Technische Universität Darmstadt
Fachbereich Informatik

`roboViewLib`: Eine echtzeitfähige Bibliothek zur Visualisierung, Animation und Kollisionserkennung für laufende Roboter und andere Mehrkörpersystemen

Diplomarbeit von Martin Friedmann

Aufgabensteller:	Prof. Dr. O. v. Stryk
Betreuerin:	Dipl.-Math. J. Kiener
Abgabetermin:	01.09.2003

Vorwort

In der vorliegenden Arbeit wird die Bibliothek `roboViewLib` beschrieben. Diese Bibliothek stell ein objektorientierten Rahmen bereit, der die flexible Entwicklung von Programmen zur Visualisierung, Animation und Kollisionserkennung für laufende Roboter und anderer Mehrkörpersysteme gestattet.
Die Architektur ist dabei sehr offen gehalten. Durch ein modulares Konzept ist es möglich, die Komponenten zur Steuerung der Animation, zur Kollisionserkennung und zur Visualisierung der Systeme frei auszutauschen.
Im Rahmen der Arbeit wurden mit Hilfe der Bibliothek die Anwendungen `roboMovieMaker` und `roboRTView` entwickelt:
`roboMovieMaker` erlaubt es Mehrkörpersysteme mit Hilfe vorgegebener Trajektorien zu animieren und als Filmdateien abzuspeichern.
`roboRTView`, dient der Echtzeitvisualisierung von Steuerdaten, wie sie z. B. bei Simulationen entstehen. Durch den Einsatz entsprechender Verfahren ist es möglich, auftretende Kollisionen in Echtzeit zu erkennen und anzuzeigen.
Beide Anwendungen greifen auf verschiedene Module zur Visualisierung und Kollisionserkennung zurück, die im Rahmen der Arbeit entwickelt wurden. Dabei wurde ein Verfahren zur Kollisionserkennung hergeleitet, das eine einheitliche Betrachtung von durch Superquadriken angenäherten quader- und zylinderförmigen Körpern gestattet.

Inhaltsverzeichnis

Vorwort **i**

1 Einleitung **1**

2 Anforderungen **3**

3 Grundlagen **7**
3.1 Mathematische Beschreibung von Mehrkörpersystemen 7
3.2 Prozesse, Threads und Synchronisation 12
3.3 Echtzeit 13
3.4 Implizite Beschreibung geometrischer Objekte 14
3.5 Struktur von XML-Dateien 16
3.6 Templates 17
3.7 3D-Graphik 17
3.7.1 Verfahren 18
3.7.2 Programmierschnittstellen zur Abstraktion von der Graphikhardware 22
3.7.3 Bibliotheken zur Szenenverwaltung 24
3.8 Erzeugen von Filmdateien mit `transcode` 25

4 Architektur der Software **27**
4.1 Vorüberlegung 27
4.2 Model-View-Controller-Konzept 28
4.3 MVC und Multithreading 29
4.4 Konkrete Implementierung 31
4.4.1 Auswahl der Programmiersprache 31
4.4.2 Verkapselung des Multithreading 31
4.4.3 Aspekte der Speicherverwaltung 33
4.4.4 Implementierung des MVC-Frameworks 34
4.4.5 Abbildung der Anwendung auf das Framework 35
4.4.6 Mathematische Datentypen 36

5 Repräsentation der Szene **37**
5.1 Interne Repräsentation der Szene 37
5.1.1 Mehrkörpersysteme 37
5.1.2 Zusatzdaten 42
5.1.3 Gesamtszene 43
5.2 XML-Beschreibung der Szene 44
5.2.1 Mehrkörpersysteme und Umgebung 44
5.2.2 Zusatzdaten 47
5.3 Repräsentation der Steuerdaten 48

6 Kollisionserkennung **51**
6.1 Schnittstelle zur Kollisionserkennung 51
6.2 Kollisionserkennung für Kugeln 53
6.3 Kollisionserkennung für Quader 54
6.3.1 Grundgedanke für das Verfahren 54
6.3.2 Schnitt zweier konvexer Polyeder 54
6.3.3 Teilen eines konvexen Polyeders an einer Ebene 55
6.3.4 Implentierung für Quader 56
6.4 Verwendung von Superquadriken als Hüllkörper 57
6.4.1 Annähern von Quadern und Zylindern durch Superquadriken 57
6.4.2 Strategie zur Kollisionserkennung für implizit definierte Körper 58
6.4.3 Kollisionserkennung für angenäherte Quader 60
6.4.4 Übertragung auf angenäherte Zylinder 63
6.4.5 Übertragung auf angenäherte Zylinder und Quader 64
6.4.6 Zusammemfassung 64

7 Graphische Darstellung der Animation **67**
7.1 Auswahl einer 3D-Graphik-Programmierumgebung 67
7.1.1 Maverik 68
7.1.2 OpenSG 69
7.1.3 Sigma 71
7.1.4 Vergleich und Auswahl 71
7.2 Implementierung der Darstellung 73
7.2.1 Die Basisklasse `CRoboView` 73
7.2.2 Integration der OpenGL-Ausgabe 74
7.2.3 Repräsentation der Szene in OpenSG 75
7.2.4 Erzeugen der graphischen Ausgabe 80
7.3 POV-Ray-Export 82

7.3.1 Zusammenwirken von Szenenbeschreibung und Bildparameterdateien 82
7.3.2 Aufbau der POV-Ray-Szenenbeschreibung 83
7.3.3 Shell-Script zur Filmerzeugung 84

8 Erzeugen von Filmen **87**
8.1 Steuerung der Animation 87
8.1.1 Verwendetes Dateiformat 87
8.1.2 Die Klasse `CMovie` 88
8.2 Kern der Applikation 89
8.2.1 Initialisierung 90
8.2.2 Einbindung in eine Anwendung 90
8.2.3 Ablaufsteuerung des Films 91
8.3 Graphische Benutzungsoberfläche 91
8.3.1 Aufbau der Benutzungsoberfläche 92
8.3.2 Implementierung der Benutzungsoberfläche 94

9 Echtzeitvisualisierung von Simulationsdaten **97**
9.1 Implementierumg von `roboRTView` 97
9.2 Protokoll zum Steuern von `roboRTView` 98
9.3 Betrieb von `roboRTView` 99

10 Zusammenfassung **101**
10.1 Ergebnisse 101
10.2 Erzeugte Bilder 102
10.3 Ausblick 105

Literatur **107**

Tabellenverzeichnis

5.1 Baumrepräsentation in `CSceneElement` 41
5.2 Repräsentation der räumlichen Lage in `CSceneElement` 42
5.3 Repräsentation der Form in `CSceneElement` 42
5.4 Parameter der Zusatzdaten 43
5.5 Datenelemente der Szenebeschreibung `CSceneDescription` . . . 44
5.6 Datenelemente des Steuerdatenobjekts `CFrameData` 48

7.1 Vergleich zwischen Maverik, OpenSG und Sigma 72
7.2 Darstellungsoptionen in `CRoboViewOptions` 74
7.3 Datenobjekte zum Manipulieren bestehender Szenen 77
7.4 Repräsentation der Zusatzdaten 80

8.1 Bedeutung der Bits in den Darstellungsoptionen 90
8.2 Methoden von `CMoviePlayer` zur Kommunikation mit der Anwendung 91

Abbildungsverzeichnis

3.1 Superquadriken 15

4.1 Grundkomponenten der Software 28
4.2 Interaktion im MVC-Entwurfsmuster 29
4.3 Erweiterte MVC-Interaktion 30

5.1 Struktur des Baumes in `CSceneElement` 41

6.1 Durch Superquadriken angenäherte Quader und Zylinder 58

7.1 Teilgraph für ein Element eines Mehrkörpersystemes 76
7.2 Texturen auf den Standardkörpern von `COSGLView` 79

8.1 Aufbau und Kommunikation von `CMoviePlayer` 89
8.2 Benutzungsoberfläche 93
8.3 Interaktion zwischen Benutzungsoberfläche und `CMoviePlayer` . 95

10.1 Humanoider Roboter in Seitenansicht 102
10.2 Humanoider Roboter in Vorderansicht 103
10.3 Humanoider Roboter mit OpenGL und POVRay visualisiert . . . 103
10.4 Kollisionserkennung im Echtzeitbetrieb 104

Kapitel 1

Einleitung

Am Fachgebiet Simulation und Systemoptimierung des Fachbereichs Informatik der TU Darmstadt werden verschiedene Forschungsaktivitäten im Bereich Robotik durchgeführt. Beispiele hierfür sind die Entwicklung optimaler Trajektorien für Industrieroboter, der Entwurf eines humanoiden Roboters und die Roboter-Fußballmannschaft „Darmstadt Dribbling Dackels“, die den Roboterhund Aibo von Sony einsetzt.
Im Rahmen dieser Forschungstätigkeit werden Werkzeuge zur dreidimensionalen Visualisierung benötigt. Anwendungen sind dabei sowohl die Generierung von Filmen als auch die Visualisierung von Simulationergebnissen in Echtzeit.
Bisher wird zum Erzeugen von Filmen eine Software (vgl. [Hel03]) verwendet, die auf die Graphikbibliothek Sigma (vgl. 7.1.3) aufsetzt. Da diese Bibliothek jedoch seit längerem nicht gewartet wird, wurde ein neues Programmpaket zum Erstellen von Animationen entwickelt, dass zusätzlich auch zur Echtzeitvisualisierung geeignet ist.
Dieses Programmpaket zeichnet sich durch eine starke Modularisierung aus. Es ist möglich, mit Hilfe der entwickelten Architektur verschiedene Programme zum Lösen von Visualisierugnsaufgaben zu realisieren. Im Rahmen der vorliegenden Arbeit wurden basierend auf derer zunächst entwickelten Programmbibliothek `roboViewLib` zwei Anwendungen entwickelt. Die eine, `roboMovieMaker`, dient dem Erstellen von Filmen, in denen Mehrkörpersysteme vordefinerte Bewegungen ausführen. Die zweite, `roboRTView`, dient der Visualisierungen der Bewegungen von Mehrkörpersystemen in Echtzeit.
Die entwickelte Architektur ist dabei so offen gehalten, dass es möglich ist, eine Verarbeitung der zu visualisierenden Daten einzufügen.
Eine Möglichkeit, die sich dadurch eröffnet, ist die Integration von Kollisionserkennungsverfahren in die Software, die es gestatten, während der Echtzeitvisualisierung von Simulationsdaten auftretende Probleme frühzeitig zu erkennen.
Dazu wurde eine Schnittstelle zur Integration von Kollisionserkennungsverfahren

