UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Universitätsklinikum Hamburg-Eppendorf
Institut für Medizinische Informatik

Computergestützte 3D-Operationsplanung zur präoperativen Repositionierung von Knochenfragmenten bei komplizierten Knochenbrüchen

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Kai Bestmann

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Prof. Dr. Heinz. Handels
(Institut für Medizinische Informatik, UKE)

Koblenz, im September 2006

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

Inhaltsverzeichnis ii

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Ziele . . . 1
1.3 Problemstellung . . . 2

2 Grundlagen 4
2.1 Unfallchirurgie . . . 4
2.2 Verwendete Daten . . . 4
2.2.1 Computertomographie . . . 4
2.2.2 Segmentierung . . . 6
2.2.3 Modellgenerierung . . . 9
2.2.4 Erzeugung von Schnittkonturen . . . 11

3 Kollisionserkennung 13
3.1 Allgemeine Anforderungen . . . 14
3.2 Anwendungsspezifische Anforderungen . . . 15
3.3 Raumunterteilung (broad phase) . . . 16
3.3.1 Uniforme Raumunterteilung . . . 16
3.3.2 KD-Bäume und-Binary-Space-Partitioning-Bäume . . . 17
3.3.3 Oktonärbäume . . . 17
3.4 Hüllkörper (broad phase) . . . 17
3.4.1 Separating-Axis-Theorem . . . 19
3.4.2 Kugel als Hüllkörper . . . 20
3.4.3 Axis Aligned Bounding Box (AABB) . . . 21
3.4.4 Object Oriented Bounding Box (OBB) . . . 23
3.4.5 Discrete Oriented Polyeder (k-DOP) . . . 24
3.4.6 Swept-Sphere-Volumes . . . 25
3.5 Detaillierte Kollisionsbestimmung (*narrow-phase*) . . . 26
3.5.1 Linearer Ansatz . . . 26
3.5.2 Mediale Achse . . . 27
3.5.3 Lin-Canny-Algorithmus . . . 28
3.5.4 GJK . . . 30
3.6 Bibliotheken für Kollisionsbehandlung . . . 30
3.6.1 I-Collide . . . 31
3.6.2 RAPID . . . 32
3.6.3 V-Collide . . . 32
3.6.4 Swift und Swift++ . . . 32
3.6.5 PQP . . . 33
3.6.6 QuickCD . . . 33
3.6.7 Voronoi-Clip (V-Clip) . . . 33
3.6.8 Solid und FreeSolid . . . 34
3.6.9 vtkCollisionDetectionFilter . . . 34

4 Realisierung / Implementierung 35
4.1 Verwendete Software und Bibliotheken 35
4.1.1 V-Collide als Bibliothek für die Kollisionsbehandlung 35
4.1.2 CMake 36
4.1.3 Visualization ToolKit (VTK) 36
4.1.4 QT 40
4.1.5 mihLib 41
4.2 Beschreibung der Funktionalitäten 42
4.3 Softwaretechnischer Entwurf 45
4.4 Realisierung 48
4.4.1 Ein- und Ausgabe der Daten 48
4.4.2 Schnittstelle zu Qt 50
4.4.3 Schnittstelle zu V-Collide 50
4.4.4 Verhaltensmuster in Abhängigkeit der Modii 52
4.4.5 Verwaltung der Transformationen 53
4.4.6 Visualisierung durch Explosionsdarstellung 59
4.4.7 Darstellung der Schichtdaten 61
4.4.8 Erweiterte Interaktionen 62

5 Ergebnisse und Diskussion 64
5.1 Beschreibung der vorhandenen Datensätze 64
5.2 Laufzeit der Komponenten 65
5.2.1 V-Collide 65
5.2.2 Konturbestimmung 71
5.3 Zusammenfassung und Ausblick 71
5.3.1 Zusammenfassung 71
5.3.2 Ausblick 73

6 Danksagung 75

Literatur 76

A Ausschnitt einer XML-Datei 79

B Ausgabe im HTML-Format 80

1 Einleitung

1.1 Motivation

Die Unfallchirurgie ist die erste Instanz, die medizinische Notfälle, bei denen es sich um Knochenbrüche handelt, entgegennimmt, stabilisiert und analysiert. Bei inneren Verletzungen werden für gewöhnlich zu Beginn radiologische Daten erzeugt, um einen Überblick über die beschädigten Strukturen im Körperinneren des Patienten zu erhalten. Je nach Verdachtsfall werden die verschiedenen radiologischen Aufnahmemöglichkeiten genutzt. Bei Frakturen, die eventuell eine hohe Komplexität aufweisen, werden mittlerweile alternativ zu der Röntgenübersicht 3D-Volumendaten mit Hilfe der Computertomographie (CT) ermittelt. Dadurch ist es möglich, an Details über die Fraktur zu gelangen, die selbst mehrere gewöhnliche Röntgenübersichten nicht bieten.

Ist die Art der Fraktur dem behandelnden Chirurgen durch Analyse der Bilder bekannt, wird die Operation (OP) und damit die Repositionierung und Fixierung der einzelnen Knochenfragmente geplant und vorbereitet. Anschließend erfolgt der Eingriff. Zwischen Einlieferung und Operation liegen meist nicht mehr als 24 Stunden [Rue03]. Die Zeit, die zur Analyse der Fraktur und Planung der OP zur Verfügung steht, ist demnach begrenzt.

Bei einigen schwierigen Frakturen bieten aber selbst die schichtweise angezeigten CT-Volumendaten unzureichenden Aufschluss über die räumlichen Details der Fraktur. In solchen Fällen kann letzten Endes nur während der Operation der tatsächliche Sachverhalt festgestellt und behandelt werden.
Eine virtuelle 3D-Darstellung der Fraktur mit verschiedenen Interaktionsmöglichkeiten, die vor Beginn der Operation vom Arzt genutzt wird, kann einige hilfreiche Informationen über die Beschaffenheit der Fraktur geben. Komplexe räumliche Zusammenhänge der Fraktur werden veranschaulicht. Die gedankliche Rekonstruktion der Bruchfragmente wird unterstützt.

1.2 Ziele

Die im Rahmen dieser Diplomarbeit entwickelte Anwendung soll in erster Linie der präoperativen Planung und Simulation von Operationen dienen, bei denen komplizierte Knochenbrüche behandelt werden müssen. Sie soll den chirurgischen Eingriff und damit letztlich den Genesungserfolg des Patienten verbessern, indem der operierende Chirurg bereits im Vorfeld der Operation die Fraktur anhand eines 3D-Modells untersucht und die Position einzelner Knochenfragmente virtuell korrigiert. Die Interaktionen sollen in Echtzeit stattfinden. Verschiebungen und Drehungen von Bruchfragmenten und die Beschaffenheit von Bruchkanten können genau betrachtet und getestet werden. Dadurch erhält der Chirurg eine bessere Vorstellung der gesamten Frakturbeschaffenheit. Mögliche Probleme und Schwierigkeiten bei dem Eingriff können vorher erkannt und bei der Planung berücksichtigt werden. Eine integrierte, geeignete Kollisionserkennung soll dabei die Repositionierung der einzelnen Frakturfragmente realistischer erscheinen lassen.

Des Weiteren soll die Anwendung sinnvoll in der chirurgischen Ausbildung ge-

nutzt werden können, indem typische und untypische Frakturen visualisiert werden. Anhand der veränderbaren Frakturdarstellung sollen die notwendigen Schritte der Korrekturmaßnahmen von Studierenden der Medizin am Computer trainiert werden können.

Das Programm soll einfach, schnell und intuitiv zu bedienen sein. Der virtuelle Arbeitsvorgang soll gespeichert werden können. Zudem soll eine Ausgabe in einer sinnvollen Form ermöglicht werden, die ein Chirurg gegebenenfalls mit in den Operationssaal oder der Studierende mit nach Hause nehmen kann.

1.3 Problemstellung

Die Hauptaufgabe bei dieser Arbeit besteht darin, eine kompakte, einfach bedienbare und echtzeitfähige Anwendung zu entwickeln, die einen Chirurgen bei seiner Arbeit unterstützt. Im Rahmen dessen sollen eine Kollisionserkennung sowie geeignete Visualisierungstechniken implementiert und die Anwendung in das institutseigene Framework mit dem Namen *mihLib* integriert werden. Dabei können die Bibliotheken von VTK, *mihLib*, Qt und OpenGL genutzt und die von *mihLib* gegebenenfalls auch verändert und erweitert werden. Die zu verwendende Programmiersprache ist C++.
Aufgrund dessen müssen folgende Probleme bzw. Aufgaben gelöst werden:

- Die 3D-Visualisierung der Fraktur und die Navigation der Knochenfragmente innerhalb der Szene ist für einen Arzt normalerweise ungewohnt und bedarf somit besonderer Aufmerksamkeit, wogegen Schichtansichten - meist axial - bekannt sind.
  Ein Problem ist demnach das Implementieren und Einsetzen ausgewählter Visualisierungstechniken für die 3D-Szene und das An- und Verknüpfen an bekannte Visualisierungen wie eben diese Schichtbilder.

- Zudem gilt es, eine geeignete Kollisionserkennung durch Analyse und Vergleich verschiedener Kollisionsbehandlungssysteme zu ermitteln und zu implementieren. Eine Kollisionserkennung dient dem Bestimmen von Objekten, die durch eine Objektrepositionierung miteinander kollidieren. Bei Kollision kann entsprechend darauf reagiert werden, wodurch das Verhalten von Objekten zueinander realistischer gestaltet werden kann. Die Kollisionserkennung muss den speziellen Anforderungen der vorliegenden 3D-Szenen-Eigenschaften genügen.

- Die Repositionierung und damit ebenso die Kollisionserkennung soll in Echtzeit erfolgen, was bei der Interaktion eine Frequenz um die 25 Hz und mehr voraussetzt. Zudem soll die Möglichkeit einer Erweiterung durch haptische Ein- und Ausgabegeräte gegeben werden, was eine Frequenz von mindestens 1000 Hz bedingt.

- Wie bereits erwähnt, sollen die durchgeführten Veränderungen an einer Fraktur-Szene in einer allgemeingültigen und für den Anwender verständlichen Form gespeichert und ausgegeben werden können. Ziel ist es, ein Format zu entwickeln, das neben den optionalen Repositionierungsdaten andere Informationen aus institutseigenen Projekten enthält. Zusätzlich soll eine,

für den Chirurg geeignete, grafische Ausgabe der Korrekturmaßnahmen für den Bereich der OP-Planung erstellt werden.

- Die implementierten Module sollen in einer Form programmiert werden, die sie auch für andere institutseigene Anwendungen nutzbar macht.

2 Grundlagen

In diesem Kapitel wird auf verschiedene Themen eingegangen, die in die Planung und Realisierung der Programmentwicklung direkt oder indirekt involviert und somit relevant sind. Es wird kurz auf den möglichen Einsatzbereich - die Unfallchirurgie - eingegangen. Daraufhin wird erläutert, wie die relevanten Daten zustande kommen und wie sie im Vorfeld bearbeitet werden, um für das Programm nutzbar zu sein.

2.1 Unfallchirurgie

Medizinische Notfälle, bei denen es sich um Knochenbrüche handelt, werden zuerst in der Unfallchirurgie stationär untersucht. Zuvor wird die medizinische Erstversorgung von einem Notarzt direkt am Unfallort oder auf dem Weg ins Krankenhaus durchgeführt. Mittlerweile wird in der Unfallchirurgie laut [Rue03] zur Diagnose bei der Mehrheit der Schockraumpatienten an mindestens einer Körperstelle (z.B. Schädel) ein CT durchgeführt; bei kreislaufstabilen Patienten immer öfter sogar ein Ganzkörper-CT. Vorteile sind hier die Zeitersparnis und die Möglichkeit einer präzisen Diagnostik. Nachteilig ist die unter Umständen unnötig hohe Strahlenbelastung für den Patienten. Die Entwicklung der Datenerstellung mittels CT stellt eine notwendige Grundlage für die zu erstellende Anwendung dar.

Die Dringlichkeit einer Operation hängt von der Verletzung des Patienten ab. Bei lebensbedrohlichen Verletzungen wie inneren Blutungen oder zum Entlasten pathologischer Druckverhältnisse im Thorax wird eine lebensrettende Sofort-OP durchgeführt. Zur Sicherung der Vitalfunktionen wird ein dringlicher Primäreingriff durchgeführt. Die Indikatoren hierfür können unter anderem offene Frakturen, offene Gelenke, grobe Skelettinstabilitäten, Frakturen langer Röhrenknochen (Femurschaft, Unterschenkelfraktur, etc.), Luxationsfrakturen und instabile Beckenringfrakturen sein. Beide OP-Arten werden in den Versorgungs-Stufenplan integriert und auf jeden Fall bereits am ersten Tag durchgeführt (*Day-1-Surgery*), denn der Zeitrahmen ist durch ein enges immunologisches Fenster vorgegeben.

Die zu entwickelnde Anwendung ist vornehmlich für derartige Verletzungen vorgesehen und muss bereits vor der Operation einsatzfähig sein. Das heisst, dass auch die dafür notwendigen Daten in diesem Zeitfenster erstellt werden müssen.

2.2 Verwendete Daten

Im Folgenden wird beschrieben, wie die Daten (CT-Daten und 3D-Modelle der Knochenfragmente), die für das Programm notwendig sind, erzeugt und verarbeitet werden.

2.2.1 Computertomographie

Der Bildgebungsprozess mit Hilfe eines modernen CT-Gerätes dauert mittlerweile nur noch wenige Minuten. Der Patient wird auf den Tisch des Gerätes gelegt, der sich anschließend durch einen Detektorring bewegt.

Handelt es sich um einen sequentiellen CT, werden die Aufnahmen pro Schicht bei fester Tischposition durchgeführt, Röhren und Detektoren bewegen sich dabei um 360 Grad um den Patienten. Diese Art der Aufnahme ist mittlerweile veraltet und wird für gewöhnlich nur noch selten eingesetzt.

Bei einem modernen Spiral-CT wird der Tisch samt Patienten craniocaudal (in Kopf-Fuß-Richtung) durch den Detektionsring bewegt. Die Scanner rotieren dabei gleichmäßig um den Tisch, wodurch eine spiralförmige Abtastung erreicht wird. Aktuelle Geräte können durch den Einsatz von Mehrzeilendetektoren bis zu 64 Schichten parallel aufnehmen. Das ermöglicht das Herausrechnen von kleine Bewegungen des Patienten während der Aufnahme, erhöht die Detailgenauigkeit und verringert die Aufnahmezeit. Zudem konnte mit diesem Verfahren die Strahlenbelastung für den Patienten reduziert werden.
Aus den aufgenommenen Daten kann ein Volumen rekonstruiert werden, bei dem ein Voxel die kleinste Einheit darstellt. Es werden derzeit Geräte angeboten, die ein isotropisches Voxelvolumen von bis zu 0.33 mm^3 erreichen können [AG06]. Die Daten werden im DICOM-Format (.dcm) gespeichert. Das weitverbreitete DICOM-Format ermöglicht eine universelle Weiterverarbeitung [NEM].

Die Testdatensätze für die Entwicklung der Anwendung wurden von einem 'RS SOMATOM Volume Zoom'-Computertomographen der Firma Siemens aufgenommen. Dabei handelt es sich um ein 4-Zeilen Gerät mit einer minimalen Schichtdicke von 0.33 mm. Die Größe der Datensätze variiert je nach Anzahl der notwendigen Schichten. Für eine Hüfte müssen zum Beispiel circa 30 cm craniocaudal aufgenommen werden. Eine Schicht besteht aus 512 $\times$ 512 Elementen á 12 Bit, bei 150 Schichtaufnahmen á 2 mm benötigen die Daten dann rund 60 MB Speicherplatz.

Ein mit dem CT erstelltes Voxel hat einen hohen Informationsgehalt. Es kann 2^{12} verschiedene Grauwerte annehmen, wobei sie den über das Voxel-Volumen gemittelte Abschwächungskoeffizienten μ repräsentieren. Die Werte werden für gewöhnlich in einem Bereich von -1000 bis +3096 mit der Einheit Hounsfield (HE) angegeben. Zur Standardisierung nimmt man für die Referenzflüssigkeit Wasser einen Hounsfield-Wert von 0 HE und für Luft einen Wert von -1000 HE an.

$$HE = 1000 \cdot \frac{(\mu_{Obj} - \mu_{H_2O})}{\mu_{H_2O}}$$

Typische HE-Werte für menschliche Organgewebe sind zum Beispiel 50 bis 70 HE für Leber-, 20 bis 40 HE für Nieren-, -500 bis -950 HE für Lungengewebe, 50 bis 250 HE für spongiösen und ab 250 HE für kompakten Knochen (siehe auch Abb. 1).

Fensterung Das menschliche Auge kann laut [GWK06] nur zwischen circa 20 bis 35 Graustufen unterscheiden, wodurch es unmöglich ist alle 2^{12} Möglichkeiten der Dichte-Informationen aus einem CT-Datensatz durch Grauwerte zu visualisieren. Aufgrund des beschränkten Darstellungsraums ist es sinnvoll, nur die diagnostisch relevanten Dichtebereiche des CT-Datensatzes auf die für das menschliche Auge unterscheidbaren Grauwerte abzubilden, was mit der sogenannten Fensterung geschieht.

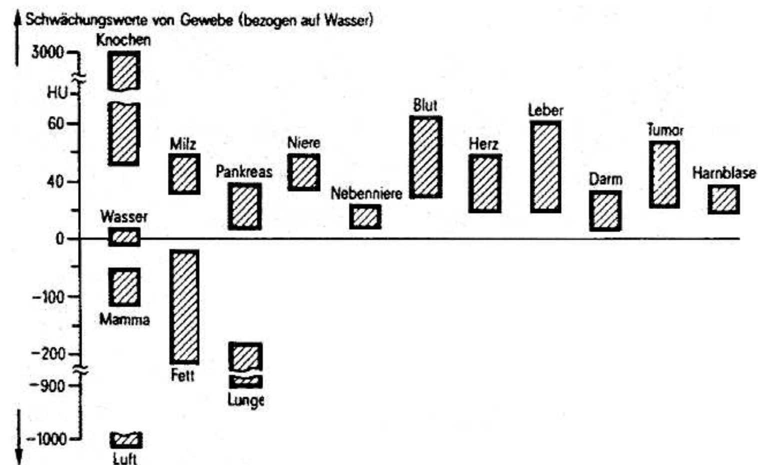


Abbildung 1: Hounsfield-Werte für verschiedene Gewebearten (Quelle: [Mor95])

Das Fenster wird definiert, indem zunächst festgelegt wird, welchem CT-Wert der mittlere Grauwert zugeordnet werden soll (*level*). Mit der Breite des Fensters werden der minimale und maximale CT-Wert festgelegt, die noch differenzierbar sein sollen ($window$). Werte außerhalb des Fensters werden schwarz bzw. weiß dargestellt. Typische Fensterungseinstellungen sind zum Beispiel $level = 1000HE$, $window = 2500HE$ für ein Knochenfenster oder $level = -626$HE, $window = 1250$HE für ein Lungenfenster.

2.2.2 Segmentierung

Bei der Segmentierung der Knochenstrukturen aus den durch den CT gewonnenen Daten werden die Teilvolumen freigestellt, aus denen später 3D-Modelle erstellt werden sollen. Realisiert wird dieser Vorgang mit MeVisLab [MeV06], einer Plattform für Bildverarbeitungsforschung und -entwicklung speziell im medizinischen Bereich. Neben zahlreichen Verarbeitungsalgorithmen und Visualisierungswerkzeugen sind bereits Module für die Segmentierung enthalten. Eine unvollständige MeVisLab-Version kann für nicht-kommerzielle Zwecke frei verwendet werden. Die aktuelle Version ist 1.4.

Derzeit bietet MeVisLab die Segmentierung mittels der Verfahren *Region-Growing*, *Live-Wire*, *Fuzzy Connectedness*, Schwellwert und manuelle Konturerstellung an. Um die 3D-Objekte für die Anwendung herzustellen, wurde ein MeVisLab-Arbeitsbereich erstellt, dessen Arbeitsschritte im Folgenden kurz erläutert werden. Dabei wird zur Segmentierung der *Region-Growing*-Ansatz gewählt, da dieser schwellwertbasierend ist, was für die spätere Oberflächenerzeugung der 3D-Modelle sinnvoll ist. Die unterschiedlichen Hounsfield-Werte von Weichteilgewebe und Knochenstrukturen bilden für die Segmentierung eine gute Ausgangslage.

Als Vorverarbeitung werden die Volumendaten zunächst durch einen 3×3-Gauß-Filter geglättet. Ein Gauß-Filter ist ein linearer Filter und dient der Rauschreduktion, wobei die Kanten jedoch besser erhalten bleiben als bei einem einfachen Mit-

telwertfilter. Anschließend erfolgt die Segmentierung mittels *Region-Growing*.

Region-Growing Ziel ist es, aus den vorliegenden CT-Volumendaten die Knochenstrukturen zu ermitteln. Wie bereits erläutert, weisen alle Voxel, die Knochen repräsentieren einen Grauwert auf, der innerhalb eines gewissen Bereichs liegt. Diese Voxel gilt es freizustellen. Beim *Region-Growing*- bzw. *Volume-Growing*-Verfahren handelt es sich um einen *bottom-up*-Algorithmus, dem zu Beginn mindestens zwei Parameter übergeben werden müssen, um mit dem Segmentieren beginnen zu können:

- Ein oder mehrere Startvoxel S (*seed points*), von denen aus der Algorithmus beginnt.
- Ein Homogenitätskriterium H bzw. ein Schwellwert, der angibt, wie hoch die Grauwert-Toleranz bei der Auswahl der Voxel ist.

Region-Growing ist durch die notwendigen Parameter nur halbautomatisch. Für gewöhnlich definiert jedes Startvoxel eine eigene Region, die im Laufe des Algorithmus expandiert wird. Es ist dagegen auch möglich, mit mehreren Startvoxeln ein und dieselbe Region zu definieren. Letzteres wurde im Rahmen der Knochensegmentierung angewandt.
Die Startvoxel werden zu Beginn in einer Menge M eingetragen. M repräsentiert die Ergebnisregion der Segmentierung. Ausgehend von den Startvoxeln werden deren benachbarte Voxel betrachtet, die die Menge N_s im initialen Zustand bilden. N_s enthält stets alle Volumenelemente, die es noch zu untersuchen gilt und an denen die aktuelle Region noch expandierbar ist. Die Grauwerte der Voxel in N_s werden mit dem der Startvoxel verglichen. Erfüllen sie das Homogenitätskriterium H, werden sie selbst M und ihre Nachbarn N_s hinzugefügt. Erfüllt ein Voxel H nicht, wird es als bearbeitet markiert. Der Algorithmus stoppt, wenn keine Voxel aus N_s das Homogenitätskriterium mehr erfüllen und es somit nicht möglich ist, die Region zu erweitern. [Han00]
Die Laufzeit ist direkt proportional zur Anzahl der zu segmentierenden Voxel. Die

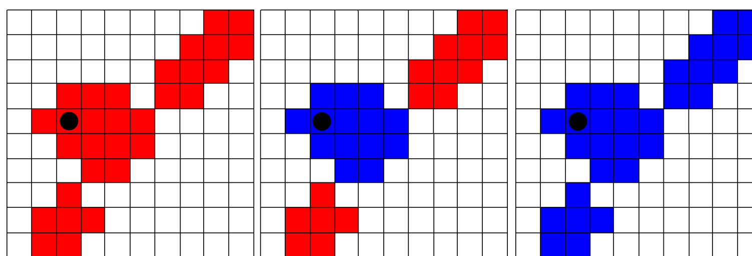


Abbildung 2: Beispiel für verschiedene *Region-Growing*-Ergebnisse bei unterschiedlicher Nachbarschaft: links die zu segmentierende 2D-Fläche, in der Mitte das Ergebnis bei 4-er Nachbarschaft, rechts das Ergebnis bei 8-er Nachbarschaft. Der Punkt repräsentiert den *seed point*

Wahl unterschiedlicher Nachbarschaften, also die Wahl der benachbarten Voxel, die es zu untersuchen gilt, hat einen großen Einfluss auf das Segmentierungsergebnis. In Abbildung 2 ist dafür ein Beispiel anhand einer 2D-Region dargestellt. Dort werden 4-er und 8-er Nachbarschaften illustriert. In 3D-Volumen sind 6-er

oder 26-er Nachbarschaften üblich. Ebenso entscheidend für die Qualität sind die Wahl der Startvoxel und des Homogenitätskriteriums.
Für den *Region-Growing*-Algorithmus gibt es diverse Erweiterungen, die für diese Arbeit jedoch nicht relevant sind und daher nicht weiter betrachtet werden (siehe [Han00]).

Die vorliegenden Daten werden mit einer 6-er Nachbarschaft segmentiert. Als Homogenitätskriterium H wird nicht wie gewöhnlich ein Toleranzbereich angegeben. Vielmehr werden ein oberer und ein unterer Schwellwert festgelegt. Diese Werte betragen 150 HE und 2000 HE. Alle Voxel, deren Grauwerte innerhalb der Schranken liegen, erfüllen H. Es bleiben hier nur Knochen und Gefäße über, weil die Dichtewerte der Gefäßinhalte aufgrund von Kontrastmittel denen der Knochen im vorliegenden Datensatz gleicht. Der daraus resultierende reduzierte Datensatz P wird zunächst gesichert.

Dilatation und Erosion Es folgen zwei morphologische Operatoren, die hintereinander auf den segmentierten Bereich angewandt werden und diesen verändern: Dilatation (*dilation*) und Erosion (*erosion*). Beide sind wie folgt definiert:

$$(f \oplus b)(x,y) = max\{f(x+s,y+t) \wedge b(s,t)\}\text{, Dilatation} \quad (1)$$
$$(f \ominus b)(x,y) = min\{f(x+s,y+t) \wedge b(s,t)\}\text{, Erosion} \quad (2)$$

Für beide Gleichungen gilt: $f(x,y), b(s,t) \in \{0,1\}$. Die Pixel $f(x,y)$ werden von einer Maske $b(s,t)$ durchlaufen. Beim $\oplus$-Operator werden alle Pixel $f(x,y)$ auf den Maximalwert gesetzt, der aus einer UND-Verknüpfung der Maskenpixel mit den Bildpixeln $f(x,y)$ resultiert. Dadurch werden Punkte an den Rändern der Objekte in Abhängigkeit der Form des strukturierenden Elements hinzugefügt. Kleine Verzerrungen und Lücken werden geschlossen, es gehen Informationen verloren. Nahe beieinander liegende Objekte werden gegebenenfalls miteinander verbunden.
Beim gegensätzlichen $\ominus$-Operator werden die $f(x,y)$ nur dann erhalten, wenn die UND-Verknüpfung aller in der darüberliegenden $b(s,t)$ eingetragenen Werte mit $f(x+s,y+t)$ eine 1 ergeben. Die Erosion verkleinert und zerlegt Regionen. Kleine Details gehen verloren, wodurch auch hier Informationen verworfen werden. Die Resultate beider Operatoren werden in Abbildung 3 illustriert.

Interaktive Nachbearbeitung Die einzelnen Knochenfragmente werden als Nächstes freigestellt. Dazu werden in MeVisLab die Originaldaten O und der auf Knochen und Gefäße reduzierte DICOM-Datensatz P geladen. Beide werden nun durch einen Aufbau diverser MeVisLab-Module derart miteinander verknüpft, dass letztlich alle im reduzierten Datensatz P enthaltenen Voxel farblich markiert transparent über dem vollständigen Originaldatensatz O liegen.
Um nun einzelne Knochenfragmente zu extrahieren, werden die durch P farbig markierten Teilflächen pro Ebene, die das zu extrahierende Objekt überlagern, isoliert und gegebenenfalls korrigiert. Anschließend wird ein visuell freigestelltes Volumen in P mit Hilfe des Region-Growing-Algorithmus extrahiert und im DICOM-Format abgespeichert.

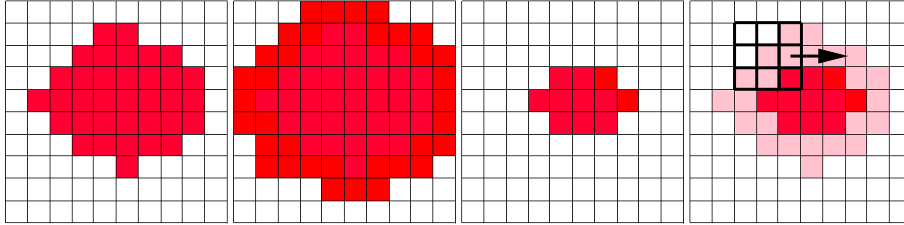


Abbildung 3: Beispiel für Erosion und Dilatation. Im linken Bild ist die Originalregion zu sehen. Im 2. Bild ist die Region durch eine Dilatation erweitert worden. Das 3. Bild zeigt die Region nach einer Erosion. Das rechte Bild visualisiert die 3×3-Maske, mit der das Bild jeweils durchlaufen wurde.

Die Dauer der Segmentierung variiert je nach Komplexität der vorliegenden Fraktur und Vorwissen des Anwenders und dürfte im Bereich von zwei bis vier Stunden liegen, wobei hier keine Referenzdaten zur Verfügung stehen. Dabei ist zu erwähnen, dass hochwertige Modelle von Knochenfragmenten bei einer komplizierten Fraktur nur mit Hilfe eines ausgebildeten Anwenders erzeugt werden können. Für einen Anwender, der wenig Wissen über die Anatomie des menschlichen Körpers besitzt und zudem keine Erfahrung beim Interpretieren von medizinischen CT-Schichtbildern besitzt, scheint es nahezu unmöglich, ein präzises 3D-Modell der Fraktur zu erstellen.

2.2.3 Modellgenerierung

Wurden im vorherigen Schritt die Frakturelemente separiert, gilt es nun, aus deren Daten Oberflächenmodelle zu erzeugen (Triangulierung). Oberflächenmodelle bestehen meist aus einer Menge von je drei Punkten (*Vertices*), die ein Dreieck definieren. Dreiecke haben die Eigenschaft, dass sie immer konvex und planar sind. Jede Dreiecksfläche besitzt eine Normale, die bei der späteren Darstellung für die Beleuchtungseigenschaften von Bedeutung ist.
Für die Oberflächengenerierung wird der *Marching-Cubes*-Algorithmus [LC87] genutzt (siehe [Han00]), der durch Verwendung von jeweils acht Voxeldaten Oberflächendreiecke erzeugt. Ein Voxel $\vec{v} = (x, y, z)$ wird nicht durch ein Volumen, sondern durch einen Punkt repräsentiert, dem im vorliegenden Fall ein Grauwert $f(x, y, z)$ zugeordnet ist. Aus dem vorhandenen Volumendatensatz entsteht durch die veränderte Art der Betrachtung ein Punktgitter.

Marching-Cubes zeichnet sich dadurch aus, dass das Problem der Oberflächenerzeugung aus acht Voxeldaten auf die Analyse von 15 lokalen Voxelkonfigurationen reduziert wird (*divide-and-conquer-approach*, siehe Abb. 4). Die Reduktion von 256 auf 15 Fälle ist durch die Symmetrieeigenschaften Rotation, Spiegelung und Inversion des Würfels möglich.
Mit einem Schwellwert $t \in \mathbb{R}$ wird eine Isofläche I_t beschrieben, die sich aus der Menge der Oberflächenpunkte zusammensetzt:

$$I_t = \{(x, y, z) | f(x, y, z) = t\} \tag{3}$$

Ob ein Voxel nun innerhalb oder außerhalb des Objektes, also hinter oder vor der Objektoberfläche liegt, wird in dem 3D-Bilddatensatz durch den Schwellwert t wie folgt beschrieben:

$$B(x, y, z) = \begin{cases} 0, \text{ falls } f(x, y, z) > t \\ 1, \text{ falls } f(x, y, z) \leq t \end{cases} \quad (4)$$

Um die Oberfläche zu konstruieren, wird der gesamte Datensatz durchwandert. Hierbei sind immer zwei benachbarte Schichten involviert, aus denen 8 benachbarte Voxel, je vier pro Schicht, betrachtet werden. Die 8 Voxel, die wie oben beschrieben, als Punkt dargestellt werden, bilden einen Quader, dessen Eckpunkte sie darstellen. Wenn für zwei benachbarte Voxel gilt: $B_{v1} \neq B_{v2}$ liegt einer innerhalb und einer außerhalb der Oberfläche. Es wird zwischen ihnen anhand der Voxel-Werte $f(\vec{v}_1)$ und $f(\vec{v}_2)$ ein Stützpunkt e linear interpoliert:

$$\vec{e} = \frac{\vec{v}_1 + (t - f(\vec{v}_1))}{(f(\vec{v}_2) - f(\vec{v}_1)) \cdot (\vec{v}_2 - \vec{v}_1)} \quad (5)$$

Punkt e wird sich auf der zu konstruierenden Oberfläche befinden. Durch die lineare Interpolation wird eine Glättung der Objektoberfläche erzielt. Mittels der 15 möglichen Konfigurationen (siehe Abb. 4) werden dann die Oberflächenpolygone mitsamt den Oberflächennormalen berechnet.

