.
Es besteht die Auswahl zwischen einer Punkt-, Achsen- und Ebenenspiegelung. Dem Benutzer ist die Möglichkeit gegeben, die Spiegelebene, den Spiegelpunkt oder die Spiegelachse frei im Raum zu positionieren. Dies wird mit den im vorherigen Abschnitt beschriebenen *Widgets* realisiert. Um Spiegelwerkzeuge zu generieren, die mit dem haptischen Gerät positionierbar sind, werden die oben beschriebenen Klassen `mihHapticSphere`, `mihHapticPlane` neben der Klasse `mihHapticLine` verwendet. Die Klas-

se `mihMirrorObject` implementiert die Funktionen um Objekte zu spiegeln.

Achsenspiegelung Die Spiegelung an einer Achse ordnet jedem Punkt P ein Bildpunkt P^* zu, so dass die Spiegelachse die Verbindungsstrecke $\overline{PP^*}$ rechtwinklig halbiert. Der Benutzer hat die Auswahl an einer der Koordi-

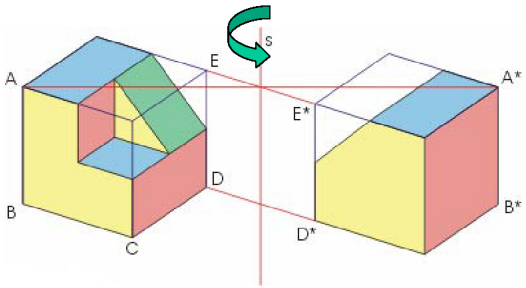

Abbildung 42: Achsenspiegelung (Quelle: [19])

natenachsen (siehe Abbildung 43) oder an einer Achse, die er frei bewegen kann, zu spiegeln (siehe Abbildung 43).
Die Spiegelung an einer Achse kann durch eine 180°-Rotation des Objektes um diese dargestellt werden. Wählt der Benutzer an einer Achse des dargestellten Koordinatensystems zu spiegeln, wird das zu spiegelnde Objekt lediglich um 180° um die angegebene Achse rotiert. Wurde dagegen ausgewählt, an einer beliebigen Achse zu spiegeln, wird eine Transformationsmatrix berechnet, die eine Translation der Spiegelachse in den Ursprung des Koordinatensystems sowie eine Rotation der Spiegelachse auf die z-Achse des Koordinatensystems beinhaltet. Daraufhin findet eine 180°-Rotation des Objektes um die z-Achse statt und die vorhergehende Translation in den Ursprung und Rotation auf die z-Achse wird rückgängig gemacht.

Ebenenspiegelung Jedem Punkt P wird ein Bildpunkt P^* zugeordnet, so dass die Spiegelebene die Verbindungsstrecke $\overline{PP^*}$ rechtwinklig halbiert. Eine Ebenenspiegelung wird durch eine Skalierungsmatrix dargestellt. Die Spiegelung eines Objektes an der xy-Ebene (siehe Abbildung 45) des Koordinatensystems, bedeutet eine Negierung der z-Koordinaten aller Punkte des Objektes. Daraus folgt die Skalierungsmatrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{17}$$

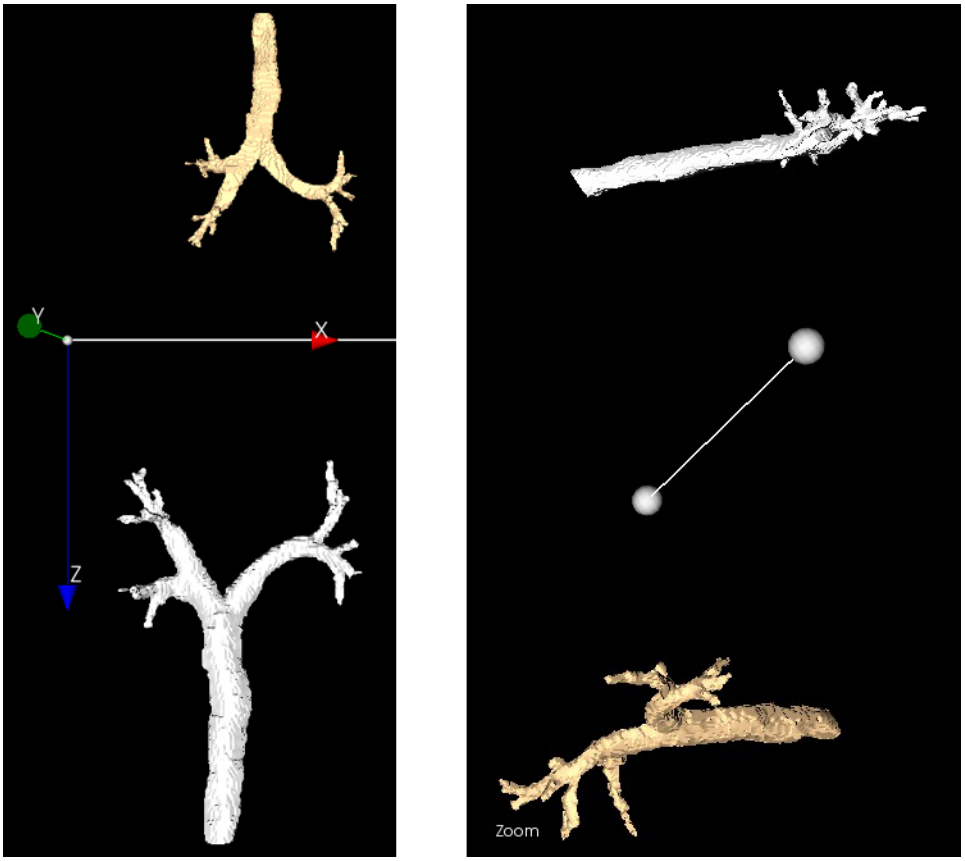

Abbildung 43: **Links:** Spiegelung an der x-Achse **Rechts:** Spiegelung an beliebiger Achse

Analog wird die Matrix für die Spieglung an der xz- oder yz-Ebene aufgebaut. Da der Benutzer die Möglichkeit hat, die „Spiegelebene“ an eine andere Position zu bewegen, wird wie bei der Spiegelung an einem beliebigen Punkt oder einer beliebigen Achse, zuerst eine Translation der Ebene in den Ursprung durchgeführt. Anschließend wird die Ebene so rotiert, dass ihre Normale auf der z-Achse liegt. Damit befindet sich die Ebene in der xy-Ebene des Koordinatensystems; die Spiegelung wird an dieser Ebene durchgeführt. Daraufhin wird die Ebene wieder an ihre ursprüngliche Position translatiert und die Rotation der Normalen auf die z-Achse rückgängig gemacht (siehe Abbildung 45).

Punktspiegelung Bei der Spiegelung an einem Punkt wird jedem Punkt P wird ein Bildpunkt P^* zugeordnet, so dass der Spiegelpunkt die Verbindungsstrecke $\overline{PP^*}$rechtwinklig halbiert. Eine Punktspiegelung (siehe Abbildung 47) kann durch drei Spiegelungen an zueinander senkrecht ste-

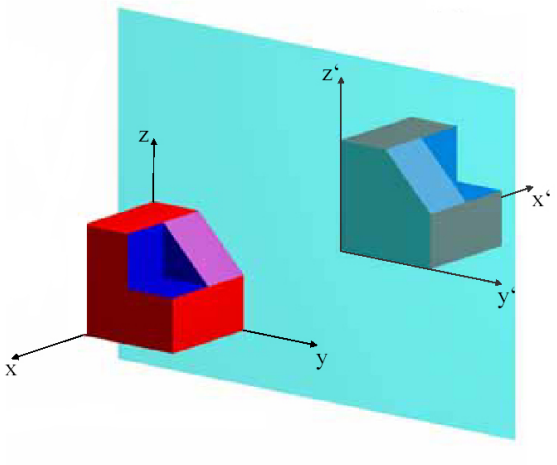

Abbildung 44: Ebenenspiegelung

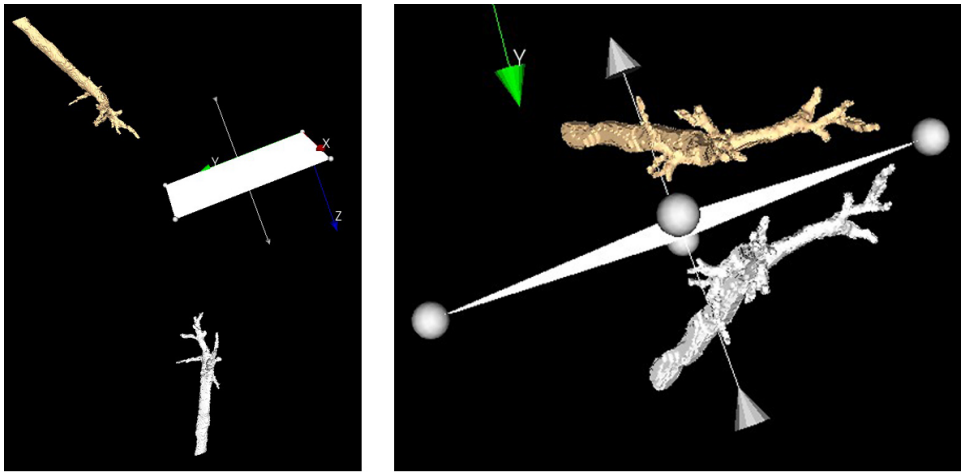

Abbildung 45: **Links:** Spiegelung an xy-Ebene **Rechts:** Spiegelung an beliebig gesetzter Ebene

henden Spiegelebenen realisiert werden. Die Matrix, mit der multipliziert werden muss, um eine Punktspiegelung zu erreichen, sieht wie folgt aus:

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{18}$$

Daraus ergibt sich für den gespiegelten Punkt P^*:

$$\begin{pmatrix} P_x^* \\ P_y^* \\ P_z^* \end{pmatrix} = \begin{pmatrix} -s_x & 0 & 0 \\ 0 & -s_y & 0 \\ 0 & 0 & -s_z \end{pmatrix} \bullet \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = \begin{pmatrix} -s_x P_x \\ -s_y P_y \\ -s_z P_z \end{pmatrix} \tag{19}$$

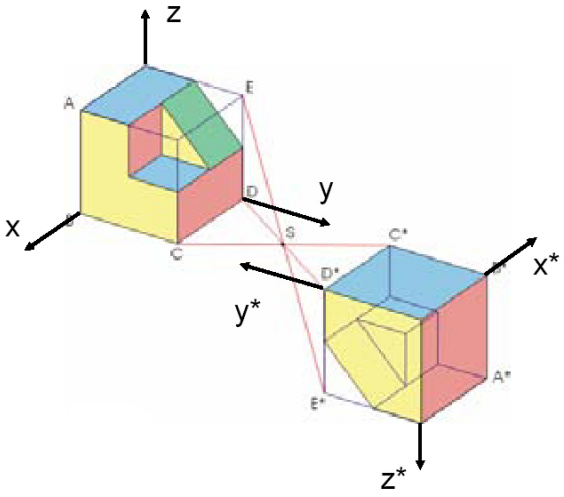

Abbildung 46: Punktspiegelung (Quelle: [19])

Um dem Benutzer zu ermöglichen, den „Spiegelpunkt“, nach Belieben zu positionieren, wird eine Matrix berechnet, die eine Translation, abhängig von der Position des „Spiegelpunktes“, in den Ursprung und zurück an die ursprüngliche Position darstellt. Diese Matritzen werden mit der oben erwähnten „Spiegelmatrix“ multipliziert und auf das zu spiegelnde Objekt angewendet.