definiert. Aufbauend auf dieser Schnittstelle wurden Verfahren zur Kollisionserkennung entwickelt und implementiert.
In den folgenden Abschnitten wird die Entwicklung der Bibliothek und der beiden Anwendungen dokumentiert. Dazu werden zunächst die Anforderungen an die Software definiert (Kapitel 2) und die für die Entwicklung wesentlichen Grundlagen dargestellt (Kapitel 3). Daraufhin wird die allgemeine Architektur der Software entwickelt(Kapitel 4).
In den nachfolgenden Abschnitten werden einzelne Aspekte der Architektur ausführlich vorgestellt. Diese sind Repräsentation der Szene (Kapitel 5), Kollisionserkennung (Kapitel 6) und Visualisierung (Kapitel 7).
Nachdem alle Komponeten der Bibliothek vorgestellt sind, werden die entwickelten Anwendungen zum Erzeugen von Filmen (Kapitel 8) und zur Echtzeitvisualisierung (Kapitel 9) vorgestellt.
Den Abschluss der Arbeit (Kapitel 10) bilden die Zusammenfassung der Ergebnisse und ein Ausblick auf weitere Möglichkeiten.

Kapitel 2

Anforderungen

Im Rahmen der vorliegenden Arbeit wurde ein Programmpaket, bestehend aus der Bibliothek `roboViewLib` und den Anwendungen `roboMovieMaker` und `roboRTView`, zur Visualisierung und Animation von Mehrkörpersystemen entwickelt. Im Folgenden werden die Anforderungen dargestellt, die die Grundlage für die Entwicklung bilden.

Architektur

Die Architektur soll es gestatten, die Visualisierung und Animation auf verschiedene Arten darzustellen. Dazu soll ein modularer Ansatz gewählt werden, der es gestattet, verschiedene Module zur Steuerung und Visualisierung der Mehrkörpersysteme zu verwenden und diese flexibel auszutauschen. Weiterhin soll es möglich sein, zusätzliche Verarbeitungsschritte wie z. B. das Erkennen von Kollsionen in die Anwendung zu integrieren.
Die einzelnen Programmkomponenten sollen in einer Bibliothek bereitgestellt werden, die es gestattet, aus den Komponenten verschiedene Anwendungen z. B. zum Erstellen von Filmen oder zur Echtzeitvisualisierung zu erstellen.

Anwendungen

Mit Hilfe der entwickelten Bibliothek sollen Anwendungen zum Erzeugen von Filmen und zur Echtzeitvisualisierung von Simulationsdaten erstellt werden.

Darzustellende Szene

Das Programmpaket soll in der Lage sein, eine Szene darzustellen, die aus einem oder mehreren Mehrkörpersystemen, deren Umgebung und Zusatzdaten besteht. Mehrkörpersysteme und Umgebung sollen dabei aus einfachen Formen wie Quadern, Ellipsoiden, Kegeln oder Ebenen bestehen. Es soll möglich sein, für jeden

angezeigten Körper einen Hüllkörper anzuzeigen. Die Zusatzdaten sind Informationen, wie Schwerpunkt oder Kraftvektoren, die sich z. B. beim Generieren einer optimierten Trajektorie ergeben. Sie sollen durch Pfeile, Geraden und Punkte dargestellt werden.
Mehrkörpersysteme, Umgebung und Zusatzdaten sollen gleichermaßen animierbar sein, wobei jeweils geeignete Parameter zur Steuerung verwendet werden sollen.

Eingabemodule

Es müssen zwei Arten von Eingabedaten verarbeitet werden: die Beschreibung der Szene und die Steuerdaten für die Animation.
Die Beschreibung der Szene soll über XML-Dateien erfolgen. Es liegt bereits ein Format zur Beschreibung von Mehrkörpersystemen vor (vgl. [Hel03]), das nach Möglichkeit weiterverwendet werden soll.
Die Steuerdaten für die Animation sollen aus verschiedenen Quellen eingelesen werden, die sich aus der konkreten Anwendung ergeben. Zum Erzeugen von Filmen soll es möglich sein, eine vorgefertigte Folge von Steuerdaten aus einer Datei einzulesen, zur Echtzeitvisualisierung muss es hingegen möglich sein, die Steuerdaten einer anderen Anwendung in Echtzeit weiterzuverarbeiten.

Verarbeitung

Es soll möglich sein, in den Anwendungen, die mit der Bibliothek erstellt werden, weitere Verarbeitungsschritte zwischen Einlesen der Steuerdaten und Ausgeben der generierten Graphik zu integrieren. Als Beispiel soll hier eine Schnittstelle zur Integration von Kollisionserkennungsverfahren integriert werden. Die Funktionsfähigkeit der Schnittstelle soll durch die Implementierung eines Kollisionsverfahrens belegt werden.

Ausgabe

Für verschiedene Anwendungsfälle werden verschiedene Ausgabemodule benötigt, die sich in der Qualität und Geschwindigkeit der Ausgabe unterscheiden.
Für den Fall der Echtzeitvisualisierung von Simulationsdaten soll ein Ausgabemodul entwickelt werden, dass in der Lage ist, animierte Graphik in der notwendigen Geschwindigkeit darzustellen. Dazu soll eine geeignete Programmierumgebung ausgewählt werden. Es soll möglich sein, einzelne Elemente der Szene ein- und auszublenden. Weiterhin ist es wünschenswert, mehrere Ansichten der Szene gleichzeitig zu visualisieren.

Neben der Erzeugung von Graphiken in Echtzeit ist das Generieren von Filmen ein weiteres wichtiges Thema. Die Ausgabemodule sollen deshalb in der Lage sein, die erzeugten Bilder nicht nur am Bildschirm darzustellen, sondern auch in Dateien abzulegen, um die Bilder zu Filmen zusammenfügen zu können. Für die Filmerzeugung soll ein weiteres Ausgabemodul bereitgestellt werden, dass die Erzeugung hochwertigerer Bilder mittels Raytracing gestattet.

Plattformen

Die Bibliothek und die Anwendungen sollen für die Linux-Plattform entwickelt werden. Dabei soll jedoch beachtet werden, dass möglichst nur Bibliotheken und Werkzeuge zum Einsatz kommen, die auch unter Windows verfügbar sind, um eine späterer Portierung auf diese Plattform zu erleichtern.

Kapitel 3

Grundlagen

3.1 Mathematische Beschreibung von Mehrkörpersystemen

In den folgenden Abschnitten wird erklärt, wie sich Roboter als Mehrkörpersysteme beschreiben lassen. Zunächst wird diskutiert, wie die räumliche Lage der einzelnen Elemente eines solchen Systems beschrieben wird. Dabei werden die Beschreibung der Orientierung durch Quaternionen und die Beschreibung von Transformationen durch homogene Koordinaten eingeführt, die im weiteren Verlauf der Arbeit benötigt werden. Im Anschluss daran wird beschrieben, wie die zeitabhängige Bewegung der Elemente eines Mehrkörpersystems über eine parametrisierte Matrix beschrieben wird.

Soweit nicht anders angegeben folgt die Darstellung [Str01] und [Cra89].

Mehrkörpersysteme

Robotersysteme lassen sich als Mehrkörpersysteme beschreiben. Mehrkörpersysteme sind Systeme mehrerer starrer Objekte im $\mathbb{R}^3$, die sich relativ zueinander bewegen können. Die Objekte sind z. B. die Glieder eines Roboters, die Bewegung erfolgt üblicherweise durch rotatorische und translatorische Gelenke.

Mehrkörpersysteme lassen sich hierarchisch als Baum beschreiben. Wird ein Objekt bewegt, so werden alle ihm untergeordneten Objekte mitbewegt. Die Bewegung eines Objektes wird relativ zu dem ihm dirket übergeorneten Objekt beschrieben.

Räumliche Lage

Im Folgenden bezeichnet der Begriff „Lage“ die Position und Orientierung eines Objekts.
Zur Beschreibung der räumlichen Lage der Objekte wird jedes Objekt fest mit einem Koordinatensystem S_{Obj} verbunden. Der Ursprung des Koordinatensystems gibt die Position eines festen Punktes des Objekts an, die Richtungen der Basisvektoren bestimmen die Ausrichtung des Objekts.
Die Koordinatensysteme sind orthogonale, kartesische Rechtssysteme. Vektoren beziehen sich dabei immer auf eines der Koordinatensysteme. Um zu verdeutlichen, auf welches Koordinatensystem sich ein Vektor v bezieht, wird das Koordinatensystem oben links vom Vektor angegeben. ${}^{a}v$ bezieht sich also auf S_a.
Der Wechsel von einem Ausgangskoordinatensystem S_a in ein Zielkoordinatensystem S_z wird durch die Translation des Ursprungs und die Abbildung der Einheitsvektoren des Ausgangs- in die des Zielkoordinatensystems dargestellt. Die Translation wird durch einen Vektor ${}^{z}r_a$ beschrieben, die Abbildung der Einheitsvektoren durch eine 3x3-Matrix ${}^{z}R_a$. Die Spalten der Matrix ${}^{z}R_a$ bestehen dabei aus den Einheitsvektoren von S_z, dargestellt bezüglich S_a. Um einen Punkt ${}^{a}p$ im Koordinatensystem S_z darzustellen, muss er wie folgt transformiert werden:

$$ {}^{z}p = {}^{z}r_a + {}^{z}R_a \cdot {}^{a}p. $$

Da alle verwendeten Koordinatensysteme orthogonal, rechtwinklig und rechtshändig sind und auch die Einheitsvektoren ihre Länge nicht ändern, ergeben sich spezielle Eigenschaften für die Matrix R: So sind u. a. die Spaltenvektoren der Matrix orthonormal zueinander. Solche Matrizen beschreiben Rotationen der Koordinatensysteme[1]. Es lässt sich zeigen, dass eine Rotationsmatrix durch nur drei Parameter vollständig beschreibbar ist (vgl. [Cra89]). Die drei Parameter lassen sich auf verschiendene Arten wählen. Zwei Möglichkeiten sind die unten vorgestellten Euler- und Kardanwinkel.
Der Wechsel des Koordinatensystems lässt sich somit durch sechs Parameter beschreiben: drei für die Translation entlang der drei Koordinatenachsen und drei für die Rotation.