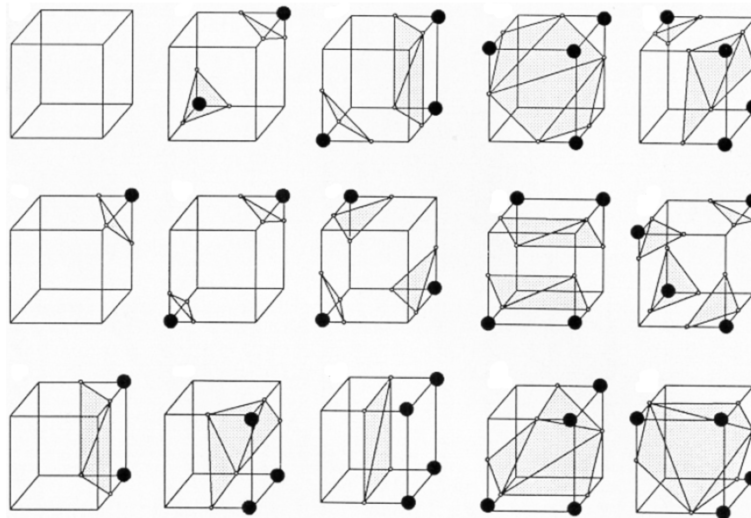


Abbildung 4: Die 15 lokalen Voxelkonfigurationen. Die schwarzen Eckpunkte repräsentieren Voxel mit $B(x, y, z) = 1$. Die eingezeichneten Oberflächen sind bereits trianguliert. Quelle: [LC87]

Ein Problem beim ursprünglichen *Marching-Cubes* ist unter anderem die Möglichkeit von Mehrdeutigkeiten und Inkonsistenzen, wodurch es zu Löchern in einer erzeugten Oberfläche kommen kann (siehe Abb. 5). Zahlreiche verschiedene Erweiterungen beheben diese Probleme. Eine einfache Lösung wäre die konsistente Anwendung bestimmter Regeln: Die Bedingung, dass Voxel mit $B(x, y, z) = 1$

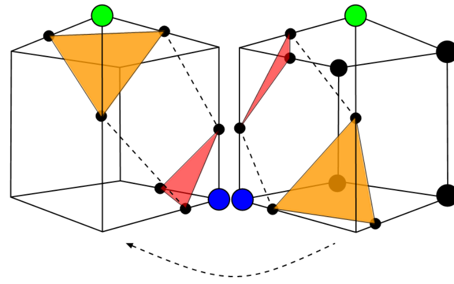


Abbildung 5: Beispiel einer Inkonsistenz, die bei der Triangulierung auftreten kann. Links wird trennend, recht verbindend trianguliert. Die gestrichelten Linien stellen das daraus resultierende Loch dar.

stets verbindend trianguliert werden, lässt zum Beispiel keine Inkonsistenzen zu.

Um letztlich aus den Daten tatsächlich eine Oberfläche zu erstellen, wurde der *mihSurfaceGenerator* verwendet. Dabei handelt es sich um eine institutseigene Anwendung, in die der *Marching-Cubes*-Algorithmus implementiert wurde. Eine mögliche Alternative bietet VTK mit seinem `vtkMarchingCubes`-Filter, der ebenfalls eine Oberflächengenerierung durchführt.
Die Modelle werden in einem Datenformat von VTK gesichert (.vtk). Die Dateien enthalten die notwendigen Daten für ein Modell, folglich alle Punktkoordinaten, die Punktindizes pro Fläche und den Normalenvektor pro Fläche. Ein erstes Beispiel illustriert die Abbildung 6.

2.2.4 Erzeugung von Schnittkonturen

Um 3D-Objekte innerhalb von 2D-Schichtenansichten darzustellen, können sie durch ihre Schnittkonturen repräsentiert werden. In Abbildung 6 wird der Verwendungszweck deutlich. Schnittkonturen entsprechen einer Abbildung von 3D in den 2D-Raum. Sie können anhand von impliziten Funktionen erstellt werden. Implizite Funktionen haben die Form

$$F(x, y, z) = c \ ,$$

wobei c eine frei wählbare Konstante ist. Implizite Funktionen weisen drei wichtige Merkmale auf:

- Sie ermöglichen die einfache geometrische Beschreibung von üblichen Objekten wie zum Beispiel Kugeln: $F(x, y, z) = x^2 + y^2 + z^2 - R^2$, wobei R den Kugelradius angibt.
- Sie unterteilen einen euklidischen Raum in drei disjunkte Bereiche. Diese Bereiche liegen innerhalb ($F(x, y, z) < 0$), auf ($F(x, y, z) = 0$) und außerhalb ($F(x, y, z) > 0$) der impliziten Funktion. Die verschiedenen Regionen können anhand der Funktion bestimmt werden.
- Sie können Raumpositionen in Skalare umwandeln. Werden die drei Koordinaten einer Position im Raum in die Funktion eingesetzt, zeigt das Ergebnis den Abstand zum Funktionskörper an.

Im Folgenden wird beschrieben, wie die Erstellung von Objektkonturen in VTK implementiert ist. Es wird davon ausgegangen, dass das 3D-Objekt, von dem die

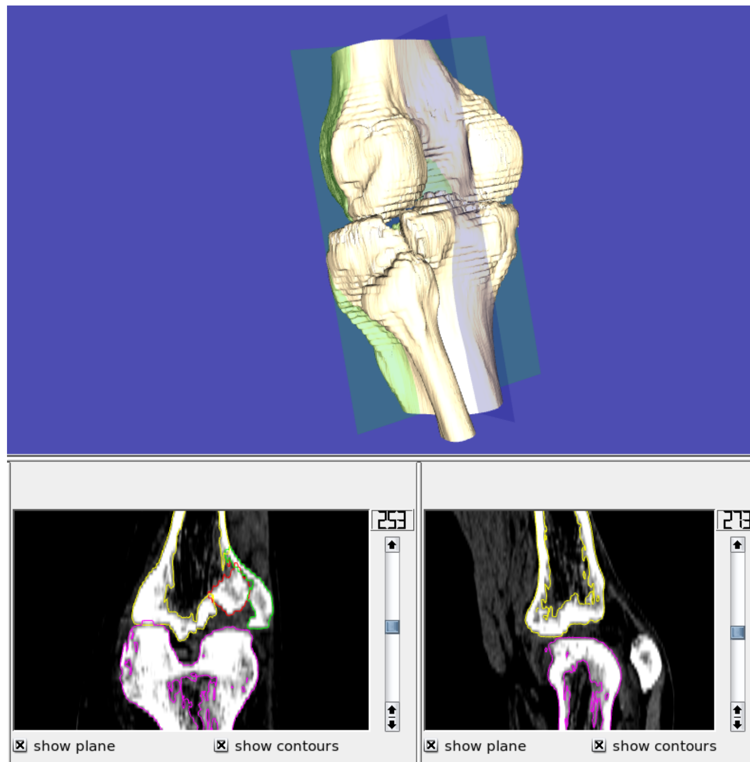


Abbildung 6: Ein Teil-Screenshot der erstellten Anwendung. Dieses Bild zeigt, in wiefern Schnittkonturen in der Applikation genutzt werden. Die Schnittebenen werden durch die transparent gezeichneten Ebenen in der 3D-Szene illustriert. Zugleich sind 3D-Modelle der Knochen zu sehen, die wie zuvor beschrieben aus den CT-Daten erstellt wurden.

Kontur erstellt werden soll, vollständig trianguliert ist.
Um die Schnittkontur anhand einer impliziten Funktion F zu ermitteln, müssen zunächst für alle Punkte des Objekts die skalaren Funktionswerte $\vec{v}$ ermittelt und zugewiesen werden. Sie repräsentieren den Abstand des Punktes zum Funktionskörper. Pro Objektdreieck werden nun dessen Linien betrachtet. Anhand v pro Linienendpunkt kann ermittelt werden, ob die Linie von F geschnitten wird. Ein Schnitt liegt dann vor, wenn einer der $\vec{v}$-Werte negativ, der andere positiv ist. Ist dies der Fall, wird durch Interpolation ein neuer Punkt auf der Dreiecks-Linie erstellt. Die Interpolation erfolgt dabei anhand der skalaren Werte $\vec{v}$ beider Punkte:

$$\vec{p}_{neu} = \vec{p}_1 + t \cdot (\vec{p}_2 - \vec{p}_1)\,, t = \frac{\vec{v}_{p_1}}{(\vec{v}_{p_1} - \vec{v}_{p_2})}$$

Dreieckspunkte, die bereits auf dem Funktionskörper liegen ($v = 0$), repräsentieren ebenfalls einen neuen Punkt p_{neu}.
Die erstellte Punktmenge muss noch durch Linien zu einer Kontur verbunden werden. Es werden stets zwei der neuen Punkte verbunden, die mit den Daten aus einem Dreieck erstellt wurden.

3 Kollisionserkennung

Bei einer Kollisionserkennung zwischen Objekten geht es um das möglichst schnelle Erkennen von Objektüberschneidungen innerhalb einer definierten Szene. Dabei existieren zwei Kategorien von Kollisionserkennungen, nämlich das statische und das dynamische Kollisionserkennungsproblem. Im ersten Fall verändern die Objekte $k_1, \ldots, k_n$ in der Welt $W = \{k_1, \ldots, k_n\}$ ihre Position nicht. Hierbei wird die Menge aller kollidierenden Objektpaare ermittelt durch:

$$S := \{(k_i, k_j) \in W \times W | k_i \cap k_j \neq 0\} \tag{6}$$

Beim dynamischen Problem werden die Objekte innerhalb der Szene W transformiert. Gesucht wird der Zeitpunkt $t_{col}, t_{col} > t_0$, bei dem mindestens zwei Objekte miteinander kollidieren. Folgendes gilt:

$$\begin{aligned} k_i(t_0) \cap k_j(t_0) &= 0, 1 \leq i < j \leq n & (7) \\ \forall t < t_{col} \, \forall k_i, k_j \in W : k_i(t) \cap k_j(t) &= 0 & (8) \\ k_i(t_{col}) \cap k_j(t_{col}) &\neq 0 & (9) \end{aligned}$$

Die Menge aller kollidierenden Paare zum Zeitpunkt t_{col} wird bestimmt durch:

$$S := \{(k_i, k_j) \in W \times W | k_i(t_{col}) \cap k_j(t_{col}) \neq 0\} \tag{10}$$

In beiden Problemfällen müsste bei n beweglichen Objekten $(n^2 - n)/2$ mal oder bei n beweglichen und m festen Objekten $\binom{n}{2} + nm$ mal eine teure Kollisionsüberprüfung für je zwei verschiedene Objekte stattfinden, was in Echtzeit nicht mehr realisierbar ist. Daher gibt es verschiedene Ansätze, bei denen die Anzahl der auf Kollision zu testenden Objektpaare stark reduziert wird. Diese Phase wird mit *broad phase* bezeichnet. Es findet ein Objekt-Culling statt. Das kann zum Beispiel durch approximierende Hüllkörper (Bounding Volumes) oder eine Raumunterteilung geschehen.

Die Phase der detaillierten und daher aufwändigen Kollisionstests für zwei Objekte wird mit *narrow phase* bezeichnet. Für beide Phasen werden einige Ansätze beschrieben, wobei eine strikte Klassifikation der Verfahren auf diese beiden Phasen nicht möglich ist.

In [Len00] werden die Kollisionserkennungsverfahren in drei Kategorien unterteilt. Die *statische Kollisionserkennung* überprüft alle in einer Szene vorhandenen Objekte, die dabei nicht transformiert werden. Die *pseudodynamische Kollisionserkennung* überprüft alle Objekte einer Szene in diskreten Zeitintervallen. Dabei kann es passieren, dass Kollisionen zwischen zwei der diskreten Zeitpunkte stattfinden, wodurch sie gänzlich übersehen oder erst spät erkannt werden. Die *dynamische Kollisionserkennung* geschieht kontinuierlich. Sie bedingt Informationen über die nachfolgenden Transformationen, also den Bewegungsverlauf der Objekte. Auch hier werden zu diskreten Zeitpunkten die Objekte auf Kollision hin überprüft. Die Kontinuität wird durch die überstrichenen Objektvolumina hergestellt, die bei der Bewegung der Objekte entstehen. Diese Volumina werden neben den aktuellen Objektvolumen bei der Kollisionsbehandlung mit einbezogen.

Im Rahmen dieser Arbeit werden nur statische bzw. pseudodynamische Verfahren

näher betrachtet. Ein Grund dafür sind die nicht vorhandenen Informationen über die nachfolgenden Transformationsformen zu den Zeitpunkten $t+n$ eines Objekts. Durch die anwenderspezifische interaktive Steuerung der Objekte können Transformationen nicht, wie z.B. bei der Bewegung von Billardkugeln, vorherbestimmt werden. Die dynamische Kollisionserkennung benötigt derartige Informationen.

Für viele der vorgestellten Lösungsansätze spielten zeitliche und räumliche Kohärenz eine wichtige Rolle. Objekte werden innerhalb eines kleinen Zeitintervalls nur wenig transformiert, was bedeutet, dass sich ihre geometrischen Beziehungen währenddessen zueinander nur minimal verändern. In diesem Fall kann die zeitliche Kohärenz ausgenutzt werden. Die räumliche Kohärenz sagt aus, dass Objekte sich innerhalb eines Zeitintervalls nur in einem bestimmten Raum bewegen, andere Räume hingegen komplett ausgeschlossen werden können.

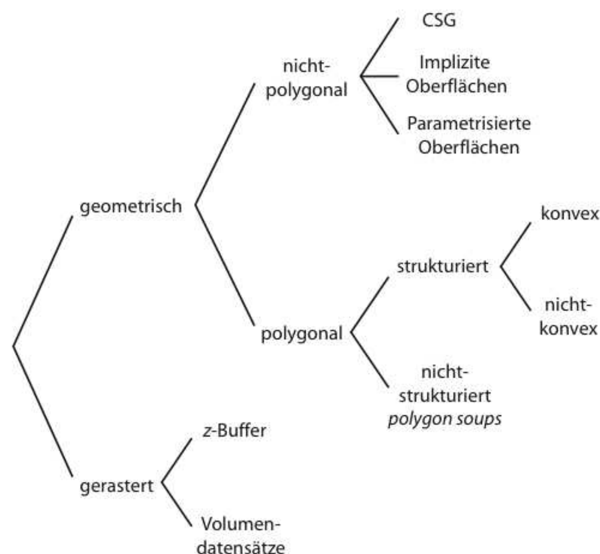


Abbildung 7: Unterschiedliche Objektrepräsentationen für die Kollisionserkennung. Die für diese Arbeit relevante Gruppe ist die der geometrischen, polygonalen und nicht strukturierten Objekte. Quelle: [ML98]

3.1 Allgemeine Anforderungen

Die allgemeinen Anforderungen an ein Kollisionsbehandlungssystem sind hoch:

- Beliebig viele Objekte, bestehend aus unterschiedlichen und komplexen Strukturen, müssen gegeneinander auf Kollision getestet werden. Die Modelle können dabei bezüglich ihrer Position und Struktur alle Freiheiten aufweisen. Löcher und andere Unregelmäßigkeiten in Modellen sind dabei keine Seltenheit.

- Der Test auf Kollision muss in Echtzeit erfolgen. Diese Anforderung wird dadurch erschwert, dass die Objekte in vielen Fällen vom Benutzer indivi-

duell transformiert werden, was eine Vorberechnung aufgrund von Schätzungen unmöglich macht.

- Bei Kollision sollen der genaue Kollisionspunkt bzw. die kollidierenden Bereiche ausgegeben werden können, um diese für eine Weitervearbeitung, z.B. für die farbliche Markierung, zu nutzen.

- Der Abstand zwischen nicht-kollidierenden Objekten pro Paar soll, genauso wie die Kollisionstiefe, zurückgegeben werden können.

3.2 Anwendungsspezifische Anforderungen

Nicht alle der allgemeinen Anforderungen sind für die hier vorgestellte Anwendung notwendig. Dafür kommen neue Anforderungen hinzu. Folgende Punkte müssen bei der Auswahl des Verfahrens für die Anwendung berücksichtigt werden:

- Bei den Objekten handelt es sich ausschließlich um *polygon-soups*. Dabei kann ein Objekt aus über 100.000 Polygonen bestehen. Die Polygone, die ein Objekt besitzt, haben keinerlei Strukturen oder Beziehungen zueinander. Eine Vorverarbeitung zur Simplifizierung der Polygonmengen ist nicht erwünscht, da durch diesen Schritt möglicherweise wichtige Fragmentdetails verloren gehen. Zudem ist der Schritt sehr zeitintensiv.

- Die Kollisionsbehandlung muss in Echtzeit erfolgen. Eine Integration von haptischen Ein- und Ausgabegeräten ist im Rahmen dieser Arbeit nicht vorgesehen, aber durchaus eine wünschenswerte Erweiterung. Eine derartige Erweiterung setzt eine Frequenz von minimal 1000 Hz voraus.

- Kollidierende Objektpolygone sollen ermittelbar sein, so dass sie zum Beispiel farblich markiert werden können. Eine Abstandsberechnung zwischen zwei nicht kollidierenden Objekten ist vorerst nicht notwendig.

- Die Anzahl der kollidierenden Objekte kann auf circa 15 beschränkt werden, da kaum Brüche mit mehr als 15 Fragmenten vorkommen. Nach einer Segmentierung der Objekte aus den CT-Daten liegen diese eng beieinander, teilweise werden sie sich von vornherein überschneiden, was für eine Bounding-Volume-Box mit hoher Präzision spricht.

- Die Transformationen werden gering sein, wodurch räumliche und zeitliche Kohärenz ausgenutzt werden kann. Zudem wird die Position zu einem Zeitpunkt t immer für ein oder mehrere Objekte vom Anwender mit der gleichen Transformation verändert. Skalierung als affine Transformation kann für diese Anwendung ausgeschlossen werden.

- Die Transformationen werden vom Benutzer individuell festgelegt, wodurch *dynamische Kollisionserkennung* in dieser Arbeit ausgeschlossen werden, denn diese benötigen vorher berechnete Transformationen für den nächsten Zeitschritt.

3.3 Raumunterteilung (broad phase)

Um den Suchraum kollidierender Objekte einzuschränken, besteht die Möglichkeit der globalen Raumunterteilung einer Szene. Objekte werden den einzelnen Volumeneinheiten des Gesamtraumes zugeordnet. Befinden sich zwei Objekte in ein und demselben Volumen, müssen sie für die nähere Kollisionsbehandlung markiert werden.
Dabei haben Methoden, die eine raumorientierte Volumenhierarchie erzeugen, den Vorteil, dass ein einfacher Test auf Kollision an einem Knoten im Baum gegebenenfalls den darunter liegenden Ast als nicht mehr relevant markieren kann. Sie müssen diesen nicht mehr in die Kollisionsbehandlung mit einbeziehen. Allerdings erfordern diese Volumenhierarchien bei Transformation von Objekten der Szene jedesmal eine Aktualisierung.

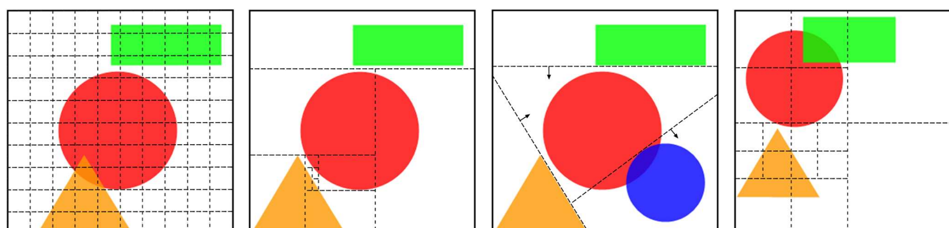


Abbildung 8: Im linken Bild wird der Raum durch *uniform-grid* aufgeteilt, im zweiten Bild anhand eines *kd-trees*. Das dritte Bild illustriert eine Aufteilung mittels *BSP-tree* und das rechte Bild zeigt eine Raumaufteilung durch einen *Oc-tree*.

3.3.1 Uniforme Raumunterteilung

Bei der uniformen Raumunterteilung (*uniform-grid*) wird der Raum in uniforme Voxel unterteilt. Es entsteht keine Hierarchie. Befinden sich Teile eines Objekts in einem dieser Voxel, wird das vermerkt, bei mehr als einem Eintrag pro Voxel muss dann auf Kollision getestet werden. Dieser Ansatz ist einfach, minimiert den Suchraum aber unzureichend.

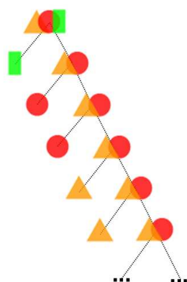


Abbildung 9: Die zum *kd-tree* gehörige Baum-Hierarchie

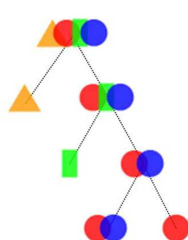


Abbildung 10: Die zum *BSP-tree* gehörige Baum-Hierarchie

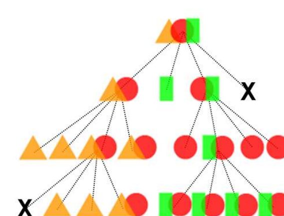


Abbildung 11: Die zum *Oc-tree* gehörige Baum-Hierarchie

3.3.2 KD-Bäume und-Binary-Space-Partitioning-Bäume

KD-Bäume (k-dimensional) sind ein Spezialfall von Binary-Space-Partitioning-Bäumen (*BSP-trees*) und teilen den Raum ungleichmäßig durch achsenparallele Trennebenen auf (nicht-uniforme Raumpartitionierung). *BSP-trees* hingegen dividieren den Raum durch Trennebenen beliebiger Richtung. Beide Verfahren erzeugen damit eine Hierarchie, wobei die erste Ebene durch den gesamten Raum geht. Alle Subebenen enden entweder am Rand der Szene oder auf der Elternebene.
Mit der Auswahl der Ebenen kann der Aufbau bzw. die Effizienz der Hierarchie beinflusst werden. So kann zum Beispiel darauf geachtet werden, dass die Ebenen möglichst keine Objekte schneiden und die Baumebenen ausbalanciert sind. Die Orientierung der Ebenen wird im BSP-Ansatz oftmals durch die Oberflächenpolygone der Objekte bestimmt.

3.3.3 Oktonärbäume

Oktonärbäume (*Oc-trees*) teilen den Raum iterativ gleichmäßig durch drei achsenparallele Trennebenen auf, wodurch eine Hierarchie entsteht. Im Baum entspricht das gesamte Volumen der Wurzel, die Knoten den Subvoxeln und die Blätter den Objekten, die in den Voxeln enthalten sind. Jeder Knoten hat demnach maximal acht Kinder. Voxel, in denen keine Objektelemente enthalten sind, werden nicht weiter unterteilt und auch nicht in den Baum eingetragen. Bei ungleichmäßiger Objektverteilung innerhalb der Szene ist ein *Oc-tree* durchaus sinnvoll. Ein Vorteil ist der einfache Aufbau der Datenstruktur, die automatisiert werden kann. Nachteilig ist, dass jede Hierarchieebene das Objekt unzulänglich umschließt.

3.4 Hüllkörper (broad phase)

Wollte man Objekte, die aus einer *polygon-soup* bestehen, auf Kollision testen, müsste man aufwändige Tests mit den Punkten, Kanten und Flächen (Features) der Objekte durchführen, unabhängig von ihrer Position in der Szene. Eine starke Vereinfachung ist es, die Objekte mit einer primitiven, konvexen, geschlossenen Form zu umhüllen, um ein Objekt-Culling im Vorfeld durchzuführen, denn ein Kollisionstest mit geometrisch einfachen Körpern ist schnell vollführt. Es gibt verschiedene Formtypen, die alle Vor- und Nachteile aufweisen. Die Anforderungen, die an solche Hüllkörper gestellt werden, sind vielfältig:

- Komplexität: Wie groß ist der Speicherbedarf zur Repräsentation und Konstruktion der Hülle?
- Kollisionserkennung: Wie schnell ist die Berechnung der Kollision bzw. der Schnittpunkt- oder Abstandsbestimmung?
- Präzision: Wie genau entsprechen Volumen und Form der Hülle den Eigenschaften des eigentlichen Objekts?
- Stabilität: Ist die Repräsentation durch den Hüllkörper stabil gegenüber Transformation?

Die Kosten F, die bei der Kollisionsbestimmung zweier Objekte entstehen, lassen sich mit folgender Formel beschreiben:

$$F = N_u \cdot C_u + N_v \cdot C_v + N_p \cdot C_p$$

N_u ist der aktualisierte Hüllkörper, C_u die Kosten der Aktualisierung von Hüllkörpern, N_v die Anzahl Tests, ob Objekte sich berühren oder überschneiden, C_v die Kosten für diese Durchdringungstests, N_p die Anzahl der Durchdringungstests für Primitive und C_p die Kosten der Durchdringungstests für Primitive.
Es gibt unterschiedliche Strategien, um den Aufwand der Berechnung zu minimieren. Wird die Präzision der Hüllkörper optimiert, sinken die Kosten für N_v und N_p. Beschleunigt man den einfachen Durchdringungstest, wird C_v gesenkt. Und wird die Aktualisierung von Hüllkörpern vermieden, können Kosten eingespart werden, die durch N_u entstehen.

Hüllkörper umschließen zunächst ein Objekt immer vollständig. Wird bei einem Test auf Kollision ermittelt, dass der Hüllkörper mit einem anderen kollidiert, muss ein aufwändiger Kollisionstest zwischen allen Polygonen zweier Objekte durchgeführt werden. Ein weiterer Kollisionsbehandlungsansatz wäre, nicht nur einen Hüllkörper für das gesamte Objekt zu erzeugen, sondern zudem Teilstücke des Objekts wiederum durch Hüllkörper zu umschließen, wodurch die Präzision steigt. Wird diese Unterteilung mehrfach durchgeführt, entsteht am Ende eine Hüllkörperhierarchie, die durch eine Baumstruktur repräsentiert werden kann. Die Baumwurzel ist der globale Hüllkörper, der das gesamte Objekt umschließt, die Knoten stehen für Hüllkörper, die Teilbereiche des Objekts umschließen und in den Blättern sind die Objektpolygone zu finden. Die einzelnen Hierarchieebenen umschließen insgesamt das Objekt. Dabei steigt durch eine *top-down*-Traversierung die Präzision.

Testet man mit Hilfe von Bounding-Volume (BV) -Hierarchien Objekte auf Kollision, wird zunächst mittels der obersten Volumen in den Hierarchien begonnen. Ist der Test positiv, werden die Bäume von oben nach unten traversiert und der Test wird mit den Sub-Hüllkörpern wiederholt. Ein aufwändiger Kollisionstest muss nur dann für die Objektpolygone durchgeführt werden, die einem Baumknoten bzw. -blatt zugeteilt sind, bei dem eine Kollision vermutet wird. Alle im Folgenden vorgestellten Hüllkörpertypen lassen sich zusammen mit Hierarchien verwenden.

Um eine BV-Hierarchie aufzubauen, kann auf unterschiedliche Weise vorgegangen werden. Dabei spielen Faktoren wie Breite, Tiefe und Balancierung der Bäume, Zerlegungsfaktoren der Hüllkörper und Aufbaurichtung für die Performanz eine Rolle. Ein Hüllkörper soll eine hohe Präzision aufweisen. Zudem ist die Eigenschaft der Konvergenz beim Aufbau wichtig. Sie sagt aus, wie schnell sich die Teilhüllkörper an das Objektfragment anpassen.

Beim *bottom-up*-Verfahren werden einzelne oder mehrere benachbarte Polygone durch einen Hüllkörper umschlossen, so dass insgesamt alle Objektpolygone durch Volumen umschlossen sind. Anschließend werden minimal zwei dieser Hüllkörper wiederum von einem Hüllkörper umschlossen. Dieser Aufteilungsprozess kann beliebig fein vollführt werden, bis hin zur Wurzel der Hierarchie.
Beim *top-down*-Verfahren läuft es genau umgekehrt. Zunächst wird die gesamte Polygonmenge durch einen Hüllkörper umschlossen. Ein Volumen wird nun sukzessive in zwei Volumen unterteilt, wobei jedesmal eine neue Hierarchieebene mit Knoten erstellt wird. Entweder endet der Vorgang in einer bestimmten Baumtiefe

oder wenn in einem Blatt nur noch ein Objektpolygon enthalten ist.
Die *top-down*-Methode wird in der Praxis häufiger verwendet. Welche Aufbauart einen für die Anwendung besseren Hierarchiebaum erstellt, kann ohne eine genaue Untersuchung nicht gesagt werden. Zu dem Thema wurden im Rahmen dieser Diplomarbeit keine anderen Studien entdeckt.

Im Folgenden werden die fünf gängigen Hüllkörpertypen, Kugeln (Spheres), Axis-Aligned-Bounding-Boxes (AABB), Object-Oriented-Bounding-Volumes (OBB), k-Discrete Oriented Poyleder (k-dop) und Swept Sphere Volumes (SSV) in Bezug auf die oben genannten Anforderungen untersucht. Außerdem wird eine Möglichkeit beschrieben, wie eine Hierarchie für den jeweiligen BV-Typ erstellt werden könnte. Zunächst jedoch wird das *Separating-Axis-Theorem* beschrieben, das in einigen der Kollisionserkennungen mittels Bounding-Volumes verwendet wird.

3.4.1 Separating-Axis-Theorem

Für die Kollisionserkennung von Bounding Volumes wird häufig das *Separating-Axis-Theorem* (SAT) verwendet. Es kann sowohl in der *broad phase* als auch in der *narrow phase* genutzt werden.

Definition: *Zwei konvexe Polyeder A und B sind genau dann disjunkt, wenn es eine Trennachse gibt, die entweder zu einer Seitenfläche von A, einer Seitenfläche von B oder zu jeweils einer Kante von A und B orthogonal ist.*

Das sind zum Beispiel bei OBB-Hüllkörpern ingesamt 15 Achsen (eine Achse pro OBB-Fläche und 9 Achsen pro Kantenpaar, errechnet mit dem Kreuzprodukt der Kanten), wobei mit den Tests anhand der Seitenflächen der Hüllkörper begonnen werden sollte. Bei AABB-Boxen sind drei Tests nötig. Dieser Satz wurde in [Got96] bewiesen.

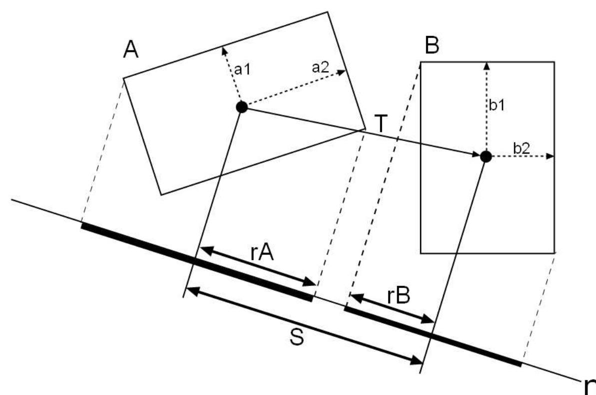


Abbildung 12: Zur Veranschaulichung eine der 15 Achsen, die beim *Separating-Axis-Theorem* verwendet werden. Als Hüllkörpertyp werden im Beispiel OBBs angewandt.