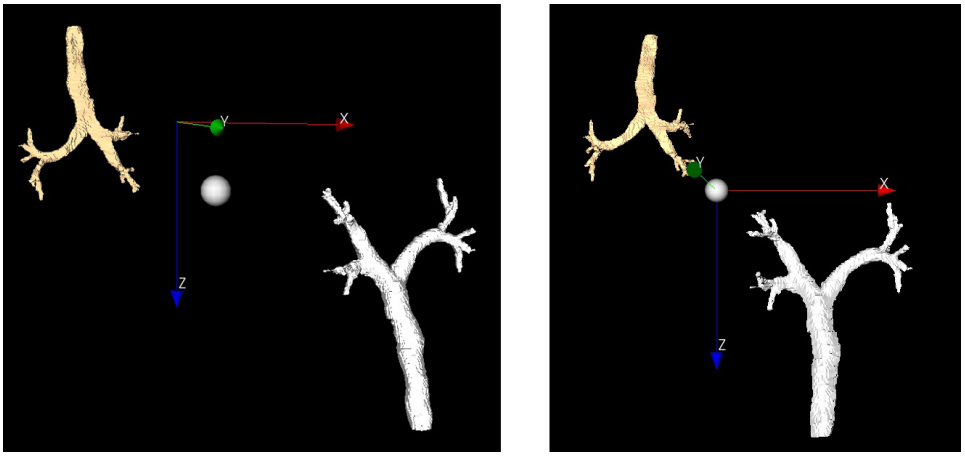

Abbildung 47: **Links:** Spiegelung an einem beliebig gesetzten Punkt **Rechts:** Punktspiegelung am Ursprung

4.2.6 Registrierung

Als Registrierung wird die Ausrichtung unterschiedlicher Datensätze an einem einheitlichen Koordinatensystem bezeichnet. Ihre Aufgabe besteht darin, eine Transformation zu finden, die einen Datensatz möglichst genau in einen anderen Datensatz überführt. Das Problem liegt darin, dass ein Zusammenhang zwischen den Punkten der beiden Datensätze nicht bekannt ist. Eine Lösung dieses Problems stellt der Iterative Closest Point Algorithmus von Besl und McKay dar.
Ein Registrierungsalgorithmus kann den Anwender unterstützen, Knochenteile anhand eines Referenzmodelles in die korrekte Position und Lage innerhalb der Skelettstruktur zu setzen.
In dieser Arbeit wird der Iterative Closest Point Algorithmus von VTK benutzt, der einen Modus zulässt, bei dem zusätzlich zu Rotation und Translation eine Skalierung berechnet wird. Im Ansatz von Besl und McKay wird lediglich die Klasse der rigiden Transformationen (Rotation und Translation) hinzugezogen. VTK stützt sich in den Berechnungen der optimalen Rotation, Translation und Skalierung auf die in [13] gemachten Lösungsvorschläge. Da K. P. Horn in [13] zur Bestimmung der Rotation eine auf Quaternionen basierende Lösung vorschlägt, befasst sich der nächste Abschnitt mit der Quaternionenmathematik. Der darauffolgende Abschnitt geht näher auf den Iterative Closest Point Algorithmus ein und erläutert die Bestimmung der optimalen Rotation, Translation und Skalierung, wie in [13] vorgestellt.

Quaternionen Ein Quaternion ist ein Vektor mit vier Komponenten, bestehend aus einem Skalar und einem dreidimensionalen Vektor. Allgemein wird ein Quaternion wie folgt geschrieben:

$$\dot{q} = \begin{pmatrix} w, & x, & y, & z \end{pmatrix} \text{ bzw. } \dot{q} = \begin{pmatrix} w, & \vec{v} \end{pmatrix} \text{ mit } \vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{20}$$

Quaternionen werden als Erweiterung der komplexen Zahlen angesehen. So besitzt das unten aufgeführte Quaternion $\dot{q}$ in q_0 einen realen Teil und in q_x, q_y und q_z drei imaginäre Teile

$$\dot{q} = q_0 + iq_x + jq_y + kq_z \tag{21}$$

Für *i*, *j* und *k* gilt

$$i^2 = j^2 = k^2 = -1 \tag{22}$$

$$ij = k, jk = i, ki = j \tag{23}$$

$$ji = -k, kj = -i, ik = -j \tag{24}$$

Addition von Quaternionen Die Addition von zwei Quaternionen $\dot{r} = r_0 + ir_x + jr_y + kr_z$ und $\dot{q} = q_0 + iq_x + jq_y + kq_z$ wird wie folgt ausgedrückt:

$$\dot{r} + \dot{q} = (r_0 + ir_x + jr_y + kr_z) + (q_0 + iq_x + jq_y + kq_z) \tag{25}$$
$$= (r_0 + q_0) + (r_x + q_x)\, i + (r_y + q_y)\, j + (r_z + q_z)\, k \tag{26}$$

Multiplikation von Quaternionen Die Multiplikation zweier Quaternionen $\dot{r} = r_0 + ir_x + jr_y + kr_z$ und $\dot{q} = q_0 + iq_x + jq_y + kq_z$ ergibt

$$\dot{r}\dot{q} = (r_0 q_0 - r_x q_x - r_y q_y - r_z q_z) \tag{27}$$
$$+ i\,(r_0 q_x + r_x q_0 + r_y q_z - r_z q_y) \tag{28}$$
$$+ j\,(r_0 q_y - r_x q_z + r_y q_0 - r_z q_x) \tag{29}$$
$$+ k\,(r_0 q_z + r_x q_y - r_y q_x - r_z q_0) \tag{30}$$

oder anders geschrieben

$$\dot{r}\dot{q} = \left(r, \quad \vec{v_0}\right)\left(q, \quad \vec{v_1}\right) = (rq - \vec{v_0} \cdot \vec{v_1}, r\vec{v_1} + q\vec{v_0} + \vec{v_0} \times \vec{v_1}) \tag{31}$$

Das Produkt zweier Quaternionen kann auch ausgedrückt werden als Produkt einer orthogonalen 4x4 Matrix und eines Vektors mit vier Komponenten:

$$\dot{r}\dot{q} = \begin{pmatrix} r_0 & -r_x & -r_y & -r_z \\ r_x & r_0 & -r_z & r_y \\ r_y & r_z & r_0 & -r_x \\ r_z & -r_y & r_x & r_0 \end{pmatrix} \dot{q} = \Re\dot{q} \tag{32}$$

Die Multiplikation von Quaternionen ist jedoch nicht kommutativ. Es gilt

$$\dot{p}\,(\dot{q}\dot{r}) = (\dot{p}\dot{q})\,\dot{r} = \dot{p}\dot{q}\dot{r} \tag{33}$$

und

$$\dot{p}\dot{q} \neq \dot{q}\dot{p} \tag{34}$$

Skalarprodukt von Quaternionen Das Skalarprodukt von $\dot{r}$ und $\dot{q}$ lautet:

$$\dot{r} \cdot \dot{q} = r_0 q_0 + r_x q_x + r_y q_y + r_z q_z \tag{35}$$

Das Quadrat der Länge eines Quaternions ist das Skalarprodukt dieses Quaternions mit sich selbst:

$$\|\dot{q}\|^2 = \dot{q} \cdot \dot{q} \tag{36}$$

Ein Quaternion der Länge 1 wird als Einheitsquaternion bezeichnet. Weiterhin gilt:

$$(\dot{q}\dot{p}) \cdot (\dot{q}\dot{r}) = (\dot{q} \cdot \dot{q})\,(\dot{p} \cdot \dot{r}) \tag{37}$$

Ist $\dot{q}$ ein Einheitsquaternion gilt $\dot{q} \cdot \dot{q} = 1$ und somit:

$$(\dot{q}\dot{p}) \cdot (\dot{q}\dot{r}) = \dot{p} \cdot \dot{r} \tag{38}$$

Inverse und Konjugierte eines Quaternions Die Konjugierte eines Quaternions negiert den imaginären Teil des Quaternions. Es folgt:

$$\dot{q}^* = q_0 - iq_x - jq_y - kq_z \tag{39}$$

Das Produkt eines Quaternions $\dot{q}$ mit seiner Konjugierten ergibt

$$\dot{q}\dot{q}^* = \left(q_0^2 + q_x^2 + q_y^2 + q_z^2\right) = \dot{q} \cdot \dot{q} \tag{40}$$

Die Inverse des Quaternions $\dot{q}$ ist

$$\dot{q}^{-1} = \left(1/\dot{q} \cdot \dot{q}\right) \dot{q}^* \tag{41}$$

Die Inverse eines Einheitsquaternions ist seine Konjugierte, da in diesem Fall der Term $1/\dot{q} \cdot \dot{q} = 1$ wird.

Darstellung von Rotationen mittels Einheitsquaternionen Ein Vektor kann als Quaternion mit einem Realteil gleich null interpretiert werden. Dieses Quaternion wird pures Quaternion oder auch rein imginäres Quaternion genannt. Ist der Vektor $\vec{r} = (x, y, z)^T$ gegeben, kann er als Quaternion

$$\dot{r} = 0 + ix + jy + kz \tag{42}$$

ausgedrückt werden.
Jede Rotation lässt sich durch einen Winkel und eine Achse darstellen, um die rotiert wird. Die Rotation um eine normierte Achse (Vektor) n um den Winkel θ wird durch das Quaternion

$$\dot{q} = \left(\cos\left(\theta/2\right)\right) + \sin\left(\theta/2\right) n_x + \sin\left(\theta/2\right) n_y + \sin\left(\theta/2\right) n_z \tag{43}$$

beziehungsweise

$$\dot{q} = \left(\cos\left(\theta/2\right)\right), \sin\left(\theta/2\right) n_x, \sin\left(\theta/2\right) n_y, \sin\left(\theta/2\right) n_z \tag{44}$$

repräsentiert. Es kann gezeigt werden, dass gilt:

$$\dot{r}^{'} = \dot{q}\dot{r}\dot{q}^* \tag{45}$$

$\dot{r}^{'}$ stellt den Vektor $\vec{r}$ dar, der bezüglich der normierten Rotationsachse n um den Winkel θ rotiert wurde. Da der Vektor $\vec{r}$ als Quaternion geschrieben einen Imaginärteil (oder Skalarteil) von null hat, muss der Imaginärteil des Vektors $\vec{r}^{'}$ ebenfalls null sein. Für den Realteil muss Folgendes gelten:

$$(E + \sin\theta D(\vec{n}) + (1 - \cos\theta)D^2(\vec{n}))\vec{r} \tag{46}$$

Dieser Term wird als Rodrigues-Formel (Drehtensor) bezeichnet.
Die Berechnung von $\dot{q}\dot{r}\dot{q}^*$ mit $\dot{q} = \left[\cos\left(\theta/2\right), \sin\left(\theta/2\right)\vec{n}\right]$ und $\dot{q}^* = \left[\cos\left(\theta/2\right), -\sin\left(\theta/2\right)\vec{n}\right]$ und $\dot{r} = [0, \vec{r}]$ ergibt:

$$\left[0, \left(E + \sin\theta D\left(\vec{n}\right)\vec{r} + \left(1 - \cos\theta\right) D^2\left(\vec{n}\right)\right)\vec{r}\right] \tag{47}$$