Rotationen

Rotationen um die Koordinatenachsen lassen sich trivial durch die folgenden Matrizen darstellen (vgl. [WNDS99]), wobei $\alpha \in \mathbb{R}$ den Drehwinkel und der Index $i \in x, y, z$ der Matrix R_i die Drehachse angibt:

[1] Es treten keine anderen linearen Abbildungen (vgl. [Wil98]) wie Scherung oder Skalierung, die sich ebenfalls durch 3x3-Matrizen beschreiben lassen, auf.

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Zur Darstellung von Rotationen um andere Achsen existieren verschiedene Möglichkeiten:

- Kardanwinkel beschreiben die Rotation durch drei verkettete Rotationen um die Koordinatenachsen eines festen Koordinatensystems.
- Eulerwinkel beschreiben die Rotation durch eine Folge von Rotationen um Koordinatenachsen des momentanen Bezugssystems.
- Quaternionen erlauben es, Rotationen um eine beliebige Achse durch den Ursprung anzugeben.

Die Darstellungen durch Kardan- oder Eulerwinkel sind für Rotationen um die Koordinatenachsen sehr anschaulich, aber nicht optimal. So existiert z.B. nicht immer eine eindeutige Darstellung einer bestimmten Rotation. Bei den Eulerwinkeln existiert zusätzlich das Problem des sog. „Gimbal Lock“ (vgl. [Ale01]): Wird bei einer der Drehungen um 90 Grad gedreht, kann eine Rotationsachse mit einer anderen zusammenfallen, so dass ein Freiheitsgrad eingebüst wird.
Quaternionen (vgl. [Beu98]) sind eine Erweiterung der komplexen Zahlen auf drei imaginäre Einheiten. Für die drei imaginären Einheiten i, j, k gelten folgende Rechenregeln:

$$i^2 = j^2 = k^2 = -1$$

$$ij = k = -ji$$

Da für die Quaternionen das Kommutativgesetz nicht gilt, bilden sie keinen *Körper*, sondern nur einen *Schiefkörper*.
Um die Rotation eines Punktes $p = \begin{pmatrix} p_x & p_y & p_z \end{pmatrix}^T \in \mathbb{R}^3$ um einen Einheitsvektor $r = \begin{pmatrix} r_x & r_y & r_z \end{pmatrix}^T \in \mathbb{R}^3$ und den Winkel $\alpha \in \mathbb{R}$ mit Quaternion durchzuführen, werden zunächst der Punkt und die Rotation als Quaternionen q_p und q_r ausgedrückt:

$$q_p = 0 + p_x i + p_y j + p_z k$$

$$q_r = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(r_x i + r_y j + r_z k)$$

Für das ebenfalls benötigte inverse Quaternion zu q_r gilt:

$$q_r^{-1} = \cos\left(\frac{\alpha}{2}\right) - \sin\left(\frac{\alpha}{2}\right)(r_x i + r_y j + r_z k)$$

Der rotierte Punkt q'_p ergibt sich dann wie folgt:

$$q'_p = q_r q_p q_r^{-1}.$$

Mit Quaternionen ist es somit möglich, Rotationen um beliebige Achsen ohne die für Kardan- oder Eulerwinkeln genannten Probleme zu beschreiben.
Aus dem Quaternion q_r lässt sich mit $c_\alpha := \cos\alpha$ und $s_\alpha := \sin\alpha$ eine Rotationsmatrix erzeugen (vgl. [Kir02]):

$$R = \begin{pmatrix} (1-c_\alpha)p_x^2 + c_\alpha & (1-c_\alpha)p_x p_y - s_\alpha p_z & (1-c_\alpha)p_x pz - s_\alpha p_y \\ (1-c_\alpha)p_x p_y + s_\alpha p_z & (1-c_\alpha)p_y^2 + c_\alpha & (1-c_\alpha)p_y p_z - s_\alpha p_x \\ (1-c_\alpha)p_x p_z - s_\alpha p_y & (1-c_\alpha)p_y p_z + s_\alpha p_x & (1-c_\alpha)p_z^2 + c_\alpha \end{pmatrix}.$$

Homogene Transformationen

Rotationen wurden bis jetzt als Multiplikationen mit einer Matrix, Translationen als Addition eines Vektors beschrieben. Durch Verwendung „homogener Koordinaten“ lässt sich hier eine Vereinheitlichung der Darstellung erzielen, bei der beide Transformationen als Multiplikation mit einer 4x4-Matrix dargestellt werden.
Dazu werden Punkte $p = \begin{pmatrix} x & y & z \end{pmatrix}^T \in \mathbb{R}^3$ wie folgt in den $\mathbb{R}^4$ eingebettet: $\hat{p} = \begin{pmatrix} x & y & z & 1 \end{pmatrix}^T \in \mathbb{R}^4$. Die Rotationsmatrix ${}^z R_a \in \mathbb{R}^{3x3}$ und der Translationsvektor ${}^z r_a \in \mathbb{R}^3$ werden in eine 4x4-Matrix eingebettet:

$${}^z T_a = \begin{pmatrix} & {}^z R_a & & {}^z r_a \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \mathbb{R}^{4x4}$$

Neben der Vereinheitlichung von Rotation und Translation ergibt sich als weiterer Effekt, dass sich eine Folge von Transformationen durch Aufmultiplizieren der homogenen Transformationsmatrizen leicht zu einer Transformation zusammenfassen lässt.

Kinematik der Objekte

Zur Beschreibung der Bewegung der Objekte eines Mehrkörpersystems müssen die Transformationsmatrizen, die die relative Lage der Objekte angeben, entsprechend der verwendeten Gelenke parametrisiert werden. Um z. B. die Bewegung, die ein rotatorisches Gelenk, das um eines der Koordinatenachsen dreht, darzustellen, kann eine der oben angegebene Rotationsmatrizen verwendet werden.
Da Gelenke i. Allg. nicht im Ursprung des Koordinatensystems des übergeordneten Körpers ansetzen, sind die zeitlich veränderlichen Transformationsmatrizen meist komplizierter. Häufig entstehen sie aus einer Kombination zeitlich fester und veränderlicher Transformationen, die verknüpft werden, um die gesamte Transformation zu beschreiben.
Für jedes Gelenk erhält man eine Matrix $T(q),\ T \in \mathbb{R}^{4x4},\ q \in \mathbb{R}$.
Um die Lage eines bestimmtes Objektes zu beschreiben, werden alle Transformationsmatrizen beginnend von der Wurzel des Baumes bis zum Objekt aufmultipliziert. Bei n Gelenken auf diesem Pfad erhält man eine Matrix $T(\bar{q}), T \in \mathbb{R}^{4x4}, \bar{q} \in \mathbb{R}^n$. Dabei ist $\bar{q}$ der Vektor der Gelenkparameter.
Um das Objekt zu bewegen, müssen die Elementen von $\bar{q}$ zeitlich verändert werden. Dazu wird der Parametervektor als zeitlich veränderliche Funktion $\bar{q}(t), t \in \mathbb{R}_0^+$ beschrieben. Die zeitlich veränderliche Lage eines Objektes zu einem Zeitpunkt $t_i \in \mathbb{R}_0^+$ wird dann durch die Transformation $T(\bar{q}(t_i))$ beschrieben.
Das Bestimmen der Lage eines Objektes für einen gegebene Parametervektor $\bar{q}$ wird als „Lösen des Vorwärstkinematik-Problems“ bezeichnet. Die Lösung lässt sich trivial durch Berechnen und Aufmultiplizieren der Transformationsmatrizen bestimmen.
Der umgekehrte Fall, das Bestimmen der notwendigen Parameter $\bar{q}$, so dass ein Objekt eine bestimmte Lage einnimmt, wird als „Lösen des Rückwärtskinematik-Problems“ bezeichnet. Das Auffinden der Lösung ist nicht trivial. Für die meisten Transformatione existieren keine algebraischen Lösungsverfahren, so dass auf numerische Methoden zurückgegriffen werden muss. Häufig existieren keine eindeutigen Lösungen.

Dynamische Betrachtung

Neben der Betrachtung der Kinematik, die die räumliche Lage bewegter Objekte beschreibt, ist zur Steuerung realer Systeme zusätzlich eine Betrachtung der Dynamik notwendig. Dabei werden die bei den Bewegungen auftretenden Geschwindigkeiten und Beschleunigungen sowie die von außen einwirkenden Kräfte (z. B. Schwerkraft) betrachtet.
Ziel dieser Betrachtungen ist es, die Bewegung der Mehrkörpersysteme so zu steuern, dass bestimmte Bedingungen erfüllt werden. Ein mögliches Ziel ist z. B. die

Einhaltung von Obergrenzen für die in den Gelenken auftretenden Kräfte.
Die bei dynamischen Betrachtungen auftretenden Differentialgleichungen werden üblicherweise mit numerischen Methoden behandelt.