Mathematisch lässt sich eine Trennachse am Beispiel der OBB-Bounding-Volumes

L wie folgt bestimmen:

$$s > r_A + r_B,\ r_A = \sum_{i=1}^{3} |\langle a_i, n\rangle| \text{ und } r_B = \sum_{i=1}^{3} |\langle b_i, n\rangle| \quad (11)$$

$$|\langle T, n\rangle| > \sum_{i=1}^{3} |\langle a_i, n\rangle| + \sum_{i=1}^{3} |\langle b_i, n\rangle| \quad (12)$$

$$T \cdot n/|n| > (a_1|r_1^A \cdot n| + a_2|r_2^A \cdot n| + a_3|r_3^A \cdot n|)/|n| + (b_1|r_1^B \cdot n| + b_2|r_2^B \cdot n| + b_3|r_3^B \cdot n|)/|n| \quad (13)$$

T ist der Abstand zwischen Objektmittelpunkten, s ist dessen Projektion auf die Trennachse n, r ist die halbe Länge eines Intervalls auf der Trennachse. Wird das SAT für die Überprüfung auf Kollision genutzt, kann lediglich festgestellt werden, ob eine Kollision vorliegt. Ausgabe der Distanz oder Penetrationstiefe bietet dieser Ansatz nicht.

3.4.2 Kugel als Hüllkörper

Mit Hilfe einer Kugel lässt sich ein Objekt leicht umschließen. Es wird ein Objektschwerpunkt ermittelt, von dem aus der größte Abstand eines Objektfragments den Kreisradius bestimmt. Dieser initiale Schritt kann je nach Objekteigenschaft sehr komplex werden.
Eine Möglichkeit, einen sinnvollen Mittelpunkt zu finden, ist das Bilden einer Menge P von Polygonpunkten, die Bestandteil von Oberflächenpolygonen des Objekts sind. Eckpunkte von Polygonen, die innerhalb des Objekts liegen, werden nicht weiter betrachtet. Anhand dieser Punktemenge wird der Schwerpunkt und damit der Mittelpunkt $\vec{M}$ einer Hülle für das gesamte Objekt berechnet. Der Radius ist schnell gefunden, indem der maximale Abstand von $\vec{M}$ zu einem Punkt aus P gesucht wird. Eine Kugel als Hüllkörper hat eine geringe Präzision. Als Beispiel hierfür dient eine beliebige Achse im Raum.

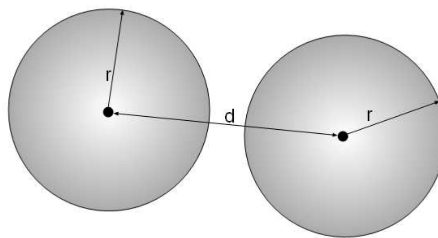


Abbildung 13: Distanz- bzw. Kollisionsbestimmung bei zwei Kugel-Hüllkörpern anhand der Radii und der Distanz zwischen den Mittelpunkten

Die Kollisionsbehandlung ist trivial. Es muss lediglich der Abstand der Kugelmittelpunkte mit der Summe ihrer Radii verglichen werden. Die Stabilität ist optimal, weil sich der Hüllkörper durch Translation und Rotation nicht verändert. Aufgrund seiner Eigenschaften wird diese Methode häufig bei Kollisionsbehandlung in einem ersten Vorverarbeitungsschritt verwendet.

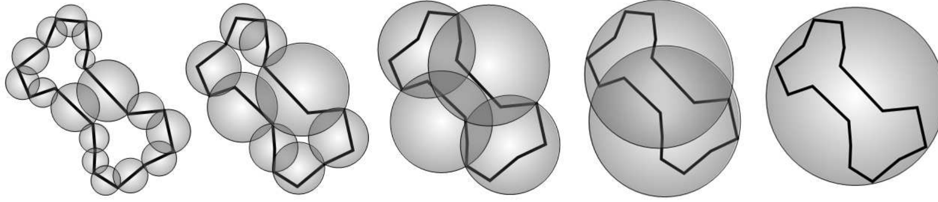


Abbildung 14: Kugel als Hüllkörper. Eine Hierarchie könnte, wie im Bild von links nach rechts zu sehen, *bottom-up* erstellt werden.

Eine Hierarchie könnte *bottom-up* erstellt werden. Aus der anfangs erwähnten Punktmenge P wird ein Voronoi-Diagramm erzeugt. Die Eckpunkte eines 3D-Voronoi-Gerüsts haben die Eigenschaft, Mittelpunkt einer Kugel zu sein, auf dessen Hülle die umliegenden Punkte liegen (siehe dazu Abschnitt 3.5.2, Medial Axis). Diese Aufteilung stellt die unterste Ebene in einem Hierarchiebaum (*Sphere-Tree*) dar. Durch Zusammenfassen dieser Kugeln können weitere Hierarchiestufen hinzugefügt werden, bis hin zu der Kugel, die das gesamte Objekt umschließt und somit die Baumwurzel repräsentiert.

3.4.3 Axis Aligned Bounding Box (AABB)

Um ein Objekt einfach zu umschließen, kann ein Quader benutzt werden, dessen Seiten achsenparallel zum Weltkoordinatensystem (WKS) ausgerichtet sind. Die Längen der Seiten bzw. die Größen der Seitenflächen werden durch die Ausmaße des zu umschließenden Objekts bestimmt.

$$A = X \times Y \times Z, \text{mit } X = [d_1^{min}, d_1^{max}], Y = [d_2^{min}, d_2^{max}], Z = [d_3^{min}, d_3^{max}] \quad (14)$$

Es lassen sich AABB-Hüllkörper mit fester und dynamischer Größe erstellen. Soll die AABB eine feste Größe haben, müssen Maxima und Minima für alle Objektorientierungen ermittelt werden, was zu einer unzureichenden Präzision führt. Bei der dynamischen Größe muss die Box nach jeder Rotation erneut berechnet werden.

Dieser Hüllkörpertyp hat sehr gute Komplexitäts- und befriedigende Stabilitätseigenschaften, wohingegen die Präzision sehr ungenau sein kann. Eine diagonal im Raum liegende Achse erhält einen viel zu großen Hüllkörper.
Der Durchdringungstest ist mit einigen wenigen Koordinatenvergleichen einfach und schnell erledigt. Hierbei werden die Hüllkörperseiten auf die drei WKS-Achsen projiziert. Die Hüllkörper von zwei Objekten $A_i = AABB(d_i^{min}, d_i^{max})$ mit $i = 1, 2$ kollidieren genau dann, wenn sich alle drei ihrer Projektionsintervalle überschneiden.

$$A_1 \cap A_2 \neq 0 \quad (15)$$
$$\Leftrightarrow \quad X_1 \cap X_2 \neq 0 \wedge Y_1 \cap Y_2 \neq 0 \wedge Z_1 \cap Z_2 \neq 0 \quad (16)$$
$$\Leftrightarrow \quad \forall j, 1 \leq j \leq 3 : d_{1j}^{min} \in [d_{2j}^{min}, d_{2j}^{max}] \vee d_{2j}^{min} \in [d_{1j}^{min}, d_{1j}^{max}] \quad (17)$$
$$\Leftrightarrow \quad \forall j, 1 \leq j \leq 3 : d_{2j}^{min} \leq d_{1j}^{min} \leq d_{2j}^{max} \vee d_{1j}^{min} \leq d_{2j}^{min} \leq d_{1j}^{max} \quad (18)$$

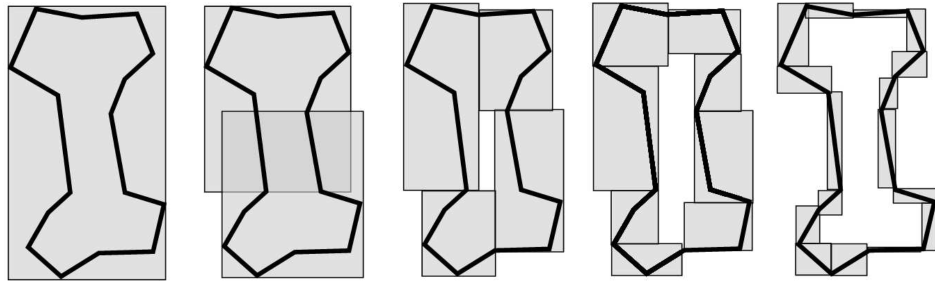


Abbildung 15: Achsenparallele Quader als Hüllkörper. Eine Hierarchie könnte, wie im Bild von links nach rechts zu sehen, *top-down* erstellt werden.

Das hier verwendete Prinzip beruht auf dem *Separating Axis Theorem* (3.4.1) und repräsentiert einen einfachen Fall dessen.

Sweep-and-Prune Eine Optimierung für diesen und alle anderen Hüllkörper, bei denen mit Projektionen auf Achsen gearbeitet wird, kann mit dem *Sweep-and-Prune* -Algorithmus realisiert werden [JC95]. Statt alle Hüllkörper-Projektionen pro Achse paarweise miteinander zu vergleichen, werden die drei Koordinatenachsen und deren natürliche Sortierung sinnvoll eingesetzt. Die Projektionsintervalle werden pro Achse in eine Liste $sort$ eingetragen, wobei die Reihenfolge aufsteigend durch deren untere Intervallgrenze bestimmt wird. Mit dem *Sweep-and-Prune*-Algorithmus und einer weiteren Liste $active$ können Projektionsintervalle, die sich überschneiden, mit einem Aufwand von $O(n\,log\,n)$ ermittelt werden, ohne alle Kombinationen durchzuspielen. n steht für die Anzahl der Hüllkörper.

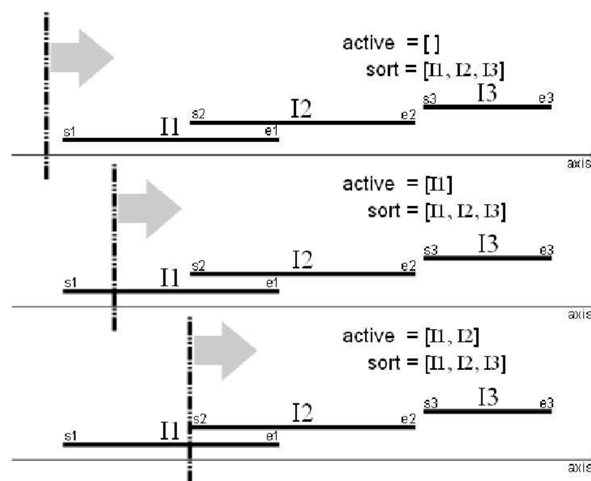


Abbildung 16: Ausnutzung der zeitlichen Kohärenz durch *Sweep-and-Prune*

Beginnend am Anfang von $sort$, wird ein Intervall I_1 in $active$ eingetragen, wenn

die Sweepline die untere Intervallgrenze erreicht hat. Ein Intervall I_1 wird aus $active$ ausgetragen, wenn ein neues Intervall I_2 hinein kommt, dessen untere Intervallgrenze größer ist als die obere Intervallgrenze von I_1. Alle Intervalle, die nach dem Eintragen von neuen und dem Austragen von alten Intervallen zusammen in der Liste $active$ stehen, überlappen sich.

Die Liste $sort$ muss anfangs mit einem Aufwand von $O(n \, log \, n)$ (Quick-Sort) neu aufgebaut werden, danach kann die zeitliche Kohärenz der Objektpositionen genutzt werden, um den Aufwand für den erneuten Aufbau bzw. das Sortieren auf bis zu $O(n)$ zu beschleunigen. Für die Sortierung bietet sich z.B. *Bubble-Sort* oder *Insertion-Sort* an. Verfahren, die *Sweep-and-Prune* verwenden, sind bekannt für ihre *n-body*-Lösungsansätze, also das Verwalten von mehr als zwei Objekten innerhalb der zu kontrollierenden Szene.
Für die AABB-Hüllkörper kann der Einfachheit halber ein *top-down* Verfahren zur Hierarchisierung verwendet werden. Die globale Box wird einheitlich unterteilt, bis maximal nur noch ein Objektfragment pro Box enthalten ist.

3.4.4 Object Oriented Bounding Box (OBB)

Um mit Hilfe der OBBs ein Objekt zu umschließen, wird ein Quader erzeugt, dessen Seiten nicht achsenparallel zum WKS sein müssen. Sie können beliebig gewählt werden. Die Präzision und damit die Konvergenzeigenschaft ist dadurch besser als bei der AABB- und Kugel-Variante, was bei Szenen mit mehreren eng aneinander liegenden Objekten von großem Vorteil ist [GLM96]. Auch ist eine optimale Stabilität gegeben, denn die Achsen, auf die später projeziert wird, hängen von den Objektorientierungen ab. Der Nachteil liegt hier bei der Komplexität. Die Erzeugung und die Kollisionsbehandlung sind teuer, der Speicheraufwand höher als bei den ersten beiden Varianten. Auch hier ist eine Ausnutzung der zeitlichen Kohärenz mittels *Sweep-and-Prune* möglich.

Die Erzeugung des Hüllkörpers ist aufwändig. Es muss zunächst die Ausrichtung bzw. Orientierung der OBB gefunden werden, die die höchste Präzision erlangt. Dazu wird die *Principle Component Analysis* (Hauptkomponenten-Analyse) aus der Statistik verwendet. Es wird der Schwerpunkt vom Objekt ermittelt, indem zum Beispiel die Eckpunkte der konvexen Hülle gemittelt werden. Punkte innerhalb des Objekts oder in konkaven Bereichen sind für den gesuchten Punkt störend und werden ignoriert. Als Beispiel wird das arithmetische Mittel μ verwendet:

$$\mu = \frac{\sum_{i=1}^{n} s^i}{n} \quad (19)$$

$$\mu = \frac{\sum_{i=1}^{n} (p^i + q^i + r^i)}{3n} \quad (20)$$

Hierbei ist n die Anzahl aller Dreiecke s, die zu der konvexen Oberfläche gehören. Ein Dreieck s hat die Eckpunkte p, q, und r. μ kann auch anders definiert werden, zum Beispiel indem das arithmetische Mittel der Dreiecksmittelpunkte, gewichtet mit der Größe der jeweiligen Dreiecksfläche, berechnet wird oder auch unter Verwendung der Kantendaten. Diese aufwändigeren Varianten sind oft eine bessere Näherung. Im Weiteren wird das bereits berechnete μ verwendet.
Mittels einer 3×3 Kovarianzmatrix C_{jk} kann eine Orientierung der OBB ermittelt

werden, denn sie repräsentiert die Ausdehnung des Objekts:

$$C_{jk} \quad = \quad \frac{\sum_{i=1}^{n}(\bar{p_j^i}\,\bar{p_k^i} + \bar{q_j^i}\,\bar{q_k^i} + \bar{r_j^i}\,\bar{r_k^i})}{3n},\ 1 \leq j,k \leq 3 \tag{21}$$

$\bar{p^i} = p^i - \mu$ steht für einen 3×1-Vektor $\bar{p^i} = (\bar{p_1^i}, \bar{p_2^i}, \bar{p_3^i})^T$. Aufgrund ihrer Definition ist eine reelle Kovarianzmatrix symmetrisch. Die Eigenvektoren stehen senkrecht zueinander und dienen fortan normalisiert als eine Basis. Mit Hilfe der Basis kann jetzt eine optimale Orientierung ermittelt werden. Entlang der Basisachsen werden die extremen Vertices bestimmt und eine Bounding Box so angelegt, dass auch diese extremen Punkte umschlossen werden. Die Achsen der Boxen liegen parallel zu den Basisachsen.
Für die Kollisionserkennung zwischen OBB-Boxen wird das in (3.4.1) beschriebene *Separating-Axis-Theorem* verwendet.

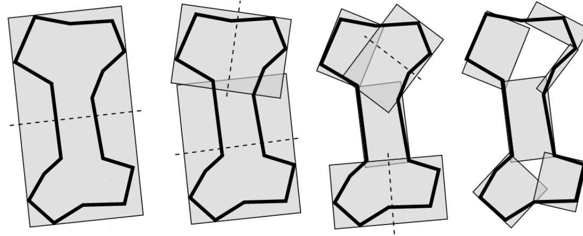


Abbildung 17: Object-Oriented-Hüllkörper. Eine Hierarchie könnte, wie im Bild von links nach rechts zu sehen, *top-down* erstellt werden.

Die Hierarchisierung kann *top-down* erfolgen, wobei die gesamte OBB zunächst die Wurzel repräsentiert. Nun wird die Box entlang der längsten Ausdehnung halbiert. Die dadurch entstandenen Boxen kommen in die darunter liegende Hierarchieebene. Die einzelnen Objektpolygone werden den zwei Boxen zugewiesen. Der Vorgang wird rekursiv wiederholt, bis maximal ein Polygon pro OBB vorhanden ist.
Aufgrund der hohen Präzision bei der Objektumschließung mittels OBB, hat der Baum weniger Hierarchiebenen als ein AABB- oder Sphere-Tree. Dadurch müssen weniger Überlappungstests durchgeführt werden.

3.4.5 Discrete Oriented Polyeder (k-DOP)

k-DOPs sind Polyeder mit k Seiten, die ein Objekt umschließen und damit eine Verallgemeinerung von AABBs. AABBs sind 6-DOP-Hüllkörper. Ein k-DOP wird definiert durch k Vektoren bzw. Richtungen D_i, wobei $i = (1, \ldots, k/2, \ldots, k)$ und k eine gerade Zahl ist. Zu jedem Vektor D_i gehört ein skalarer Wert d. Die Koeffizienten $(d_1, \ldots, d_k)$ beschreiben die Distanzen der zugehörigen Halbräume zum Ursprung, wobei dieser nicht notwendigerweise im Schwerpunkt oder innerhalb des Objekts liegen muss. Die $k/2$ Koeffizientenpaare $(d_i, d_{i+k/2})$ werden Slabs genannt. Die Schnittpunkte dieser Slabs definieren das aktuelle k-DOP und damit den Hüllkörper:

$$A_{k-dop} = \bigcap_{i=1,\ldots,k} H_i,\ H_i : \langle D_i, x\rangle - d_i \leq 0 \tag{22}$$

Die Orientierungen D_i sind für alle Objekte gleich. Pro Objekt müssen lediglich die Distanzen d gespeichert werden. Je größer die Anzahl der Orientierungen, desto genauer ist die Präzision der Hüllkörper. Durch die statischen Orientierungen kommt es zu Form-Instabilitäten bei Rotationen. Die Komplexität und der Aufwand zur Kollisionsbestimmung sind gering. k-DOPs haben eine schnellere Kollisionsbehandlung als OBBs, solange $k \leq 30$ ist und damit nicht mehr als 15 Achsen zur projizierten Intervallüberprüfung verwendet werden, da die Präzision höher ist.

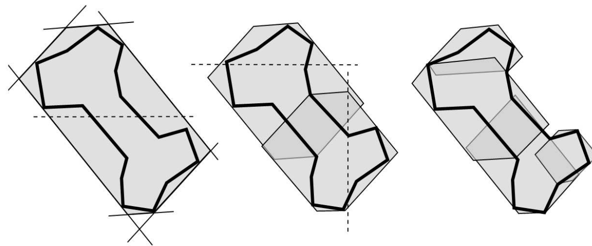


Abbildung 18: Discrete-Oriented-Polyeder-Hüllkörper. Eine Hierarchie könnte, wie im Bild von links nach rechts zu sehen, *top-down* erstellt werden.

Für die Kollisionsbehandlung wird wieder das *Separating-Axis-Theorem* verwendet, wobei die Achsen durch die k-DOP-Normalen festgelegt sind. Es müssen also auf $k/2$ Achsen die Projektionsintervalle überprüft werden. Wie schon bei den AABB-, und OBB-Hüllkörpern, kann auch hier die Optimierung auf Basis des *Sweep-and-Prune*-Algorithmus genutzt werden.

Ein k-DOP-Baum kann *top-down* erzeugt werden. Ein erstes k-DOP wird erstellt. Im nächsten Schritt wird eine Ebene orthogonal zu einer Koordinatenachse so ausgewählt, dass die neu entstehenden k-DOPs zum Beispiel ein minimales und damit ähnliches Volumen haben. Jedes Primitiv wird durch die Lage des Dreiecksschwerpunkts einem der beiden Halbräume zugewiesen. Aus diesen Primitiv-Mengen werden dann, wie im ersten Schritt, zwei neue k-DOPs erstellt. Dieser Vorgang wird wiederholt, bis minimal ein Polygon pro k-DOP vorhanden ist. Ein Vorteil gegenüber OBB-Hierarchien wurde in [Zac98] festgestellt. Dort heisst es, dass der Aufbau einer k-Dop-Hierarchie für beliebige Polygonmodelle 2- bis 4-mal schneller durchgeführt werden kann, als es bei OBB-Hierarchien der Fall ist.

3.4.6 Swept-Sphere-Volumes

Swept-Sphere-Volumes (SSV) [LGLM99] entsprechen grundlegenden geometrischen Formen wie einem Punkt, einer Linie oder einem Rechteck, wobei jeder dieser Volumentypen das Ergebnis einer Minkowski-Summe der entsprechenden Grundform und einer Kugel ist. Das Volumen wird abhängig von dem zu umhüllenden Objekt maximiert. SSV verhalten sich in Ihrer Performanz ähnlich wie OBB-Volumen. Sie weisen eine hohe Präzision und Stabilität auf. Die Komplexität bzw. die Erstellung der Boxen ist recht teuer, die Kollisionsbehandlung dagegen schnell und einfach. Da die Kollisionstests transformationsinvariant sind, ist die Stabilität akzeptabel.

Als geometrische Grundformen werden in [LGLM99] *Point Swept Sphere* (PSS), *Line Swept Sphere* (LSS) oder *Rectangle Swept Sphere* (RSS) vorgestellt. PSS entsprechen einer Kugel und lassen sich durch einen Punkt und ein Radius beschreiben. LSS entsprechen Zylindern, die an den Enden durch zwei Halbkugeln abgeschlossen sind. Die notwendigen Details sind eine Linie, ein Mittelpunkt und ein Radius. RSS sehen aus wie Boxen, bei denen die Ecken abgerundet sind. Sie lassen sich durch ein Rechteck, dessen Mittelpunkt und einen Radius beschreiben. LSS und RSS werden genau wie OBB mit Hilfe der Kovarianzmatrix erzeugt, wodurch die Hauptorientierung des 3D-Objekts gefunden wird. Ein Vorteil gegenüber OBB- oder AABB-Volumen aufgrund höherer Präzision wird für einige Testszenarien in [LGLM99] bewiesen. Allerdings wird hier auch deutlich, dass die Ergebnisse nicht verallgemeinert werden können. Je nach Beschaffenheit der Objekte ist es sinnvoller, OBB- oder AABB-Typen einzusetzen. Ein direkter Vergleich zwischen OBB- und RSS-Typen zeigt, dass OBBs etwas schneller einen Überlappungstest durchführen, für die Distanzberechnung jedoch wesentlich mehr Zeit als RSS benötigt wird. k-DOPs werden in die Vergleiche mit SSV-Typen nicht mit einbezogen.

Abbildung 19: Die drei verschiedenen Swept-Sphere-Typen: *Point Swept Sphere*, *Line Swept Sphere* und *Rectangle Swept Sphere*. Quelle: [LGLM99]

Bei der Kollisionsbehandlung zwischen zwei SSV-Hüllkörpertypen wird zunächst der Abstand zwischen den Primitiven (Punkt, Linie, Rechteck) ermittelt. Hiervon werden die beiden Radii abgezogen. Dadurch kann sowohl eine Abstandsberechnung als auch eine Penetrationstiefe durchgeführt werden.

Die SVV-Volumen lassen sich *top-down* hierarchisch unterteilen. RSS-Typen verhalten sich dabei zum Beispiel ähnlich wie OBB-Volumen. Zudem lassen sich die verschiedenen SSV-Typen in ein und dieselbe Hierarchie integrieren. In den Aussichten in [LGLM99] werden die Analysen von weiteren SSV-Primitven als Bounding-Volume-Typen erwähnt.

3.5 Detaillierte Kollisionsbestimmung (*narrow-phase*)

3.5.1 Linearer Ansatz

Bei diesem Ansatz wird die Kollisionsbehandlung zunächst als Optimierungsproblem behandelt und anschließend als lineares Programm gelöst. Der Ansatz ist laut [Ham05] schnell, kann aber nur ein Objektpaar zur Zeit behandeln. Die Objekte müssen konvex sein. Sind sie das nicht, müssen sie zunächst aufwändig in konvexe Objekte unterteilt werden.
Jedes Flächenelement i eines Objekts wird durch eine Ebenen-Ungleichung der Form $a_ix + b_iy + c_iz \leq d_i$ repräsentiert.

Jeder Punkt, der innerhalb eines Objekts liegt, muss die Ungleichungen aller Flächen i, die das Objekt beinhaltet, erfüllen. Er liegt entweder auf der Fläche oder 'hinter' der Fläche. Wenn ein Punkt den Ungleichungen von zwei Objekten entspricht, kollidieren diese Objekte. In der Praxis wird der Ansatz nicht weiter verfolgt bzw. wurde keine Implementierung hierzu gefunden. Der Vollständigkeit halber sei er hier aber erwähnt.

3.5.2 Mediale Achse

Unter der Verwendung von medialen Achsen, die ein Skelett eines 3D-Objekts beschreiben, kann ein 3D-Objekt derart repräsentiert werden, dass die Kollisionsbehandlung anhand eines Sphere-Trees schnell durchzuführen ist [Hub96]. Das Skelett weist interessante Eigenschaften auf. Es kann benutzt werden, um das 3D-Objekt approximierend zu rekonstruieren (Umkehrbarkeit). Das rekonstruierte Objekt ist eine Menge von sich überlappenden Kugeln. Das Skelett ist invariant gegenüber Translation und Rotation, was eine Neuberechnung bei diesen Transformationen erübrigt. Wie mittels Kugeln die Kollisionsbehandlung durchgeführt wird, wurde bereits in 3.4.2 erwähnt.
Da der Aufwand mediale Achsen für ein Objekt zu berechnen hoch ist, werden sie

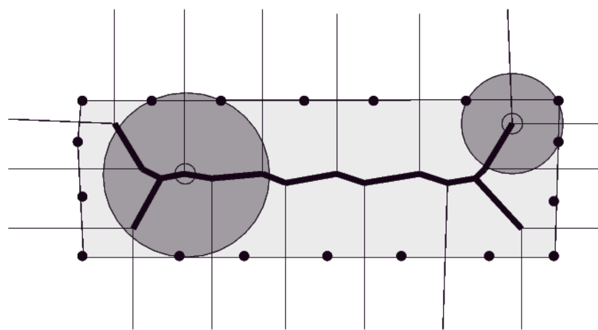


Abbildung 20: Die Abbildung zeigt den Definitionsprozess von Bounding-Volume-Spheres unter Verwendung von medialen Achsen. Anhand der schwarzen Randpunkte werden Voronoi-Regionen erstellt. Vertizes der Voronoiregionen bestimmen den Mittelpunkt einer Kugel, die drei der Randpunkte schneidet (hier drei, weil Darstellung in 2D). Quelle: [Hub96]

bei diesem Ansatz nur approximiert. Dazu werden auf der Objektoberfläche Punkte P gewählt, anhand derer das Volumen in Voronoi Regionen unterteilt wird. Die Vertizes der Voronoiregionen (siehe 20), die sich ausschließlich innerhalb des Objekts befinden, werden als Kugelmittelpunkte betrachtet. Sie befinden sich auf der approximierten medialen Achse. Demnach steigt die Präzision der errechneten medialen Achse durch die Anzahl der Oberflächenpunkte, die berücksichtigt werden. Es werden jetzt Kugeln definiert, so dass vier Punkte aus P auf deren Oberflächen liegen. Die erstellten Kugeln werden in eine Hierarchie eingetragen, wobei es mehrere Strategien gibt. Eine wäre die einzelnen Hierarchien in Abhängigkeit der verwendeten Oberflächenpunkte zur Erzeugung der Voronoiregionen zu stellen. Die Darstellung anhand verschiedener Hierarchieebenen kann in Abbildung 21 betrachtet werden. Hier wird deutlich: Je tiefer die Hierarchieebene, desto de-

taillierter das Modell, desto aufwändiger der Kollisionstest.

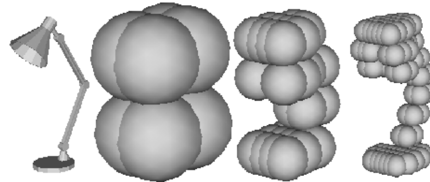


Abbildung 21: Approximation einer Schreibtischlampe unter Verwendung von Kugeln. Je mehr Oberflächenpunkte für die Herstellung der Voronoi-Regionen verwendet werden, desto mehr Kugeln, woraus eine detaillierte Darstellung resultiert. Quelle: [Hub96]

Das beschriebene Verfahren wurde in keiner der in 3.6 vorgestellten Bibliotheken implementiert. Einige Erweiterungen ([Bra03], [Bra02]) verfolgen den Ansatz von Hubbard und verbessern dabei seine Eigenschaften.

3.5.3 Lin-Canny-Algorithmus

Der Lin-Canny-Algorithmus wird in der *narrow phase* zur genauen Distanzbestimmung eingesetzt. Er basiert auf einem *Closet Feature Tracking* und ist damit *feature-based*. Merkmale oder Features eines Objekts sind dessen Punkte, Kanten und Flächen. Der Algorithmus berechnet unter Verwendung der zeitlichen Kohärenz zweier aufeinander folgenden Frames die minimale Distanz zwischen zwei disjunkten Polygon-Objekten. Dabei kommen externe, also außerhalb des Objektvolumens liegende, Voronoi-Regionen zum Einsatz.
Lin-Canny hat in seiner ursprünglichen Form zwei Schwächen: Bereits kollidierende Objekte können nicht behandelt werden, denn die Voronoi-Regionen sind nur extern definiert. Der Algorithmus gerät hierbei in eine Endlosschleife. Zudem ist eine solide Robustheit nicht immer gegeben, wenn die Punkte zweier Features mit der geringsten Distanz nicht eindeutig definiert sind [Mir98]. Das passiert zum Beispiel genau dann, wenn zwei Flächen verschiedener Objekte mit der geringsten Distanz parallel zueinander liegen. Der Aufwand beträgt bei n Objekten $O(n^2)$.

Nach dem Prinzip der Voronoi-Regionen wird das außerhalb eines Polygonobjekts liegende Gesamtvolumen mit einer Menge N von n Punkten so aufgeteilt, dass dessen Subvolumen oder Voronoi-Zellen $S_1, ..., S_M$ bestimmten Merkmalen des Objekts zugeordnet werden, wobei M dieser Merkmale existieren. Die Zuordnung erfolgt dabei nach der Regel, dass alle Punkte n innerhalb eines Subvolumens S_n das gleiche Merkmal als nächstliegendes aufweisen. Die minimale Distanz D zu einem Merkmal im Raum sorgt demnach für die Zuordnung in Voronoi-Zellen. Mathematisch wird eine Voronoi-Zelle für einen Punkt p wie folgt definiert:

$$S_n(p) = \{q \in \Re^3 \, , D(q,p) < D(q,n)\}, \, \forall n \in N$$

Um die Distanz zwischen zwei räumlich disjunkten Polygonen zu bestimmen, reicht es aus, den Raum außerhalb der Polygone durch Voronoi zu partitionieren.