Dies soll in der folgenden Berechnung nachvollzogen werden. Zur besseren Lesbarkeit wird für $\cos(\theta/2)$ c und für $\sin(\theta/2)$ s geschrieben.

$$\begin{aligned}
\dot{q}\dot{r}\dot{q}^* &= [c, s\vec{n}]\,[0, \vec{r}]\,[c, -s\vec{n}] && (48)\\
&= [c, s\vec{n}]\,[0c - \vec{r}\cdot(-s\vec{n}),\, 0(-s\vec{n}) + c\vec{r} + \vec{r}\times(-s\vec{n})] && (49)\\
&= [c, s\vec{n}]\,[s(\vec{r}\cdot\vec{n}),\, c\vec{r} - s(\vec{r}\times\vec{n})] && (50)\\
&= [cs(\vec{r}\cdot\vec{n}) - sn\cdot(c\vec{r} - s(\vec{r}\times\vec{n})), && (51)\\
&\quad c(c\vec{r} - s(\vec{r}\times\vec{n})) + s(\vec{r}\cdot\vec{n})s\vec{n} + s\vec{n}\times(c\vec{r} - s(\vec{r}\times\vec{n}))] \\
&= [cs(\vec{r}\cdot\vec{n}) - cs(\vec{r}\cdot\vec{n}) + s^2\vec{n}\cdot(\vec{r}\times\vec{n}), && (52)\\
&\quad s^2\vec{n}(\vec{r}\cdot\vec{n}) + c^2\vec{r} - cs(\vec{r}\times\vec{n}) + (s\vec{n}\times(c\vec{r} - s(\vec{r}\times\vec{n})))] \\
&= \left[0, s^2\vec{n}(\vec{r}\cdot\vec{n}) + c^2\vec{r} - cs(\vec{r}\times\vec{n}) + sc(\vec{n}\times\vec{r}) + s^2\vec{n}\times(\vec{r}\times\vec{n})\right] && (53)\\
&= \left[0, s^2\vec{n}(\vec{r}\cdot\vec{n}) + c^2\vec{r} + 2cs(\vec{n}\times\vec{r}) - s^2\vec{n}\times(\vec{r}\times\vec{n})\right] && (54)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + c^2\vec{r} + s^2\vec{n}(\vec{r}\cdot\vec{n}) - s^2\vec{n}\times(\vec{r}\times\vec{n})\right] && (55)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + c^2\vec{r} + s^2\vec{n}(\vec{r}\cdot\vec{n}) - s^2((\vec{n}\cdot\vec{n})\vec{r} - (\vec{n}\cdot\vec{r})\vec{n})\right] && (56)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + c^2\vec{r} + 2s^2\vec{n}(\vec{r}\cdot\vec{n}) - s^2(\vec{n}\cdot\vec{n})\vec{r}\right] && (57)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + (1-\cos\theta)\vec{n}(\vec{r}\cdot\vec{n}) + c^2\vec{r} - s^2\vec{r}\right] && (58)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + (1-\cos\theta)\vec{n}(\vec{r}\cdot\vec{n}) + \cos\theta\vec{r}\right] && (59)\\
&= \left[0, \sin\theta(\vec{n}\times\vec{r}) + (1-\cos\theta)(\vec{n}(\vec{r}\cdot\vec{n}) - \vec{r} + \vec{r}) + \cos\theta\vec{r}\right] && (60)\\
&= \left[0, \sin\theta D(\vec{n})\vec{r} + (1-\cos\theta)D^2(\vec{n})\vec{r} + \vec{r}\right] && (61)\\
&= \left[0, \vec{r} + \sin\theta D(\vec{n})\vec{r} + (1-\cos\theta)D^2(\vec{n})\vec{r}\right] && (62)\\
&= \left[0, \left(E + \sin\theta D(\vec{n}) + (1-\cos\theta)D^2(\vec{n})\right)\vec{r}\right] && (63)
\end{aligned}$$

Anmerkungen:
zu (52) auf (53): $\vec{n}\cdot(\vec{r}\times\vec{n}) = \vec{n}\cdot(\vec{n}\times\vec{r}) = 0$
zu (53) auf (54): $(\vec{r}\times\vec{n}) = -(\vec{n}\times\vec{r})$
zu (54) auf (55): $2\sin\cos\theta = \sin(2\theta)$
zu (55) auf (56): $\vec{n}\times(\vec{r}\times\vec{n}) = (\vec{n}\cdot\vec{n})\vec{r} - (\vec{n}\cdot\vec{r})\vec{n}$
zu (57) auf (58): $2\sin^2\theta = 1 - \cos 2\theta$
zu (57) auf (58): da $\vec{n}$ normiert, folgt: $\vec{n}\cdot\vec{n} = 1$
zu (58) auf (59): $\cos^2\theta - \sin^2\theta = \cos(2\theta)$
zu (59) auf (60): erweitern mit: $+\vec{r} - \vec{r}$
zu (60) auf (61): $D(\vec{n})\vec{r}$ bezeichnet das Kreuzprodukt $\vec{n}\times\vec{r}$
zu (60) auf (61): $D^2(\vec{n})\vec{r} = D(\vec{n})D(\vec{n}) = \vec{n}\times(\vec{n}\times\vec{r}) = (\vec{n}\cdot\vec{r})\vec{n} - (\vec{n}\cdot\vec{n})\vec{r} = (\vec{n}\cdot\vec{r})\vec{n} - \vec{r}$
zu (61) auf (62): Ausklammern von $\vec{r}$
zu (63): E bezeichnet die 3x3 Einheitsmatrix

Das Ergebnis ist ein pures Quaternion, dessen Vektorteil den Drehtensor darstellt. Einheitsquaternionen können somit zur Darstellung von Rotationen herangezogen werden.

Der Iterative Closest Point Algorithmus (ICP) Der Iterative Closest Point-Algorithmus durchläuft drei Schritte um eine Transformation zu finden, die eine Punktmenge *P* in eine Punktmenge *M* überführt.

1. Suche zu jedem Punkt aus *P* den nächsten Nachbarn in *M*.
2. Berechne die Transformation, die den quadratischen Fehler minimiert.
3. Wende die Transformation auf P an und aktualisiere den quadratischen Fehler.

Diese drei Schritte werden solange durchgeführt, bis der quadratische Fehler einen Schwellwert unterschreitet oder eine andere Abbruchbedingung erfüllt ist. Im weiteren werden die Punktmengen P und M durch $P = \{\vec{p}_i\}\ {}^{N_P}_{i=1}$ und $M = \{\vec{p}_i\}\ {}^{N_M}_{i=1}$ beschrieben, wobei angenommen wird, dass die Anzahl der Punkte in *P* und *M* gleich sind und somit $N_M = N_P$ gilt. Zusätzlich wird angenommen, dass zu jedem Punkt in *P* ein korrespondierender Punkt in *M* existiert. Um die optimale Transformation von *P* auf *M* zu finden, benötigt der ICP-Algorithmus zwei Verfahren: eines, das zu einem gegebenen Punkt aus *P* den nächsten Nachbarn in *M* findet und ein anderes welches die Transformation bestimmt, die benötigt wird, um *P* auf *M* abzubilden.

Bestimmung des nächsten Nachbarn Um den nächsten Nachbarn von *P* in *M* zu bestimmen wird der euklidische Abstand eines gegebenen Punktes aus *P* zu allen Punkten aus *M* berechnet. Der Punkt mit dem geringsten Abstand ist der nächste Nachbar. Der euklidische Abstand zwischen zwei Punkten $\vec{r_1}\begin{pmatrix} x_1 & y_1 & z_1 \end{pmatrix}$ und $\vec{r_2}\begin{pmatrix} x_2 & y_2 & z_2 \end{pmatrix}$ lautet:

$$d\left(\vec{r}_1, \vec{r}_2\right) = |\vec{r}_1 - \vec{r}_2| = \sqrt{\left(x_2 - x_1\right)^2 + \left(y_2 - y_1\right)^2 + \left(z_2 - z_1\right)^2} \tag{64}$$

Die euklidische Distanz zwischen einem Punkt aus P und der Punktmenge M wird mit

$$d\left(p_{P,i}, M\right) =_{i \in \{1 \ldots N_M\}}^{min} d\left(p_{P,i}, p_{M,i}\right) \tag{65}$$

beschrieben. Für jeden Punkt in P durchgeführt, erhält man eine Menge von Nachbarpunkten, die Teilmenge aus *M* ist.

Bestimmung der Transformation Um die Transformation zu finden, die die Punktmenge P in die Punktmenge M überführt, muss eine Fehlerfunktion minimiert werden. Diese Funktion legt fest, wie groß der Fehler zwischen zwei Punkten der beiden Punktmengen ist.
Beide Punktmengen besitzen die gleiche Anzahl an Punkten. Die Koordinaten der Punkte in P werden mit $\{r_{P,i}\}$, die Koordinaten der Punkte in M

mit $\{r_{M,i}\}$ bezeichnet, mit $i \in 1, \ldots, n$, wobei n die Anzahl der Punkte ist. Gesucht wird eine Transformation der Form:

$$\vec{r}_M = sR\left(\vec{r}_P\right) + \vec{t} \tag{66}$$

$R\left(\vec{r}_P\right)$ bezeichnet den rotierten Vektor $\vec{r}_P$, $\vec{t}$ eine Translation und s eine Skalierung. Im Allgemeinen ist es nicht möglich ist, eine Transformation zu finden, die die Gleichung für jeden Punkt erfüllt. Deshalb wird sich ein Fehler abzeichnen:

$$\vec{e}_i = \vec{r}_{M,i} - sR\left(\vec{r}_{P,i}\right) - \vec{t} \tag{67}$$

Der quadratische Fehler

$$\sum_{i=1}^{n} \|\vec{e}_i\|^2 \tag{68}$$

soll minimiert werden.
K.P. Horn stellt fest, dass es nützlich ist, alle Punkte in Bezug zum Schwerpunkt der zugehörigen Punktmenge zu setzen:

$$\mu_P = \frac{1}{n} \sum_{i=1}^{n} \vec{r}_{P,i} \tag{69}$$

$$\mu_M = \frac{1}{n} \sum_{i=1}^{n} \vec{r}_{M,i} \tag{70}$$

Die neuen Koordinaten werden als

$$\vec{r}^{\,'}_{P,i} = \vec{r}_{P,i} - \mu_P \tag{71}$$

und

$$\vec{r}^{\,'}_{M,i} = \vec{r}_{M,i} - \mu_M \tag{72}$$

geschrieben. Der Fehlerterm wird umgeschrieben in

$$\vec{e}_i = \vec{r}^{\,'}_{M,i} - sR\left(\vec{r}^{\,'}_{P,i}\right) - \vec{t}^{\,\prime} \tag{73}$$

mit

$$\vec{t}^{\,\prime} = \vec{t}^{\,\prime} - \mu_M + sR(\mu_P) \tag{74}$$

Rotation Um die optimale Rotation zu finden, muss der Term

$$\sum_{i=1}^{n} \vec{r}^{\,'}_{M,i} \cdot R\left(\vec{r}^{\,'}_{P,i}\right) \tag{75}$$

maximiert werden. (Wird der Winkel zwischen dem ursprünglichen Punkt (Vektor) und dem rotierten Punkt (Vektor) kleiner, liegen ursprünglicher

und rotierter Punkt näher beieinander und das Skalarprodukt wird größer). Der Ansatz von K.P. Horn stützt sich auf Einheitsquaternionen. Im Folgenden wird beschrieben, wie das Einheitsquaternion $\dot{q}$ berechnet wird. In dem Term

$$\sum_{i=1}^{n} \left(\dot{q} \dot{r}'_{P,i} \dot{q}^* \right) \cdot \dot{r}'_{M,i} \tag{76}$$

wurden die Vektoren in pure Quaternionen umgewandelt und $R\left(\vec{r}'_{P,i}\right)$ als Multiplikation von Einheitsquaternionen ausgedrückt. Die Umformung des obigen Ausdruck ergibt:

$$\sum_{i=1}^{n} \left(\dot{q} \dot{r}'_{P,i} \right) \cdot \left(\dot{r}'_{M,i} \dot{q} \right) \tag{77}$$