3.2 Prozesse, Threads und Synchronisation

Im folgenden Abschnitt wird beschrieben, wie ein Programm in mehrere, scheinbar parallel ablaufende Ausführungspfade aufzuteilt werden kann und wie diese synchronisiert werden. Die Darstellung orientiert sich an [Tan01] und [SG98].
Die meisten modernen Betriebssysteme gestatten es, mehrere sequentielle Programmabläufe parallel auszuführen.
Prozesse sind die ursprüngliche Form der Parallelität. Ein Prozess beschreibt dabei ein einzelnes Programm in Ausführung. Werden mehrere Prozesse parallel ausgeführt, spricht man von Multitasking oder Multiprogramming. Dabei gibt es verschiedene Vorgehensweisen des Betriebssystems: Beim kooperativen Multitasking wird ein Prozess solange ausgeführt, bis er die Ausführung von selbst unterbicht, d. h. Ein- oder Ausgabe ausführt, auf Ressourcen wartet oder über spezielle Systemaufrufe den Prozessor abgibt. Bei diesem Vorgehen kann theoretisch ein Prozess unendlich lange rechnen und damit das Vorankommen aller anderen Prozesse verhindern. Dies ist nicht wünschenswert, weshalb moderen Betriebssysteme üblicherweise verdrängendes Multitasking einsetzen. Hierbei kann das Betriebssystem einen laufenden Prozess an einer beliebigen Stelle anhalten und einen anderen Prozess zur Ausführung bringen.
Wird beim Multitasking ein laufender Prozess unterbrochen, so muss vom Betriebssystem ein neuer Prozess bestimmt werden, der ausgeführt werden soll. Dies wird durch die sog. „Scheduling-Strategie“ bestimmt. Für verschiedene Anwendungszwecke gibt es dabei stark unterschiedliche Verfahren, die z. B. die Antwortzeit oder den Gesamtdurchsatz des Systems beeinflussen.
Die einzelnen Prozesse werden üblicherweise streng voneinander abgeschirmt, indem sie in getrennten Adressräumen ausgeführt werden. Durch diese Abgrenzung wird verhindert, dass sich die Prozesse im Ablauf gegenseitig stören. Gleichzeitig wird dabei aber eine einfache Nutzung gemeinsamer Daten erschwert [2].
Für viele Anwendungsfälle ist es wünschenswert, einen gemeinsamen Datenbestand durch verschiedene parallel ablaufende Programmteile zu verarbeiten. Typische Beispiele hierfür sind z. B. die Erfassung und Verarbeitung von Messdaten oder Serverprogramme, die verschiedene Benutzer gleichzeitig bedienen.
Um hierbei den Aufwand für den Datenaustausch zwischen den einzelnen parallelen Abläufen zu erleichtern, wurden Threads eingeführt. Threads sind parallele

[2] Sollen Daten direkt über den Hauptspeicher ausgetauscht werden, stellen die meisten Betriebssysteme spezielle Verfahren wie „shared memory“ zur Verfügung.

Ausführungspfade eines einzelnen Prozesses. Im Gegensatz zu parallelen Prozessen laufen Threads in einem gemeinsamen Adressraum ab, so dass ein unkomplizierter Austausch gemeinsam genutzter Daten möglich ist.
Greifen mehrere parallele Threads [3] gleichzeitig auf gemeinsame Daten zu, müssen diese Zugriffe synchronisiert werden, um Inkonsistenzen zu vermeiden. Ein typisches Problem sind „race-conditions“, die auftreten, wenn mehrere Threads gleichzeitig den gleichen Speicherbereich ändern. Abhängig davon, wann das System zwischen den Threads wechselt, können völlig verschiedene Ergebnisse nach Abschluss aller Threads in den betroffenen Speicherbereichen stehen.
Zur Vermeidung von race-conditions muss beim Zugriff auf gemeinsame Speicherbereiche der „wechselseitige Ausschluss“ sichergestellt werden. Dies bedeutet, dass, solange ein Thread einen Codeabschnitt ausführt, der auf den Speicherbereich zugreift, kein anderer Thread einen derartigen Codeabschnitt ausführen darf.
Das Standardwerkzeug zum Lösen dieser und anderer Synchronisationsaufgaben sind Semaphore. Semaphore sind spezielle Zählvariablen, die üblicherweise vom Betriebssystem bereitgestellt werden und nur über Systemfunktionen manipuliert werden können.
Dabei stehen die Operationen „Inkrementieren“ und „Dekrementieren“ zur Verfügung. Versucht ein Thread oder Prozess den Wert eines Semaphors, der bereits Null ist, weiter zu dekrementieren, wird der Thread solange blockiert, bis ein anderer Thread den Wert mindestens einmal inkrementiert hat. Erst dann kann der blockierte Thread den Wert dekrementieren und seine Arbeit fortsetzen.
Semaphore eignen sich zur Lösung einer Vielzahl von Synchronisationsproblem (vgl. [Tan01] und [SG98]).

3.3 Echtzeit

In [SG01] werden Echtzeitsysteme als Betriebssysteme beschrieben, die z. B. bei der Steurung und Überwachung zeitkritischer technischer Prozesse zum Einsatz kommen. Solche Systeme zeichnen sich dadurch aus, dass sie bei der Programmausführung vorgegebene Zeitschranken einhalten.
In [Tan01] wird zusätzlich zwischen „harter“ und „weicher“ Echtzeit unterscheiden. Dabei bedeutet harte Echtzeit die unbedingte Einhaltung der Zeitvorgaben, wie es zur Steurung technischer Prozesse nötig ist. Bei weicher Echtzeit hingegen sind gelegentliche Verzögerungen tolerabel. Sie eignen sich z. B. zum Einsatz für Aufgaben im Multimediabereich.
Nach [SG98] sind für harte Echtzeitanforderungen spezielle Betriebssysteme und häufig auch spezielle Hardware erforderlich. Weiche Echtzeit wird hingegen auch

[3] Die Probleme treten natürlich auch bei der Verwendung von Prozessen auf.

von modernen Universalbetriebssystemen wie Linux oder Windows NT unterstützt.
Für die vorliegende Arbeit bedeutet Echtzeit somit weiche Echtzeit. Die konkrete Anforderung ist, dass das System in der Lage ist, eintreffende Steuerdaten ohne Verzögerung zu visualisieren bzw. eine konstante Bildrate aufrechtzuerhalten.

3.4 Implizite Beschreibung geometrischer Objekte

In diesem Abschnitt wird beschrieben, wie geometrische Objekte durch Funktionen $f : \mathbb{R}^3 \mapsto \mathbb{R}$ implizit beschrieben werden. Eine solche Darstellung gestattet es, für jeden Punkt $p \in \mathbb{R}^3$ direkt zu entscheiden, ob er Teil des Objektes ist oder nicht, was in Abschnitt 6 für die Kollisionserkennung ausgenutzt wird.
Betrachtet man geometrische Objekte als Teilmengen des $\mathbb{R}^3$, so gibt es verschiedene Arten, solche Objekte zu beschreiben. Neben der expliziten Darstellung, bei der mit Hilfe einer Funktion aus einem oder mehreren Parametern die Punkte einer Kurve oder einer Oberfläche erzeugt werden, gibt es die alternative Möglichkeit der impliziten Definition von Objekten.
Um ein geometrisches Objekt implizit zu definieren, wird eine Funktion $f : \mathbb{R}^3 \mapsto \mathbb{R}$ angegeben, die für jeden Punkt aus dem Raum entscheidet, ob der Punkt Teil des Objektes ist oder nicht. Bei geschlossenen Körpern ist über diese Funktion eine Entscheidung möglich, ob ein Punkt auf der Oberfläche, im Inneren oder außerhalb des Körpers liegt.
Im Folgenden werden zwei implizite Beschreibungen angegeben, die im weiteren Verlauf der Arbeit benötigt werden:

- Die Hesse-Normalform zum Beschreiben von Ebenen und Halbräumen.
- Superquadriken, die eine bestimmte Klasse geschlossener Körper darstellen.

Hesse-Normalform

Die Hesse-Normalform (vgl. [BSMM95]) beschreibt eine Ebene E durch Angabe eines Ortsvektors $\vec{r} \in E$ und eines Einheitsvektors $\vec{n}$, der senkrecht auf der Ebene steht und somit die Normale der Ebene angibt. Über die Funktion

$$d(\vec{p}) := \vec{n}(\vec{p} - \vec{r})$$

lässt sich der Abstand eines beliebigen Punktes $\vec{p}$ von der Ebene berechnen. Positive Werte bedeuten dabei, dass der Punkt sich auf der Seite der Ebene befindet, in die die Normale zeigt.

Damit lässt sich die Menge aller Punkte, die auf der Ebene liegen wie folgt angeben:

$$E = \{\vec{p} \in \mathbb{R}^3 \,|\vec{n}(\vec{p} - \vec{r}) = 0\}.$$

Superquadriken

Superquadriken (vgl. [Ale02]) werden durch die Funktion

$$f_{(e_1,e_2)}(x,y,z) = \left(x^{\frac{2}{e_2}} + y^{\frac{2}{e_2}}\right)^{\frac{e_2}{e_1}} + z^{\frac{2}{e_1}}$$

implizit beschrieben. Alle Punkte $p = (x,y,z)$, für die $f(p) \leq 1$ gilt, sind dabei Teil des Körpers. Durch die Parameter e_1 und e_2 lässt sich die Gestalt des Körpers beeinflussen.
Superquadriken sind gewissermassen Abstandfunktionen, bei denen der Abstand entlang der verschiedenen Koordinatenachsen verschieden definiert ist. Für $e_1 = e_2 = 1$ ergibt sich z. B. der normale euklidische Abstand vom Mittelpunkt, für $e_1 = e_2 = 2$ ergibt sich die 1-Norm (vgl. p-Norm, [For96]), bei der der „Abstand“ die Summe der Abstände in Richtung der Koordinatenachsen ist. Lässt man e_1 und e_2 gegen Null gehen, kann man die ∞-Norm annähern, die dem Maximum der Abstände in Richtung der Koordinatenachsen entspricht.
Die folgenden Abbildungen geben einen Eindruck der Beschreibungsmöglichkeiten, die Superquadriken bieten:

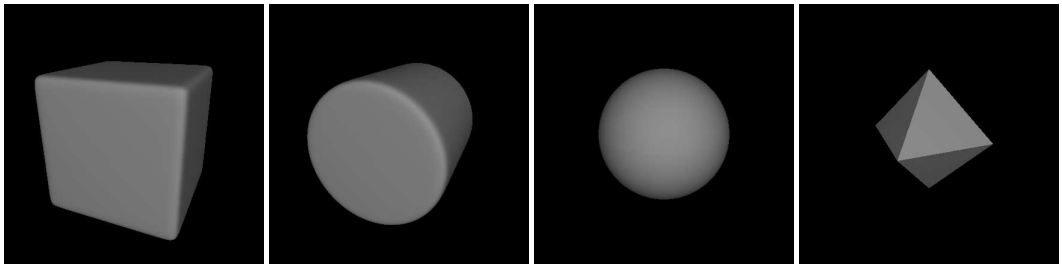

Abbildung 3.1: Die Abbildungen zeigen von links nach rechts Superquadriken mit den Parametern $e_1 = 0.1$ und $e_2 = 0.1$, die einen Würfel annähern, $e_1 = 1$ und $e_2 = 0.1$, die einen Zylinder annähern, $e_1 = 1$ und $e_2 = 1$, die eine Kugel beschreiben und $e_1 = 2$ und $e_2 = 2$, die einen Oktaeder beschreiben.