Definition: *Der minimale Abstand zwischen zwei konvexen, sich nicht schneidenden Polygonen A und B ist genau dann gefunden, wenn zwei Punkte $a \in A$ und $b \in B$ auf zwei Features $m_a \in M_A$ und $m_a \in M_B$ den minimalen Abstand zwischen diesen Features besitzen und wenn sie in der Voronoi-Zelle des jeweils anderen Features vorkommen.*

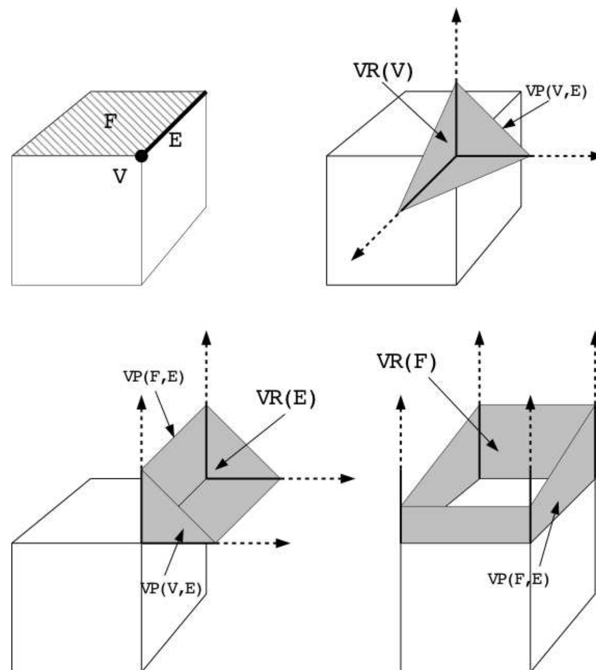


Abbildung 22: Die Aufteilung der außerhalb liegenden Regionen anhand der Objekt-Features mittels Voronoi. F, E und V bezeichnen drei der unterschiedlichen Merkmale. $VR(V)$ beschreibt eine Voronoi-Region des V-Merkmals. $VP(V,E)$ beschreibt die Trennebene zwischen den Regionen der Merkmale V und E. Quelle: [Mir98]

Beim einfachen Lin-Canny-Algorithmus wird zunächst eine Initialisierung vorgenommen, indem zwei beliebige Merkmale zweier Polygone (Objekte) ausgewählt werden. Anschließend läuft der Algorithmus zur Distanzbestimmung permanent während einer Anwendung:

1. Ermittle die Punkte von zwei Merkmalen, die eine minimale Distanz aufweisen.
2. Wenn die beiden gewählten Punkte in der Voronoi-Zelle des jeweils anderen Merkmals liegen, gebe die beiden Punkte als die minimale Distanz zwischen den Objekten aus.
3. Wähle eine neue Merkmalskombination aus und führe den Algorithmus bei 1. weiter durch. Dabei sollten die nächsten Merkmale räumlich nahe bei den vorherigen liegen, um die zeitliche Kohärenz auszunutzen.

Eine Kollision wird in [LC91] durch eine vorher festgelegte minimale Distanz ε zwischen den beiden Punkten mit minimaler Distanz signalisiert. Es gibt Erwei-

terungen, die die Schwächen des Lin-Canny-Algorithmus erfolgreich korrigieren. Dabei sei auf [Mir98] verwiesen.

3.5.4 GJK

Der GJK-Algorithmus [EGG88] ist *simplex-based*. Es können mit dem Verfahren neben der Kollision auch Distanzen und Kollisionstiefen zwischen Objekten behandelt werden. Bei diesem iterativen Anatz wird von einem Polygon-Objekt nur die äußere konvexe Hülle betrachtet bzw. alle Punkte p, die diese definieren. Die Minkowski-Differenz M zwischen zwei Objekthüllen O_1 und O_2 wird wie folgt gebildet:

$$M = O_1 - O_2 = \{\vec{p} - \vec{q} : \vec{p} \in O_1, \vec{q} \in O_2\}$$

M besteht aus Vektoren, die im Folgenden als eine Menge von Punkten oder ein Polytop betrachtet werden. Wenn O_1 und O_2 Punktmengen sind, die eine konvexe Hülle bilden, dann bilden die Punkte in M ebenfalls einen konvexen Körper. Der Nullpunkt des Raumes, in dem sich M befindet, wird durch den Nullvektor dargestellt. Enthält M einen Punkt im Ursprung, berühren sich die zwei Objekte an genau einem Punkt. Liegt der Ursprung innerhalb von M, kollidieren die Objekte, liegt er außerhalb, schneiden sie sich nicht. Je nachdem ist die Distanz oder Kollisionstiefe zwischen O_1 und O_2 durch den minimalen Vektor aus M definiert.

- Kollision: $O_1 \cap O_2 \neq \{\} \Leftrightarrow 0 \in M$
- Distanz: $d(O_1, O_2) = min\{\|x\| : x \notin M\}$
- Penetration: $p(O_1, O_2) = min\{\|x\| : x \in M\}$

M vollständig zu erzeugen, ist bei entsprechend vielen Objekt-Vertices sehr aufwändig ($O(m \cdot n)$) und wird deshalb im GJK trickreich vereinfacht, wobei der *simplex-based* Ansatz ins Spiel kommt.

Wie bereits erwähnt, bilden die Punkte in M ein konvexes Polytop. Eine Menge von Punkten wird affin unabhängig genannt, wenn keiner der Punkte durch eine affine Kombination der anderen Punkte dargestellt werden kann. Im 3D-Raum kann eine affin unabhängige Punktmenge aus maximal vier Punkten bestehen. Ein Simplex, wie er nun im GJK verwendet wird, ist eine Menge genau dieser Art, bestehend aus einer Untermenge von M. Die Simplices können also Punkte, Geraden, Dreiecke oder Tetraeder sein.

GJK konstruiert nun in mehreren Iterationschritten eine Folge von Simplices pro Objektpaar. Die iterative Auswahl der Simplices erfolgt so, dass sich die Simplices dem Ursprung im M-Raum nähern. Ein terminiertes Simplex umschließt optimalerweise den Ursprung oder liegt so nah wie möglich bei ihm.
Durch zahlreiche Erweiterungen und Optimierungen des GJK, zum Beispiel in [Cam97] oder [VB99], ist der *enhanced GJK* entstanden, mit denen der Aufwand für die Distanz- bzw. die Kollisionsberechnung zweier Polytope auf bis zu $O(1 + \varepsilon)$ reduziert werden kann.

3.6 Bibliotheken für Kollisionsbehandlung

Es existieren eine Reihe Kollisionsbibliotheken, die genutzt werden können, um in bestehende Systeme integriert zu werden. Dabei verwenden die Bibliotheken

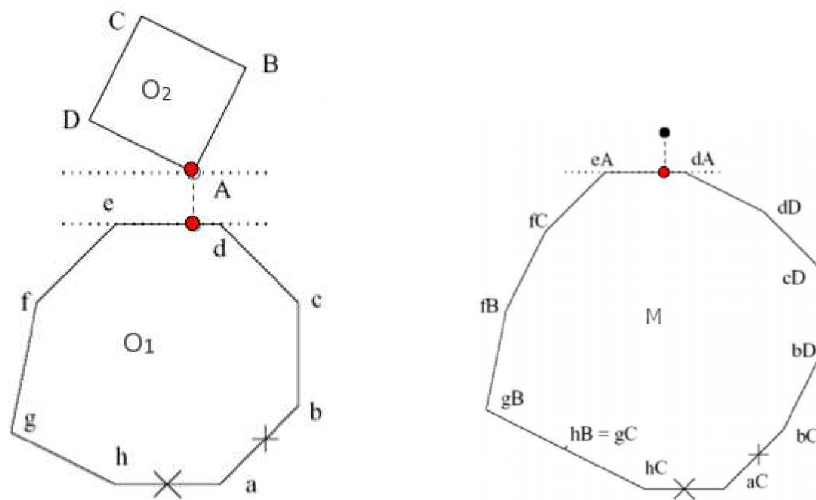


Abbildung 23: Links: Zwei nicht kollidierende Objekte im Koordinatenraum. e ist der Abstand zwischen den Objekten. Rechts: Die Objektrepräsentation im Konfigurationsraum nach der Minkowski-Differenz. Der Nullpunkt ist nicht im Objekt enthalten. Der minimale Abstand der Features ist der Abstand zwischen O_1 und O_2.

unterschiedliche Methoden, um Kollisionen zu ermitteln. Im Folgenden werden einige Bibliotheken und ihre Vor- und Nachteile näher beschrieben. Es wird analysiert, ob die Bibliotheken die Anforderungen sinnvoll erfüllen, die durch die hier vorgestellte Anwendung gegeben sind.

Die mit Abstand meisten Kollisionsbehandlungssysteme wurden von der Forschungsgruppe *team gamma* (Geometric Algorithms for Modeling, Motion, and Animation) entwickelt, einer Gruppe der Universität von North-Carolina.

3.6.1 I-Collide

I-Collide [JC95] verwendet in der *broad phase* AABB-Hüllkörper und die *Sweep-and-Prune* Optimierung, wodurch die zeitliche Kohärenz ausgenutzt wird. Für die *narrow phase* wird der Lin-Canny-Algorithmus verwendet, wobei auch die Objektvolumina durch Pseudo-Voronoi-Regionen unterteilt werden. Dadurch können kollidierende Objekte weiter behandelt werden. Die Objektpolygone werden *bottom-up* zu einer AABB-Hierarchie hinzugefügt, wodurch einfache nicht-konvexe Polyeder aus konvexen Teilen zusammengesetzt und mit I-Collide behandelt werden können. In [JC95] wurden sowohl statische als auch dynamische AABB-Volumen untersucht. Die statischen Volumen ergaben eine minimal bessere Untersuchungszeit, was auf die einfachen, in der Untersuchung verwendeten Objekte zurückzuführen ist.
I-Collide wurde 1996 vom *team gamma* entwickelt. Als Alternative für konvexe Objekte empfiehlt es das SWIFT-System, da es schneller und stabiler sei. Als Alternative für nicht-konvexe Objekte und *polygon soups* wird V-Collide zusammen mit RAPID empfohlen.

3.6.2 RAPID

RAPID [GLM96] kann mit Objekten umgehen, die aus *polygon soups* bestehen. Es bietet keinen direkten Support für mehr als zwei Objekte. Zudem lässt es aufgrund der verwendeten OBB-Hierarchie und dem *Separating-Axis-Theorem* keine Distanzmessungen zu. Die Kollisionsbehandlung von RAPID ist schnell. Es arbeitet in zwei Schritten. Zuerst wird *top-down* eine binäre OBB-Hierarchie der Objektpolygone erstellt. Während der *narrow-phase* wird das *Separating-Axis-Theorem* an den OBB-Volumen in der Hierarchie eingesetzt.

RAPID wurde bis 1997 vom *team gamma* entwickelt und ist laut Entwickler das kleinste und einfachste Modul aus eigener Herstellung des *team gamma*. Die aktuelle Version ist 2.01. Es bietet bei Kollision bereits die Rückgabe der Modell-Triangles an, die in die Kollision zweier Objekte verwickelt sind.

3.6.3 V-Collide

V-Collide [Man97] ist ein *n-body*-Prozessor für RAPID und ermöglicht damit die Kollisionserkennung zwischen mehr als zwei Objekten, basierend auf einer binären OBB-Hierarchie und dem *Separating-Axis-Theorem*. Dies ist allerdings mit dem Nachteil verbunden, dass keine Entfernungen und Penetrationstiefen gemessen werden können. V-Collide ermittelt, ähnlich wie I-Collide anhand von AABB-Hüllkörpern und dem *Sweep-and-Prune*-Algorithmus, welche Objekte möglicherweise kollidieren und übergibt diese Objektpaare an RAPID zur genaueren Untersuchung.
V-Collide bietet die Möglichkeit, Objekte nachträglich in die Szene zu laden oder aus ihr zu löschen. Zudem können Objektpaare deaktiviert werden, sie werden dann bei der Kollisionsbetrachtung ignoriert.

V-Collide wurde vom *team gamma* entwickelt und zuletzt 1998 aktualisiert. Die aktuelle Version ist 1.1. Die Bibliothek bietet alles, was im Rahmen der zu entwerfenden Anwendung gefordert wird. Zudem wird V-Collide im VTK-Forum [Dev06] häufig als sinnvolle Bibliothek für die Kollisionsbehandlung genannt.

3.6.4 Swift und Swift++

Swift [Lin00] und Swift++ [Lin01] sind eine Weiterentwicklung von I-COLLIDE und sollten laut *team gamma* stattdessen verwendet werden. Da Swift nur mit konvexen Objekten umgehen kann, werden nicht-konvexe Objekte durch einen, in Swift++ mitgelieferten, *Decomposer* zu mehreren konvexen Objekten aufgeteilt und anschließend an Swift übergeben. Sie unterstützen eine Kollisionserkennung, eine abschätzende und eine genaue Distanzmessung zwischen mehreren Objekten. Zudem können die kollidierenden Features wie Punke, Linien oder Flächen ausgegeben werden.

Swift benutzt *Sweep-and-Prune*, um uninteressante Objekte in der *broad phase* auszuschließen. Dabei entscheidet Swift beim Einlesen pro Objekt, ob es eine statische oder dynamische AABB-Bounding-Box verwendet. Für die *narrow phase* wird eine verbesserte Lin-Canny-Version verwendet. Hierbei wird eine Level-Of-Detail-Hierarchie eines Objekts erstellt. Sie ist als eine Folge von konvexen Polyedern $P_0, \dots P_k$ aufgebaut, wobei P_0 das Original ist. Alle davon abgeleiteten $P_i, i > 0$

haben sukzessive weniger Features und umschließen jeweils den Vorgänger. Eine gute Objektqualität bedeutet gleichzeitig ein zeitintensives Voronoi-Marching und umgekehrt. Mittels Voronoi-Marching auf den qualitativ unterschiedlichen konvexen Hüllen können wahlweise exakte oder approximierende Objektdistanzen errechnet werden.

Swift und Swift++ wurden von *team gamma* entwickelt und zuletzt 2001 verändert. Die aktuelle Versionsnummer von Swift ist 1.3 und von Swift++ 1.2. Dieses System ist laut des *team gamma* signifikant schneller als zum Beispiel I-Collide, V-Clip oder der erweiterte GJK. Wie schon V-Collide bietet Swift/Swift++ alles, was im Rahmen dieser Arbeit benötigt wird und fällt damit ebenfalls in die nähere Auswahl.

3.6.5 PQP

PQP basiert auf RAPID und erweitert dieses Paket, indem Distanzen ermittelt und überprüft werden können. Zudem bietet PQP die Möglichkeit, eine Toleranzzone um Objekte anzugeben, die eine Überschneidung melden. Für die Realisierung werden zusätzlich Rectangle-Swept-Spheres als Hüllkörper verwendet, die *top-down* den Hierarchiebaum aufbauen. Bei Kollision können alle beteiligten Polygone ausgegeben werden.

PQP wurde vom *team gamma* entwickelt, die letzte Veränderung fand 1999 statt. Die aktuelle Version ist 1.3. Laut [Maz02] verhält sich PQP ähnlich wie RAPID.

3.6.6 QuickCD

QuickCD [Mit96] stellt für *polygon soups* ein Kollisionsbehandlungssystem zur Verfügung, wobei keine Distanzen zwischen zwei Objekten ermittelt werden können. Dabei verwendet QuickCD 6-dop, 14-dop, 18-dop und 24-dops, um eine Hierarchie *top-down* für Szenen-Objekte aufzubauen. Die Verwendung von k-dops bei der Hierarchieerstellung ist laut [Zac98] zwei bis viermal schneller als eine OBB-Hierarchisierung, wie sie RAPID bietet. Dennoch ergeben Benchmarks, die von Zachmann durchgeführt wurden, dass eine Kollisionserkennung mit RAPID bzw. OBB-Trees schneller Ergebnisse liefert [Zac06]

QuickCD wurde 1996 von Jim Klosowski, Martin Held und Joe Mitchell an der State University of New York at Stony Brook entwickelt. Die aktuelle Version ist seither 1.0.

3.6.7 Voronoi-Clip (V-Clip)

V-Clip [Mir98] verwendet eine eigens optimierte Version des Lin-Canny Algorithmus. Der modifizierte Lin-Canny weist eine geringere Komplexität als das Original auf, kann mit Intersektionen von Objekten umgehen, ist robust und besitzt keine Schleifenprobleme. Dabei kann V-Clip nur unzulänglich mit nicht-konvexen Objekten umgehen.
Die Bibliothek verwendet QHull [BDH98], um eine Hierarchie pro Objekt zu erzeugen. Die Hierarchie besteht aus konvexen Vertice-Objekten, die jeweils Teilbe-

reiche der gesamten Objekt-Vertices beinhalten.

In der Veröffentlichung von V-Clip wird erwähnt, dass RAPID für komplexe Szenenverhältnisse besser geeignet ist als V-Clip. Letzteres ist wiederum schneller, wenn die Modelle wohlgeformt, nicht konkav und nicht all zu groß sind. Die Tests in dieser Ausarbeitung ergeben, dass V-CLIP zwei Objekte effizienter auf Kollision überprüft als der einfache oder erweiterte Lin-Canny-Algorithmus.

V-Clip wurde vom Mitsubishi Electric Research Lab (MERL) 1997 entwickelt. Die letzte Änderung fand 2003 statt.

3.6.8 Solid und FreeSolid

Solid [VB99] wurde 1999 erstmals vorgestellt, FreeSolid [VB01], eine Erweiterung, im Jahr 2001. Beide bieten Kollisionserkennungen für konvexe Objekte und arbeiten in zwei Phasen. Zunächst wird in der *broad-phase* eine AABB-Hierarchie für die Objekte aufgebaut. Die *narrow-phase* wird mit einer manipulierten Version des GJK [EGG88] gelöst. Daher kann mit Solid sowohl eine Kollision als auch die Distanz und Kollisionstiefe ermittelt werden. Um die notwendigen konvexen Hüllen zu errechnen, wird die QHull-Bibliothek [BDH98] verwendet. Angaben des Entwicklers zufolge läuft Solid 1.4 bis 2.6 mal langsamer als RAPID.

Solid und FreeSolid wurden von Gino van den Bergen entwickelt. Die aktuelle Version von FreeSolid ist 2.1.1.

3.6.9 vtkCollisionDetectionFilter

Eine Kollisionserkennung, die sofort innerhalb der VTK-Pipeline genutzt werden kann, wurde von der Arbeitsgruppe Bioengeneering der University College Dublin [oUCD06] erstellt. Der Filter hat als Input zwei `vtkPolyData`-Objekte. Intern wird eine OBB-Hierarchie mittels der Klasse `vtkOBBTree` erstellt, anhand derer beide Objekte auf Kollision getestet werden. Als Ausgabet erhält man ebenfalls `vtkPolyData`-Objekte. Dabei sind die Flächen farbig markiert.

Obwohl diese System innerhalb der VTK-Architektur benutzt werden kann, wurden im Rahmen dieser Arbeit keine Hinweise ermittelt, dass es jemals erfolgreich implementiert und genutzt wurde. Von einer Implementierung wurde daher abgesehen.

4 Realisierung / Implementierung

Dieses Kapitel beschreibt zunächst die Software und Bibliotheken, die zur Erstellung der hier vorgestellen Anwendung mit dem Namen *mihMovinBones* verwendet wurden. Anschließend werden die Funktionen der Anwendung aufgeführt. Eine Beschreibung der Realisierung und Implementierung ist im Anschluss daran zu finden.

4.1 Verwendete Software und Bibliotheken

4.1.1 V-Collide als Bibliothek für die Kollisionsbehandlung

Welche Bibliothek zur Kollisionserkennung letztlich für die hier vorgestellte Anwendung verwendet werden sollte, hing von den speziellen Anforderungen der Anwendung ab (siehe 3.2). Ein Implementieren aller Bibliotheken und eine anschließende Evaluierung der Systeme hätte den Rahmen dieser Arbeit gesprengt. Es gibt andere Ausarbeitungen, die derartige Vergleiche bereits teilweise durchgeführt haben.
In [Maz02] werden V-Collide, RAPID, Solid, V-Clip und PQP miteinander verglei- chen. Das Ergebnis dieser Untersuchung ist, dass für wohlgeformte und einigermaßen konvexe Objekte, die auch nicht zu groß sind, V-Clip die optimale Lösung bietet. Für den Fall, dass die Anforderungen anders aussehen, nämlich so wie es bei *mihMovinBones* der Fall ist, wird das Kombipaket RAPID/V-Collide als beste Lösung genannt. FreeSolid kommt für die Anwendung nicht in Frage, da das System nur mit konvexen Objekten umgehen kann, was eine Vorverarbeitung bedeuten würde.
Dass V-Collide den Anforderungen hinsichtlich der Performanz genügt, haben Tests gezeigt, die in 5.2.1 beschrieben und analysiert werden.

Gegenüber Swift++ konnte V-Collide sich letztlich aufgrund der kompatibleren Schnittstelle für VTK-Datensätze durchsetzen. V-Collide liest die einzelnen Objektdaten ein, indem jedes Dreieck, bzw. jeweils drei Dreieckspunkte mit einer ID übergeben werden. VTK und *mihLib* können nach dem Einlesen auf diese Informationen der 3D-Daten komfortabel über Methodenaufrufe zugreifen und an V-Collide weiterreichen. Dabei ist es unerheblich, welche Strukturen V-Collide übergeben werden, solange das genannte Prinzip eingehalten wird. Die Möglichkeit bietet eine optimale Verarbeitung von *polygon-soups* ohne jegliche Vorverarbeitung. Die Klasse `mihCollisionWorld` stellt eine Schnittstelle zwischen V-Collide und VTK, bzw. *mihLib* dar. Die Eigenschaften der Klasse werden in 4.1.5 beschrieben.

Swift++ dagegen fordert bei komplexen und nicht-konvexen Objektmodellen, wie sie in *mihMovinBones* verwendet werden, zunächst eine Vorverarbeitung durch einen *Decomposer*. Dabei handelt es sich um eine zu Swift++ mitgelieferte separate Anwendung, die in vier Schritten die Objekte in mehrere konvexe Objekte unterteilt. Durch die Tatsache der notwendigen Vorverarbeitung stand Swift++ somit nur an zweiter Stelle in der Kandidatenliste der Kollisionsbibliotheken. Hätten die Tests mit V-Collide ergeben, dass es den Anforderungen nicht gerecht wird, wäre wahrscheinlich Swift++ als Alternative implementiert worden.

4.1.2 CMake

CMake ist ein in C++ geschriebenes, erweiterbares und plattformunabhängiges Open-Source Make-System, das die Projekterstellung verwaltet und vereinfacht. Dabei ist CMake kein selbstständiger Build-Prozess, sondern verwendet die vorhandenen Prozesse der verschiedenen Plattformen, z.B. make auf Unix-Systemen. Durch Einfügen und Verwenden von Konfigurations-Dateien (`CMakeLists.txt`) in jedem Projektverzeichnis, in denen die sogenannten *cache values* enthalten sind, wird eine Build-Datei (z.B. makefile bei Unix-Systemem) erstellt. CMake kann Quellcode kompilieren, Bibliotheken erstellen, Wrapper generieren und ausführbare Dateien erstellen. Für viele dieser Aufgaben wird CMake im Rahmen der vorgestellten Arbeit verwendet.
CMake wurde im Rahmen des Visible Human Projekts eigens für das Insight Segmentation and Registration Toolkit (ITK) erstellt. Die erste Version erschien im Jahr 2000 und wird seither weiter entwickelt. Die aktuelle Version ist 2.4.3.

4.1.3 Visualization ToolKit (VTK)

VTK ist ein Open-Source Softwaresystem für den Bereich der Computergrafik, Bildverarbeitung und Visualisierung. Es wird seit 1997 fortwährend entwickelt und erweitert. Die objektorientierte Bibliothek besteht mittlerweile aus über 700 C++ Klassen, Applikationen können in C++, Java, tcl oder Python erstellt werden. Die aktuelle Version ist 5.0.

VTK lässt sich in zwei Bereiche unterteilen. Der erste Bereich beinhaltet eine Visualisierungs-Pipeline (*visualization model*). Der zweite Bereich umfasst eine abstrakte Version der Rendering-Pipeline (*graphics model*). Als Schnittstelle zwischen beiden Bereichen werden sogenannte Mapper verwendet.

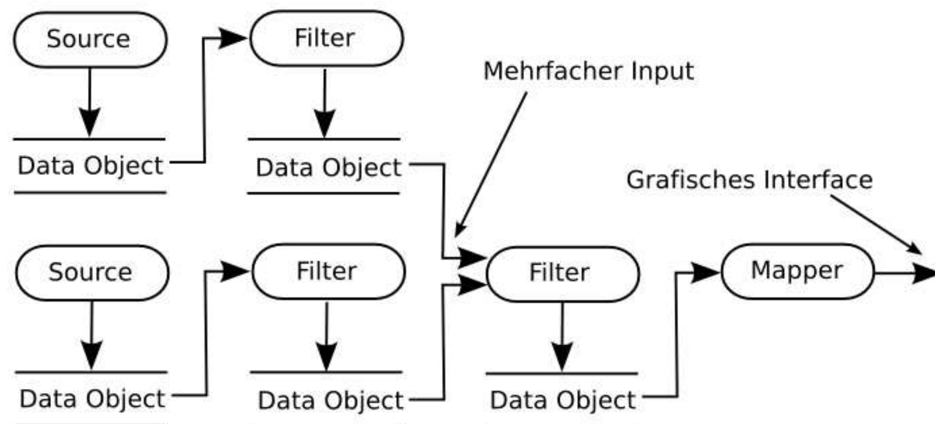


Abbildung 24: VTK-Visualisierungs-Pipeline. Die Pfeile zeigen in Richtung des Datenflusses.

Die Visualisierungs-Pipeline (siehe Abb. 24) hat zur Aufgabe Informationen in grafische Daten zu transformieren. Innerhalb der Pipeline sind sowohl Daten als

auch Algorithmen in Klassen verpackt. Es werden zwei Basistypen verwendet, `vtkDataObject` und `vtkProcessObject`. `vtkDataObject` ist eine umfassende Repräsentation aller möglichen Visualisierungsdaten. Die Daten können als gewöhnliches Bild, strukturiertes oder unstrukturiertes Gitter, Punktmenge, Polygonmodelle und in vielen anderen Formaten vorliegen und mit VTK verarbeitet werden. `vtkProcessObject` sind Objekte bzw. Filter, die diese Datenmengen einlesen, verarbeiten und verändert ausgeben. Daten und Filter werden hintereinander geschaltet, wobei der Output eines Objekts der Input für das folgende Objekt ist:

```
aFilter->SetInput(anotherFilter->GetOutput() );
```

Dabei besteht die Möglichkeit beliebig viele Filter hintereinander zu verwenden. Ein Filter kann mehrere Ein- und Ausgabekanäle besitzen. Die Visualisierungs-Pipeline nutzt das Prinzip der '*lazy evaluation*', das heisst, dass Veränderungen aufgrund von Filtern an den Daten nur dann tatsächlich durchgeführt werden, wenn diese Daten angefordert werden. Das Prinzip ist aufgrund der oftmals großen Datenmengen in VTK durchaus sinnvoll. Die Daten werden am Ende der Visualisierungspipeline einem Mapper übergeben, der die Daten aus dem `vtkDataObject` liest und sie für die grafische Ausgabe bzw. den Renderer konvertiert.

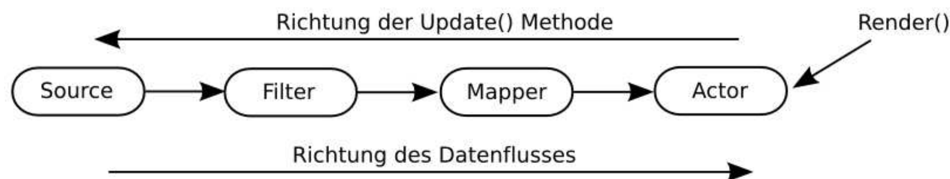


Abbildung 25: VTK-Rendering-Pipeline. Veranschaulichung des Pipeline-Konzepts.

Der zweite Bereich (siehe Abb. 25) umfasst eine abstrakte Version der Rendering-Pipeline (*graphics model*). Sie konvertiert letztlich grafische Daten in das Bild der Szene, abhängig von den Mappern, der Kameraposition, den Lichtquellen usw. Sie abstrahiert dabei die Grafikbibliotheken wie OpenGL. Die Tabelle 1 beschreibt die relevanten Klassen, die in diesem Bereich notwendig sind.

Ein Beispiel-Code (siehe Listing 1) soll den Pipeline-Aufbau und das '*lazy evaluation*'-Prinzip verdeutlichen. Die hier verwendeten `vtkSmartPointer` werden anschließend erläutert. In dem Beispiel wird ein Linie (`vtkLineSource`) mit Anfangs- und Endpunkt erzeugt. Ein `vtkTubeFilter` konvertiert diese Linie in eine Röhre. Anschließend erfolgt die Übergabe an den Mapper und damit an die Rendering-Pipeline mit den in Tabelle 1 beschriebenen Objekten.

Durch die beiden `std::cout`-Ausgaben wird das *lazy evaluation*-Prinzip deutlich. Die erste Ausgabe wird für die Anzahl der verwendeten Streifen eine Null sein, obwohl der `vtkTubeFilter` bereits einen Input und die notwendigen Parameter besitzt. Erst das `tubeFilter->Update()` erzwingt eine Ausführung des Filters,

Tabelle 1: Gebräuchliche Objekte innerhalb der Rendering-Pipeline

vtkRenderWindow	Grafisches Ausgabefenster, in das ein oder mehrere vtkRenderer ihre Inhalte einzeichnen können.
vtkRenderer	Renderer, der die Lichtquellen, Kameras, Objekt-Aktoren usw. während des Renderprozesses einer Szene verwaltet.
vtkLight	Lichtquelle mit den üblichen Eigenschaften wie Position, Farbe, Helligkeit, usw.
vtkCamera	Kamera mit den Eigenschaften wie Position, Blickrichtung, usw.
vtkActor	Repräsentiert Objekte (3D-Oberflächenmodell) innerhalb des vtkRenderer mit den Eigenschaften Position, Transformation, Texturen, Sichtbarkeit usw.
vtkProperty	Erscheinungseigenschaften eines vtkActor wie zum Beispiel Farbe, Transparenz, und Lichteigenschaften.
vtkMapper	Schnittstelle zwischen den Daten und der visuellen Datenausgabe. Dient der geometrische Darstellung von Daten, Wird für die Visualisierung einem vtkActor übergeben.
vtkRenderWindow-Interactor	Ermöglicht eine Interaktion mit der Szene per Maus oder Tastatur.

der wiederum die Daten für seinen Input anfordert. Die zweite Ausgabe für die Streifenanzahl des Objekts ist die Zahl Sechs.
Für gewöhnlich muss Update() nicht manuell aufgerufen werden, wenn die Pipeline vollständig ist. Sobald der vtkActor eine Aufforderung zum Rendern bekommt, schickt der Mapper einen Update-Aufruf durch die gesamte Pipeline an alle Objekte, um das aktuelle Objekt darzustellen.

SmartPointer Im Listing 1 fällt auf, dass die erzeugten Objekte durch die Wrapperklasse vtkSmartPointer umhüllt sind. SmartPointer erübrigen beim Arbeiten mit Pointern das Dereferenzieren von Pointern und das Löschen von Objekten, die im dynamischen Speicher angelegt wurden und nicht mehr verwendet werden bzw. auf die nicht mehr referenziert wird. Ein Zeiger auf ein Objekt, der von vtkSmartPointer umschlossen ist, speichert intern mit einem Zähler (reference count) die Anzahl der Zeiger, die auf das Objekt verweisen. Sobald dieser Zähler Null ist, ruft der SmartPointer die delete()-Methode des Objekts auf.