Die Multiplikation zweier Quaternionen kann als Multiplikation einer 4x4-Matrix mit einem vierkomponentigen Vektor (Quaternion) dargestellt werden. Für $\left(\dot{q} \dot{r}'_{P,i} \right)$ ergibt sich mit $\dot{r}'_{P,i} = (0, x'_{P,i}, y'_{P,i}, z'_{P,i})$:

$$\dot{q} \dot{r}'_{P,i} = \begin{bmatrix} 0 & -x'_{P,i} & -y'_{P,i} & -z'_{P,i} \\ x'_{P,i} & 0 & z'_{P,i} & -y'_{P,i} \\ y'_{P,i} & -z'_{P,i} & 0 & x'_{P,i} \\ z'_{P,i} & y'_{P,i} & -x'_{P,i} & 0 \end{bmatrix} \dot{q} = \bar{\Re}_{P,i} \dot{q} \tag{78}$$

und für $\left(\dot{r}'_{M,i} \dot{q} \right)$ mit $\dot{r}'_{M,i} = (0, x'_{M,i}, y'_{M,i}, z'_{M,i})$:

$$\dot{r}'_{M,i} \dot{q} = \begin{bmatrix} 0 & -x'_{M,i} & -y'_{M,i} & -z'_{M,i} \\ x'_{M,i} & 0 & z'_{M,i} & -y'_{M,i} \\ y'_{M,i} & -z'_{M,i} & 0 & x'_{M,i} \\ z'_{M,i} & y'_{M,i} & -x'_{M,i} & 0 \end{bmatrix} \dot{q} = \Re_{M,i} \dot{q} \tag{79}$$

Eingesetzt in Gleichung (77) und umgeformt ergibt sich:

$$\sum_{i=1}^{n} \left(\dot{q} \dot{r}'_{P,i} \right) \cdot \left(\dot{r}'_{M,i} \dot{q} \right) = \sum_{i=1}^{n} \left(\bar{\Re}_{P,i} \dot{q} \right) \cdot \left(\Re_{M,i} \dot{q} \right) \tag{80}$$

$$= \sum_{i=1}^{n} \dot{q}^T \bar{\Re}_{P,i}^T \Re_{M,i} \dot{q} \tag{81}$$

$$= \dot{q}^T \left(\sum_{i=1}^{n} \bar{\Re}_{P,i}^T \Re_{M,i} \right) \dot{q} \tag{82}$$

$$= \dot{q}^T \left(\sum_{i=1}^{n} N_i \right) \dot{q} = \dot{q}^T N \dot{q} \tag{83}$$

Es gilt $N = \sum_{i=1}^{n} N_i$ und $N_i = \bar{\Re}_{P,i}^T \Re_{M,i}$. Um das Einheitsquaternion zu finden, das $\dot{q}^T N \dot{q}$ maximiert, führt K.P. Horn die Matrix M ein. Die Elemente dieser Matrix sind Summen der Produkte der Koordinaten in P mit den Koordinaten in M.

$$M = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{bmatrix} \tag{84}$$

wobei zum Beispiel gilt:

$$S_{xx} = \sum_{i=1}^{n} x'_{P,i} x'_{M,i} \tag{85}$$

$$S_{yx} = \sum_{i=1}^{n} y'_{P,i} x'_{M,i} \tag{86}$$

Die folgende Berechnung zeigt, dass sich die Matrix N aus Elementen der Matrix M zusammensetzen lässt:

$$\Re_{P,i}^T = \begin{bmatrix} 0 & x'_{P,i} & y'_{P,i} & z'_{P,i} \\ -x'_{P,i} & 0 & -z'_{P,i} & y'_{P,i} \\ -y'_{P,i} & z'_{P,i} & 0 & -x'_{P,i} \\ -z'_{P,i} & -y'_{P,i} & x'_{P,i} & 0 \end{bmatrix} \tag{87}$$

und

$$\Re_{M,i} = \begin{bmatrix} 0 & -x'_{M,i} & -y'_{M,i} & -z'_{M,i} \\ x'_{M,i} & 0 & -z'_{M,i} & -y'_{M,i} \\ y'_{M,i} & z'_{M,i} & 0 & -x'_{M,i} \\ z'_{M,i} & -y'_{M,i} & x'_{M,i} & 0 \end{bmatrix} \tag{88}$$

$$N_i = \Re_{P,i}^T \bullet \Re_{M,i} = \begin{bmatrix} 0 & x'_{P,i} & y'_{P,i} & z'_{P,i} \\ -x'_{P,i} & 0 & -z'_{P,i} & y'_{P,i} \\ -y'_{P,i} & z'_{P,i} & 0 & -x'_{P,i} \\ -z'_{P,i} & -y'_{P,i} & x'_{P,i} & 0 \end{bmatrix} \begin{bmatrix} 0 & -x'_{M,i} & -y'_{M,i} & -z'_{M,i} \\ x'_{M,i} & 0 & -z'_{M,i} & -y'_{M,i} \\ y'_{M,i} & z'_{M,i} & 0 & -x'_{M,i} \\ z'_{M,i} & -y'_{M,i} & x'_{M,i} & 0 \end{bmatrix} \tag{89}$$

$$= \begin{bmatrix} x'_{P,i}x'_{M,i}+y'_{P,i}y'_{M,i}+z'_{P,i}z'_{M,i} & \dots & \dots & x'_{P,i}y'_{M,i}-y'_{P,i}x'_{M,i} \\ y'_{P,i}z'_{M,i}-z'_{P,i}y'_{M,i} & \dots & \dots & x'_{P,i}z'_{M,i}+z'_{P,i}x'_{M,i} \\ z'_{P,i}x'_{M,i}-x'_{P,i}z'_{M,i} & \dots & \dots & y'_{P,i}z'_{M,i}+z'_{P,i}y'_{M,i} \\ -y'_{P,i}x'_{M,i}+x'_{P,i}y'_{M,i} & \dots & \dots & z'_{P,i}z'_{M,i}-y'_{P,i}y'_{M,i}-x'_{P,i}x'_{M,i} \end{bmatrix} \tag{90}$$

Daraus ergibt sich:

$$N = \begin{bmatrix} (S_{xx}+S_{yy}+S_{zz}) & S_{yz}-S_{zy} & S_{zx}-S_{xz} & S_{xy}-S_{yx} \\ S_{yz}-S_{zy} & (S_{xx}-S_{yy}-S_{zz}) & S_{xy}+S_{yx} & S_{zx}+S_{xz} \\ S_{zx}-S_{xz} & S_{xy}+S_{yx} & (-S_{xx}+S_{yy}-S_{zz}) & S_{yz}+S_{zy} \\ S_{xy}-S_{yx} & S_{zx}+S_{xz} & S_{yz}+S_{zy} & (-S_{xx}-S_{yy}+S_{zz}) \end{bmatrix} \tag{91}$$

Horn zeigt, dass das Einheitsquaternion, welches den Term $\dot{q}^T N \dot{q}$ maximiert, der Eigenvektor ist, der zum größten Eigenwert der Matrix N gehört. Die symmetrische 4x4 Matrix N hat vier Eigenwerte $\lambda_1, \ldots, \lambda_4$. Zu diesen vier Eigenwerten kann eine Menge von orthogonalen Einheitseigenvektoren $\dot{e}_1, \ldots, \dot{e}_4$ gefunden werden, so dass gilt:

$$N\dot{e}_i = \lambda \dot{e}_i \text{ für } i = 1, 2 \ldots, 4 \tag{92}$$

Der Eigenvektor ist ein Vektor, der durch eine Abbildung seine Richtung nicht ändert, sondern in ein Vielfaches (hier Lambda) übergeht. Da die Eigenvektoren einen vierdimensionalen Raum aufspannen, kann ein beliebiges Quaternion $\dot{q}$ als Linearkombination geschrieben werden

$$\dot{q} = \alpha_1 \dot{e}_1 + \alpha_2 \dot{e}_2 + \alpha_3 \dot{e}_3 + \alpha_4 \dot{e}_4 \tag{93}$$

Weiterhin gilt:

$$\dot{q} \cdot \dot{q} = \alpha_1^2 + \alpha_2^2 + \alpha_3^2 + \alpha_4^2 \tag{94}$$

Diese Gleichung muss 1 sein, da $\dot{q}$ ein Einheitsquaternion ist. Da $\dot{e}_1, \ldots, \dot{e}_4$ Eigenvektoren von N sind gilt:

$$N\dot{q} = \alpha_1 \lambda_1 \dot{e}_1 + \alpha_2 \lambda_3 \dot{e}_3 + \alpha_3 \lambda_3 \dot{e}_3 + \alpha_4 \lambda_4 \dot{e}_4 \tag{95}$$

Horn formt den zu maximierenden Term $\dot{q}^T N \dot{q}$ zu $\dot{q} \cdot (N\dot{q})$ um, setzt das Ergebnis von oben ein und erhält

$$\dot{q}^T N \dot{q} = \dot{q} \cdot (N\dot{q}) = \alpha_1^2 \lambda_1 + \alpha_2^2 \lambda_2 + \alpha_3^2 \lambda_3 + \alpha_4^2 \lambda_4 \tag{96}$$

denn

$$\begin{aligned}
\dot{q} \cdot (N\dot{q}) &= (\alpha_1 \dot{e}_1 + \alpha_2 \dot{e}_2 + \alpha_3 \dot{e}_3 + \alpha_4 \dot{e}_4) \cdot (\alpha_1 \dot{e}_1 \lambda_1 + \alpha_2 \dot{e}_2 \lambda_2 + \alpha_3 \dot{e}_3 \lambda_3 + \alpha_4 \dot{e}_4 \lambda_4) \\
&= \alpha_1^2 \dot{e}_1^2 \lambda_1 + \alpha_2^2 \dot{e}_2^2 \lambda_2 + \alpha_3^2 \dot{e}_3^2 \lambda_3 + \alpha_4^2 \dot{e}_4^2 \lambda_4 \\
&= \alpha_1^2 \lambda_1 + \alpha_2^2 \lambda_2 + \alpha_3^2 \lambda_3 + \alpha_4^2 \lambda_4
\end{aligned}$$

da $\dot{e}_i^2 = 1$. Anschließend sortiert Horn die Eigenwerte:

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \lambda_4 \tag{97}$$

Der Term $\alpha_1^2 \lambda_1 + \alpha_2^2 \lambda_2 + \alpha_3^2 \lambda_3 + \alpha_4^2 \lambda_4$ kann nicht größer werden als der größte positive Eigenwert:

$$\dot{q}^T N \dot{q} \leq \alpha_1^2 \lambda_1 + \alpha_2^2 \lambda_2 + \alpha_3^2 \lambda_3 + \alpha_4^2 \lambda_4 = \lambda_1 \tag{98}$$

Wird zum Beispiel $\alpha_1 = 1$ und $\alpha_2 = \alpha_3 = \alpha_4 = 0$ gewählt, dann ist $\dot{q} = \dot{e}_1$. Horn schließt, dass der zum größten positiven Eigenwert gehörende Einheitseigenvektor den Term $\dot{q}^T N \dot{q}$ maximiert. Das Quaternion, das die Rotation repräsentiert, ist ein Einheitsvektor, der in die gleiche Richtung

zeigt, wie der gefundene Einheitseigenvektor. Die Eigenwerte λ_i der Matrix N sind die Nullstellen der Gleichung

$$\det(N - \lambda I) = 0 \tag{99}$$

$$\det(N - \lambda I) = \begin{pmatrix} n_{11} - \lambda & n_{12} & \dots n_{1n} \\ n_{21} & n_{22} - \lambda & \dots n_{2n} \\ \vdots & & \ddots \\ n_{n1} & \dots & n_{nn} - \lambda \end{pmatrix} = 0 \tag{100}$$

wobei I die 4x4 Einheitsmatrix darstellt. Um den Eigenvektor $\vec{e}_n$ zu dem größten Eigenwert λ_n zu erhalten, muss

$$(N - \lambda_n I)\vec{e}_n = 0 \tag{101}$$

gelöst werden. Damit wurde der gesuchte Eigenvektor gefunden, welcher das Einheitsquaternion darstellt, das den Term $\dot{q}^T N \dot{q}$ maximiert und somit die Rotation repräsentiert.

Translation Wird weiterhin der Fehlerterm

$$\vec{e}_i = \vec{r}'_{M,i} - sR\left(\vec{r}'_{P,i}\right) - \vec{t}' \tag{102}$$

betrachtet, folgt daraus der quadratische Fehler

$$\sum_{i=1}^{n} \left\| \vec{r}'_{M,i} - sR\left(\vec{r}'_{P,i}\right) - \vec{t}' \right\|^2 \tag{103}$$

der zu minimieren ist. Durch Umformungen lässt sich der Term schreiben als:

$$\sum_{i=1}^{n} ||\vec{r}'_{M,i} - sR\left(\vec{r}'_{P,i}\right)||^2 - 2t \sum_{i=1}^{n} \left[\vec{r}'_{M,i} - sR\left(\vec{r}'_{P,i}\right)\right] + n \left\|\vec{t}'\right\|^2 \tag{104}$$

Der Fehler wird mit $\vec{t}' = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$ minimiert, da der erste Ausdruck unabhängig von $\vec{t}'$ ist, der letzte Term nicht negativ werden kann und der mittlere Ausdruck des Terms aufgrund:

$$\sum_{i=1}^{n} \vec{r}'_{M,i} = 0 \qquad\qquad \sum_{i=1}^{n} \vec{r}'_{P,i} = 0 \tag{105}$$

null ist. Wird $\vec{t}' = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$ in Gleichung (74) eingesetzt, folgt:

$$\vec{t}' = \mu_M - sR(\mu_P) \tag{106}$$

Somit wird die Translation als Differenz zwischen dem Schwerpunkt von M und dem rotierten und skalierten Schwerpunkt von P beschrieben.

Skalierung Da $\vec{t}' = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$ ist, wird der quadratische Fehler als

$$\sum_{i=1}^{n} ||\vec{r}'_{M,i} - sR\left(\vec{r}'_{P,i}\right)||^2 \tag{107}$$

geschrieben, unter Beachtung, dass gilt:

$$||R(\vec{r}'_{P,i})||^2 = ||\vec{r}'_{M,i}||^2 \tag{108}$$

Durch Umformungen ergibt sich:

$$\sum_{i=1}^{n} ||\vec{r}'_{M,i}||^2 - 2s\sum_{i=1}^{n} \vec{r}'_{M,i} \cdot R(\vec{r}'_{P,i}) + s^2 \sum_{i=1}^{n} ||\vec{r}'_{P,i}||^2 \tag{109}$$

K.P. Horn ersetzt $\sum_{i=1}^{n} ||\vec{r}'_{M,i}||^2$ durch S_M, $\sum_{i=1}^{n} ||\vec{r}'_{P,i}||^2$ durch S_P und $\sum_{i=1}^{n} \vec{r}'_{M,i} \cdot R(\vec{r}'_{P,i})$ durch D und erhält

$$S_M - 2sD + s^2 S_P \tag{110}$$

Durch quadratische Ergänzung entsteht:

$$\left(s\sqrt{S_P} - \frac{D}{\sqrt{S_P}}\right)^2 + \frac{\left(S_M S_P - D^2\right)}{S_P} \tag{111}$$

Um diesen Term in Bezug auf s zu minimieren, muss der erste Term null sein, beziehungsweise gelten, dass

$$s = \frac{D}{S_P} \tag{112}$$

$$s = \frac{\sum_{i=1}^{n} \vec{r}'_{M,i} \cdot R(\vec{r}'_{P,i})}{\sum_{i=1}^{n} ||\vec{r}'_{P,i}||^2} \tag{113}$$

Horn weist darauf hin, dass die Symmetrie verloren geht, wenn statt der besten Anpassung für die Transformation

$$\vec{r}_M = sR\left(\vec{r}_P\right) + \vec{t} \tag{114}$$

die beste Anpassung für die inverse Transformation

$$\vec{r}_P = s_{inv} R_{inv}\left(\vec{r}_M\right) + \vec{t}_{inv} \tag{115}$$

mit

$$s_{inv} = \frac{1}{s} \qquad R_{inv} = R^{-1} \qquad \vec{t}_{inv} = -\frac{1}{s} R^{-1}(\vec{t}) \tag{116}$$

gesucht wird. Anstelle des erwarteten s_{inv} ergibt sich:

$$s_{inv} = \sum_{i=1}^{n} \vec{r}'_{P,i} \cdot R_{inv}(\vec{r}'_{M,i}) / \sum_{i=1}^{n} ||\vec{r}'_{M,i}||^2 \tag{117}$$

Um die Symmetrie zu wahren, schlägt K.P. Horn als beste Lösung zu Berechnung der Skalierung vor, den symmetrischen Ausdruck

$$\vec{e}_i = \frac{1}{\sqrt{s}} \vec{r}'_{M,i} - \sqrt{s} R \left(\vec{r}'_{P,i} \right) \tag{118}$$

für den Fehlerterm zu benutzen. Daraus folgt für den quadratischen Fehler:

$$\frac{1}{s} S_M - 2D + s S_P \tag{119}$$

Durch quadratische Ergänzung folgt:

$$\left(\sqrt{s} S_P - \frac{1}{\sqrt{s}} S_M \right) + 2 \left(S_P S_M - D \right) \tag{120}$$

Wird der erste Ausdruck null, beziehungsweise gilt, dass

$$S = S_M / S_P, \tag{121}$$

ist der Term in Bezug auf s minimiert.
Da der symmetrischen Ausdruck des Fehlerterms mit $\sqrt{s}$ multipliziert werden muss, um zu dem ursprünglichen Fehlerterm zu gelangen, gilt für die Skalierung s:

$$s = \left(\frac{\sum_{i=1}^{n} ||\vec{r}'_{M,i}||^2}{\sum_{i=1}^{n} ||\vec{r}'_{P,i}||^2} \right)^{\frac{1}{2}} \tag{122}$$

Der ICP-Algorithmus in der Anwendung Die Klasse `vtkIterativeClosestPointTransform` bildet zusammen mit `vtkLandmarkTransform` die Grundlage für die Implementierung des Iterative Closest Point Algorithmus in VTK.
Der Benutzer hat die Möglichkeit Modell (Target) und Szene (Source), die aufeinander abgebildet werden sollen, aus je einer Combo-Box auszuwählen. Das Modell wird rot und leicht transparent dargestellt, die Szene dagegen opak und weiß. Über einen Slider kann die Anzahl der Iterationen, die der Algorithmus durchlaufen soll, festgelegt werden. Desweiteren bietet VTK zwei Möglichkeiten, das Abstandsmaß für die Berechnung der „closest points“ zu bestimmen: In der ersten Methode wird der mittlere Abstand als Wurzel des Durchschnitts der aufsummierten Quadrate der „closest point“-Abstände definiert. Die zweite Methode berechnet den

mittleren Abstand aus der Summe der absoluten Werte der „closest point“-Abstände. Der Abstand zwischen zwei Punkten wird mit der Funktion `Distance2BetweenPoints` berechnet. Für die Punkte (x_1, y_1, z_1) und (x_2, y_2, z_2) wird der Abstand d wie folgt berechnet:

$$d = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 \tag{123}$$

In der Combobox *Mean Distance Mode* kann der Benutzer zwischen den beiden Modi auswählen. Per „default“ wird der Algorithmus gezwungen zwischen zwei Iterationen den mittleren Abstand der „closest points“ mit dem ausgewählten Modus (RMS Mode oder AbsoluteValue) zu überprüfen. Ist der aktuelle Abstand kleiner oder gleich dem Wert in dem Feld *Maximum Mean Distance* bricht der Algorithmus ab. Weiterhin besteht mit dem Markieren der Checkbox *Start By Matching Centroids* die Möglichkeit, den Algorithmus mit dem Matchen der Schwerpunkte von Modell und Szene zu beginnen. Für Datensets mit hoher Punktemenge ist es nicht notwendig alle Punkte für die Bestimmung der Transformation heranzuziehen. Mit dem Wert in dem Feld *Maximum Number of Landmarks* kann der Benutzer festlegen, wie viele Punkte der Algorithmus für die Berechnungen nutzen soll. Zur Bestimmung der Transformation kann der Benutzer zwischen drei Modi in der Combobox *Constrain* auwählen: *Rigid Body*, *Similarity* und *Affine*. *Rigid Body* berechnet, wie im ursprünglichen Algorithmus von Besl und McKay vorgestellt, lediglich eine Translation und Rotation zur Registrierung der beiden Datensätze, während *Similarity* zusätzlich eine uniforme Skalierung einfließen lässt. Der Modus *Affine* beinhaltet eine Scherung. Eine Übersicht der beschriebenen Parameter, die für die Registrierung eingestellt werden können, bietet Abbildung 48.