Die so beschriebenen Superquadriken beschreiben Körper, deren Mittelpunkt sich im Ursprung des Koordinatensystems befindet und die einem achsparallelen Würfel der Seitenlänge 2 einbeschrieben sind. Sollen Lage und Seitenverhältnisse eines Körpers verändert werden, so müssen die Koordinaten, die in die Funktion eingegeben werden, entsprechend transformiert werden. Dabei kommen neben

Translation und Rotation (vgl. 3.1) zusätzlich Skalierung und Scherung in betracht, die sich leicht durch entsprechende Matrizen (vgl. [ESK96]) darstellen lassen.

3.5 Struktur von XML-Dateien

In diesem Abschnitt wird kurz beschrieben, wie XML-Dateien strukturiert sind und wie sie eingelesen werden können. Dabei werden nur diejenigen Aspekte diskutiert, die für die vorliegende Arbeit relevant sind. Die Darstellung bezieht sich auf [Bag97] und [XML1].
XML, die „**Ex**tensible **M**arkup **L**anguage“ erlaubt es Dokumente, hierarchisch zu strukturieren. Dabei entsteht ein Baum von durch sog. „Tags“ begrenzten Textabschnitten, die beliebig geschachtelt werden können.
Ein Knoten dieses Baumes wird durch ein Start-Tag eingeleitet und durch ein End-Tag abgeschlossen. Beide Tags tragend den gleichen, frei wählbaren Bezeichner, der in spitze Klammern gestellt wird. Im End-Tag wird dem Bezeichener ein Schrägstrich vorangestellt. Eine einfache Baumstruktur, bestehend aus einem Knoten mit zwei Kindknoten, lässt sich also wie folgt beschreiben:

```
<Tag1>
  <Tag2>
  </Tag2>
  <Tag3>
  </Tag3>
</Tag1>
```

Zwischen Start- und End-Tag der Knoten lassen sich beliebige Inhalte einfügen.
Es ist möglich, in den Start-Tags zusätzliche Parameter anzugeben. Dazu wird ein Bezeichner angegeben und durch ein Gleichheitszeichen ein Wert zugewiesen. Der zugewiesene Wert muss dabei immer in Anführungszeichen stehen. Um für einem Knoten vom Typ `Knoten` den Parameter `Parameter` zu setzen, wird also folgende Syntax verwendet:

```
<Knoten Parameter=''Wert''>
  Hier stehen die Daten des Knotens
</Knoten>
```

Knoten, deren einziger Inhalt die Parameter sind und die keine Unterknoten benötigen kein End-Tag Stattdessen wird aus dem Start-Tag nach dem Namen des Knotens und den Parametern einem abschliessenden Schrägstrich hinzugefügt:

```
<EinfacherKnoten Parameter1=''Wert'' Parameter2=''Wert'' />
```

Um mit Hilfe von XML Daten zu repräsentieren, müssen ledgilich die benötigten Tags und Parameter definiert werden.

Ein einfach nutzbares Mittel zum Einlesen von XML-Dokumenten ist die Bibliothek `libXML` (vgl. [XML2]). Sie generiert beim Einlesen eines Dokumentes automatisch eine Baumrepräsentation des Dokuments. Dieser Baum kann dann traversiert werden, um die strukturierten Daten auszuwerten. Die Bibliothek ist für viele Plattformen, darunter auch Linux und Windows, verfügbar.

3.6 Templates

Die Programmiersprache C++ bietet mit dem Sprachkonzept der „Templates“ (vgl. [Str92]) die Möglichkeit, zur Übersetzungszeit parametrisierbare Klassen zu definieren. Diese Parameter können z. B. Zahlenwerte und Datentypen sein. Man kann z. B. ein Template für quadratische Matrizen zu erzeugen, mit dem es möglich ist, Klassen für Matrizen beliebiger Dimension und beliebiger Zahlentype zu erzeugen.

Eine weitere häufige Anwendung von Templates ist das Erzeugen sog. „Containerlassen“, wie z. B. verkettete Listen oder dynamische Felder. Dabei wird der Programmcode zur Verwaltung der Daten nur einmal geschrieben und ist zunächst unabhängig vom zu speichernden Datentypen. Soll eine Containerklasse für einen konkreten Datentypen erstellt werden, geschieht dies mit Hilfe des Templates automatisch.

In der „Standard Template Library“ (STL) (vgl. [STL]) ist eine Vielzahl von Templates zusammengefasst, die auf einheitliche Art und Weise Containerklassen und darauf operierende Algorithmen bereitstellen.

In der vorliegenden Arbeit wird häufig vom `vector`-Template aus der STL Gebrauch gemacht. Dieses stellt ein dynamisches Feld bereit, dessen Größe zur Laufzeit geändert werden kann. Die Zugriffsoperatoren der mit dem Template erzeugten Klassen sind dabei so beschaffen, dass sie syntaktisch kompatibel mit den normalen, in C (vgl. [ER94]) verwendeten Feldern sind.

3.7 3D-Graphik

In diesem Abschnitt wird ein kurzer Überblick über das Erzeugen von 3D-Graphik am Rechner gegeben. Den ersten Schwerpunkt der Darstellung bilden die derzeit angewandten Verfahren. Von besonderem Interesse ist dabei zum einen die Realitätsnähe der generierten Graphik und zum anderen die Geschwindigkeit der Ausgabe. Daraufhin wird ein Überblick über Programmierkonzepte und Bibliotheken gegeben, die zur Erzeugung von 3D-Graphik herangezogen werden können.

3.7.1 Verfahren

Beim Erzeugen von 3D-Graphik wird die Beschreibung einer dreidimensionalen Szene in ein zweidimensionales Bild umgesetzt. Hierzu wurden im Laufe der Jahre verschiedene Verfahren vorgestellt und eingesetzt. Für die vorliegende Aufgabenstellung von besonderem Interesse sind Verfahren, die in Echtzeit ausgeführt werden können. Im Folgenden wird deshalb ein grober Überblick über die Möglichkeiten und Einschränkungen moderner Graphikhardware gegeben. Als alternatives, derzeit jedoch nur eingeschränkt echtzeitfähiges Verfahren, wird Raytracing präsentiert.
Die nachfolgende Darstellung orientiert sich an [ESK96], [ESK97], [FDFHP94] und [Ber02].

Das Darstellungsverfahren aktueller Graphikhardware

Seit einigen Jahren werden im PC-Bereich Graphikkarten entwickelt, die für die Erzeugung von 3D-Graphik optimiert sind. Die Karten übernehmen dabei immer mehr Aufgaben der Graphikerzeugung selbständig, um die eigentliche Anwendung zu entlasten. Aktuelle Modelle sind dazu mit erheblicher eigener Rechenleistung und großen Mengen an Speicher ausgestattet.
Die aktuelle Graphikhardware ist daraufhin optiomiert, Szenen darzustellen, die als Menge von Polygonen, meist Dreiecken, beschrieben werden.
Die einzelnen Elemente der Szene werden durch die Eckpunkte ihrer Polygone beschrieben. Üblich ist dabei die Verwendung homogener Koordinaten. Diese erlauben es, alle relevanten Transformationen (z. B. Rotation, Translation, Skalierung, aber auch Projektion und perspektivische Verzerrung) einheitlich durch Multiplikationen mit Matrizen zu beschreiben. Hierdurch ist es möglich, zunächst alle Transformationen, denen eine Szene unterworfen werden soll, durch Aufmultiplizieren der Matrizen zusammenzufassen und anschließend alle zu transformierenden Eckpunkte nur mit jeweils einer Matrix zu multiplizieren. Die ist erheblich effizienter als jeden einzelnen Eckpunkt einer Folge von Transformationen zu unterziehen.
Nachdem die Polygone der Szene entsprechend der gewünschten Abbildungsvorschriften transformiert sind, müssen sie am Bildschirm dargestellt werden. Ein wesentliches Problem dabei ist es, dass sich Polygone gegenseitig verdecken können. Für dessen Lösung stehen eine Vielzahl verschiedener Verfahren bereit; von der Graphikhardware wird nur eines dieser Verfahren, der z-Buffer (vgl. [ESK96]), verwendet. Dazu wird neben dem Frame-Buffer, der die Farbwerte der Bildpunkte speichert, ein weitere Puffer, der z-Buffer verwaltet, der für jeden Bildpunkt festhält, wie weit der gezeichnete Punkt vom Betrachter entfernt ist. Der z-Buffer wird mit dem maximal möglichen Entfernungswert für jeden Bildpunkt initiali-

siert. Anschließend werden alle Polygone, die sich innerhalb des „virtuellen Gesichtfeldes “ befinden, gezeichnet[4]. Dabei wird für jeden Bildpunkt zusätzlich der Entfernungswert berechnet. Farb- und Entfernungswert werden nur dann in Frame- und z-Buffer eingetragen, wenn der Enfernungswert kleiner ist als der bereits für den Bildpunkt eingetragene.