Listing 1: Beispiel für eine Datenvisualisierung mittels VTK. Anhand der beiden Konsolenausgaben wird das Prinzip der *lazy evaluation* deutlich: die erste Ausgabe wird die Zahl 0, die zweite 6 ausgegeben

```
#include "vtkLineSource.h"
#include "vtkTubeFilter.h"
#include "vtkPolyData.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkRenderWindow.h"
#include "vtkRenderer.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkSmartPointer.h"

int main(int argc, char **argv)
{
  // erstelle die Geometrie einer Linie
  vtkSmartPointer<vtkLineSource> line = vtkSmartPointer<vtkLineSource>::New();
  line->SetPoint1(1.0, 2.0, 3.0);
  line->SetPoint2(10.0, 10.0, 10.0);

  // Filter, der eine Linie in eine Roehre konvertiert
  vtkSmartPointer<vtkTubeFilter> tubeFilter = vtkSmartPointer<vtkTubeFilter>::New();
  tubeFilter->SetNumberOfSides(6);
  tubeFilter->SetInput(line->GetOutput());

  // erst durch Update() wird der Filter ausgefuehrt
  std::cout<<"Polygonanzahl vor Update(): "<<
        tubeFilter->GetOutput()->GetNumberOfStrips() << std::endl;
  tubeFilter->Update();
  std::cout<<"Polygonanzahl nach Update(): "<<
        tubeFilter->GetOutput()->GetNumberOfStrips() << std::endl;

  // uebergibt die Geometriedaten dem Mapper
  vtkSmartPointer<vtkPolyDataMapper> map = vtkSmartPointer<vtkPolyDataMapper>::New();
  map->SetInput(tubeFilter->GetOutput());

  // Mapper uebergibt Actor Informationen zur Darstellung
  vtkSmartPointer<vtkActor> aLine = vtkSmartPointer<vtkActor>::New();
  aLine->SetMapper(map);
  aLine->GetProperty()->SetColor(0,0,1);

  // Renderer und Render-Window werden erzeugt und verknuepft
  vtkSmartPointer<vtkRenderer> ren1 = vtkSmartPointer<vtkRenderer>::New();
  vtkSmartPointer<vtkRenderWindow> renWin = vtkSmartPointer<vtkRenderWindow>::New();
  renWin->AddRenderer(ren1);

  // Interactor wird erzeugt und dem RenderWindow zugeteilt
  vtkSmartPointer<vtkRenderWindowInteractor> iren =
        vtkSmartPointer<vtkRenderWindowInteractor>::New();
  iren->SetRenderWindow(renWin);
  // Actor der Kugel wird dem Renderer hinzugefuegt
  ren1->AddActor(aLine);
  ren1->SetBackground(1,1,1);
  // Ausgabebild rendern (Licht und Kamera wurde automatisch erstellt)
  renWin->Render();
  iren->Start();
}
```

```
Output:
Polygonanzahl vor Update(): 0
Polygonanzahl nach Update(): 6
```

Interaktion Um per Maus und Tastatur Einfluss auf eine gerenderte Szene zu nehmen, kann ein `vtkRenderWindowInteractor` dem `vtkRenderWindow` zugewiesen werden. Diese Klasse besitzt vordefinierte Paare von abzufragenden Ereignissen und davon abhängigen Aufrufen und ist in seiner elementaren Form bereits sehr umfangreich in Bezug auf Interaktionsmöglichkeiten. Die Reaktionen auf zu beobachtende Ereignisse können je nach Anforderung durch zugewiesene `vtkInteractorStyle`-Objekte variiert oder selbst definiert werden, was auf Basis der Subject/Observer- und Command-Entwurfsmuster [Gam01] geschieht.

Alle von vtkObject abgeleiteteten Objekte haben die Methode AddObserver(), um Ereignisse von außerhalb beobachten zu lassen. Geschieht das Ereignis, auf das der Observer angesetzt wurde, wird ein zugeordneter Callback durchgeführt. So kann zum Beispiel permanent die Mausposition in einem Bild zurückgegeben werden, indem ein Observer das MouseMoveEvent des Interaktors beobachtet und auf eine Methode verweist, die die Mausposition ermittelt und ausgibt.

4.1.4 QT

Qt ist eine Klassenbibliothek, die es möglich macht graphische Benutzeroberflächen (GUIs) plattformübergreifend unter C++ zu erstellen. Dabei besteht Qt mittlerweile aus über 400 Klassen, die unter anderem die GUI-Programmierung und -Design, Datenbank-Programmierung, den Einsatz im Netzwerk und die Verwendung von XML unterstützt.

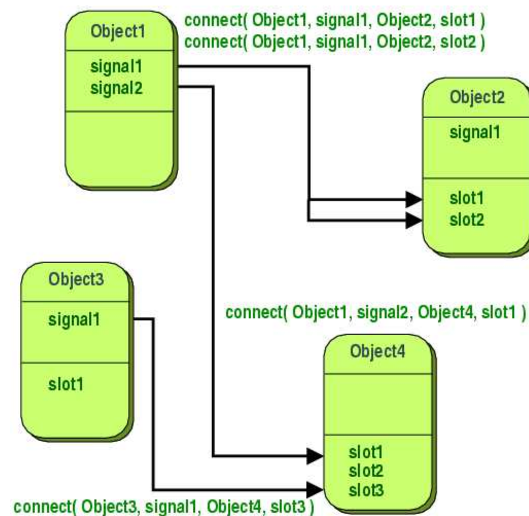


Abbildung 26: Eine abstrakte Ansicht einiger Signal-Slots-Verbindungen. Quelle: [Tro05]

Der Qt-Designer ist das dazugehörige WYSIWYG-Tool, dass das Zusammenstellen von Oberflächen mittels Qt erleichtert. Im Designer können zudem bereits die Signals und Slots definiert werden, die das Event-Handling realisieren. Ein Signal wird bei einer bestimmten Aktion als Reaktion ausgesendet, zum Beispiel stateChanged() bei einer QCheckBox. Ein Slot ist der Signalempfänger, der auf ein Signal angesetzt wird und als Reaktion auf dieses Signal eine Slot-Methode aufruft. Dabei kann auch eine Parameterübergabe stattfinden. Eine Verbindung zwischen Signals und Slots geschieht durch den connect-Befehl.

Signals und Slots werden, genau wie Methoden, im Header der jeweiligen Klassen definiert. Signals werden mit dem Schlüsselwort signals: , Slots mit slots: deklariert. Die Anzahl der Signals und Slots ist dabei beliebig. Damit dieses Prinzip in einer Klasse genutzt werden kann, muss diese von QObject abgeleitet sein und das Macro QOBJECT aufrufen.

4.1.5 mihLib

Die institutseigene Bibliothek *mihLib* dient zur Unterstützung von institutsspezifischen Anwendungen und Verknüpfung von verschiedenen Bibliotheken wie VTK, ITK, xerces und Qt. Dabei stehen zahlreiche Klassen zur Verfügung, die das Einlesen, Verarbeiten und Ausgeben von medizinischen Daten vereinfachen. Unter Verwendung einiger *mihLib*-Klassen wurde auch die hier vorgestellte Anwendung programmiert. Im Rahmen dessen wurde zudem *mihLib* um die Klassen `mihGroupObject`, `mih3DTransformation`, `mih3DTransformationHistory` erweitert. Die Klassen `mihPatientData`, `mihXMLWriter`, `mihQtSceneWidget` und `QVTKImageSliceViewer2` wurden verändert. Die Funktionalität der neuen Klassen und die Veränderungen werden später erläutert.

Kapselung der VTK-Pipeline durch mihLib-Klassen Die Hauptaufgabe der *mihLib* ist die Bereitstellung derjenigen VTK-Strukturen, die für gewöhnlich bei der Visualisierung verwendet werden. Dadurch ist es mit *mihLib* möglich, sich auf die eigentlichen Aufgaben, z.B. die Verwendung von speziellen Filtern, zu konzentrieren, ohne dass jedes mal erneut die VTK-Pipeline zusammengebaut werden muss. Die folgenden Klassen der *mihLib*-Bibliothek abstrahieren die VTK-Pipeline, indem die relevanten VTK-Klassen innerhalb der *mihLib*-Objekte gekapselt werden.

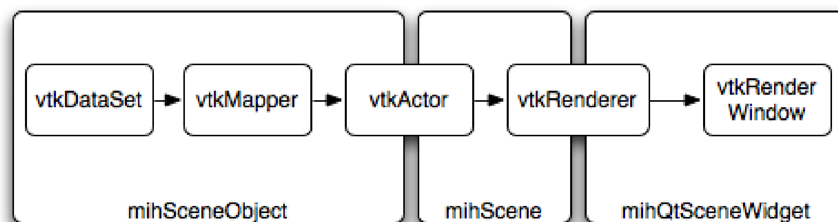


Abbildung 27: Verallgemeinerte Darstellung der Kapselung von VTK-Elementen durch die Bibliothek mihLib.

Die Klasse `mihSceneObject` ist die Basisklasse für alle Objekttypen, die mit der *mihLib* verarbeitet werden. Dazu zählen `mihVolumeObject`, `mihVectorFieldObject` und `mihSurfaceObject`, dass wiederum in die Klassen `mihSmoothedSurfaceObject` und `mihSurfaceSliceObject` unterteilt wird. In `mihSceneObject` sind demnach neben anderen die Klassen `vtkDataSet`, `vtkMapper`, `vtkActor` und `vtkProperty` gekapselt. `mihSceneObject` ist eine rein virtuelle Klasse. Alle Methoden sind abstrakt und müssen in den abgeleiteten Klassentypen implementiert werden. Ein `mihSceneObject` kann entsprechend seiner Eigenschaften über Methodenaufrufe umfangreich manipuliert werden.

`mihScene` ist eine Klasse, die zur Verwaltung der `mihSceneObject`-Objekte und Kameras innerhalb einer Szene dient. Entsprechend werden in `mihScene` und `mihSceneBase` die Klassen `vtkRenderer` und `vtkCamera` inkludiert. Alle vorhandenen Objekte einer Szene werden dem Renderer zugewiesen. Zahlreiche Me-

thoden ermöglichen das Hinzufügen, Auswählen, Deselektieren und Löschen von 3D-Modellen, die in Form von `mihSceneObjects` vorliegen. Des Weiteren kann bei Bedarf über den Renderer die Pipeline zur Aktualisierung aufgefordert werden.

Mit der Klasse `mihQtSceneWidget` wird ein Ausgabefenster angeboten, das die 3D-Modelle aus einer Szene darstellt. Als Input ist eine `mihScene` vorgesehen. `mihQtSceneWidget` ist von `QVTKWidget` abgeleitet und kapselt `vtkRenderer` und `vtkRenderWindow` aus der Pipeline. Interaktoren werden durch `QVTKWidget` verwaltet, in dem ein `QVTKInteractor` von vornherein existiert.
Schließlich realisiert `QVTKImageSliceViewer2` die Visualisierung von Schichtdaten und zusätzlichen Objekten, wobei einige Hilfsmittel integriert wurden. Die Klasse ist ebenfalls von `QVTKWidget` abgeleitet und beinhaltet zudem den `vtkImageSliceViewer2` und damit einen `vtkRenderer` und ein `vtkRenderWindow`.

Mit `mihQtSceneWidget` und `QVTKImageSliceViewer2` wurden nur zwei der zahlreichen Möglichkeiten beschrieben, die *mihLib* zur visuellen Ausgabe von medizinischen Daten bereitstellt. Da die anderen Klassen aber nicht im Rahmen dieser Arbeit verwendet wurden, werden sie auch nicht näher erläutert.

4.2 Beschreibung der Funktionalitäten

Die im Rahmen dieser Diplomarbeit erstellte Anwendung mit dem Namen *mihMovinBones* stellt einen ersten Ansatz für eine präoperative Repositionierungssoftware für komplizierte Knochenbrüche dar. Ein Screenshot der Programm-GUI zeigt die Abbildung 29. Die Funktionalitäten, die im Folgenden beschrieben werden, basieren teilweise auf den Ideen eines praktizierenden Unfallchirurgen. Die Beschreibung der praktischen Umsetzung erfolgt in 4.4.

- Die CT-Daten der gesamten Fraktur und die 3D-Modelle der einzelnen Knochenfragmente, die der Anwender mittels der Software verarbeitet, werden über INFO- und XML-Dateien eingelesen. Innerhalb dieser Datenformate wird wiederum auf DICOM-Dateien und VTK-Dateien verwiesen. Eine zusätzliche Speicherung der durch den Anwender veränderten Frakturеigenschaften wird lediglich im XML-Format angeboten.

- Der wesentliche Bestandteil der Anwendung ist das 3D-Widget, indem die Knochenmodelle in einer Szene dargestellt werden. Per Interaktion durch Maus und Tastatur können die in VTK üblichen Veränderungen der Kameraposition und die Objektauswahl durchgeführt werden. *mihMovinBones* bietet die Möglichkeit die einzelnen Fragmente unabhängig voneinander in ihrer Farbe und Transparenz zu verändern (siehe Abbildung 28). Eine unterschiedliche Farbe für die Objekte dient dann der Übersicht, wenn die Objekte eng beieinander liegen und dadurch nur noch schwerlich voneinander zu unterscheiden sind. Das Setzen der Objekttransparenz erlaubt die Ansicht innerer Strukturen, die durch ein Objekt verdeckt werden. Auch ist die Transparenz im Zusammmenhang mit der Kollisionsbehandlung sinnvoll, wenn kollidierende Objektdreiecke farbig markiert werden (siehe Abbildung 34).

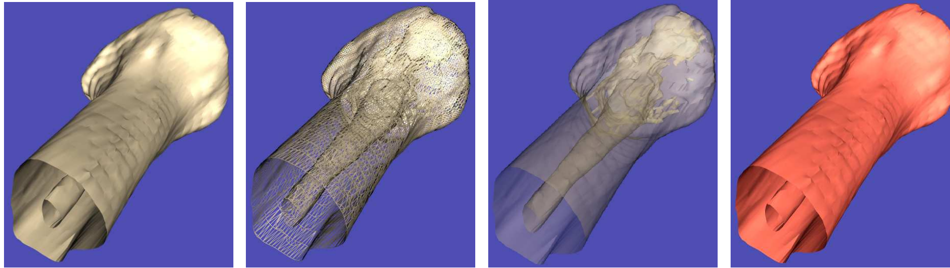


Abbildung 28: Vier verschiedene Visualisierungen eines Knochenfragments

Das visuelle Hervorheben (*Highlighting*) des aktuell ausgewählten Objekts kann auf unterschiedliche Art und Weise geschehen. Auch ist das Einblenden vom objektbezogenen Koordinatensystem des jeweiligen Objekts möglich.

- Wie bereits aus dem vorherigen Punkt zu entnehmen ist, beinhaltet die Anwendung eine Kollisionserkennung zwischen den einzelnen Objekten, die ein- und ausgeschaltet werden kann. Als Kollisionsbibliothek wurde aus den in 4.1.1 genannten Gründen V-Collide gewählt. Die beiden von V-Collide angebotenen Arten der Informationsrückgabe bezüglich der Kollisionserkennung sind dabei berücksichtigt worden. Tests der eingebundenen Kollisionsbibliothek überprüften die Performanz von V-Collide in Abhängigkeit verschiedener Szene- und Objekttypen. Die Testergebnisse sind in 5.2.1 aufgeführt.
- *mihMovinBones* bietet durch eine Explosionsdarstellung der vorliegenden Fraktur die Gelegenheit sich eine Übersicht über alle Objekte und deren Konstellationen zueinander zu verschaffen. Hierbei kann der Anwender die Szenenobjekte sternförmig stufenlos auseinanderziehen und wieder zusammenfügen. Die eingezeichneten Pfade, die die Bewegungsrichtungen der Objekte visualisieren, verdeutlichen gleichzeitig die geometrischen Verhältnisse zueinander. Die zusätzlich eingeblendeten Beschriftung der Fragmente ist hilfreich für eine Übersicht der Objekt-Name-Zuordnungen, die zum Beispiel bei der optionale Objektauswahl über das Menu von Bedeutung sind.
- Um die beschriebene 3D-Darstellung der Szene mit einer dem Anwender vertrauten Betrachtungsweise zu kombinieren, beinhaltet das Programm drei Widgets, die die Volumendaten schichtweise in den drei ortogonal zueinander liegenden Richtungen, sagittal, coronal und axial, anzeigen. Ein direkter Bezug zur 3D-Szene ist gegeben, indem die Objektkonturen über den Schichtdaten wahlweise illustriert werden können. Eine weitere Beziehung zwischen beiden Visualisierungsarten wird durch die Anzeige der aktuellen Schichtpositionen im 3D-Widget ermöglicht. Der Anwender kann damit permanent sehen, welcher Datenausschnitt in Form einer Ebene aus dem gesamten Datensatz gerade angezeigt wird. Die Fensterung der schichtweise angezeigten Volumendaten erfolgt mittels Mausinteraktionen.

- Alle Objekte können einzeln oder gruppenweise verschoben und rotiert werden. Für die Rotation kann dabei ein beliebiger Punkt oder eine beliebige Achse als Rotationsobjekt gewählt werden. Diese werden entweder im 3D-Widget oder in einem der Schichtbilder per Mausinteraktion festgelegt und darauf hin angezeigt. Dem Anwender ist damit die Möglichkeit gegeben ein Objekt durch mehrere Teilschritte anhand verschiedener Rotationsobjekte korrekt zu repositionieren. Die einzelnen Teiltransformationen werden einer Historie zugefügt und können in einem Drop-Down-Menu nachträglich betrachtet und bei Bedarf gelöscht werden. Der Anwender kann dadurch den Repositionierungsprozess anschließend betrachten und gegebenenfalls korrigieren.

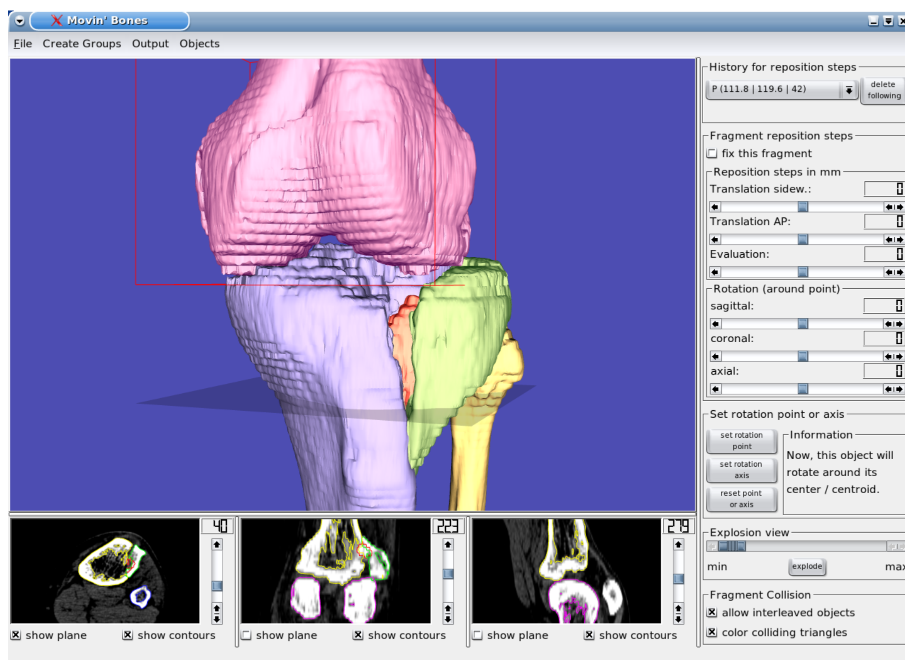


Abbildung 29: Kompletter Screenshot der mihMovinBones-Anwendung. Der rote Rahmen hebt das aktuelle Objekt hervor. Die transparente Ebene visualisiert die Schichtposition der sagittalen Ansicht. Die Schichtansichten beinhalten jeweils die Konturen der Objekte.

- Um einzelne Schritte der Korrekturmaßnahmen auch visuell zu speichern, können jederzeit Screenshots des 3D-Widgets erzeugt werden, die als PNG-Datei gesichert werden. Außerdem kann bei Bedarf ein HTML-Dokument erzeugt werden lassen, dass Informationen über die bearbeiteten Daten, die Transformationshistorie und beliebig viele zuvor erstellte Screenshots beinhalten kann. Diese Ausgabe soll dem Arzt während der Operation helfen, die präoperativ durchgeführten Schritte gegebenenfalls nochmals betrachten zu können. Das HTML-Format wurde aufgrund seiner hohen Kompatibilität gewählt.
- Um ungültige Interaktionen des Anwender zu unterbinden, wurden die

Funktionalitäten drei unterschiedlichen Modii zugeordnet. Der aktive Modus legt damit die aktuellen Funktionsmöglichkeiten fest.

4.3 Softwaretechnischer Entwurf

In diesem Abschnitt wird der softwaretechnische Aufbau von *mihMovinBones* erläutert. Die Klasse `framework` stellt dabei das elementare Gerüst der Anwendung dar. Das Zusammensetzen der GUI, also die grafische Anzeige der Applikation, bestehend aus den einzelnen Widgets, wird hier durchgeführt. `framework` ist für das Managen des notwendigen Datenaustausches zwischen den einzelnen Instanzen verantwortlich.
Die für die Anwendung nötigen Daten, die über eine Instanz der Klasse `mihPatientData` eingelesen werden und anschließend über Methodenschnittstellen dieser Klassen erreichbar sind, werden an die passenden Klasseninstanzen in `framework` weitergereicht. Als Beispiel seien hier die Widgets für die visuellen Ausgaben oder die Schnittstellenklasse für die Kollisionserkennung `mihCollisionWorld` genannt, die zu Beginn die Szenedaten benötigen. Anhand des aktuellen Modusstatus, der sich zu Beginn im Standardmodus befindet, werden die Qt-Elemente in den verschiedenen Widgets initialisiert.

Alle Widgets besitzen Interaktionsmöglichkeiten für den Anwender. Zudem sind alle Widgets mit der Hauptklasse `framework` über diverse `connect()`-Verbindungen angeschlossen. Für gewöhnlich findet nach einer Aktion des Anwenders, die ein Signal auslöst, in den Slot-Methoden der Widget-Instanzen eine Vorverarbeitung der Ereignisse statt. Anschließend teilen diese wiederum über Signale dem Framework die Art und gegebenenfalls die Inhalte mit. Die Informationen werden in `framework` innerhalb diverser Slot-Methoden weiterverarbeitet und anschließend an die entsprechenden Instanzen verteilt. Exemplarisch wird nun der Vorgang der Auswahl von Rotationsobjekten aufgeführt:

Der Anwender möchte ein neues Rotationsobjekt für das aktuell ausgewählte Knochenfragment festlegen. Im `mihQtRotationPointAxisWidget` wird aufgrund der Buttonbenutzung der Slot `slotSetRotationPoint()` aktiviert. Der Slot setzt eine interne Flag derart, dass sie Auskunft darüber gibt, dass der Anwender einen Punkt als Rotationsobjekt setzen möchte. Anschließend teilt der Slot der Instanz von `framework` über ein emittiertes Signal `timeForSetRotationFeatures()` mit, dass ein neues Rotationsobjekt gesetzt werden soll. Die Information hat damit den Verteiler erreicht und setzt ihren Weg über verschiedene Kanäle fort, denn sowohl die Instanz des 3D-Ausgabefensters als auch die drei schichtendarstellenden Widgets müssen informiert werden, da hier die Möglichkeit besteht Rotationsobjekte zu setzen.

Der Slot `framework::slotChangeRotationPointOrAxis()` wird aufgrund des Signals aufgerufen. Zunächst wird das Widget, in dem der Anwender festgelegt hat, dass ein Rotationsobjekt gesetzt werden soll per Methodenaufruf `getTypeOfRotation()` abgefragt, welchen Rotationsobjekttypen der Anwender setzen möchte. In Abhängigkeit dessen werden sowohl Flags in `framework` gesetzt, also auch den für das Setzen von Rotationsobjekten relevanten Widgets mitgeteilt, dass ein Punkt gesetzt werden soll. Diese veranlassen intern die notwendigen

Schritte. Letztlich wird dem Interactor der 3D-Szene über die Methode `listenForRotPoint()` mitgeteilt, dass der nächste Mausclick auf ein Objekt der Position des neuen Rotationspunktes des aktuellen Objekts entspricht.

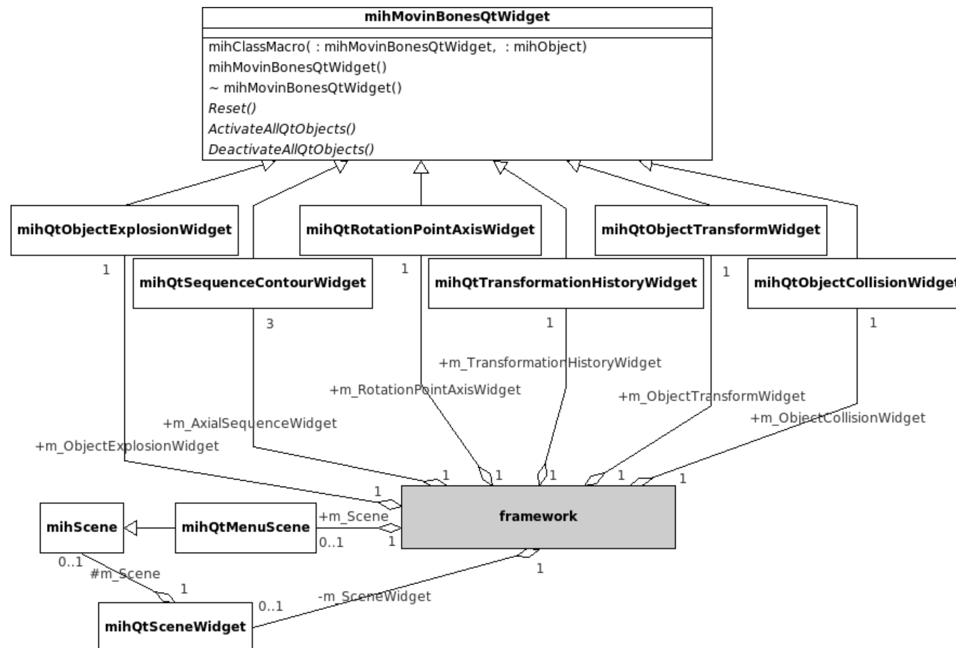


Abbildung 30: Die Struktur aller Klassen, die für die Interaktion relevant sind. Alle 8 erstellten Widgets haben die gleiche Vaterklasse. mihQtScene erweitert mihScene um die Möglichkeit der Interaktion über Qt-Elemente. mihQtSceneWidget ist die Instanz für die 3D-Visualisierung. Die Klasse framework ist für den Informationsaustausch zwischen den Interaktionsinstanzen verantwortlich.

Um das Beispiel fortzuführen, wird angenommen, dass der Anwender im axialen Widget einen Rotationspunkt per Mausclick festlegt. Darauf hin wird der interne Slot `slotSetPointOrAxis()` ausgeführt, der nach einigen Verarbeitungschritten ein Signal `rotationObjectIsSet()` aussendet, dass wiederum in der Instanz von `framework` abgehört und an den Slot `framework::slotRotObjectIsSet()` weitergereicht wird. Der Vorgang erreicht die Verteilerklasse `framework` zum zweiten mal. Die Aufgabe der aktuellen Methode ist nun, allen relevanten Instanzen die Information über das neu gesetzte Rotationsobjekt mitzuteilen. Die Historie muss aktualisiert werden, die Slider werden auf ihre Nullwerte gesetzt und in allen Darstellungsansichten wird das aktuelle Rotationsobjekt eingezeichnet.

Die Objektverwaltung wird genau wie in 4.1.5 beschrieben, gehandhabt. Alle darzustellenden Objekte befinden sich in den `mihScene`-Instanzen, denen sie zu Beginn zugeordnet worden sind. Objekteigenschaften können entweder direkt, über die zugehörigen Aktoren oder über die zugehörigen `vtkProperty`-Instanzen erreicht werden. Lediglich die Transformationen werden vorerst noch separat ge-

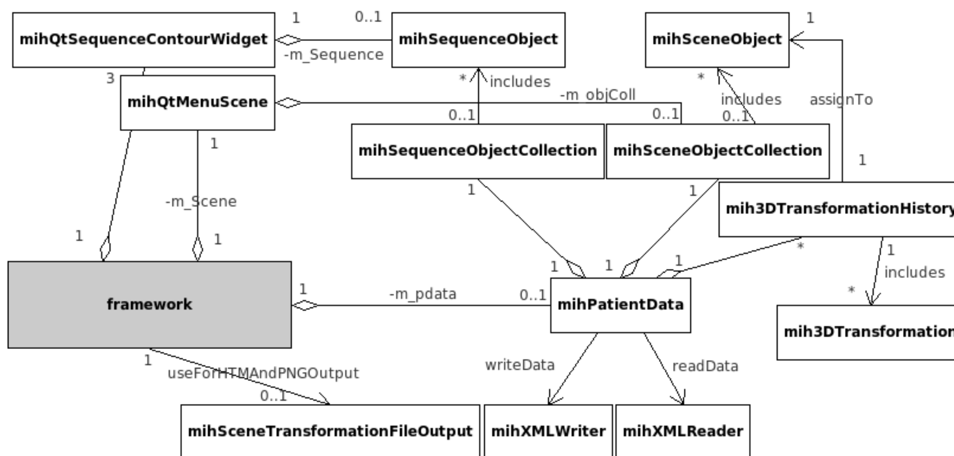


Abbildung 31: Dieses UML-Diagramm verdeutlicht die Verwaltung der vorhandenen Daten. Die Klasse framework dient als Organisator, besitzt selbst nur Zeiger auf die Daten, die sich in der Klasse mihPatientData befinden.

halten. Pro Objekt werden die Transformationen durch die Klassen `mih3DTransformation` und `mih3DTransformationHistory` realisiert. Sie sind nicht direkt in die Klasse `mihSceneObject` implementiert, sondern werden über eine Map in `mihPatientData` den Objekten zugeordnet.

Die Datenausgabe wird in *mihMovinBones* vom Rest abgekapselt. Ein Sichern der Daten im XML-Format erfolgt innerhalb von `mihPatientData`, das die gesamte Anwendungsdauer über die Modell- und Sequenzdaten beinhaltet. Eine genauere Prozessbeschreibung erfolgt in 4.4. Das Ausgeben der Daten im HTMl-Format erfolgt ebenfalls in einer von `framework` abgekapselten Klasse `mihSceneTransformationFileOutput()`. Lediglich die Screenshots des 3D-Widgets werden innerhalb der Hauptklasse `framework` angefertigt.

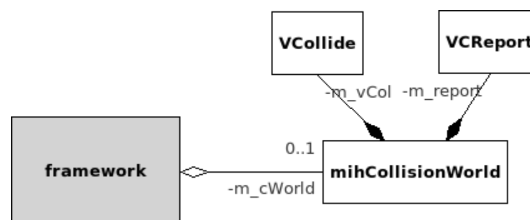


Abbildung 32: Darstellung der Schnittstelle von *mihMovinBones* und V-Collide.

Mit der Klasse `mihCollisionWorld` wurde eine klare Schnittstelle zu V-Collide definiert, die das Kommunizieren zwischen V-Collide und *mihLib*-Objekten auf einfache und schnelle Weise erledigt.

4.4 Realisierung

In diesem Kapitel wird auf die Schwerpunkte der Implementierung eingegangen. Dabei handelt es sich um die Ein- und Ausgabe der Daten, verschiedene Schnittstellen, das Prinzip der unterschiedlichen Modii, die Realisierung der Visualisierungen, und die Verwaltung und Umsetzung der Transformationsdaten.

4.4.1 Ein- und Ausgabe der Daten

Im Rahmen der Entwicklung von *mihMovinBones* wurde die Bibliothek *mihLib* um die Möglichkeit erweitert, Informationen über die vorhandene Szene, insbesondere Informationen über Transformationen der enthaltenen Objekte und Visualisierungseigenschaften, im kompatiblen XML-Format zu laden und zu speichern. Dazu wurde die Klasse `mihPatientData` um diverse Funktionen erweitert. Sie bot bisher die Möglichkeit, Szenedaten aus INFO-Dateien zu laden. Listing 2 zeigt anhand einer vollständigen INFO-Datei die stark begrenzten Möglichkeiten der Informationssicherung auf, die mit diesem Format gegeben sind.