Um zu visualisieren wie groß die Abstände zwischen den nächsten Punk-

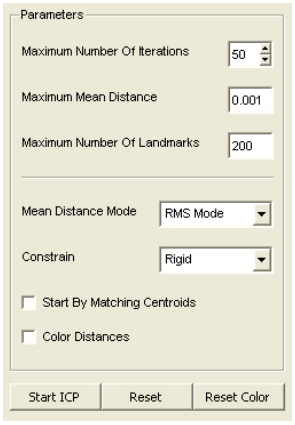

Abbildung 48: Auswahl der Parameter für die Registrierung

ten von Modell und Szene sind, hat der Benutzer die Möglichkeit die Checkbox *Color Distances* vor dem Starten des Algorithmus anzuwählen. Dies

bewirkt, dass nach dem Beenden des Alorithmus für alle Punkte der Szene die Distanz zum nächsten Nachbarn im Modell berechnet wird. Realisiert wird dies mit der Klasse `vtkPointLocator`. Die Methode `FindClosestPoint` dieser Klasse findet zu einer gegebenen Position x den nächsten Punkt und gibt die id dieses Punktes zurück. Für diese beiden Punkte wird die Distanz mit der oben beschriebenen Methode `Distance2BetweenPoints` berechnet, die Wurzel daraus gezogen und als Skalar in einem Array abgelegt. Anschließend werden die Skalarwerte den Punkten der Source zugeordnet. VTK bietet die Möglichkeit des *color mapping*. Der Mapper eines Actors kann mittels `ScalarVisibilityOn` angewiesen werden, die Skalarwerte, die den Punkten des Actors zugeordnet wurden, darzustellen. Durch eine *lookup table* werden den Skalarwerten Farben zugeordnet. Diese Farben werden während des Rendering-Prozesses auf die Punkte des Objektes angewendet. Das Ergebnis dieses Prozesses ist die farbige Darstellung der Distanzen zwischen den nächsten Nachbarn von Source und Target. Große Distanzen werden blau und kleine Distanzen rot dargestellt. Mittels einer *Skalarbar* wird die Zuordung der Farben zu der Größe der Distanzen verdeutlicht (siehe Abbildung 49). Je höher die An-

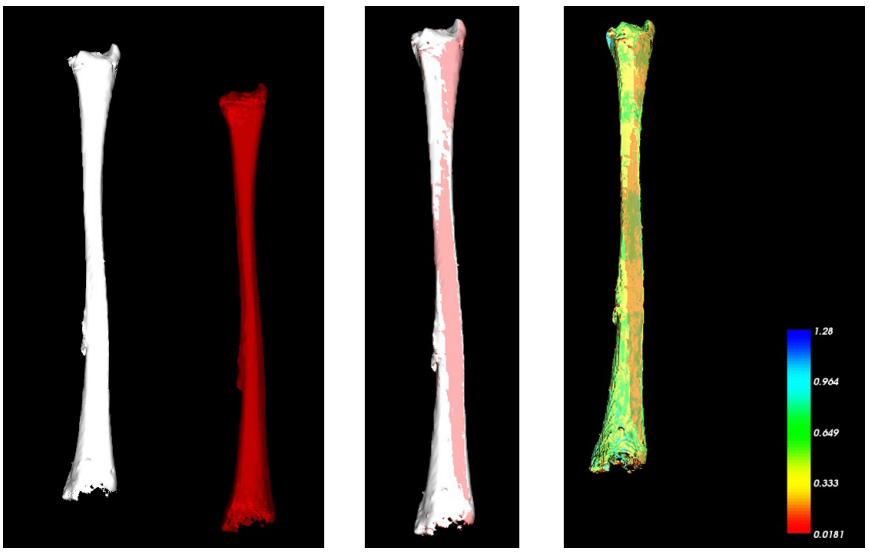

Abbildung 49: **Links:** Vor der Registrierung **Mitte:** Nach der Registrierung ohne farbige Darstellung der Distanzen **Rechts:** Nach der Registrierung mit farbiger Darstellung der Distanzen

zahl der Punkte ist, über die iteriert wird, desto zeitintensiver wird der Prozess.
Ein Nachteil des *Iterative Closest Point*-Algorithmus besteht darin, dass Source und Target grob manuell aufeinander ausgerichtet werden müssen, um eine korrekte Transformation zu berechnen. Sind die Objekte nicht ausge-

richtet, findet der Algorithmus eventuell lediglich ein lokales Minimum, was die Anwendung einer „falschen“ Transformation auf die Source zur Folge (siehe Abbildung 50) hat. Für die Berechnung der optimalen Trans-

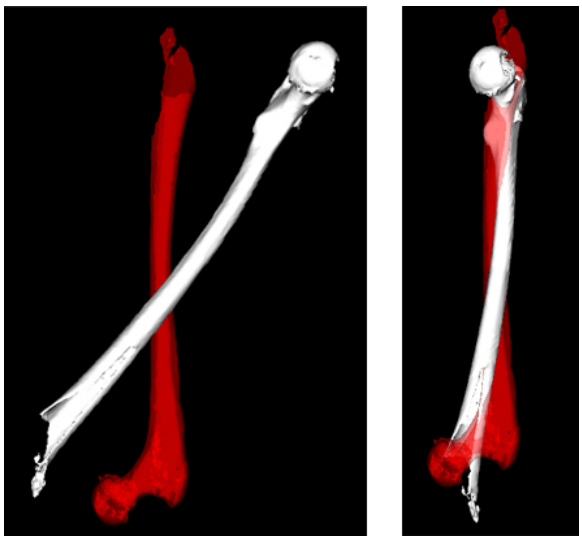

Abbildung 50: Berechnung einer falschen Transformation (lokales Minimum)

formation ist es von Vorteil, wenn die Anzahl der Punkte in Source und Target gleich sind und jeder Punkt in der Szene einen korrespondierenden Punkt im Modell besitzt. Die Daten, die zur Registrierung in der vorgestellten Arbeit Verwendung finden, besitzen diese Eigenschaften nicht. Um den Aufbau des Skelettes einer Moorleiche zu rekonstruieren, sollen die vorhandenen Knochenfragmente auf die Skelettstrukturen eines Koreaners registriert werden. Da in diesem Fall die Registrierung von zwei Skelettstrukturen erfolgt, die nur in ihrer groben Struktur übereinstimmen, ist auch die resultierende Registrierung entsprechend grob (siehe Abbildung 51).

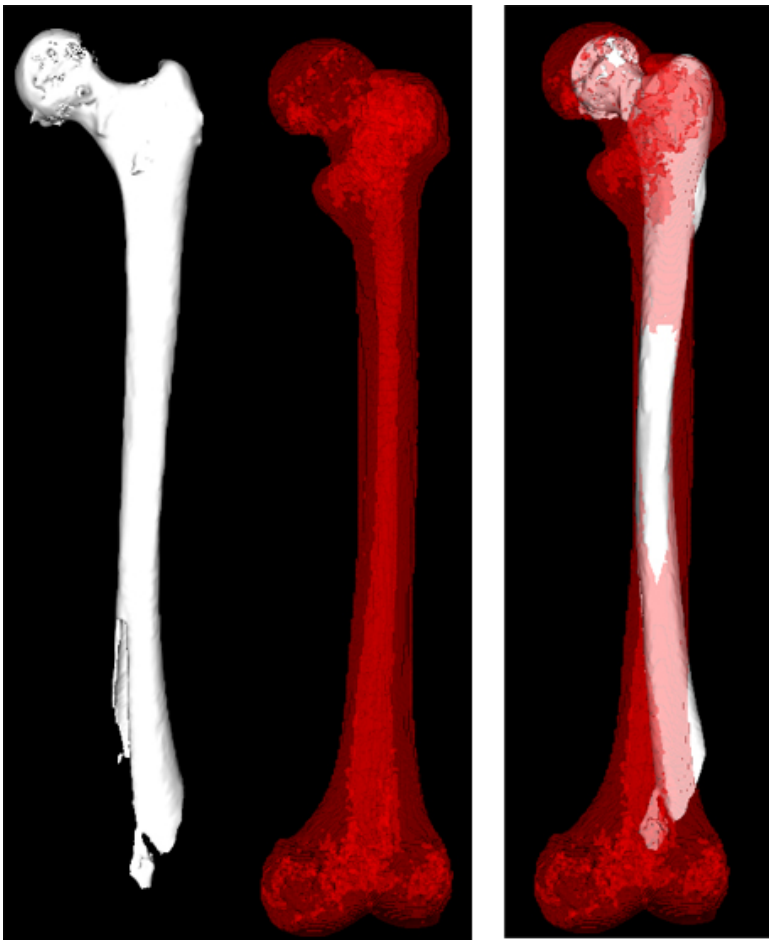

Abbildung 51: Grobe Registrierung

5 Anwendungsszenario

In den folgenden Abschnitten wird konkreter auf ein Anwendungsszenario für die vorliegende Arbeit eingegangen. Die zur Anwendung kommenden Datensätze werden genauer beschrieben und es wird näher auf die Verwendungsmöglichkeiten der implementierten Funktionalität eingegangen.

5.1 Die Moorleiche „Moora“

Im Jahr 2000 wurde beim Torfabbau im Uchter Moor eine Moorleiche gefunden. Eine Radiokarbondatierung der gefundenen Skelettteile ergab, dass diese aus dem Zeitraum um 650 v. Chr. stammen. Weitere Untersuchungen zeigten, dass es sich bei den Knochenteilen um die eines circa 16-19jährigen Mädchens handelt.
Das Skelett weist an vielen Stellen Verbiegungen, Knickbildungen, Schrumpfungen und Zerstörungen von Knochen auf.
Zur ersten groben Rekonstruktion des Skeletts wurden die gefundenen Knochen sortiert und in anatomischer Anordnung auf einer festen Unterlage fixiert. Die anschließenden CT-Aufnahmen ergaben Schichtebenen in etwa $0.8mm$ Abstand. Die so erhaltenen Bildfolgen bildeten die Grundlage für eine Segmentierung (räumliche Abgrenzung der einzelnen Knochenteile). Diese wurde halbautomatisch mit Hilfe von Volume-Growing-Algorithmen [10] durchgeführt. Auf Basis der Segmentierung konnten daraufhin Oberflächenmodelle erzeugt werden: Ein virtuelles Abbild der Knochen von „Moora“ wurde generiert [6].

5.2 Problemstellung

Über die 3D-Visualisierung hinaus, soll das Skelett der Moorleiche „Moora“ realitätsnah rekonstruiert werden. Da das Skelett teilweise unvollständig, fragmentiert und verformt ist, besteht die Notwendigkeit der Anwendung von Schneide- und Spiegelfunktionen: Eine Schneidefunktionen kann verwendet werden, um zerstörte oder deformierte Knochenteile zu entfernen. Fehlende Teile können durch Spiegelung korrespondierender Teile der anderen Körperhälfte ersetzt werden.
Um ohne präzise anatomische Kenntnisse die Knochenteile in ihre richtige Lage bringen zu können, ist die Anwendung eines Registrierungsalgorithmus hilfreich.