Beim Zeichnen der Polygone in den Frame-Buffer muss zusätzlich die Farbe jedes Bildpunktes bestimmt werden. Dazu existieren verschiedenste Techniken, die vom einfachen Einfärben des gesamten Polygons mit einer Farbe über das Erzeugen von Farbverläufen bis zum Aufbringen zweidimensionaler Bilder (sog. Texturen, vgl. [ESK97]) auf die Oberfläche reichen. Seit kurzem ist es möglich, hier eigene Algorithmen in die Graphikhardware zu laden, um spezielle Effekte erzielen zu können.

Zusätzlichen Einfluss auf die Berechung der Farbwerte der einzelnen Bildpunkte hat die Beleuchtungsrechnung (vgl. nächster Abschnitt). Hierzu werden lokale Beleuchtungsmodelle eingesetzt. Dies bedeutet, dass zur Berechnung der Lichtintensität die Lage der einzelnen Polygone zu vorher festgelegten Lichtquellen herangezogen wird. Dabei wird jedoch (im Gegensatz zu globalen Modellen) nicht berechnet, ob sich Hindernisse zwischen Objekt und Lichtquelle befinden[5]. Üblicherweise wird zur Beleuchtungsrechnung mit Hilfe des Phong-Modells (vgl. [ESK96]) ein Farbwert für jeden Eckpunkt des Polygons berechnet. Anschliessend wird aus diesen Werten mittels Gouraud-Shading (vgl. [ESK96]) der Farbwert (oder der Helligkeitswert, falls Texturen verwendet werden) jedes Bildpunktes berechnet.

Das hier grob umrissene Verfahren hat den entscheidenden Vorteil, dass sich die einzelnen Schritte leicht in speziellen Prozessoren ausführen lassen. Dabei wurden die Aufgaben Schritt für Schritt von der Anwendersoftware in die Graphikhardware verlagert. Zunächst war es der Hardware nur möglich, schattierte Dreiecke unter Beachtung der Verdeckungsrechnung mit hoher Geschwindigkeit zu erzeugen. Weitere Aufgaben, wie die Transformation der Szene oder die Beleuchtungsrechnung mussten zunächst noch von der Software vorgenommen werden, wurden aber Schritt für Schritt in die Hardware integriert. Parallel zu dieser Entwicklung wurden auch die Möglichkeiten der Hardware bei der Berechnung der Farbwerte der einzelnen Bildpunkte erweitert: Vom Berechnen von Farbverläufen über das Aufbringen von Texturen bis zu frei programmierbaren sog. „Pixel-Shadern“(vgl. [Ber02]).

Durch die weitgehende Integration in die Hardware sind sehr hohe Geschwindigkeiten erreichbar, so dass es möglich ist, selbst komplexe Szenen in Echtzeit

[4]Durch geschickte Vorauswahl der zu zeichnenden Polygone lässt sich hier die benötigte Rechenzeit zum Erzeugen eines Bildes erheblich reduzieren.

[5]Folglich werden keine Schatten generiert.

zu animieren. Dennoch hat das Verfahren auch einige Nachteile. Durch das globale Beleuchtungsmodell müssen realistischere Beleuchtungseffekte wie Schatten aufwändig simuliert werden. Ebenso lassen sich weitere Effekte, die sich aus der Lichtausbreitung ergeben, wie z. B. transparente Objekte und Spiegeleffekte, nur über Umwege generieren. Ein weiterer Nachteil ist, dass die Szene als Menge von Polygonen beschrieben wird. Gekrümmte, stetig differenzierbare Oberflächen, wie sie z. B. durch den Einsatz von Splines und Tensorproduktflächen (vgl. [ESK96]) enstehen, müssen dazu in Polygone aufgeteilt werden. Die entstehenden Körper weisen dann an den Berührungskanten der Polygone Knicke auf. Um hier den Eindruck einer glatten Oberfläche aufrechtzuerhalten, muss eine entsprechend feine Unterteilung gewählt werden. Hierdurch entstehen sehr große Polygonmengen, so dass der Rechenaufwand entsprechend steigt.
Der Berechnungsaufwand für Graphiken hängt hauptsächlich von der Anzahl der Polygone und der Größe des zu erzeugenden Bildes ab.

Beleuchtungsrechnung und Materialrepräsentation

Ziel der Beleuchtungsrechnung ist es, den Einfluss einer oder mehrerer Lichtquellen auf das Aussehen einer Szene zu bestimmen. Eng mit der Beleuchtungsrechnung verknüpft ist die Beschreibung der Materialien, die das Reflexionsverhalten der dargestellten Oberflächen bestimmen.
Generell werden globale und lokale Beleuchtungsmodelle unterschieden. Globale Beleuchtungsmodelle betrachten dabei die gesamte Szene, so dass jedes Objekt die Beleuchtung bzw. Abschattung der anderen Objekte beeinflusst. Bei lokalen Modellen hingegen wird die Beleuchtungssituation für jedes Objekt der Szene berechnet, als sei es das einzige vorhandene Objekt. Dies schließt natürlich die einfache Berechnung von Schatten aus.
Beim üblicherweise eingesetztes lokalen Beleuchtungsmodell (vgl. [ESK96]) gehen in die Berechnung der Leuchtintensität für einen beliebigen Punkt einer Oberfläche die Position des Betrachters, die Normale der Oberfläche im betrachteten Punkt und die Lage der Lichtquelle(n) ein. Es werden drei Arten von Lichtreflexionen unterschieden:

- Ambiente Reflexionen, deren Intensität unabhängig von der Lage des Betrachters und der der Lichtquellen ist. Diese Reflexionen beschreiben ungerichtetes Licht, das in der Umbebung vorhanden ist.

- Diffuse Reflexionen, deren Intensität vom Einfallswinkel des Lichtes abhängt. Diese Reflexionen beschreiben Licht, das von einer bestimmten Quelle ausgeht, aber an der Oberfläche diffus in alle Richtungen gleichmäßig gestreut wird.

- Spekuläre Reflexionen, deren Intensität vom Winkel zwischen dem ideal reflektierten Lichtstrahl (der sich aus Einfallswinkel des Lichtes und der Oberflächennormalen ergibt) und dem Sehstrahl des Beobachters ergibt. Diese Reflexionen beschreiben Licht, das von einer bestimmten Quelle ausgeht und an der Oberfläche gerichtet diffus gestreut wird, wobei die Intensität in Richtung der idealen Reflexion am höchsten ist. Der spekuläre Anteil des Lichtes wird üblicherweise dazu verwendet, Glanzeffekte und ähnliches zu erzeugen.

Für jeden dieser Reflexionstypen wird dabei ein eigener Farbwert für die Oberfläche verwendet, der mit der Intensität des entsprechend reflektierten Lichtes moduliert wird. Ebenso wird für jeden Reflexionstyp eine eigene Lichtfarbe verwaltet.
Das Modell kann zur Berechnung der durch die Beleuchtung beeinflussten Farbwerte beliebiger Oberflächenpunkte verwendet werden. Um nun eine Oberfläche einzufärben, muss für jeden Bildpunkt der Oberfläche ein Farbwert berechnet werden. Dazu existieren verschiedene Verfahren.
Beim Gouraud-Shading, das üblicherweise auf der Graphikhardware implementiert ist, wird das Modell für jeden Eckpunkt eines Polygons ausgewertet. Im Anschluss daran werden die Farbwerte der Eckpunkte linear über das Polygon interpoliert. Problematisch dabei ist, dass Glanzeffekte, die in der Mitte des Polygons auftreten würden, ignoriert werden, da die Mitte des Polygons nicht für die Berechnung herangezogen wird. Ebenso werden Glanzeffekte, die an einer der Ecken auftreten, über das ganze Polygon „verschmiert“.
Beim Phong-Shading hingegen werden die Normalen, die in den Ecken des Polygons gegeben sind, linear über dessen Oberfläche interpoliert, und das Modell für jeden Bildpunkt explizit ausgewertet. Phong-Shading ist aufwändiger und wird üblicherweise nicht in der Hardware implementiert.

Raytracing

Ein Visualisierungsverfahren, dass die oben beschriebenen Nachteile bei der Berechnung der durch Beleuchtung und Lichtausbreitung bedingten Effekte nicht hat, ist das Raytracing (vgl. [FDFHP94]).
Die grundlegende Idee beim Raytracing ist, die Ausbreitung einzelner Lichtstrahlen zu simulieren. Dabei wird jedoch nicht von den Lichtquellen ausgegangen, sondern statt dessen die Ausbreitung vom Auge des (virtuellen) Betrachters aus rückwärts verfolgt.
Für jeden zu erzeugenden Bildpunkt wird dabei ein sog. „Sehstrahl“ in die Szene geschickt. Trifft dieser Strahl innerhalb der Szene auf ein oder mehrere Objekte, so wird der Farbwert für den Auftreffpunkt berechnet, der dem Betrachter am