Listing 2: Beispiel einer vollständigen INFO-Datei. Neben einigen Patientendaten wird lediglich auf die Volumendaten in einer LIST-Datei und die Modelldaten in VTK-Dateien hingewiesen.

```
[Info]
hh02012005ifmi
Racho
Volker
01 01 1977
24 01 2005
12 15 30
[Models]
1
Head
/path/to/3D/model/data/in/VTK/format/3D_example.vtk
[Sequences]
1
CT_Data
/path/to/directory/with/DCM/slice/data/ct_example.list
```

Mit Hilfe der Klasse `mihXMLReader` wurde ein XML-Parser integriert, der XML-Dateien nach relevanten Informationen durchsucht und diese in globalen Instanzen sichert. VTK-Informationen werden an `mihSceneObject`-Instanzen übergeben, die Volumendaten werden an `mihSequenceObject`-Instanzen weitergereicht. Das Format einer gültigen XML-Datei kann auszugsweise im Anhang betrachtet werden. Für das Parsen wird ein Zeiger auf eine Instanz vom `mihXMLReader` in Abhängigkeit der verschiedenen XML-Tags an unterschiedliche Methoden übergeben:

```
bool ReadSequencesFromXMLFile(mihXMLReader*);
bool ReadModelsFromXMLFile(mihXMLReader*);
bool ReadElementar3DModelDataFromXMLFile(mihXMLReader*);
bool ReadAdditional3DModelDataFromXMLFile(mihXMLReader*);
bool Read3DModelTransformationHistory(mihXMLReader*,
        const char*);
```

Die Methoden liefern boolesche Werte zurück, die dem Erfolg des Einlesens entsprechen. Die Implementation ist derart gestaltet, das die Reihenfolge der XML-Tags in einer Hierarchieebene gemäß der Spezifikation aus der zugehörigen XSD-Datei irrelevant ist. *mihMovinBones* erhält die vollständigen Daten über eine Instanz der `mihPatientData`-Klasse, in der alle notwendigen Daten enthalten sind

und auf die über Methodenschnittstellen zugegriffen werden kann.
Falls INFO-Dateien geladen werden, ordnet *mihMovinBones* die Szenedaten analog zu den Daten aus XML-Dateien den `mihSceneObjects`- und `mihSequenceObjects`-Instanzen zu. Jeder Instanz eines 3D-Modells wird zu Beginn eine `mih3DTransformationsHistory` zugeordnet, da diese nicht in den INFO-Dateien gesichert werden können. Aufgrund der optionalen Speicherung zusätzlicher Daten mittels XML kann diese Zuordnung innerhalb der `mihPatientData` gegebenenfalls auch beim Laden von XML-Dateien erfolgen.
Für das Schreiben der Daten in eine XML-Datei kann zunächst über eine Methoden-Schnittstelle von `mihPatientData` definiert werden, welche Daten gesichert werden sollen. Es können Volumendaten, Modelldaten, Modelleigenschaften und Modelltransformationen separat selektiert werden. Informationen über Gruppenzugehörigkeiten können nicht gespeichert werden. Nach Selektion wird der `mihXMLWriter` verwendet, um die Informationen aus den veränderten `mihSceneObjects`, `mih3DTransformationHistory` und den unveränderlichen `mihSequenceObjects` in eine XML-Datei strukuriert hineinzuschreiben und unter einem vom Anwender bestimmten Namen zu sichern. Die Volumen- und Objektdaten werden beim Laden, Verarbeiten oder Sichern niemals direkt verändert. Sie werden anhand einer Pfadangabe in die XML-Ausgabe integriert. Das hat zum einen den Vorteil, dass die XML-Daten eine kleine Datengröße aufweisen. Für ein und denselben Datensatz einer Knochenfraktur können beliebig viele XML-Dateien erzeugt werden, die den gleichen Datensatz auf unterschiedliche Weise von außen manipulieren, ohne durch die Speicherung redundante Datensätze zu erzeugen.

Eine weitere Ausgabemöglichkeit der Szenedaten erfolgt im HTML-Format. Der Anwender kann während des Arbeitsverlaufs jederzeit Screenshots der 3D-Szene erstellen, die im PNG-Format gesichert werden. Nach Beendigung der Korrekturmaßnahmen kann über das Menu eine HTML-Ausgabe angefordert werden, die unter dem vom Anwender gegebenen Namen gesichert wird. Der CSS-Teil befindet sich im Header der HTML-Datei und nicht in einer separaten Datei. Die HTML-Ausgabe beinhaltet zunächst elementare Daten wie zum Beispiel das Datum der Erzeugung, Name der geladenen Szene-Datei und gegebenenfalls Name der gesicherten Szene-Datei. Anschließend folgen die gewünschten Screenshots. Unterhalb der Bilder sind tabellarisch die Repositionierungsschritte für die einzelnen Knochenfragmente aufgelistet, wobei pro Rotationsobjekt jeweils die faktorisierten Gesamttransformationswerte angegeben werden. Pro Knochenfragment werden abschließend noch die 3 Translations- und 3 Rotationsschritte aufgelistet, die das zugehörige Objekt in die finale Position und Orientierung transformieren. Diese Informationen über Knochenfragment-Transformationen werden mittels Methodenschnittstellen per Zeiger übergeben. Die Erstellung der einzelnen Parts wird durch verschiedene Methoden innerhalb der Klasse `mihSceneTransformationFileOutput` realisiert. Im Anhang ist die grafische Ausgabe einer solchen Datei zu finden.

Interface für den Anwender Das Laden und Speichern von Szenedaten, sowie das Erstellen von Screenshots im PNG-Format und die Ausgabe der Transformationen im HTML-Format kann der Anwender über das Menu des Mainframes von *mihMovinBones* auswählen. Anschließend erscheinen vordefinierte Qt-Dialog-

Elemente, die Eingabe der notwendigen Daten, zum Beispiel den Namen der Datei, ermöglichen.

4.4.2 Schnittstelle zu Qt

Die Schnittstelle zu den Qt-Objekten ist einerseits durch *mihLib*-Klassen wie `mihQtSceneWidget` gegeben. Alle im Rahmen von *mihMovinBones* erstellten Qt-Widgets stellen weitere Schnittstellen zu Qt dar. Die Klassen haben die Eigenschaft, dass sie von der Klasse `mihMovinBonesQtWidget` abgeleitet sind. Die virtuelle Klasse dient als Schnittstellenklasse und bedingt das Implementieren von einigen Methoden innerhalb der abgeleiteten Klassen. Die bedingten Methoden ermöglichen das Aktivieren und Deaktivieren aller Qt-Objekte innerhalb eines Widgets. Eine weitere zu implementierende Methode setzt die Widgets und damit auch die Qt-Elemente auf den initialen Zustand zurück. Besondere Verhaltensmuster der Qt-Elemente innerhalb der Widgets werden in den von `mihMovinBonesQtWidget` abgeleiteteten Klassen implementiert (siehe Abbildung 30).

4.4.3 Schnittstelle zu V-Collide

Mit der Klasse `mihCollisionWorld` wurde für das institutseigene Framework *mihLib* eine Schnittstelle für die Kollisionsbibliothek V-Collide geschaffen. Über die Methode `AddCollisionObject(int,mihSceneObject*)` können Objekte der Kollisionsdetektion zugefügt werden. Der Integer-Wert beinhaltet die ID für das Objekt, die für die Identifizierung der Objekte notwendig ist. Innerhalb der Methode werden die Punkte aller Dreiecke des Objekts mit einer eindeutigen ID an V-Collide übergeben. Das Objekt wird anschließend aktiviert, das heisst es wird bei der Kollisionserkennung berücksichtigt. Sollte ein vorliegendes Szeneobjekt neben Dreiecken noch andere Polygontypen enthalten, muss zuvor eine Vorverarbeitung zum Beispiel durch den `vtkTriangleFilter` stattfinden.

Die Übergabe von Transformationen von VTK zu V-Collide geschieht mit der Methode `SetTransformation(int,vtkMatrix4x4*)`. Der Integer-Wert gibt das Objekt an, für das die Transformation ausgeführt werden soll. Alle Matrixwerte werden vor der Übergabe an V-Collide bis auf 5 Stellen hinter dem Komma auf- oder abgerundet, da die Matrix-Werte ansonsten innerhalb von V-Collide zu Endlosschleifen führen können. Für die Übergabe werden die Werte aus der Matrix, verpackt in einem 2D-Array an V-Collide übergeben und die Kollisionserkennung gestartet. Geschieht aufgrund der Transformation eine Kollision, gibt die Methode `FALSE` zurück, andernfalls `TRUE`. In welchem Mode V-Collide die Kollisiondetektion durchführt, kann mit der Methode `SetColoringCollidingTriangles(bool)` bestimmt werden. Der Anwender hat die Wahl, ob V-Collide lediglich die IDs der kollidierenden Objekte oder pro Kollisionspaar alle kollidierenden Dreieck-IDs zurückgeben soll. Letzteres ist enorm zeitaufwändig (siehe 5.2.1). Der Zugriff auf die Kollisionsergebnisse erfolgt über verschiedene Methoden der `mihCollisionWorld`-Klasse. Um die Effizienz zu optimieren, kann über die Methode `SetActiveObject(int)` der Schnittstellenklasse das Objekt ausgewählt werden, dass transformiert wird. Das hat zur Folge, dass nur die Objektpaare auf Kollision überprüft werden, in denen sich das ausgewählte Objekt befindet. Alle anderen Kollisionen werden ignoriert.

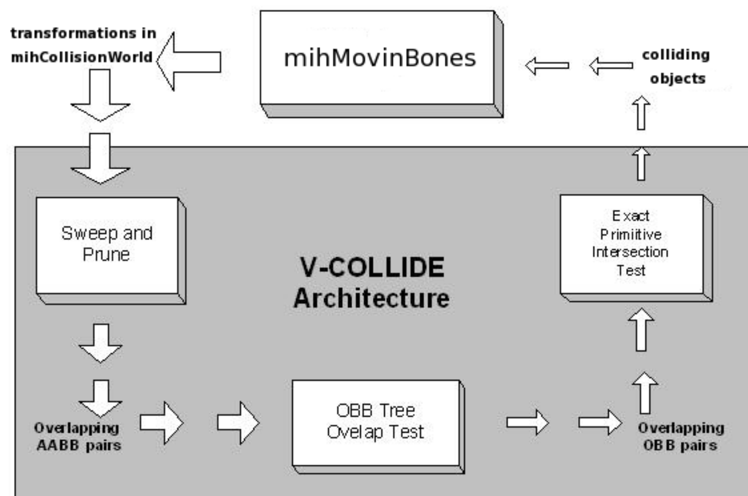


Abbildung 33: Die Grafik verdeutlicht die Schnittstelle zu *mihMovinBones*. V-Collide werden die Objekttransformationen übergeben. Als Rückgabe werden die IDs der kollidierenden Objekte und wahlweise die in die Kollision verwickelten Objektpolygone entgegengenommen.

Interface für den Anwender Die notwendigen Einstellungen an V-Collide kann der Anwender über das `mihQtObjectCollisionWidget` durchführen. Über eine Checkbox wird die Detektion aktiviert oder deaktiviert. In beiden Fällen werden die Transformationsdaten an die Klasse `mihCollisionWorld` übergeben, nur werden bei deaktivierter Kollisionsbehandlung die Objekte unabhängig von den Ergebnissen der Untersuchung transformiert.
Ist die Kollisionserkennung hingegen aktiv, werden die Objekte in Abhängigkeit der Kollisionsergebnisse transformiert. Eine Deaktivierung ist häufig sinnvoll, da durch die Segmentierung einige Knochenfragmente bereits zu Beginn minimal kollidieren. Eine detaillierte Repositionierung der Knochenfragmente wäre somit nicht möglich.

Über eine weitere Checkbox kann *mihMovinBones* aufgefordert werden, Polygone eines 3D-Modells, die aktiv in eine Kollision verwickelt sind, zu markieren. Dabei bezieht sich die Einstellung jeweils nur auf das aktuelle Objekt, nicht auf die Kollisionspartner, denn der Prozess der farblichen Polygonmarkierung ist je nach Kollisionsumfang sehr zeitintensiv. Nach einer Transformation zum Zeitpunkt t müssen pro Objekt mehrere Daten berücksichtigt werden:

- Ermittlung der Polygon-IDs, die bei $t-1$ kollidieren und bei t nicht mehr. Die Farben der Polygone müssen anschließend zurückgesetzt werden.

- Ermittlung der Polygon-IDs, die bei $t-1$ nicht kollidieren, jedoch bei t genau dies tun. Sie müssen anschließend als farbig markiert und gesichert werden.

Im Listing 3 wird durch Pseudocode kurz die Realisierung erläutert. Wenn zum Zeitpunkt t das aktuelle Objekt nicht kollidiert oder die Färbung ausgeschaltet ist, müssen alle gefärbten Polygone zurückgesetzt werden. Wenn das aktuelle Objekt

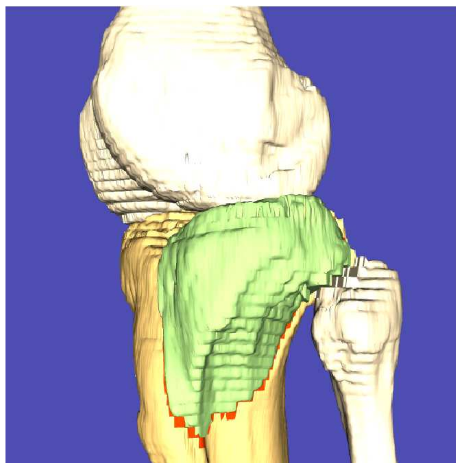


Abbildung 34: Das grün eingefärbte Fragment wurde durch Transformation korrigiert. Anschließend wurden vom orange markierten Fragment die kollidierenden Polygone farbig markiert

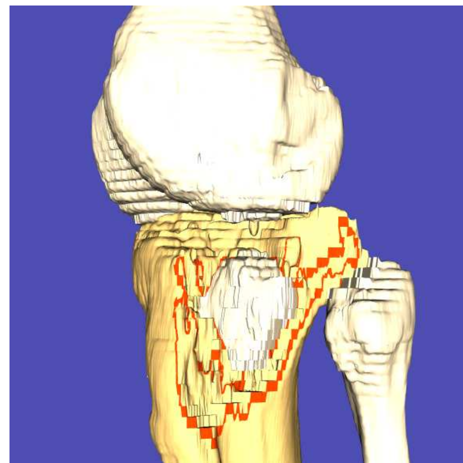


Abbildung 35: Gleicher Sachverhalt wie im Bild links. Zudem wurde die Transparenz vom grünen Fragment auf 100% gesetzt, um die Kollisionsmarkierungen zu betrachten.

bei t kollidiert und die Färbung aktiviert ist, dann muss überprüft werden, welche Polygone farblich verändert werden müssen. Die entsprechenden IDs werden zunächst in zwei Vektoren eingetragen, denn eine unmittelbare Veränderung der Farbeigenschaften der Polygone würde zu einem falschen Resultat führen. Anschließend werden die Polygone, deren IDs in den Vektoren stehen, farblich verändert.

Listing 3: Pseudocode für die Färbung kollidierender Polygone

```
transformObject();
if((noCollision || !coloringFlag) && someTrianglesAreColored)
  setAllColoredTrianglesToStandardColor();
  setPipelineModified();
else if (collision && coloringFlag)
  for(loop over all triangles that collided before)
    if(triangle does not collide actually)
      AddTriangleIDToVector changeToStdColor[];
  for(loop over all actual colliding triangles)
    if(actualTriangleDoesNotCollideBefore)
      AddTriangleIDToVector changeToColColor[];
  for(loop over changeToStdColor[])
    resetTriangleColor();
  for(loop over changeToColColor[])
    colorTriangleColor()
```

Durch die Realisierung werden nur die Polygone angefasst, die bei $t-1$ oder t kollidieren. Trotzdem ist der Rechenaufwand hoch, wodurch unter Umständen die Echtzeitfähigkeit nicht aufrecht erhalten werden kann.

4.4.4 Verhaltensmuster in Abhängigkeit der Modii

mihMovinBones bietet unterschiedliche Funktionalitäten in Abhängigkeit vom aktuellen Modus an. Drei verschiedene Modii sind derzeit in der Anwendung implementiert (siehe Abbildung 36), der aktuelle Modustyp wird stets in der unteren Programmleiste angezeigt. Jeder Modus setzt ein besonderes Verhaltensmuster

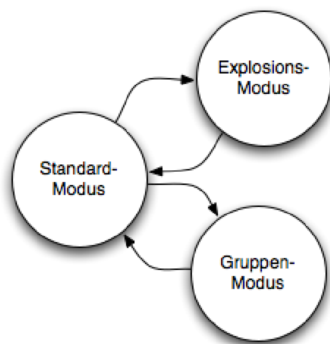


Abbildung 36: Die Grafik zeigt die in *mihMovinBones* existierenden Modii und die zugehörigen gültigen Moduswechsel.

für Qt-Elemente der gesamten Anwendung voraus, um fehlerhafte Interaktionen des Anwenders von vornherein zu unterbinden. Die verschiedenen Qt-Elemente befinden sich in den einzelnen Widgets, über die der Anwender für bestimmte Funktionen interagieren kann. Zum Beispiel kann im Gruppierungsmodus nicht die Explosionsdarstellung aktiviert werden, der entsprechende Button im Widget für die Explosionsdarstellung muss deaktiviert werden.
Es wird zwischen Normalmodus, Gruppierungsmodus und Explosionsmodus differenziert. Welche Moduswechsel erlaubt sind, zeigt die Abbildung 36. Jeder Modus wird letztlich durch eine Flag in der Klasse `framework` realisiert. Beim Wechsel von einem in den anderen Modus wird anhand einer Methode `bool SetMode(const char*)` überprüft, ob der aktuelle Modus den Wechsel gewährt. Nach akzeptiertem Moduswechsel werden innerhalb der Methode den involvierten Instanzen der Widgets die notwendigen Verhaltensmuster für deren Qt-Elemente mitgeteilt.
Bei einer Anwendungserweiterung durch zusätzliche Modii bedarf es lediglich einer Anpassung dieser Methode, um das Verhalten der einzelnen Widgets bei Aktivierung des neu implementierten Modus zu definieren. Entsprechende Schnittstellen der einzelnen Widgets müssen dabei vorhanden sein. Einige dieser Schnittstellen werden stets durch die Schnittstellenklasse `mihMovinBonesQtWidget` (siehe 4.4.2) implementiert.

4.4.5 Verwaltung der Transformationen

Mit den Klassen `mih3DTransformation` und `mih3DTransformationHistory` werden die rigiden Transformationen (Rotation und Translation) für die einzelnen Knochenfragmente verwaltet. Dabei sind die Klassen vorerst nicht direkt in `mihSceneObject` integriert. Die Zuordnung und Verwaltung findet in `mihPatientData` statt. Jede Instanz einer Transformation zeichnet sich durch ein eigenes Rotationsobjekt aus, anhand dem die Rotationen durchgeführt werden. Das kann ein Punkt mit Punktkoordinaten (x, y, z) oder eine Achse mit Anfangs- und Endpunktkoordinaten (x_1, y_1, z_1) und (x_2, y_2, z_2) sein.
Die Verwaltung und Realisierung der Transformationen erfolgt mit 4×4-Matrizen.

Eine Instanz von `mih3DTransformation` beinhaltet zahlreiche globale Matrizen vom Typ `vtkMatrix4x4`, die je nach Bedarf zusammengesetzt werden. Da Matrizenmultiplikationen einige Schwächen aufweisen (nicht kommutativ, uneindeutig, Gefahr von Gimbal Locks), bedarf der Umgang mit den Transformation, die willkürlich vom Anwender vorgegeben werden, einige Überlegungen.

Sobald das Rotationsobjekt für ein Szeneobjekt über Methodenschnittstellen festgelegt ist, beginnt in der Instanz von `mih3DTransformation` das Berechnen von einigen inneren Matrizen durch Aufruf der `setMatrices()`-Methode. Falls es sich beim Rotationsobjekt um einen Punkt handelt, werden zwei Transformationsmatrizen $M_{T_{to}}$ und $M_{T_{from}}$ berechnet, die eine Translation, abhängig von der Position des Rotationspunktes, in den Ursprung und zurück durchführen. Die Rotation um einen Punkt wird mit der Matrix $M_{R_{tmp}}$

$$M_{R_{tmp}} = M_{T_{from}} \cdot M_{R_{ges}} \cdot M_{T_{to}} \,, \tag{23}$$

durchgeführt. Wie genau die Berechnung von $M_{R_{ges}}$ geschieht, wird im nächsten Absatz behandelt. Wurde als Rotationsobjekt eine Achse festgelegt, kommen weitere 4 Matrizen hinzu. $M_{R1_{to}}$ und $M_{R1_{from}}$ sind Rotationsmatrizen, die die gegebene Achse um die X-Achse in die XZ-Ebene hin- und zurückrotieren. Die Matrizen $M_{R2_{to}}$ rotiert die in der XZ-Ebene liegende Achse um die Y-Achse derart, dass sie anschließend genau auf der Z-Achse liegt. $M_{R2_{from}}$ entspricht der inversen Matrix von $M_{R2_{to}}$. Eine Rotationsmatrix $M_{R_{tmp}}$ um eine Achse wird wie folgt berechnet:

$$M_{R_{tmp}} = M_{T_{from}} \cdot M_{R1_{from}} \cdot M_{R2_{from}} \cdot M_{R_{ges}} \cdot M_{R2_{to}} \cdot M_{R1_{to}} \cdot M_{T_{to}} \,, \tag{24}$$

$M_{T_{to}}$ beschreibt in diesem Fall die Translation der Achse bzw. des ersten Achsenpunktes in den Ursprung, $M_{T_{from}}$ dessen Inverse.

Bei der Realisierung der Rotationen war eine Steuerung gewünscht, bei der über drei Slider anhand der fixen Achsen des Weltkoordinatensystems rotiert werden kann. Diese Art der Steuerung ist intuitiver, als wenn der Anwender um eine bereits rotierte Achse rotieren muss. Zudem kann die Information über eine Rotation verständlicher angegeben werden. Eine Implementierung in Form einer Eulertransformation mit einer festen Matrizen-Reihenfolge während der Akkumulation würde wie folgt berechnet werden:

$$M_{R_{ges}} = M_{R_z} \cdot M_{R_y} \cdot M_{R_x}$$

Hierbei wird der aktuelle Rotationswert in der entsprechenden Matrix eingetragen. Eine Rotation anhand der festen Koordinatenachsen kann auf diese Weise nicht geboten werden. Würde hier in einer ersten Rotation um die X-Achse mit 45° gedreht, würde eine anschließende Rotation um die Z-Achse nicht mehr um die Achse des WKS, sondern um die transformierte Z-Achse stattfinden.

In der Implementierung werden nun Rotationswerte durch die Methode `setRotationValue(double,int)` der Transformationsinstanz mitgeteilt, wobei der `double`-Wert dem Rotationswert entspricht und der Integer-Wert die Rotationsachse bestimmt. Pro Rotationsschritt erfolgt eine Übergabe. Wird erst 4° axial, dann 2° axial und anschließend 3° sagittal rotiert, sind die Übergabeparameter $(4, 2)$, $(2, 2)$ und $(3, 0)$. Aufgrund der Beschaffenheit der Slider kann in einem

Schritt eine Maximalrotation von 10° erfolgen. Je nach Rotationsobjekt wird eine temporäre Rotationsmatrix $M_{R_{new}}$ erstellt und mit der Gesamtmatrix $M_{R_{ges}}$ multipliziert. Die Konsequenz ist eine fortwährende Matrizenmultiplikation, wobei eine zeitliche Reihenfolge unabhängig von der Rotationsachse besteht:

$$M'_{R_{ges}} = M_{R_{new}} \cdot M_{R_{ges}} \tag{25}$$

Translationen werden durch die Methode `setTranslation(double,double,double)` der Instanz von `mih3DTransformation` mitgeteilt, wobei alle drei Translationswerte übergeben werden. Wird also erst um 4 mm axial und anschließend 6 mm coronal translatiert, sind die Übergabeparameter beim ersten Schritt $(0, 0, 4)$ und beim zweiten Schritt $(0, 6, 4)$. Die Werte in einer internen Transformationsmatrix M_T werden an den entsprechenden Positionen aktualisiert.

Die Gesamttransformation für ein Objekt wird sodann durch eine Akkumulation der Rotationsmatrix $M_{R_{tmp}}$ und der Transformationsmatrix M_T in Form einer Matrix ermittelt und an den stellvertretenden Objekt-Aktor übergeben:

$$M_{ges} = M_{R_{tmp}} \cdot M_T$$

Das Objekt nimmt die vom Anwender gewünschte Position ein.

Faktorisierung Durch die Art und Weise der Matrizenakkumulation gehen die Teiltransformationen verloren. Es wäre möglich die Teilschritte zu sichern, was bei genauerer Betrachtung des Anwendungsfall aber nicht notwendig ist. Der Anwender wird eine korrekte Knochenfragmentposition letztlich durch Ausprobieren einer Vielzahl kleiner Teiltransformationen erreichen. Relevant sind am Ende nur die Einzelwerte für die Rotation und Translation pro Rotationsobjekt, die aus der Gesamtmatrix gewonnen werden können. Die Translationswerte werden durch Addition der Werte aus M_T und $M_{R_{ges}}$, jeweils aus der 4. Spalte, bestimmt. Eine Faktorisierung von M_{ges} ermittelt die gewünschten Rotationswerte.

Da Eulertransformationen nicht eindeutig sind, kann eine beliebige Kombination von Teiltransformationen auf eine Matrizenakkumulation mit fester Reihenfolge zurückgeführt werden, was folgende Gesamtmatrix liefert:

$$M_{R_{ges}} = M_{R_x} \cdot M_{R_y} \cdot M_{R_z} = \tag{26}$$

$$\begin{pmatrix} c(r_y)c(r_z) & -c(r_y)s(r_z) & s(r_y) & 0 \\ c(r_z)s(r_x)s(r_y) + c(r_x)s(r_z) & c(r_x)c(r_z) - s(r_x)s(r_y)s(r_z) & -c(r_y)s(r_x) & 0 \\ -c(r_x)c(r_z)s(r_y) + s(r_x)s(r_z) & c(r_z)s(r_x) + c(r_x)s(r_y)s(r_z) & c(r_x)c(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{27}$$

Die drei Rotationswerte werden wie folgt ermittelt:

$$\varphi_y = asin(s(r_y)) \tag{28}$$

$$\varphi_x = atan2(-c(r_y)s(r_x),\ c(r_x)c(r_y)) \tag{29}$$

$$\varphi_z = atan2(c(r_y)s(r_z),\ c(r_y)c(r_z)) \tag{30}$$

Bei der Faktorisierung wird `atan2(a,b)` anstelle von `atan(a/b)` verwendet. `atan2(a,b)` kann Fälle mit $b = 0$ behandeln, während ansonsten eine Division durch 0 zu einem Fehlverhalten der Anwendung führte. Der Zugriff auf diese Werte erfolgt über Funktionsaufrufe. Die Ergebniswerte sind nicht eindeutig. Das heisst, für ein und dieselbe Objektposition und -ausrichtung können unterschiedliche Werte für die Rotation und Translation ermittelt werden. Wichtig ist zudem, dass der Anwender die Reihenfolge der Rotationen einhält, die aufgrund der Faktorisierung gegeben ist: Sie bedingt eine Rotation erst in axialer (y), dann in coronaler (z) und schließlich in sagittaler (x) Richtung. Letztlich kann der Anwender mit den ausgegebenen Werten und in korrekter Rotationsreihenfolge das gewählte Knochenfragment auf die gewünschte Position repositionieren. Da es bei der Faktorisierung zu einem Gimbal Lock kommen kann, bedarf es einiger Vorkehrungen bei der Faktorisierung von $M_{R_{ges}}$. Es folgt eine allgemeine Beschreibung des Gimbal Locks:

Gimbal Lock Beim Gimbal-Lock wird durch eine 90° Rotation um eine Achse eine andere überlagert. Eine anschließende Rotation um die überlagerte Achse rotiert nicht um die gewünschte Achse, sondern um die zuvor gedrehte Achse. Dadurch kommt es zu fehlerhaften Resultaten in der Gesamttransformation. Grund dafür ist der Verlust eines Freiheitsgrades, da nach der 90° Rotation nur noch zwei Achsen zur Verfügung stehen. In der Anwendung würde sich das in der Form auswirken, dass einer der Slider nicht mehr um die ihm zugeordnete Achse rotiert. Als Beispiel sei die folgende Rotationsmatrix aufgeführt, die eine 90°-Rotation um die y-Achse beinhaltet:

$$M_{R_{ges}} = M_{R_x} \cdot M_{R_y} \cdot M_{R_z} \tag{31}$$

$$= \begin{pmatrix} 0 & 0 & 1 & 0 \\ c(r_z)s(r_x) + c(r_x)s(r_z) & c(r_x)c(r_z) - s(r_x)s(r_z) & \mathbf{0} & 0 \\ -c(r_x)c(r_z) + s(r_x)s(r_z) & c(r_z)s(r_x) + c(r_x)s(r_z) & \mathbf{0} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{32}$$

s und c stehen aus Platzgründen für sin und cos. In der dritten Spalte resultieren die zwei hervorgehobenen Nullen aus $cos(90°)$. Sie sind bei weiteren Multiplikation für den Freiheitsverlust verantwortlich.
Bei der Faktorisierung innerhalb von *mihMovinBones* treten Gimbal-Locks auf, wenn die Gesamtrotation 90°-Rotationen enthält. Der Octave-Code in Listing 4 zeigt die Behandlung für diese Sonderfälle. Ein deartiges Ereignis passiert äußerst selten während der Repositionierung, da zahlreiche Multiplikationen von double-Werten selten exakt den Wert 0 ergeben. Dennoch werden in diesem Fall die Werte festgelegt.

Listing 4: Sonderbehandlung bei der Faktorisierung einer Matrix Mges. Die Variable rad führt eine Umwandlung der Winkelwerte von Bogenmaß in Grad aus.

```
if(Mges(2) > 0.998) # singularity at north pole
        roty = atan2(Mges(9), Mges(11)) / rad
        rotz = PI/2 / rad
        rotx = 0
elseif(Mges(2) < -0.998) # singularity at south pole
        roty = atan2(Mges(9), Mges(11)) / rad
        rotz = -PI/2 / rad
        rotx = 0
else # no singularities
        roty = asin( Mges(9)) / rad
        rotz = atan2( -Mges(5), Mges(1)) / rad
        rotx = atan2( -Mges(10), Mges(11)) / rad
endif
```

Abschließened noch die Beschreibung des Zusammenfügens von den Einzeltransformationen. Mehrere Transformationen eines Objekts anhand verschiedener Rotationsobjekte werden mit der Klasse `mih3DTransformationHistory` verwaltet. Hier kann sowohl auf einzelne, als auch auf miteinander multiplizierte Teiltransformation zurückgegriffen werden. Dadurch besteht die Möglichkeit sich die Einzelschritte der Repositionierung ausgeben zu lassen. Des Weiteren kann die Gesamtmatrix zurückgegeben werden, die stets die aktuelle Position des Objekts repräsentiert. Die Klasse ermöglicht das Löschen von beliebigen Teiltransformationen. Dabei ist zu beachten, dass beim Löschen einer inneren Transformation auch alle nachfolgenden Transformationen verworfen werden. Ein Löschen einer einzelnen inneren Transformation würde eine für den Anwender nicht nachvollziehbare Änderung der Gesamttransformation zur Folge haben.

Interface für den Anwender Das Drop-Down-Menu in `mihQtTransformationHistoryWidget` erhält pro neu gewähltem Rotationsobjekt einen neuen Eintrag. Nach dem Einlesen der Daten aus einer INFO-Datei, in der keine Informationen über Transformationen gesichert werden können, ist zunächst der Mittelpunkt des Fragments als Rotationsobjekt eingetragen. Ansonsten wird stets die letzte Teiltransformation als die aktuelle ausgewählt. Werden im Laufe der Bearbeitung eines Frakturfragments verschiedene Rotationsobjekte hinzugefügt, füllt sich das Menu. Die Namensgebung erfolgt anhand der Koordinaten. Für einen Punkt wird der Text 'P(x,y,z)' eingetragen, für eine Achse 'A(x,y,z)-(x,y,z)'.
Mit dem Drop-Down-Menu ist dem Anwender die Möglichkeit gegeben, sich seine einzelnen Transformationsschritte nachträglich noch einmal anzuschauen. Denn sobald ein Eintrag bzw. ein Rotationsobjekt im Menu ausgewählt wird, verändern sich entsprechend Position und Orientierung des 3D-Objektes im Render-Fenster. Dazu aktualisieren sich die Konturen in den drei `mihQtSequenceContourWidget`-Instanzen.

Die sechs Slider für Rotation und Translation im `mihQtObjectTransformWidget` nehmen die Werteposition ein, die aufgrund der beschriebenen Faktorisierung ermittelt wurden. Die Slider können in diesem Mode aber nicht manipuliert werden, da die Konsequenzen der Veränderung für den Anwender nicht nachvollziehbar wären. Eine Veränderung der Sliderwerte entspräche einer Veränderung einer Matrix inmitten einer Verkettung von Matrixmultiplikationen. Lediglich bei Auswahl des letzten Menueintrages und damit des aktuellen Rotationsobjekts können die Slider in gewohnter Weise verändert werden, da in Folge dessen auch

nur die letzte Matrix in der Verkettung verändert wird.

Wie bereits erwähnt, ermöglicht das Durchwandern des Drop-Down-Menus die zeitliche Abfolge der einzelnen Transformationsschritte anhand der verschiedenen Rotationsobjekte. Rotationsobjekte, anhand der keine Transformationen durchgeführt wurden, werden verworfen. Zusätzlich kann der Anwender durch Drücken des 'delete following' Buttons nachfolgende Rotationsobjekte und damit die dazugehörigen Transformationen verwerfen. Diese Möglichkeit ist durchaus sinnvoll, wenn sich beim Arbeiten herausstellen sollte, dass der gewählte Transformationsansatz bestehend aus mehreren Teiltransformationen zu keinem zufriedenstellendem Ergebnis führt.