5.3 Rekonstruktion mit der Anwendung *ReModelVR*

Die folgenden Abschnitte zeigen beispielhaft, wie die Anwendung *ReModelVR* zur Rekonstruktion von Skelettstrukturen der Moorleiche verwendet werden kann.

5.3.1 Rekonstruktion des Unterkiefers

Die Anwendung kann beispielsweise bei der Rekonstruktion des Schädels von „Moora“ Unterstützung bieten. So ist der linke Teil des Unterkiefers stark deformiert. In Abbildung 52 ist der Verlauf der Rekonstruktion des Unterkiefers dargestellt. Das erste Bild (erste Reihe) zeigt den ursprünglichen verformten Unterkiefer. Im zweiten Bild wird eine Schnittebene, die zum Entfernen des deformierten Teils dient, dargestellt. Im dritten Bild sind die mit Hilfe dieser Ebene zerschnittenen Teile zu sehen. Das vierte Bild (zweite Reihe) zeigt den Kiefer, nachdem der linke Teil entfernt und an diese Stelle der rechte Teil des Kiefers gespiegelt wurde. Im vorletzten Bild werden Segmentationsfehler markiert, die entfernt werden sollen. Das letzte Bild zeigt den fertig rekonstruierten Unterkiefer.

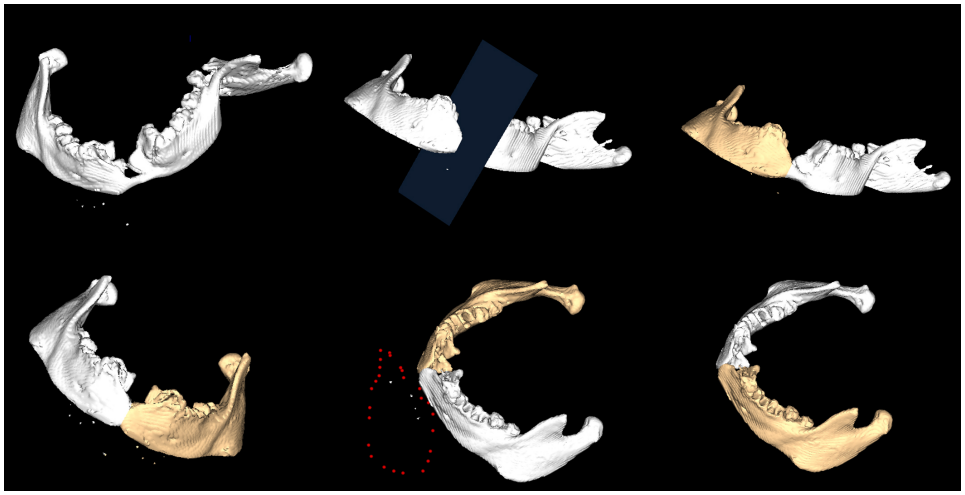

Abbildung 52: Rekonstruktion des Unterkiefers

5.3.2 Positionierung von Knochenteilen

Eine Anwendung für den Registrierungsalgorithmus findet sich in der Unterstützung zur Positionierung einzelner Knochenteile in den Skelettaufbau. Um den Registrierungsalgorithmus anwenden zu können, besteht die Notwendigkeit eines Referenzmodelles. Der Datensatz dieses Modelles wird in dem nächsten Abschnitt eingehender betrachtet.

„Visible Korean“ Der „Visible Korean“ wurde als Datensatz für das Referenzmodell ausgewählt, da er aufgrund seiner Skelettstruktur (geringe Körpergröße, geringe Knochendicke) am besten zu den Skelettteilen „Mooras“ passt.

Entstehung der Daten Der Leichman eines Koreaners wurde tiefgefroren und mit Hilfe einer Abfräsetechnik in $0.2mm$ dicke Scheiben zerlegt. Nach jeweils einem Schnitt wurde eine digitale Fotografie des verbleibenden Körperstückes angefertigt. Auf diese Weise sind mehr als 8500 „Schichtbilder“ mit einer Auflösung von je 2468 x 1407 Pixel entstanden. Aus jeder fünften Schicht wurden innere Organe und Knochen segmentiert. Aus diesen segmentierten Daten lassen sich Oberflächenmodelle generieren [7].

Abbildung 53 zeigt die Positionierung der Oberschenkel und Wadenbeine von „Moora“ an dem Referenzmodell des „Visible Human“. Im ersten Bild (erste Reihe) sind die nicht ausgerichteten Knochen von „Moora“ (links) neben dem Referenzmodell abgebildet. Das zweite Bild zeigt die Knochen, nachdem sie mit dem haptischen Gerät grob ausgerichtet wurden. Im dritten Bild (zweite Reihe) ist das Stadium nach der Ausrichtung der Beinknochen von Moora mit dem Registrierungsalgorithmus zu sehen. Das letze Bild zeigt die Knochen ohne das Referenzmodell. Da in diesem Fall die Registrierung von jeweils zwei Skelettstrukturen erfolgt, die nur in ihrer groben Struktur übereinstimmen, ist auch die resultierende Registrierung entsprechend grob.

5.4 Ergebnisse

Durch die Verwendung des haptischen Gerätes zur Positionierung der Knochenteile zueinander, kann dieses sehr intuitiv erfolgen. Es ist vorstellbar, dass weitere Teile der Skelettstruktur mit Hilfe der Anwendung rekonstruiert werden können. Fehlende Teile können durch Spiegelung vorhandener korrespondierender Strukturen der anderen Körperhälfte ersetzt werden, während verformte Knochen abgeschnitten werden können, um daraufhin das verbleibende Knochenstück durch Anwendung der Spiegelfunktion zu vervollständigen.

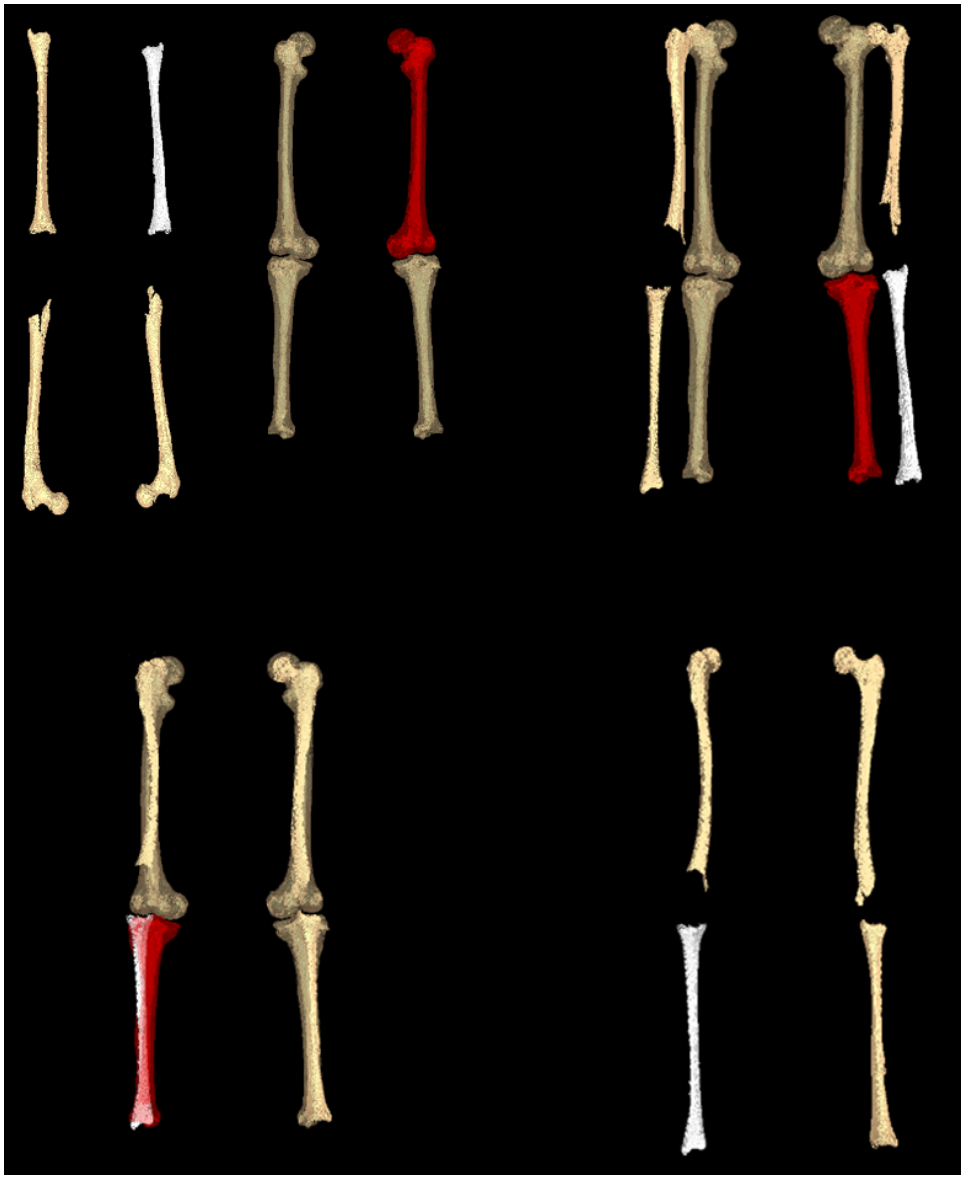

Abbildung 53: Positionierung mit dem haptischen Gerät und dem Registrierungsalgorithmus

6 Zusammenfassung und Ausblick

Zum Abschluss werden die erreichten Ziele noch einmal kurz zusammengefasst und auf Erweiterungs- und Verbesserungsmöglichkeiten eingegangen.