nächsten liegt. In diese Berechnung können verschiedene Parameter wie die Beleuchtung der Szene und die Materialeigenschaften des Objektes eingehen.
Im Folgenden soll kurz umrissen werden, wie die Beleuchtungsrechnung durchgeführt wird und wie Reflexionen realisiert werden.
Zur Durchführung der Beleuchtungsrechnung wird zunächst untersucht, von welchen Lichtquellen der betrachtete Punkt beleuchtet wird. Dazu wird für jeden Strahl vom Auftreffpunkt zu einer der Lichtqullen untersucht, ob dieser Strahl ein Objekt schneidet. Schneidet der Strahl kein Objekt, so geht die jeweilige Lichtquelle in die Beleuchtungsrechnung ein. Üblicherweise wird danach mit Hilfe des Phong-Models und der Materialbeschreibung des Objektes ein Farbwert berechnet. Da nur diejenigen Lichtquellen in die Berechnung eingehen, die nicht durch Objekte verdeckt sind, werden so implizit Schattenrisse erzeugt.
Haben Objekte reflektierende Oberflächen, so wird vom Auftreffpunkt eines Strahls ein neuer Strahl ausgerechnet, dessen Richtung sich durch Spiegeln an der Oberfläche ergibt. Der Farbwert dieses neuen Sehstrahls wird dann mit dem Farbwert für den betrachteten Punkt überlagert. Auf diese Weise kann ein Strahl beliebig oft innerhalb der Szene reflektiert werden. Üblicherweise wird hierbei eine maximale Rekusionstiefe vorgegeben oder die Intensität des reflektierten Strahls als Abbruchkriterium herangezogen.
Der Berechnungsaufwand beim Raytracing wird von der Anzahl der Objekte in der Szene, der Materialbeschaffenheit der einzelnen Objekte, der Anzahl der Lichtquellen und der Anzahl der zu berechnenden Bildpunkte beeinflusst. Der Einfluss der ersten drei Faktoren hängt dabei stark von den verwendeten Algorithmen ab. Die Anzahl der zu berechnenden Bildpunkte hingegen geht unabhängig vom sonstigen Verfahren linear in den Berechnungsaufwand ein.
Raytracing ist sehr aufwändig. Zur Zeit ist es nicht möglich, auf einem einzelnen PC Bilder in Echtzeit zu generieren. Es gibt jedoch Bestrebungen, einen Echtzeitbetrieb mit Hilfe von PC-Clustern (vgl. [WBS02]) oder spezieller Hardware (vgl. [SWS02]) zu ermöglichen.
Ein frei verfügbares Raytracing-Programm ist POV-Ray ([PovDoc]). POV-Ray wandelt eine als Klartextdatei bereitgestellte Szenenbeschreibung mittels Raytracing in eine Bilddatei um (vgl. Abschnitt 7.3).

3.7.2 Programmierschnittstellen zur Abstraktion von der Graphikhardware

Zur Programmierung von 3D-Graphik wurden im Laufe der Zeit verschiedene Programmierschnittstellen und -bibliotheken entwickelt. Dabei spielt neben dem Bereitstellen möglichst effizienter Visualisierungsverfahren auch die Abstraktion von der verwendeten Graphikhardware eine wichtige Rolle.

Derzeit sind vor allem die Schnittstellen *OpenGL* und *Direct3D* sehr weit verbreitet und werden im Folgenden kurz vorgestellt.

OpenGL

OpenGL (vgl. [WNDS99] und [Ber03]) ist eine Programmierschnittstelle, die sich aus der proprietären Schnittstelle IrisGL von Silicon Graphics entwickelt hat. Der OpenGL-Standard wird von einer unabhängigen Arbeitsgruppe, dem ARB (Architecture Review Board), in der verschiedene Hersteller von Hard- und Software vertreten sind, gepflegt und genormt. Die Schnittstelle steht auf verschiedenen Plattformen (unter anderem Linux, Windows, MacOS und diverse UNIXe) zur Verfügung.

Die von OpenGL bereitgestellte Schnittstelle ist prozedural. OpenGL verwaltet intern den sog. „Rendering-Zustand“, der vorgibt, wie einzelne graphische Primitive dargestellt werden können. Dieser Zustand beschreibt z. B. die affine und perspektivische Transformation, der die Objekte vor dem Zeichnen unterzogen werden, aber auch die Farbe und Lichtquelle(n), mit denen das jeweilige Objekt dargestellt werden soll. Der Rendering-Zustand wird über eine Menge von Funktionen beeinflusst und wird von OpenGL selbst nicht verändert.

Soll eine Graphik erzeugt werden, muss zunächst der Rendering-Zustand gesetzt werden. Im Anschluss daran können über verschiedene Funktionen graphische Primitive wie Punkte, Linien und Polygone erzeugt werden. Es ist dabei möglich, den Rendering-Zustand für jedes Primitiv zu ändern.

OpenGL dient ausschließlich der plattformunabhängigen Visualisierung der Graphik. Es stellt keine weitergehenden Funktionen bereit, mit denen sich komplexe Szenen verwalten lassen. Stattdessen erfolgen alle Operation direkt auf den graphischen Primitiven. Die einzige Möglichkeit, komplexere graphische Objekte direkt mit OpenGL zu verwalten, sind die sog. „Display-Lists“, mit denen es möglich ist, eine fest zusammenhängende Folge graphischer Primitive zu vereinigen und gemeinsam zu manipulieren.

Sollen komplexere Szenen verwaltet werden, müssen dazu weitere Bibliotheken herangezogen werden (vgl. Abschnitt 3.7.3).

Soll eine Graphikhardware über OpenGL angesprochen werden, muss der Hersteller einen Treiber bereitstellen, der die OpenGL-Schnittstelle implementiert. Zum Einbinden herstellerspezifischer Erweiterungen stellt OpenGL eine spezielle Schnittstelle bereit. Häufig werden solche Erweiterungen vom ARB in spätere Versionen des Standards aufgenommen.

Direct3D

Direct3D (vgl. [Ber03]) ist eine objektorientierte Programmierschnittstelle, die eng mit dem COM (Component Object Model) von Microsoft verknüpft ist. Direct3D ist nur auf den 32-Bit Versionen von Windows verfügbar und wird von Microsoft gepflegt und genormt.
Direct3D ist eine mächtige und vielseitige Schnittstelle, die neben der reinen Darstellung dreidimensionaler Graphik auch Methoden zur Verwaltung der Szene bereitstellt.
Da Microsoft den Standard alleine verwaltet, werden neue Trends im Graphikbereich schneller integriert, als dies bei OpenGL geschehen kann.
Aufgrund der Bindung an die Windows-Plattform spielt Direct3D für die vorliegende Arbeit keine Rolle.

3.7.3 Bibliotheken zur Szenenverwaltung

OpenGL stellt lediglich eine Schnittstelle zum schnellen Zeichnen einzelner Polygone oder Polygonmengen bereit. Es verarbeitet keinerlei Informationen, in welchem Zusammenhang diese Polygone stehen.
Daraus ergeben sich verschiedene Probleme. Zum einen muss das Anwendungsprogramm selbst für die Verwaltung der Szene sorgen, zum anderen ist es für graphische Effekte wie Transparenz oder Schatten notwendig, die Szene als Ganzes zu betrachten.
An dieser Stelle werden Programmbibliotheken eingesetzt, die die Verwaltung und Darstellung der Szene übernehmen. Eine übliche Semantik ist die des „Szenengraphen“.
In einem Szenengraphen wird die Gesamtszene als eine Menge verknüpfter Knoten eines üblicherweise gerichteten Graphen betrachtet. Die Blätter dieses Graphen enthalten die Geometriedaten der darzustellenden Szene. Die inneren Knoten beschreiben alle anderen Daten über die Szene, wie z. B. relative Transformationen zum übergeordneten Knoten, Informationen über Lichtquellen, Materialeigenschaften usw.
Um eine Szene darzustellen, wird der Szenengraph beginnend bei seinem Wurzelknoten mittels Tiefensuche traversiert. Dabei werden die Geometriedaten der Blätter allen Transformationen und sonstigen Ausgabeanweisungen unterworfen, die sich in den Knoten auf dem Pfad von der Wurzel zum jeweiligen Blatt befinden.
Werden besondere Darstellungstechniken eingesetzt, muss der Graph ggf. mehrfach traversiert werden. Sollen z. B. transparente Objekte dargestellt werden, so werden im ersten Durchgang nur die nicht-transparenten Objekte gezeichnet. Im zweiten Durchgang werden die transparenten Objekte in der richtigen Reihenfol-

ge (die sich aus der Lage der Objekte zum Betrachter ergibt) über die anderen Objekte gezeichnet.
Beispiele für solche Bibliotheken sind:

- Maverik
- OpenSG
- Sigma

Im Abschnitt 7.1 werden diese Bibliotheken ausführlicher diskutiert.

3.8 Erzeugen von Filmdateien mit `transcode`

Ein universell einsetzbares Werkzeug zur Bearbeitung von Filmdateien auf Linux-Rechnern ist das Kommandozeilenprogramm `transcode` (vgl. [Ost03]). Das Programm ist in der Lage, Filmdateien sehr vieler Formate zu öffnen, zu bearbeiten und anschließend als neue Filmdatei im selben oder einem anderen Format abzuspeichern.
Welche Dateiformate `transcode` konkret unterstützt, ist vom System abhängig, auf dem es übersetzt und installiert wird. Zum Übersetzungszeitpunkt werden alle verfügbaren Programmbibliotheken für Audio- und Videodateien eingebunden. Dabei wird für jedes verfügbare Dateiformat ein Ein- und ein Ausgabefilter generiert. Zur Laufzeit wird über die Kommandozeile die entsprechenden Filter ausgewählt.
Typische Bearbeitungsmöglichkeiten, die `transcode` bereitstellt, sind unter anderem

- Ändern der Bildhöhe und -breite
- Ändern der Bildrate
- Extraktion von Audiodaten
- Zerlegen von Filmen in Einzelbilder
- Montieren von Einzelbildern zu einem Film

Der letzte aufgeführte Punkt ist dabei von besonderem Interesse für die vorliegende Arbeit. Um aus einer Folge von Einzelbildern eine Filmdatei zu generieren, stellt `transcode` einen speziellen Filter bereit, der anstelle einer Filmdatei eine Liste von Bilddateien einliest. Die in dieser Liste angegebenen Bilddateien werden von `transcode` geladen und mit Hilfe des gewählten Ausgabefilters als Filmdatei gespeichert.

Kapitel 4

Architektur der Software

In diesem Abschnitt wird die Softwarearchitektur entwickelt, die in der Bibliothek `roboViewLib` verwendet wird. Dazu wird zunächst das bekannte *Model-View-Controller*-Entwurfsmuster (vgl. [GHJV95]) an die speziellen Gegebenheiten im parallelen Betrieb mit mehreren Threads angepasst. Im Anschluss daran werden die Details der Implementierung vorgestellt.