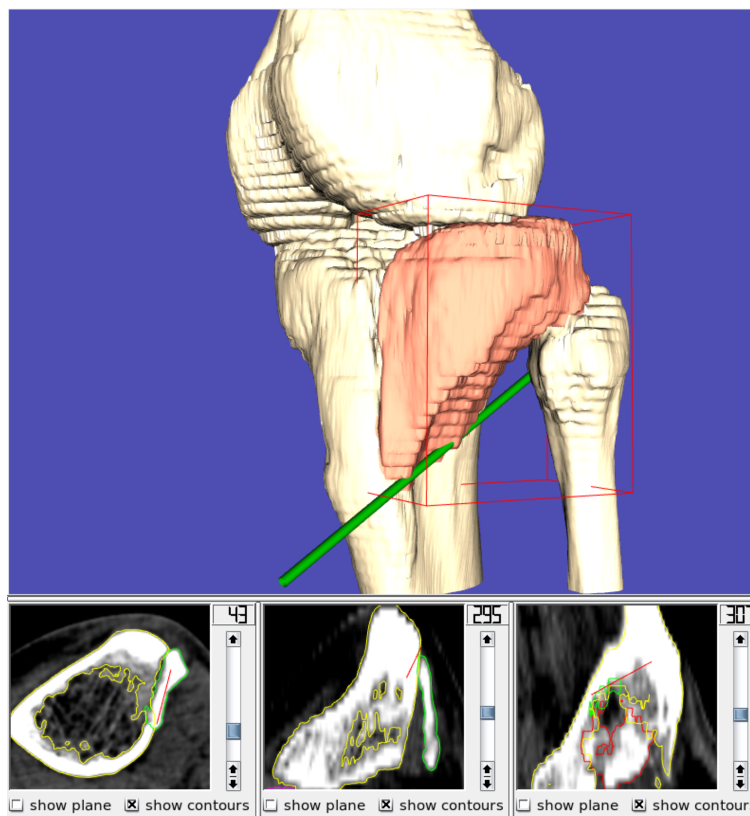


Abbildung 37: Für das farbig markierte Objekt wurde eine Rotationsachse gesetzt, die in der 3D-Szene und in den drei Schichtbildern eingezeichnet ist.

Setzen der Rotationsobjekte Knochenfragmente können mit *mihMovinBones* an Rotationsachsen- und punkten rotiert werden. Diese Rotationsobjekte können sowohl in den drei Schichtdarstellungen, als auch direkt in der 3D-Szene gesetzt werden. Vor dem Auswählen der Rotationsobjekte muss in dem `mihRotationPointAxisWidget` der Typ ausgewählt werden, also Punkt oder Achse. Entsprechend werden interne Flags gesetzt, die darauf hinweisen, dass die nächsten Interaktionen per Maus die Positionsdaten für die Rotationsobjekte verschaffen.

In der 3D-Szene werden die Positionen der Rotationsobjekte mit einem Picker festgelegt. Die (x, y)-Position des Cursors auf dem Fenster der dargestellten Szene wird ermittelt. Von dort aus wird ein Strahl entlang der Kamerablickrichtung in die Szene verschossen. Alle von diesem Strahl getroffenen Knochenfragment-Aktoren werden in einer Kollektion zurückgegeben. Zudem kann auf die Schnittpunkte zugegriffen werden. Die Koordinaten des ersten Schnittpunktes vom ersten geschnittenen Objekts werden als neue Koordinaten des Rotationsobjektes ge `wählt.` Dadurch ist es möglich, die Rotationspunkte und -Achsen auf den Knochenmodelloberflächen zu positionieren.

Die Rotationsobjekte können neben der 3D-Szenenansicht auch in den verschiedenen Schichtbildern gesetzt werden. Hier wird die Tiefe der Position bereits durch die Schichtposition bestimmt. Es ist daher möglich Rotationspunkte und -Achsen auch innerhalb von Knochenmodellen und an andere beliebige Raumpositionen zu setzen. Das Setzen der Achsen und Punkte in den Schichtbilder ist häufig präziser, da der Tiefenwert exakt durch die aktuelle Schichtposition festgelegt ist.
Zur Visualisierung existieren zwei globale Aktorinstanzen, die die Rotationsobjekte aller Fragmente repräsentieren. Deren zugeordnete Mapper bekommen als Datenquelle VTK-Primitive (`vtkSphereSource` und `vtkLineSource`) übergeben. Sie werden zu Beginn in die `mihScene` eingetragen, jedoch zunächst aufgrund der Invisibilität nicht gerendert. Bei Auswahl eines Knochenmodells werden ihnen die gerade aktuellen Koordinaten zugewiesen. Erst dann werden sie gerendert. Bei Änderung der Rotationsobjekte bekommen sie aus den entsprechenden Instanzen per Signal/Slot-Technik die Koordinaten übergeben.

Interface für den Anwender Mit Hilfe von drei Button kann der Anwender dem Programm mitteilen, dass er für das aktuell ausgewählte Objekt das zugehörige Rotationsobjekt verändern möchte. Zur Auswahl stehen dabei ein Rotationspunkt, eine Rotationsachse oder der Mittelpunkt des Objekts. Letzterer wird extra aufgeführt, weil dieser Punkt als Standardpunkt betrachtet wird und manuell nur schwer präzise gesetzt werden kann.

4.4.6 Visualisierung durch Explosionsdarstellung

Um von der aktuellen Szenenansicht in die Explosionsdarstellung zu wechseln, sind einige Berechnungen nötig. Zunächst wird der Mittelpunkt der Szene $p_c = (x, y, z)$ aus den n vorhandenen Objektmittelpunkten p_o berechnet:

$$p_c = \frac{1}{n} \cdot \sum_{i=0}^{n} (x_i, y_i, z_i)$$

Anschließend wird pro Objekt ein normierter Richtungsvektor $\vec{v}$ berechnet, der dessen Bewegungsrichtung während der Explosionsdarstellung beschreibt:

$$\vec{v_i} = \sqrt{(x_{p_c} - x_{p_o})^2 + (y_{p_c} - y_{p_o})^2 + (z_{p_c} - z_{p_o})^2}$$

Die einzuzeichnenden Bewegungspfade der Objekte werden durch `vtkLine - Source`-Objekte und deren Mapper und Aktoren realisiert. Pro Objekt wird der `vtkLineSource`-Instanz als Startpunkt p_c und als Endpunkt $p_c + s \cdot m_{max} \cdot \vec{v}$

übergeben, wobei m_{max} den maximalen Sliderwert und s einen Skalierungswert repräsentiert. Die Parameter der Aktoren werden über Methodenzugriffe wie gewünscht (nicht selektierbar, Objektfarbe) verändert und anschließend dem Renderer übergeben. Solange die Frakturelemente in der Szene nicht verändert werden, bedarf es keiner Neuberechnung. Sobald ein Fragment transformiert wird, werden die oben genannten Berechnungsschritte beim nächsten Aufruf einer Explosionsdarstellung erneut ausgeführt.

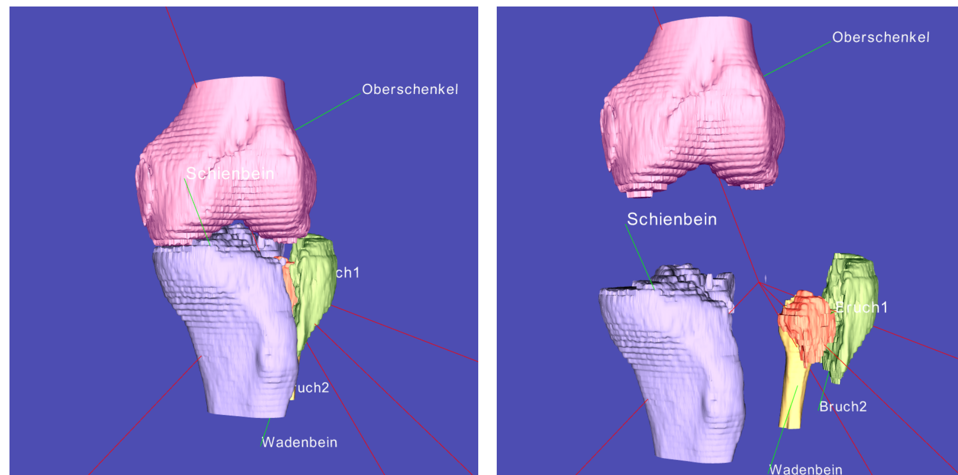


Abbildung 38: Das links Bild zeigt eine Szene bei eingeschalteter Explosionsdarstellung, jedoch liegt der Grad der Explosion bei 0. Im rechten Bild wurde der Explosionsgrad erhöht, wodurch die Objekte sich auseinander bewegen.

Während der Explosionsdarstellung werden die Bezeichner der einzelnen Objekte in die Szene eingefügt und visualisiert. Dazu wird jeweils von p_o eine zu $\vec{v_i}$ orthogonale Linie eingezeichnet, die wiederum mit einem `vtkLineSource`-Objekt realisiert wird. Der Richtungsvektor $\vec{v_t}$ für die Linie kann mit dem Kreuzprodukt zwischen $\vec{v_o}$ und dem Einheitsvektor $\vec{e}$ bestimmt werden:

$$\vec{v_t} = \vec{v_o} \times \vec{e}$$

Für die Positionierung der Bezeichnerlinie pro Objekt ist demnach p_o der Startpunkt und $p_o + b \cdot \vec{v_t}$ der Endpunkt. b ist dabei ein Skalierungsfaktor, abhängig von der Breite der Objekt-Bounding-Box. Direkt am Endpunkt wird der Objektname gesetzt. Dafür wird die Klasse `vtkVectorText` genutzt. Zwischen den Text-Mapper und den Aktor wird ein `vtkFollower` in die Pipeline gesetzt. Das hat zur Folge, dass der Text dauerhaft in Richtung der Kamera ausgerichtet ist. Zuvor muss ein Zeiger auf die aktive Kamera dem Follower übergeben werden, damit die Kameraposition bekannt ist.

Während der Explosionsdarstellung werden die Fragmente in Abhängigkeit des Wertes translatiert, der bei Sliderbewegungen im `mihQtObjectExplosionWidget` übergeben wird. Dazu wird eine Translationsmatrix $M_{T_{exp}}$ erstellt, die multpliziert mit der Gesamtmatrix dem Objekt zugewiesen wird. $M_{T_{exp}}$ wird jedoch

nicht in die Transformationshistorie des Objekts eingetragen. Verlässt der Anwender die Explosionsdarstellung, werden die Objekte wieder an ihre ursprüngliche Position gesetzt, indem ihnen die Gesamtmatrix aus ihrer Transformationshistorie zugewiesen wird.
Die aktive Explosionsdarstellung repräsentiert einen reinen Visualierungsmodus, in dem die Objekte weder ausgewählt noch transformiert werden können. Entsprechend sind die dafür notwendigen Interaktionsobjekte deaktiviert.

Interface für den Anwender Die Interaktion erfolgt über einen Qt-Slider, in dessen Abhängigkeit die Objekte während des Explosionsmodus transformiert werden.

4.4.7 Darstellung der Schichtdaten

Die CT-Daten werden innerhalb von *mihMovinBones* nicht verarbeitet, sondern lediglich visualisiert und dienen damit der Orientierung des Anwenders. Das `mihQtSequenceContourWidget` stellt die Volumendaten aus beliebiger Betrachtungsrichtung (axial, coronal, saggital) schichtweise dar. Die Ansichtsrichtung und der Daten-Spacing-Wert müssen der Instanz neben den Volumendaten zu Beginn übergeben werden. Die Schichtansicht geschieht durch die Klasse `QVTKImageSliceViewer2` , die wiederum `vtkImageViewer2` verwendet. Durch den enthaltenen `vtkInteractorStyleImage`-Style erfolgt die Manipulation der *level*- und *window*-Werte, die für die Fensterung relevant sind, durch Interaktion mit der Maus. Das erspart das Einbinden von zwei weiteren Slidern als Eingabeobjekte.

Berechnen der Objektkonturen Zusätzlich zu den CT-Volumendaten können verschiedene Objekttypen hinzugefügt und visualisiert werden. Im Rahmen vom *mihMovinBones* handelt es sich dabei um `mihSurfaceContourObjects` für die Objektkonturen, die dem Renderer übergeben werden. Die Möglichkeit der Konturüberblendung wurde implementiert, um die dynamischen Transformationsprozesse aus der 3D-Szene auf die sonst statischen Schichtbilder zu übertragen (siehe Abb. 39 und 40). Der anwendende Chirurg kann dadurch bei Bedarf nach wie vor mit der gewohnten Schichtenansicht arbeiten. Zur besseren Unterscheidung werden die Konturen farblich kodiert.
Die Berechnung und das Setzen der Konturen ist in der Klasse `mihQtSequenceContourWidget` gekapselt. Nach Einlesen der Modelldaten bedarf es lediglich der Übergabe eines Zeigers auf die zugehörigen `mihSceneObjects` an die Klasseninstanz. Die Konturen werden mit Hilfe der Klasse `vtkCutter` erzeugt, die als Parameter eine implizite Funktion und die Modelldaten benötigt. Eine Beschreibung des Vorgangs ist in 2.2.4 zu finden. Als impliziete Funktion wird hier eine Ebene (`vtkPlane`) übergeben, die der aktuellen Schichtposition entspricht.
Wird ein Objekt in der Szene transformiert, so aktualisiert sich die Kontur gemäß der Schichtposition. Dieser Vorgang ist aufgrund der teilweise großen Knochenfragmentmodelle teuer (siehe 5.2.2) und beeinträchtigt das Echtzeitverhalten.

Berechnen der Ebenenparameter Dem Anwender ist die Möglichkeit gegeben die aktuellen Schichtpositionen in der 3D-Szenen-Ansicht ein- und auszublenden. Dafür werden bereits im Konstruktor die dafür notwendigen Aktoren hin-

Abbildung 39: Die Originalszene ohne Transformationen. Die Konturen umschließen gemäß der Segmentierung die Objekte

Abbildung 40: Die gleiche Szene wie links, wobei eine Fragmentposition korrigiert wurde. Die grüne Kontour hat sich entsprechend der Objekttransformation aktualisiert

zugefügt, jedoch zunächst nicht dargestellt. Die Positionen der einzublendenen Ebenen ist durch die Betrachtungspositionen der Schichtansichten festgelegt und kann dort ermittelt werden. Die Ausmaße der Ebenen soll so gewählt werden, dass alle Objektprojektionen vollständig darauf abgebildet werden.
Dazu werden alle Bounding-Box-Werte der Szenenobjekte miteinander verglichen und die minimalen und maximalen Werte pro Dimension an die Aktoren der Flächen übergeben.

4.4.8 Erweiterte Interaktionen

Die meisten Interaktionsformen geschehen über die Qt-Elemente, die entweder in den einzelnen Widgets oder im Mainframe-Menu enthalten sind. Die Interaktion über die VTK-Render-Windows (3D-Szene und Volumendatendarstellung) erfolgt über `vtkInteractorStyle`-Typen. Im Rahmen der Entwickung von *mihMovinBones* wurde mit der Klasse `mihSpecialInteractorStyle` ein neuer InteractorStyle für das 3D-Widget erstellt, der den Anforderungen bezüglich der Anwender-Interaktionen genügt. Die Klasse ist abgeleitet von `vtkInteractorStyleSwitch`, die bereits zahlreiche Interaktionen anhand von Maus und Tastatur implementiert. Eine Erweiterung fand in sofern statt, als dass das Setzen von Rotationsobjekten als gesondertes Ereignis hinzugefügt wurde. Über diverse Methodenschnittstellen wurde das Festlegen und die Rückgabe von Rotationsobjekten realisiert:

```
void listenForRotPoint()
void listenForRotAxis()
void getRotationPoint(double*)
void getRotationAxis(double*)
int getTypeOfEvent()
bool rotationPointWasSet()
bool rotationAxisWasSet()
```

Zusätzlich wurden Methoden der Vaterklasse überschrieben, um einige Standardfunktionen zu sperren. So ist zum Beispiel zunächst nicht möglich, vom Kamera-Mode, indem nur die Kamera bewegt werden kann, in den Actor-Mode zu wechseln. Als Grund dafür steht die Verarbeitung der Objekttransformationen. Die-

se sollen vorerst nur über die Slider im Transformations-Widget gesetzt werden dürfen. Eine zusätzliche Manipulation der Transformationsdaten durch Mausinteraktionen im 3D-Widget würde zu einer Inkonsistenz der Transformationsdaten führen.

Erweiterung der Szenenverwaltung Durch die neu erstellte Klasse `mihQt-MenuScene` wird die für gewöhnlich verwendete `mihScene` um die Möglichkeit der Interaktion anhand von Qt-Objekten realisiert. Die Szene und damit die enthaltenen Objekte können über ein Menu verwaltet werden, das im Mainframe eingebunden wird. Dadurch wird die Objektauswahl und -Manipulation und die Kamerapositionierung übersichtlicher gestaltet. Bei Veränderungen bezüglich der enthaltenen Objekte, zum Beispiel durch Erstellung einer Gruppe oder Löschen von Objekten, wird ein Submenu aktualisiert, das stets alle Objekte, aufgelistet mit Namen, beinhaltet.

5 Ergebnisse und Diskussion

Im abschließenden Kapitel werden zunächst die Testdatensätze, die für die Überprüfung der Programmfunktionen verwendet wurden, näher beschrieben. Daraufhin werden die Programmlaufzeiten hinsichtlich der genannten Anforderungen in Testläufen untersucht und ausgewertet.
Im Anschluß sind in einer Zusammenfassung die letztlich realisierten Programmfunktionen nochmals aufgelistet. Abschließend folgt ein Ausblick über Verbesserungen und erweiterte Funktionalitäten, die bei einer Fortführung der Entwicklung von *mihMovinBones* in Betracht gezogen werden können.

5.1 Beschreibung der vorhandenen Datensätze

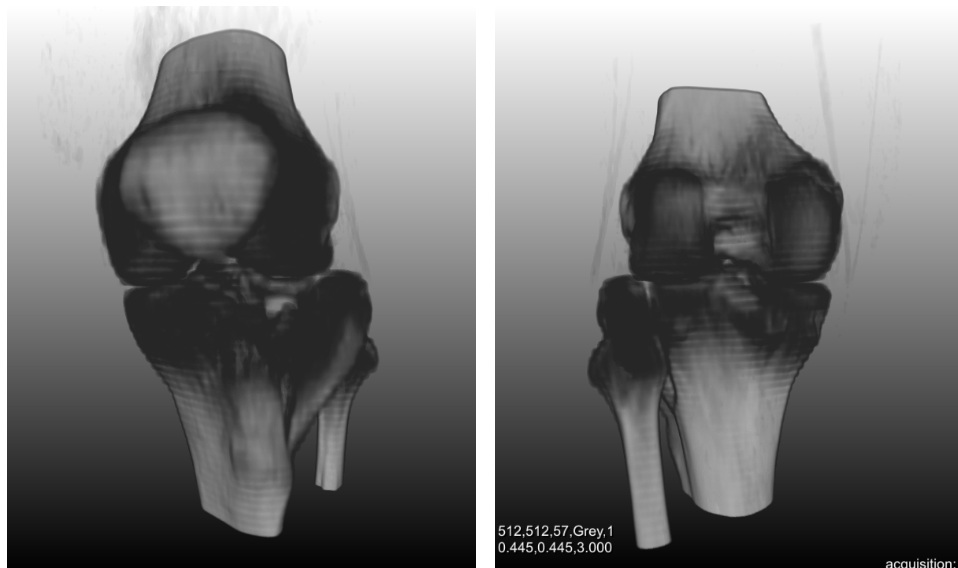


Abbildung 41: Screenshots des ersten Volumendatensatzes in MeVisLab. Zu sehen ist eine komplizierte Kniefraktur aus zwei entgegengesetzen Perspektiven.

Für die exemplarischen Daten standen zwei Datensätze zur Verfügung, die komplizierte Frakturen enthalten. Bei dem ersten Datensatz handelt es sich um eine Torsion im Kniegelenk, die mit einer Schichtdicke von 3 mm per CT gescannt wurde. Das Voxelvolumen beträgt (0.4453mm $\times$ 0.4453mm $\times$ 3mm). Durch die Segmentierung der Knochen wurden fünf Knochenfragmente freigestellt: der vollständige Femur, die unvollständige Tibia, zwei Frakturelemente, die im Bereich des Condylus der Tibia abgebrochen sind (proximale Tibiafraktur) und die vollständig erhaltene Fibula. Die Komplexität dieses Bruches bestand darin, dass das kleinere Bruchstück der Tibia am Kopf abgebrochen und in den keilförmigen Bruchspalt zwischen Tibia und zweites Bruchstück gerutscht war und sich dabei stark gedreht hatte.

Der zweite Datensatz beinhaltet eine komplizierte Hüftfraktur. Die fachlich korrekte genaue Bezeichnung dafür lautet 'Protrusion des Caput femoris in das Os Coxae'. Die aufgenommene Schichtdicke beträgt 1.5mm, das anisotrope Voxelvolumen hat die Ausmaße (0.734mm $\times$ 0.734mm $\times$ 1.5mm). Hier wurden durch die Segmentierung sieben Knochenfragmente ermittelt: der Femur, der ungefähr in der Mitte schräg gebrochen war und das Becken, bestehend aus Sacrum sowie rechtem und linkem Os coxae. Das rechte Os Coxae war im Bereich der Gelenkpfanne in mehrere Teile zerbrochen. Bei dieser Fraktur bestand die Schwierigkeit in dem Ausmaß der Splitterfarktur, was auch die Segmentierung stark erschwerte.

Abbildung 42: Screenshots des zweiten Volumendatensatzes in MeVisLab. Zu sehen ist ein komplexer Beckenbruch aus zwei verschiedenen Perspektiven.

Aus dem Datensatz mit einer Schichtdicke von 3mm lässt sich nur schwerlich qualitativ ausreichend gute 3D-Fragment-Modelle konstruieren, mit denen anschließend zufriedenstellend in der Anwendung gearbeitet werden kann. Durch die grobe Abtastung entstehen starke Stufeneffekte in den Fragmentmodellen, die gerade bei Bruchstellen die Rekonstruktion stark verfälschen. Ebenso würde ein nachträgliches Smoothing der 3D-Objekte den tatsächlichen Sachverhalt verfälschen. Mit dem zweiten Datensatz, der eine Schichtdicke von 1.5 Millimetern aufweist, lassen sich die Knochenfragmente realistischer modellieren.

5.2 Laufzeit der Komponenten

5.2.1 V-Collide

Die ausgewählte Kollisionsbibliothek *V-Collide* wurde anhand von drei Testszenarien auf ihre Performanz hin überprüft. Dabei unterlagen die Testszenarien folgenden Gegebenheiten:

- Die 3 Szenen beinhalteten maximal 10 Objekte. Dabei handelte es sich um die beiden konstruierten Szenen aus den vorliegenden Datensätzen für die ersten beiden Tests. Für den dritten Test wurden besondere Szenen mit je 10 identischen Objekten angefertigt. Die Polygonanzahl wurde in Abhängigkeit der Ergebnisse aus den ersten beiden Tests bestimmt.

- Die für die Tests verwendeten Objekte bestanden aus unterschiedlich vielen Polygonen mit unterschiedlich komplexem Aufbau. In den Tabellen 2 und 3 sind die Objekte und ihre Polygonanzahl der ersten beiden Szenen aufgelistet. Die hervorgehobenen Objekte wurden im Rahmen der Tests transformiert.

- Die Objekte wurden willkürlich durch die Szene bewegt, unabhängig von realistischen Bewegungen. Dabei war eine Kollision der Objekte erlaubt, wobei keine Rücksicht auf den Grad der Objektüberschneidungen genommen wurde.

- Gemessen wurde die Dauer, die *mihMovinBones* für die Anpassung der Transformationsdaten für das Übergabeformat von *V-Collide* benötigte, sowie die Zeit, die *V-Collide* für die einfache Kollisionserkennung braucht. Die Vorverarbeitung der Übergabeparameter konnte dabei als nicht relevant betrachtet werden. *V-Collide* befand sich bei den Tests im `VC_FIRST_CONTACT`-Mode, in dem nur die IDs der kollidierenden Objekte zurückgegeben werden, nicht jedoch die IDs der kollidierenden Polygone pro Objekt. Die Zeiten, die *V-Collide* in diesem Mode (`VC_ALL_CONTACT`) erreicht, sind aufgrund ihrer inakzeptabel hohen Laufzeiten von bis zu 0.2 Sekunden pro Transformation (gemessen in der ersten Testszene) irrelevant.

- Die Tests wurden auf einem 64bit-Prozessor-Rechner (AMD Opteron(tm) Processor 848 / 2.2GHZ / 16GB MM) durchgeführt. Bei der Grafikkarte handelte es sich um eine NVIDIA Quadro FX 3400.

Die Testergebnisse wurden in der Form ausgegeben, wie sie in Abbildung 43 bis 45 zu sehen sind. Auf der Abszisse wurde die Anzahl der kollidierenden Objekte mit dem aktuellen Objekt aufgetragen, auf der Ordinate die Zeit in Mikrosekunden, die *V-Collide* für die Rückgabe der Kollisionsereignisse benötigte. Es wurde stets ein und dasselbe Objekt durch die Szene transformiert. Die Polygonanzahl des Objekts ist jeweils über dem Graphen abzulesen. Bei jeder Übergabe der Transformationsdaten an *V-Collide* wurde die Anzahl derjenigen Objekte, die mit dem aktuell transformierenden Objekt kollidieren, in Beziehung zu der Zeit gesetzt, die *V-Collide* zur Kollisionserkennung benötigt. Dadurch wurde pro Anzahl der kollidierenden Objekte das Zeitfenster visualisiert, in dem *V-Collide* bei unterschiedlichen Transformationsereignissen die Kollisionsergebnisse zur Weiterverarbeitung zurückliefert. Der kritische obere Frequenzwert, den es einzuhalten galt, wurde grün markiert. Er befindet bei 1000 Hz. Es folgen Beschreibungen der verschiedenen grafischen Ergebnisse.

Tabelle 2: Objekte und ihre Polygonanzahl aus der ersten Szene

Name	Anzahl der △
fragment01.vtk	**30882**
fragment02.vtk	**91404**
fragment03.vtk	77354
fragment04.vtk	18716
fragment06.vtk	**104034**
fragment07.vtk	6384
fragment08.vtk	21394
fragment09.vtk	13382
Summe der △	363550

Tabelle 3: Objekte und ihre Polygonanzahl aus der zweiten Szene

Name	Anzahl der △
bruch1.vtk	19104
bruch2.vtk	50644
schienbein.vtk	**179964**
wadenbein.vtk	**33782**
oberschenkel	**225516**
Summe der △	509010

In Abbildung 43 ist das Testergebnis für Transformationen innerhalb der ersten Szene mit maximal acht Objekten zu sehen. Es fällt auf, dass der Zeitbedarf von *V-Collide* bei steigender Kollisionsobjektmenge zunimmt, was nachvollziehbar ist. Dieses Verhalten ist bei den anderen Tests ebenfalls vorhanden, jedoch in unterschiedlicher Ausprägung. Beim ersten Test überschreiten die Zeiten für die Kollisionsbehandlung bei einer Anzahl von sechs und mehr kollidierenden Objekten den Grenzwert von 1000 Hz recht häufig. Als oberer Grenzwert aller Einträge kann 666 Hz genannt werden. Vereinzelt gibt es Einträge für Zeitwerte, die sich ungewöhnlich weit von den anderen Einträgen befinden.
Beim zweiten Testergebnis in Abbildung 44 fällt auf, dass die Einträge sich bis auf vereinzelte Ausreisser alle unter dem 1000 Hz Grenzwert befinden. Da jedoch die Szene lediglich aus fünf Objekten besteht, sind entsprechend auch nur maximal fünf Objekte an der Kollision beteiligt.

Bei dem dritten Test handelte es sich um drei besondere Einzeltests. Die zwei bereits besprochenen Test ließen offen, inwieweit die Performanz wirklich in Abhängigkeit der Objektanzahl und der Objektkomplexität steht. Es sollte zunächst überprüft werden, wie sich die Kollisionsbibliothek verhält, wenn die Anzahl der Objekte steigt. Zudem wurde überprüft, welche Auswirkungen die Wohlgeformtheit der *polygon soups* auf die Kollisionserkennung hat. Der erste und zweite Graph in Abbildung 45 zeigen Ergebnisse für Szenen mit jeweils zehn identischen Objekten, wobei es sich um Kugelmodelle handelte. Die Modelle bestanden aus 100000 bzw. 243000 Polygonen. Vergleicht man die Polygonanzahl mit denen aus Test 1 und Test 2, dann handelte es sich um Modelle, die im oberen Größenbereich eingeordnet werden können. Beide Ausgaben zeigen, dass trotz der außerordentlich großen Objektmodelle und der hohen Objektanzahl sich die Werte für die Kollisionserkennung bis auf wenige Ausnahmhen unterhalb der 1000 Hz befinden.

Im Test, dem der untere Graph als Resultat zugeordnet werden kann, wurden zehn Oberschenkelmodelle aus dem zweiten Test zu einer neuen Szene hinzugefügt. Jedes Objekt bestand nach Tabelle 3 aus 225516 einzelnen Polygonen. Der Unterschied zum Test mit den zehn Kugeln, bestehend aus 243000 Polygonen, bestand somit in der Komplexität der Modelle. Wie dem in Abbildung 45 enthaltenen rechten Graphen zu entnehmen ist, kann *V-Collide* bei dieser Anforderung die Zeit für Kollisionserkennung nicht mehr unter $1000 \mu s$ halten.

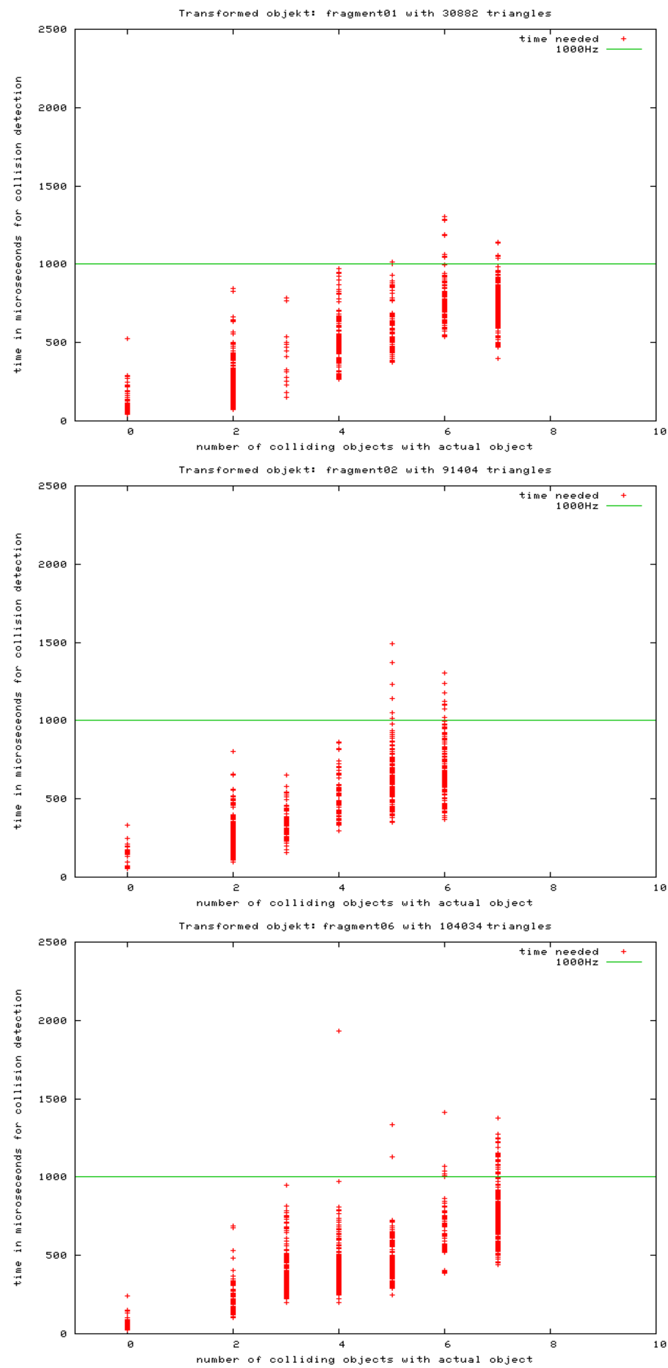


Abbildung 43: Testwerte für die Szene mit der Hüftfraktur. Je mehr Objekte mit dem aktuellen Objekt kollidieren, desto größer die Zeitdauer, die *V-Collide* für die Kollisionserkennung benötigt. Die Werte liegen im akzeptablen Bereich. Lediglich bei Kollisionen, in die sechs und mehr Objekten involviert sind, kann eine Frequenz von 1000 Hz nicht dauerhaft erreicht werden.