6.1 Zusammenfassung

Im Rahmen der Diplomarbeit wurde eine Anwendung entwickelt, die es ermöglicht, Skelettstrukturen aus Knochenfragmenten zu rekonstruieren. Dabei wurde auf einen in [4] vorgestellten Ansatz zur haptischen Interaktion mit 3D-Modellen zurückgegriffen. Dieser Ansatz wurde softwaretechnisch neu gestaltet und in die institutseigene Bibliothek *mihLib* integriert.
Um den häufigen Wechsel zwischen 2D-Maus und haptischem Gerät zu reduzieren, wurde der Ansatz zur Steuerung der virtuellen Kamera verbessert. Die Generierung der Kraft, die den Stift des haptischen Gerätes im Mittelpunkt des Arbeitsbereiches hält, findet nicht mehr statt, da diese die Kamerasteuerung erschwerte.
Verschiedene Bewegungen des Gerätearmes führen zu unterschiedlichen Kamerabewegungen. Zusätzlich besteht die Möglichkeit die Elemente der grafischen Benutzeroberfläche mit dem haptischen Gerät zu bedienen; dieses verringert ebenfalls den häufigen Wechsel zwischen den verschiedenen Eingabegeräten.
Im Laufe der Programmierung entstand die Notwendigkeit, Teile der schon bestehenden Integration der haptischen Komponente in VTK ([4]) zu korrigieren. Durch diese Korrektur entfällt das bisher erforderliche Anwenden der Gerätetransformation auf einen nicht-haptischen *Dummy* beim Bewegen der Objekte mit dem haptischen Gerät. Ferner wurde dadurch die Möglichkeit geschaffen, zusätzlich den von VTK bereitgestellten Interaktorstil zu nutzen, um das Objekt mit der 2D-Maus bewegen zu können. In der vorherigen Implementierung führte dies zu Fehlern in der Bewegung der Objekte.
Die Schneidefunktion wurde dahingehend erweitert, dass Objekte mit verschiedenen Schneidewerkzeugen geschnitten werden können. Dazu zählt das Schneiden mit einer Ebene, einer Box oder einer Kugel. Der Benutzer hat die Auswahl zwischen der Positionierung der Schneidewerkzeuge mit dem haptischen Gerät oder der 2D-Maus. Desweiteren erfolgte eine Verbesserung der bereits bestehenden Schneidefunktion. Die bereits bestehende Implementierung erlaubt es, Schnittmarken mit dem haptischen Gerät auf der Objektoberfläche zu setzen. Das Schneiden erfolgt jedoch anhand einer Ebene, die durch die erste und letzte Schnittmarke gelegt wird. Die verbesserte Funktion schneidet anhand der gesetzten Marken. Dabei kann der Benutzer Schnittmarken auf der Objektoberfläche setzen und so Teile der Oberfläche herausschneiden oder Schnittmarken in den freien Raum

vor der Oberfläche zu setzen, um das Modell ganz durchzuschneiden.
Eine Erweiterung erfolgte mit der Implementierung einer Spiegelfunktion, die zum Ersetzen fehlender Strukturen eingesetzt werden kann. Die bekannten Spiegelungsarten (Ebenenspiegelung, Achsenspiegelung, Punktspiegelung) wurden umgesetzt. Für den Benutzer besteht wiederum die Möglichkeit, die Spiegelwerkzeuge mit der 2D-Maus oder dem haptischen Gerät zu positionieren.
Eine zusätzliche Erweiterung bietet die Integration eines Registrierungsalgorithmus, der es erlaubt, Skelettstrukturen an einem Referenzmodell auszurichten. Zu diesem Zweck wurde der von VTK bereitgestellte ICP-Algorithmus in die Anwendung integriert. Eine visuelle Unterstützung bietet hierbei die farbige Darstellung der Distanzen der nächsten Punkte zueinander.

6.2 Ausblick

Die vorgestellte Anwendung bietet eine Vielzahl von Erweiterungs- und Verbesserungsmöglichkeiten:

- **Interaktion mit den Oberflächenmodellen**
  Bisher besteht lediglich die Möglichkeit der Interaktion zwischen opaken Modellen und dem haptischen Eingabegerät. Transparente Strukturen können nicht erfühlt und bewegt werden. Um dies zu erlauben, muss der Verlauf des Rendering-Prozesses von VTK für transparente Strukturen genauer analysiert werden, um die korrekte Position der HLAPI-Befehle zum Rendern dieser Strukturen zu finden.
  Eine zusätzliche Erweiterung wäre die Integration einer Kollsionserkennung und -behandlung zwischen den haptischen Objekten. Vorstellbar ist die Generierung einer Kraft, die bei der Kollision zweier Objekte den Aufprall zwischen diesen Objekten widerspiegelt. Damit der Benutzer die Kraft zum Zeitpunkt der Kollision spürt, muss die Kollisionserkennung ausreichend schnell ablaufen.

- **Schneidewerkzeuge und Schneidefunktion**
  Um die Schneidewerkzeuge in ihrer Größe zu verändern, müssen *Slider* der grafischen Oberfläche bedient werden. Die Manipulation mit der 2D-Maus bietet dagegen die Möglichkeit, die Objekte mittels an ihnen angebrachten *Handles* zu vergrößern oder verkleinern. Dieses wäre auch für die Änderung der Größe mit dem haptischen Gerät wünschenswert.
  Das Einzeichnen von Schnittmarken auf einer Oberfläche gestaltet sich durch die Struktur der Modelle als relativ schwierig. Es kommt zu häufigem Abrutschen mit dem haptischen Gerät. Aus diesem Grund wurde eine Funktion implementiert, die es ermöglicht im freien Raum vor dem Objekt Schnittmarken einzuzeichnen. Hierbei ist es kaum

möglich in einer Ebene zu bleiben. Um dies zu gewährleisten und damit ein genaueres Setzen von Schnittmarken möglich zu machen, ist die Generierung einer fühlbaren halbtransparenten Ebene vor dem Objekt erstrebenswert, auf der die Marken eingezeichnet werden können. Die Ebene würde dem Benutzer einen Halt beim Zeichnen geben und eine Erleichterung bieten. Schnittmarken könnten genauer eingezeichnet werden.

- **Registrierung**
  Um Knochenstrukturen auf ein Referenzmodell abzubilden, wurde der *Iterative Closest Point*-Algorithmus von VTK in die Anwendung integriert. Weichen die beiden zu „matchenden" Punktmengen zu stark voneinander ab, das heißt besitzen sie eine sehr unterschiedliche Anzahl an Punkten oder weist ein Datensatz starke Formdeffekte auf, versagt der *ICP*-Algorithmus häufig. Eine Verbesserung soll beispielsweise der *Trimmed ICP*-Alorithmus bieten. Eine gleiche Anzahl der Punkte in den aufeinander zu registrierenden Datensätzen ist nicht notwendig. Es besteht keine Notwendigkeit, dass zu jedem Punkt aus P ein korrespondierender Punkt in M existiert. Bei der Anwendung des *trimmed ICP*-Algorithmus wird zuerst eine Überschneidungsrate ξ berechnet. Es können damit zu $N_{P_\xi} = \xi N_P$ Punkten aus P korrespondierende Punkte in M gefunden werden. Daraufhin erfolgt die Bestimmung der nächsten Nachbarn und die Berechnung der individuellen Distanzen. Die Distanzen werden in absteigender Reihenfolge sortiert, die N_{P_ξ} kleinsten Distanzen ausgewählt und ihre Summe berechnet. Es erfolgt die Bestimmung der Transformation, die die berechnete Summe minimiert. Ob der *Trimmed ICP*-Algorithmus eine Verbesserung, bei den in dieser Arbeit verwendeteten stark fehlerhaften und deformierten Daten bringt, müsste evaluiert werden.
  Eine nützliche Erweiterung würde darin bestehen, eine „haptische Registrierung" einzubauen. Sie könnte versuchen, eine Kraft so zu generieren, dass der Benutzer mit der richtigen Translation und Rotation auf das Referenzmodell geleitet wird.

Literatur

[1] P. J. Besl and N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[2] D. Böhme and M. Sotoodeh. Haptik. http://www.informatik.uni-bremen.de/ nostromo/haptik/index.html, 1999. last checked: December 2006.

[3] G. C. Burdea. Haptic Feedback for Virtual Reality. Based on the paper with the same title presented at the Virtual Reality and Prototyping Workshop, June 1999, Laval (France).

[4] F. Drescher. Integration von haptischen Ein-/Ausgabegeräten mit Kraftrückkopplung in OP-Planungssysteme. Master's thesis, Universität Koblenz-Landau, 2006.

[5] Dr. K. Drewing. Hautsinne. www.allpsych.uni-giessen.de/karl/teach/Wahrnehmung/Wahr-12-haut.pdf. last checked: December 2006.

[6] K. Püschel et al. Radiologische Untersuchungen am Mädchen aus dem Uchter Moor - erste Ergebnisse aus dem Universitätsklinikum Hamburg-Eppendorf. In *Berichte zur Denkmalpflege in Niedersachsen*, pages 34–37, 2 2006.

[7] M. Riemer et al. Improving the 3D Visualization of the Visible Korean Human via Data Driven 3D Segmentation in RGB Colour Space. In *Proceedings of the World Congress on Medical Physics an Biomedical Engineering*, pages 4057–4060, 2006.

[8] S. Frings. http://www.sinnesphysiologie.de/hvsinne/haut/mechano.htm, 2003. last checked: December 2006.

[9] http://wwwicg.informatik.uni-rostock.de/Lehre/HCG/scripte0304/05_VRAR.pdf. last checked: December 2006.

[10] H. Handels. *Medizinische Bildverarbeitung*. Teubner Verlag, 2000.

[11] http://www.davidszondy.com/future/robot/hardiman.htm. last checked: November 2006.

[12] http://faculty.etsu.edu/currie/images/homunculus1.JPG. last checked: December 2006.

[13] B. K. P. Horn. Closed-Form solution of absolute orientation using unit quaternions. *Optical Society of America*, 4:629–642, 1987.

[14] R. Huch and C. Bauer. *Mensch, Körper, Krankheit*. Urban & Fischer, 2003.

[15] Immersion. http://www.immersion.com. last checked: December 2006.

[16] Inc. Kitware. *The VTK User's Guide*. Kitware, Inc., 2004.

[17] T. H. Massie and J. K. Salisbury, editors. *The PHANTOM Haptic Interface: A Device for Probing Virtual Objects*, Proceedings of the ASME Winter Annual Meeting, 1994.

[18] H. Morgenbesser and M. Srinivasan. Force Shading for Haptic Shape Perception. In *Proceedings of the ASME Dynamics Systems and Control Division*, 1996.

[19] H. Pottmann. Grundkurs Architektur & Darstellung: Darstellende Geometrie. www.geometrie.tuwien.ac.at/peternell/arch/grundkursdg05.pdf. last checked: December 2006.

[20] A. Rettig. Haptisches Rendering. geri.uni-koblenz.de/ws0506/vrarfolien/haptik.pdf. last checked: January 2007.

[21] D. C. Ruspini, K. Kolarov, and O. Khatib, editors. *The Haptic Display of Complex Graphical Environments*, COMPUTER GRAPHICS Proceedings, Annual Conference Series, 1997.

[22] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 3rd edition edition, 2004.

[23] Somatische Sensibilität und Geschmacksinn. http://www.biologie.uni-duesseldorf.de/Institute/Neurobiologie/Lehrangebot/Grundstudium/Dokumente/Bio5_Somatosensorik Skript.pdf. last checked: December 2006.

[24] M. A. Srinivasan. What is Haptics? Laboratory for Human and Machine Haptics: The Touch Lab Massachusetts Institute of Technology.

[25] K. V. Stolpe. *Einfluss eines kinästhetischen Trainings auf das Erlernen des Golfschwungs*. PhD thesis, Universität der Bundeswehr München, 2002.

[26] R. J. Stone. Haptic Feedback: A Potted History, From Telepresence to Virtual Reality.

[27] SensAble Technologies. http://www.sensable.com. last checked: December 2006.

[28] Sensable Technologies. *OpenHaptics Toolkit, API Reference*.

[29] Sensable Technologies. *OpenHaptics Toolkit, Programmers Guide*.

[30] Visualization ToolKit. http://www.vtk.org/. last checked: January 2007.

[31] Wikipedia. Haptische Wahrnehmung. http://de.wikipedia.org/wiki/Haptische_Wahrnehmung, 2006. last checked: December 2006.

[32] C. B. Zilles and J. K. Salisbury, editors. *A Constraint-Based God-Object Method for Haptic Display*, Proc. IEEE International Conference on Intelligent Robots and Systems, 1995.