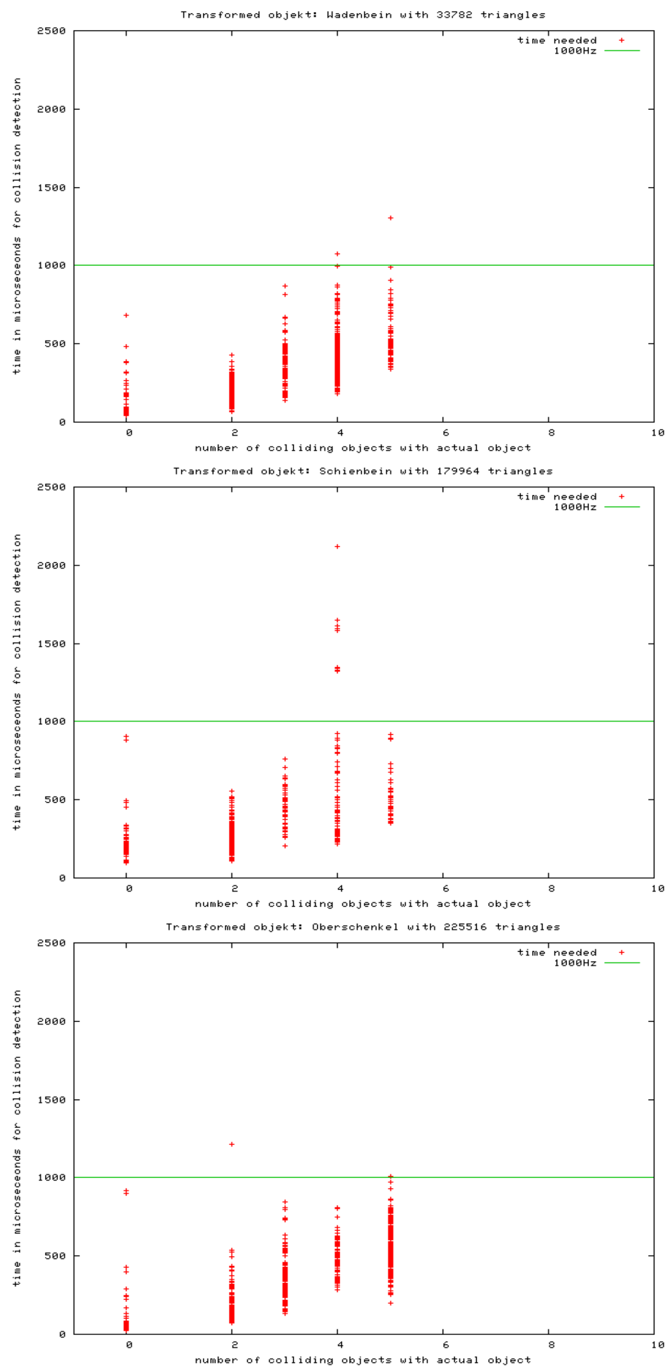


Abbildung 44: Testwerte für die Szene mit der Kniefraktur. Je mehr Objekte mit dem aktuellen Objekt kollidieren, desto größer die Zeitdauer, die *V-Collide* für die Kollisionserkennung benötigt. Die Werte liegen allesamt im akzeptablen Bereich von unter 1000 Hz.

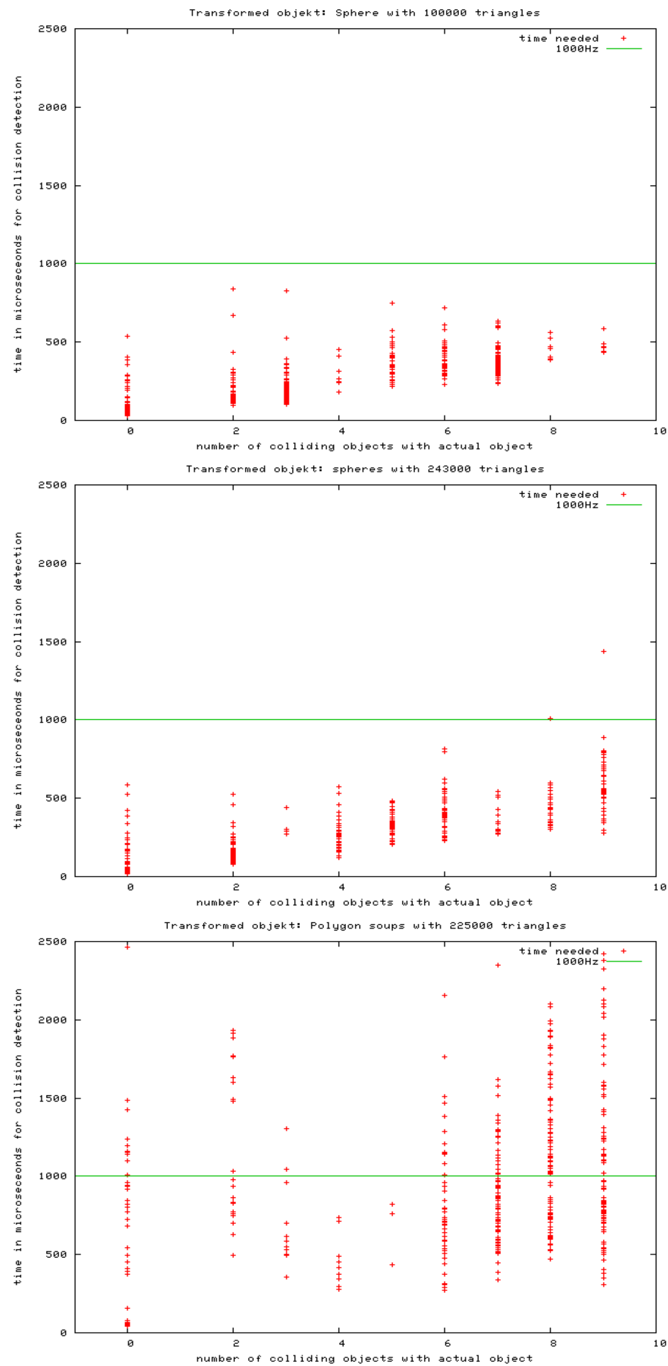


Abbildung 45: Testwerte aus abstrakten Szenen mit wohlgeformten Kugelmodellen (Graph 1 und 2) und einer komplexen *polygon soup* (Graph 3). Die ersten beiden Graphen zeigen, dass trotz der hohen Anzahl von Objektpolygonen und Objekten eine Frequenz von 1000 Hz nahezu kontinuierlich erreicht wird. Wird dagegen ein komplexes Objekt mit ähnlich vielen Polygonen verwendet, kann die Frequenz nicht mehr gehalten werden.

Abschließend kann die Aussage getroffen werden, dass die Kollisionserkennung mit V-Collide für die Anforderungen, die *mihMovinBones* stellt, ausreichend sind. Allerdings ist wenig Spielraum nach oben gegegeben. Bei einer hohen Anzahl von komplexen Objekten kann V-Collide die Anforderungen nicht mehr erfüllen.
Eine Optimierung könnte erreicht werden, indem die 3D-Polygonmodelle einer Vorverarbeitung unterzogen werden. In dem Schritt wäre es sinnvoll und möglich, innere Polygonstrukturen der Knochen zu eliminieren und so die Polygonanzahl erheblich zu reduzieren.

5.2.2 Konturbestimmung

Die Berechnung der Objektkonturen erfolgt wie in 4.4.7 beschrieben. Der Prozess ist sehr zeitintensiv. Tabelle 4 zeigt pro Objekt einer Szene bei 100 willkürlichen Transformationen den mittleren Zeitwert, den es bedarf, um die Kontur dieses Objekts für die axiale Ansicht zu ermitteln. Die Resultate der Messungen stellen klar, dass die Anwendung bei eingeschalteter Konturanzeige eine Aktualisierungsfrequenz von 1000 Hz nicht mehr realisieren kann. Da die Anwendung zudem drei der Widgets besitzt, in denen Konturen angezeigt werden können, kann sich die Dauer der Konturbestimmung verdreifachen. Dadurch steigt der Zeitbedarf Aktualisierung pro Transformation so stark, dass ein Arbeiten in Echtzeit ohnehin nicht mehr möglich ist.

Fragment	Transf.	$\triangle$	t in Sekunden
Oberschenkel	100	225516	0.488
Schienbein	100	179644	0.451
Bruch02	100	50644	0.176
Wadenbein	100	33782	0.163
Bruch01	100	19104	0.139

Tabelle 4: Die mittlere Zeit t einer Konturberechnung pro Objekt

5.3 Zusammenfassung und Ausblick

5.3.1 Zusammenfassung

Mit der Umsetzung der in den Anforderungen genannten Inhalte stellt *mihMovinBones* eine solide Grundlage einer Anwendung für die präoperative Repositionierung von Knochenfragmenten dar.
Dem Anwender wird die Möglichkeit geboten, 3D-Modelle von Knochenfragmenten in einer Simulation in Echtzeit zu repositionieren und damit deren Position und Orientierung zu korrigieren. Die dabei integrierte Kollisionserkennungsbibliothek, die sich anhand theoretischer Vergleiche gegenüber anderen vorhandenen und frei verfügbaren Bibliotheken als die beste Wahl profilieren konnte, erhöht dabei den Grad der simulierten Realität.

Durch Kombinieren von neuen und dem Anwender bekannten Darstellungsformen einer Knochenfraktur und dem Illustrieren von Objekten der einen Darstel-

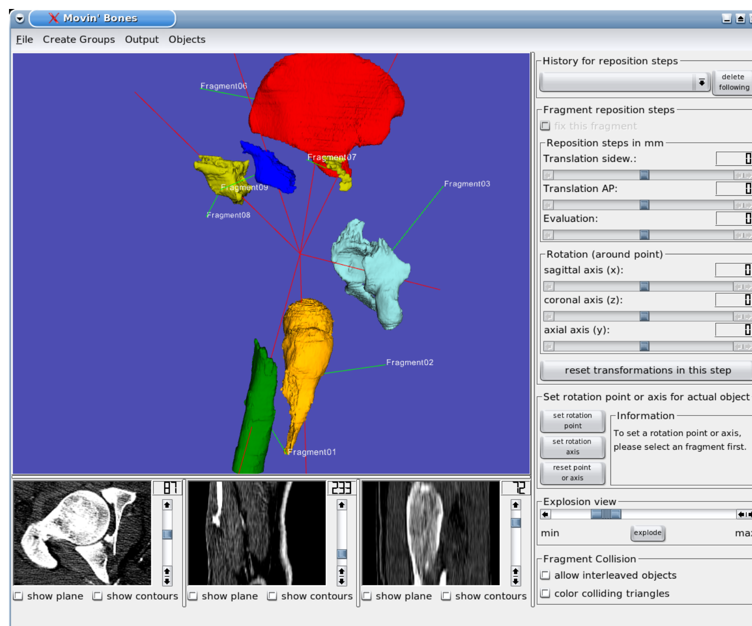


Abbildung 46: Screenshot der gesamten GUI von mihMovinBones. Das 3D-Widget stellt die Fraktur in der Explosionsdarstellung dar. Bei der Szene handelt es sich um die Hüftfraktur.

lungsform in der anderen, werden dem Anwender neue Möglichkeiten der Betrachtungs- und Bearbeitungsweise geboten, ohne dabei die alten und gewohnten zu ignorieren. Zugleich besteht eine nachvollziehbare Verbindung beider Visualisierungsarten.

Intuitive Interaktionsmöglichkeiten über eine klar strukturierte GUI und durch Maus- und Tastaturereignisse helfen dem Anwender sich in kurzer Zeit in die Bedienung von *mihMovinBones* einzuarbeiten. Auch das Repositionieren der Knochenfragmente anhand der sechs Slider macht in dieser Anwendung aufgrund der gegebenen Präzision Sinn.

Die Korrekturmaßnahmen, die an einem komplizierten Knochenbruch anhand von mehreren Repositionierungsschritten durchgeführt werden, lassen sich komfortabel laden und speichern. Zudem ermöglicht eine Ausgabe des Korrekturprozesses im HTML-Format mit beliebig vielen Screenshots und ein anschließendes Ausdrucken eine Mitnahme zu Studienzwecken oder direkt in eine Operation.
Die Art der Ausgabe der Transformationsinformationen anhand verschiedener Rotationsobjekte bedarf einer Überarbeitung. Hier muss zunächst ein intensiver Austausch mit der Anwendergruppe, also praktizierenden Chirurgen aus dem Bereich der Unfallchirurgie, stattfinden. Im Ausblick (5.3.2) ist kurz eine erste Idee dazu aufgeführt.

Abschließend ist zu erwähnen, dass solange die Vorverarbeitung der Daten, also das Segmentieren der einzelnen Knochenfragmente, das Erstellen der 3D-Knochen-

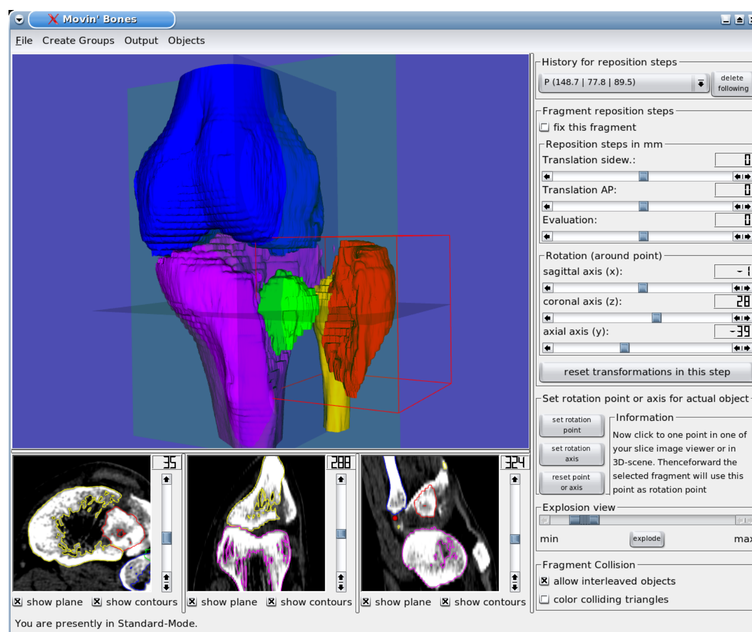


Abbildung 47: Screenshot der gesamten GUI von mihMovinBones. Im 3D-Widget sind die Schichtpositionen eingeblendet. Bei der Szene handelt es sich um die Kniefraktur.

modelle und das Erzeugen einer für *mihMovinBones* lesbaren Datei inklusive der Dateiverwaltung etc. jedoch noch die Zeit beansprucht, die derzeit benötigt wird, kann die mit dieser Arbeit vorgestellte Art der Anwendung aufgrund des kleinen Zeitfensters keinen Einsatz im Alltag der Unfallchirurgie finden. Zudem lässt die Qualität der segmentierten Fragmentmodelle zu wünschen übrig. Auch hier muss ein Trade-Off zwischen Erhalt der Knochenstrukturinformationen und der Darstellungsqualität festgelegt werden.

5.3.2 Ausblick

In diesem Kapitel werden einige Möglichkeiten der Erweiterung beschrieben, die noch wünschswert wären. Ein Bereich, den es vorerst weiter zu entwickeln gilt, nicht aber in den Bereich einer Erweiterung von *mihMovinBones* fällt, umfasst die Vorverarbeitung der Daten. Wie bereits in der Zusammenfassung beschrieben, wäre es für viele Anwendungen der hier vorgestellten Art sicherlich sinnvoll, den Bildgebungsprozess für die notwendigen Daten zeitlich zu optimieren. Die Entwicklung von Segmentierungsverfahren und Verfahren zur Konstruktion authentischer 3D-Modelle auf Basis von Bilddaten wird seit Jahren stark voran getrieben, so dass die durch die Medizin gegebenen unterschiedlichen Zeitvorgaben in naher Zukunft sicherlich erfüllt werden können. Mit dem Erreichen dieser Vorgaben wird der praxisbezogene Einsatz von Anwendungen der hier vorgestellten Art real werden. Dies ist eine Motivation, sich Gedanken über mögliche Programmerweiterungen zu machen.

- Der erste Schritt der Erweiterung wird sicherlich die Möglichkeit der Ob-

jekttransformation innerhalb der dargestellten Szene per Mausinteraktion sein. Die Bedienung würde dadurch anwenderfreundlicher.

- Eine weitere Möglichkeit der Erweiterung wäre das Einbinden von haptischen Ein- und Ausgabegeräten, mit Hilfe derer die Interaktion der Szene nochmals intuitiver gestaltet werden könnte. Bei der Auswahl der Kollisionserkennung wurde dieser Hintergedanke bereits berücksichtigt. V-Collide kann bei den getesteten Szenen eine Frequenz von über 1000 Hz aufrecht erhalten (siehe 5.2.1), wodurch eine Bedingung für die Integration von haptischen Geräten bereits erfüllt ist.
- In Zusammenarbeit mit praktizierenden Chirurgen muss eine einheitliche Basis für das Ausgabeformat der Repositionierungsschritte für die einzelnen Knochenfragmente gefunden werden. Die hier vorgestellte Lösung kann nur ein erster Ansatz sein, da sie zwar mathematisch korrekt, aber für die praktische Anwendung kaum nutzbar ist. Der Chirurg, mit dem zusammen gearbeitet wurde, schlägt hierfür eine Reduzierung bei der Auswahl der Rotationsobjekte vor. Sinnvoll ist seiner Meinung nach sicherlich der Objektmittelpunkt als Rotationsobjekt. Als weitere Rotationsobjekte macht eine begrenzte Auswahl an Punkten auf der Objektoberfläche Sinn, die an markante Positionen gesetzt werden. Welche Positionen in Abhängigkeit der Knochenfragmentform das sind und ob diese automatisch oder manuell gesetzt werden sollen, bedarf der Diskussion.
- Eine Optimierung bei der Erzeugung der Daten wurde bereits angesprochen. Eine Art der Objektmanipulation, die innerhalb von *mihMovinBones* sinnvoll sein kann, ist die Reduzierung der Polygonanzahl der einzelnen Fragmente. Die Modelle umfassen innerhalb ihrer äußeren Hülle sehr viele Polygonstrukturen. Dabei handelt es sich um innere, spongiöse Knochenstrukturen, die bei der Segmentierung dem Knochen zugeordnet werden. Die Tests in 5.2.1 haben gezeigt, dass Polygonmodelle, die keine inneren Strukturen haben, besser handbar für die Kollisionserkennung und die Konturerstellung sind. Diese Art der Optimierung würde der Echtzeitfähigkeit zu Gute kommen. Ein Problem dabei ist das Bestimmen der Bruchflächen von Knochen, da an diesen Stellen häufig komprimiertes oder anderweitig verändertes spongiöses Knochengewebe die Bruchfläche darstellt.
- Bei der Transformation von Knochenfragmenten werden innerhalb der Ausgabefenster für die Schichtdaten lediglich die überlagerten Konturen mittransformiert, die Volumendaten selbst sind statisch. Es wäre wünschenswert, dass bei einer Transformation das zugehörige Subvolumen innerhalb der Volumendaten ebenfalls gemäß der Objektrepositionierung transformiert würde. Der Anwender könnte dadurch in der ihm gewohnten Visualisierungsart das Resultat betrachten und beurteilen. Dies setzte eine exakte Segmentierung voraus.
- Um eine abschließende der zahlreichen Erweiterungsmöglichweiten zu nennen, sei das Setzen von Fremdkörpern zur Fixation von Knochenfragmenten genannt. Häufig werden Frakturen durch Schrauben, Platten oder andere Objekte stabilisiert, um ein gegenseitiges Verschieben der Frakturelemente zu verhindern. Mit dieser Erweiterung würde sich die Anwendung weiter in Richtung unfallchirurgischen Praxisalltags orientieren.

6 Danksagung

Ich möchte mich zuletzt noch bei einigen Menschen bedanken: Danke an Angela für alles. Danke an meine Eltern, die mir das Studieren ermöglichten. Danke an das gesamte IMI-Team für die angenehme Zeit.

Literatur

[AG06] Siemens AG. *Siemens Medical Solutions*, 2006. Webpage: http://www.medical.siemens.com.

[BDH98] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 1998.

[Bra02] Gareth Bradshaw. *Bounding volume hierarchies for level-of-detail collision handling*. PhD thesis, Trinity College Dublin, 2002.

[Bra03] Gareth Bradshaw. Adaptive medial-axis approximation for sphere-tree construction. 2003.

[Cam97] Stephen A. Cameron. Enhancing gjk: Computing minimum and penetration distances between convex polyhedra. *IEEE Journal of Robotics and Automation*, pages 3112–3117, April 1997.

[Dev06] VTK User & Developer. *VTK User & Developer Forum*, 09 2006. Forum: http://www.vtk.org/cgi-bin/mailman/listinfo.

[EGG88] S. S. Keerthi E. G. Gilbert, D. W. Johnson. A fast procedure for computation the distance between complex objects in three dimensional space. *IEEE Journal of Robotics and Automation*, page 193, 1988.

[Gam01] Erich Gamma. *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 1996.

[Got96] S. Gottschalk. Separating axis theorem. *Technical Report TR96-024*, 1996.

[GWK06] Rolf Sauer Guenter W. Kauffmann, Ernst Moser. *Radiologie*, volume 2. Urban & Fischer Verlag, 2006.

[Ham05] C. Fares Y. Hama. Collision detection for rigid bodies: A state of art review. *Graphicon*, 2005.

[Han00] Heinz Handels. *Medizinische Bildverarbeitung*. B.G.Teubner Stuttgart - Leipzig, 1 edition, 2000.

[Hub96] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, pages 179–210, 1996.

[JC95] D. Manochaand M.Poamgi J. Cohen, M.C. Lin. I-collide: An interactiveand exact collisions detection system for large scale environment. *Proceedings Symposium on Interactive3D Graphics*, pages 189–196, 1995.

[LC87] W. E. Lorensen and H. Cline. Marching cubes: a high resolution 3D surface construction algorithm. In *Proc. SIGGRAPH 1987*, volume 21, pages 303–312, 1987.

[LC91] M. Lin and J. Canny. A fast algorithm for incremental distance calculation. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[Len00] Christian Lennerz. *Impulse-based dynamics simulation of rigid bodies using bounding-volume-accelerated distance computation*. PhD thesis, University of Saarland, Saarbruecken, Februar 2000.

[LGLM99] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast proximity queries with swept sphere volumes. 1999.

[Lin00] S. A. Ehmann M. C. Lin. Swift: Accelerated proximity queries between convex polyhedra by multi-level voronoi marching. *Technical Report*, 2000.

[Lin01] S. A. Ehmann M. C. Lin. Accurate and fast proximity queries between polyhedra using surface decomposition. *Computer Graphics Forum (Proceedings of Eurographics)*, 2001.

[Man97] T. Hudson M. Lin J. Cohen S. Gottschalk D. Manocha. V-collide: Accelerated collision detection for vrml. *Proceedings of VRML*, 1997.

[Maz02] S. Caselli M. Reggiani M. Mazzoli. Exploiting advanced collision detection libraries in a probabilistic motion planner. *WSCG, Proceedings EVL-2002-21*, 2002.

[MeV06] MeVisLab. *MeVisLab - Medical Image Processing and Visualization*, 2006. MeVisLab: http://www.mevislab.de.

[Mir98] Brian Mirtich. Vclip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, pages 177–208, July 1998.

[Mit96] J.T. Klosowski M. Held J.S.B. Mitchell. Real-time collision detection for motion simulation within complex environments. *SIGGRAPH '96 Visual Proceedings*, 1996.

[ML98] S. Gottschalk M. Lin. Collision detection between geometric models: a survey. *Proceedings of the IMA Conference on Mathematics of Surfaces*, pages 61–63, 1998.

[Mor95] H. Morneburg. *Bildgebende Systeme fuer die medizinische Diagnostik. Roentgendiagnostik und Angiographie, Computertomographie, Nuklearmedizin, Magnetresonanztomographie, Sonographie, Integrierte Informationssysteme*. Publicis MCD Verlag, Erlangen, 1995.

[NEM] NEMA. *DICOM - Digital Imaging and Communications in Medicine*. NEMA. MeVisLab: http://medical.nema.org/.

[oUCD06] Bioengeneering of University College Dublin. *vtkCollisionDetectionFilter*, 2006. vtkCollisionDetectionFilter: http://www.bioengineering-research.com/.

[Rue03] Axel Rueter. *Unfallchirurgie*. Urban & Fischer, 2 edition, 2003.

[Sed92] Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, neunte edition, Januar 1992.

[Tro05] Trolltech. *Signals and Slots*, 2005. Signals and Slots: http://doc.trolltech.com/3.3/signalsandslots.html.

[VB99] Gino Van and Den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *Journal of Graphics Tools*, 1999.

[VB01] Gino Van and Den Bergen. Proximity queries and penetration depth computation on 3d game objects. *Game Developers Conference*, 2001.

[Zac98] Gabriel Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 90–97, Atlanta, Georgia, March 1998.

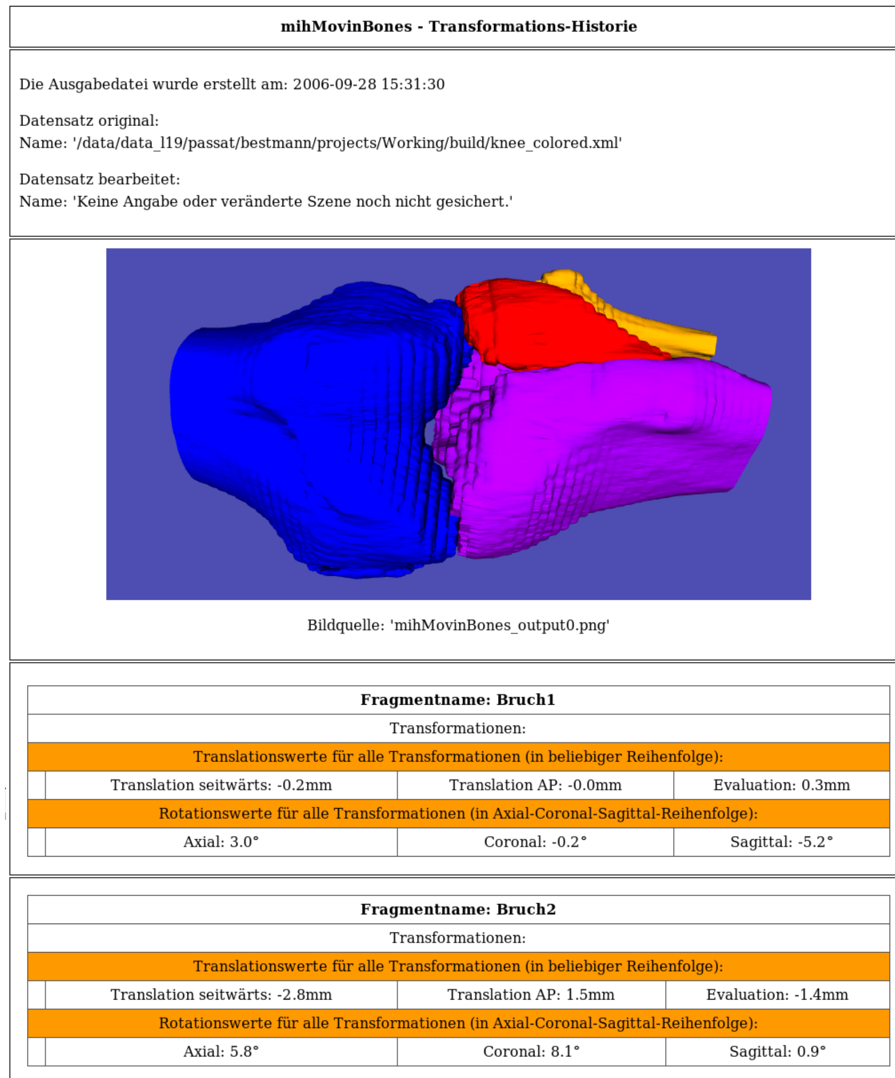
[Zac06] Gabriel Zachmann. *Benchmarking Collision Detection Algorithms*, 08 2006. Benchmarks: http://zach.in.tu-clausthal.de/˜zach/coldet/index.html.

A Ausschnitt einer XML-Datei

Listing 5: Beispielcode einer XML-Datei für eine Szene mit nur einem 3D-Modell

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<mihXMLImageFile xmlns="http://mih.org/images">
  <scene>
    <sequence>
      <seqlist>
        <dataname>CT_Data</dataname>
        <src>/passat/Bilddaten/KnochenFraktur/ott001/data/ct3.list</src>
      </seqlist>
    </sequence>
    <mih3DModels>
      <model>
        <modelname>Bruch1</modelname>
        <src>/passat/ehrhardt/Projects/KnochenFraktur/Data/ott001/model/ott001_segm_ct_003_bruch1.
            vtk</src>
        <surfaceColor>
          <r>0</r>
          <g>1</g>
          <b>0</b>
        </surfaceColor>
        <ambientColor>
          <r>0</r>
          <g>1</g>
          <b>0</b>
          <value>0</value>
        </ambientColor>
        <diffuseColor>
          <r>0</r>
          <g>1</g>
          <b>0</b>
          <value>0.7</value>
        </diffuseColor>
        <specularColor>
          <r>0</r>
          <g>1</g>
          <b>0</b>
          <value>0.5</value>
        </specularColor>
        <pickable>1</pickable>
        <dragable>1</dragable>
        <opacity>1</opacity>
        <transHistory>
          <mih3DTrans>
            <typeOfTrans>point</typeOfTrans>
            <rotPoint>
              <sagittal>140.046</sagittal>
              <coronal>110.879</coronal>
              <axial>106.615</axial>
            </rotPoint>
            <rotation>
              <sagittal>-0</sagittal>
              <coronal>0</coronal>
              <axial>-0</axial>
            </rotation>
            <translation>
              <sagittal>0</sagittal>
              <coronal>0</coronal>
              <axial>0</axial>
            </translation>
          </mih3DTrans>
        </transHistory>
      </model>
    </mih3DModels>
    <mihPatientData/>
  </scene>
</mihXMLImageFile>
```

B Ausgabe im HTML-Format



mihMovinBones - Transformations-Historie

Die Ausgabedatei wurde erstellt am: 2006-09-28 15:31:30

Datensatz original:
Name: '/data/data_l19/passat/bestmann/projects/Working/build/knee_colored.xml'

Datensatz bearbeitet:
Name: 'Keine Angabe oder veränderte Szene noch nicht gesichert.'

Bildquelle: 'mihMovinBones_output0.png'

Fragmentname: Bruch1			
Transformationen:			
Translationswerte für alle Transformationen (in beliebiger Reihenfolge):			
	Translation seitwärts: -0.2mm	Translation AP: -0.0mm	Evaluation: 0.3mm
Rotationswerte für alle Transformationen (in Axial-Coronal-Sagittal-Reihenfolge):			
	Axial: 3.0°	Coronal: -0.2°	Sagittal: -5.2°

Fragmentname: Bruch2			
Transformationen:			
Translationswerte für alle Transformationen (in beliebiger Reihenfolge):			
	Translation seitwärts: -2.8mm	Translation AP: 1.5mm	Evaluation: -1.4mm
Rotationswerte für alle Transformationen (in Axial-Coronal-Sagittal-Reihenfolge):			
	Axial: 5.8°	Coronal: 8.1°	Sagittal: 0.9°

Abbildung 48: Ausgabe im HTML-Format. In diesem Beispiel wurde nur ein Screenshot der Ausgabe hinzugefügt. Zwei Fragmente wurde transformiert. Die Gesamttransformationen sind aufgeführt