4.1 Vorüberlegung

Die entwickelte Bibliothek `roboViewLib` soll zu verschiedenen Zwecken eingesetzt werden. Die beiden Hauptanwendungen sind das Erzeugen von Filmen und das Visualisieren von Mehrkörpersystemen und deren Umgebung in Echtzeit. Für diese beiden Fälle können die Steuerdaten für die Animation aus verschiedenen Quellen, z. B. Dateien oder in Echtzeit erzeugte Daten, kommen. Hieraus ergibt sich die Notwendigkeit, die Eingabe der Steuerdaten von der sonstigen Software zu entkoppeln, um ein leichtes Austauschen des Eingabemoduls ohne Beeinflussung der anderen Module zu ermöglichen.
Speziell bei der Visualisierung in Echtzeit ist es darüber hinaus wünschenswert, zum einen mehrere parallele Ansichten zu unterstützen[1] und zum anderen die Darstellung möglichst zeitnah zu den generierten Steuerdaten zu erzeugen. Je nach Aufwand der erzeugten Szene kann dies erfordern, dass einzelne Bilder übersprungen werden.
Um die Darstellung mehrerer paralleler Ansichten zu beschleunigen, sollten Berechnungen, die von allen Ansichten benötigt werden, möglichst zentral und pro Steuerdatensatz nur einmal durchgeführt werden. So können z. B. die Berechnung der zeitabhängigen Transformationen und die Kollisionserkennung einmal zentral ausgeführt werden, um die Ergebnisse allen Ansichten zur Verfügung zu stellen.

[1] Um eine Szene aus mehreren Blickwinkeln zu beobachten.

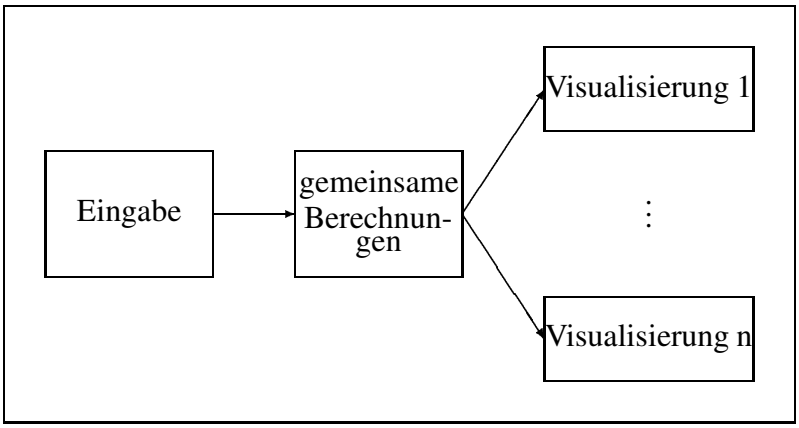

Abbildung 4.1: Grundkomponenten der Software

Um möglichst unabhängig von der Geschwindigkeit der Graphik-Generierung zu sein, ist es sinnvoll, die einzelnen Darstellungen wie auch die Dateneingabe und Vorverarbeitung in getrennten Threads durchzuführen. Hierdurch wird verhindert, dass einzelne, zeitaufwändige Berechnungen die gesamte Anwendung am Fortschreiten hindern.
Die zentralen Forderungen an die Architektur der Software sind also:

- Trennung von Eingabe, Datenverarbeitung und Visualisierung, um eine Anpassung an verschieden Anwendungsfälle zu erleichtern.
- Unterstützung von Multithreading, um den Anforderungen einer Echtzeitdarstellung gerecht zu werden.

Als grundlegendes Konzept bietet sich hierfür das Model-View-Controller-Entwurfsmuster an, das allerdings an die Anforderungen angepasst werden muss, die sich aus dem verwendeten Multithreading ergeben.

4.2 Model-View-Controller-Konzept

Das *Model-View-Controller*-Konzept (MVC-Konzept) ist ein gängiges Entwurfsmuster für interaktive Anwendungen und wird z. B. in Smalltalk zur Programmierung von Benutzungsoberflächen (vgl. [GHJV95]). Bei diesem Konzept besteht die Anwendung aus drei Klassen von Objekten: *Model*, *View* und *Controller*.
In Objekten der Klasse *Model* werden dabei die Daten der Anwendung verwaltet und notwendige Berechnungen durchgeführt. Über Objekte der Klasse *Controller* ist es möglich, die Daten zu manipulieren. Mit Objekten der Klasse *View* schließlich werden Daten und Ergebnisse dargestellt.

Dabei kann ein *Model*[2] mit mehreren *Controllern* und *Views* zusammenarbeiten: Das *Model* verwaltet üblicherweise eine Liste der *Views*, die seine Daten und Ergebnisse darstellen. Sobald die Daten des *Models* von einem *Controller* aus geändert werden, führt das *Model* die notwendigen Berechnungen durch und benachrichtigt im Anschluss daran die *Views*. Üblicherweise lesen die *Views* dann die von ihnen darzustellenden Informationen aus dem *Model* aus.

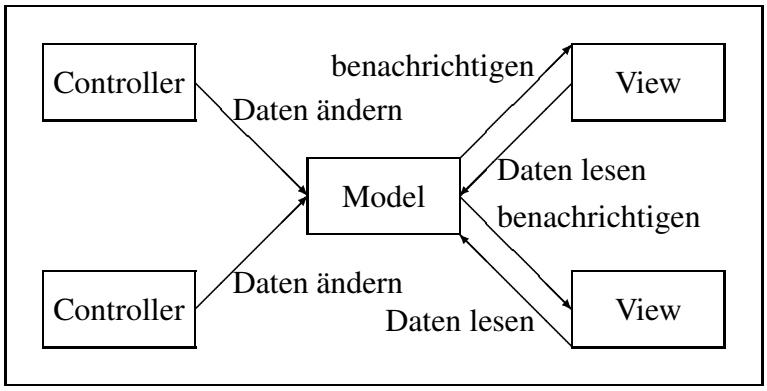

Abbildung 4.2: Interaktion im MVC-Entwurfsmuster

4.3 MVC und Multithreading

Soll das MVC-Konzept in einem Programm mit mehreren parallelen Threads eingesetzt werden, entsteht an verschiedenen Punkten die Notwendigkeit der Synchronisation, um die Konsistenz der Daten zu wahren:

- Greifen mehrere *Controller* parallel auf das *Model* zu, muss sichergestellt werden, dass die *Controller* sich nicht gegenseitig stören.
- Führt das *Model* Berechnungen an den Daten durch, dürfen sich diese nicht während der Berechnungen ändern.
- Werden Daten von einem *View* angezeigt, dürfen sich diese nicht ändern, während der *View* die Darstellung generiert.

Ein einfacher Ansatz ist, die Zugriffe von *Controllern* und *Views* so zu synchronisieren, dass immer nur ein *Controller* schreibend auf die Daten zugreifen kann und dass sich lesende Zugriffe der *Views* und schreibende Zugriffe der *Controller* gegenseitig ausschließen. Weiterhin dürfen weder *Controller* noch *Views* auf

[2]Im Folgenden bezeichnen *Model*, *View* und *Controller* jeweils Objekte der entsprechenden Klassen, um umständliche Formulierungen zu vermeiden.

die Daten zugreifen, solange das *Model* Berechnungen ausführt. Derartige Probleme werden als Leser-Schreiber-Probleme bezeichnet. Es existieren verschiedene Lösungsansätze (vgl. [Tan01]).
Der einfache Ansatz über die Synchronisation der Lese- und Schreibzugriffe auf das *Model* hat den Nachteil, dass sich *Controller* und *Views* gegenseitig bremsen. Solange die Zugriffe durch *Controller* und *Views* verhältnismäßig kurz sind, entstehen keine Probleme. Wenn jedoch ein *View* lange braucht, um die Darstellung zu generieren, können währenddessen die *Controller* nicht arbeiten, da sie blockiert werden. Dies ist in zeitabhängigen Anwendungen nicht wünschenswert, da ein langsamer *View* so die ganze Anwendung blockieren kann.
Um eine solche Blockierung zu verhindern, wird das MVC-Konzept erweitert. Die Anwendungsdaten werden nicht mehr im *Model*, sondern in einem speziellen Datenobjekt gespeichert. *Controller* können über das *Model* Zugriff auf die Daten nehmen. Solange ein *Controller* die Daten manipuliert, sind Zugriffe für andere *Controller* darauf gesperrt. Sobald der *Controller* fertig ist, verarbeitet das *Model* die Daten, signalisiert den *Views*, dass sich die Daten geändert haben, und übergibt ihnen einen Zeiger auf das Datenobjekt. An diesem Datenobjekt darf nun nichts mehr verändert werden, da es jederzeit von den *Views* asynchron dargestellt werden kann. Um den *Controllern* weitere Manipulationen zu ermöglichen, wird eine Kopie des Datenobjekts erstellt, auf die die *Controller* zugreifen können. Sobald diese Kopie von einem der *Controller* manipuliert wird, wiederholt sich der beschriebene Vorgang.

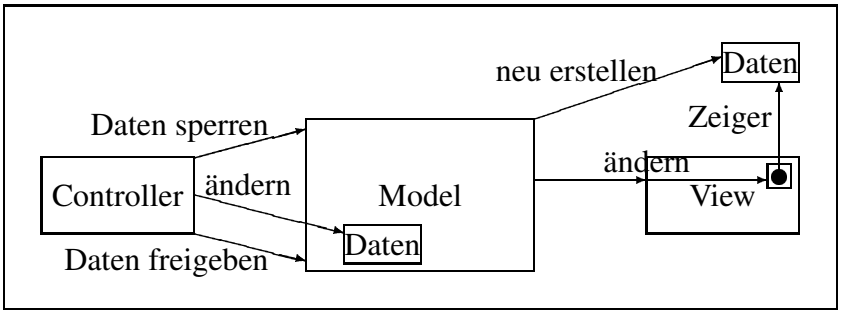

Abbildung 4.3: Erweiterte MVC-Interaktion

Der Vorteil bei diesem Verfahren ist, dass die *Controller* weiterarbeiten können, sobald die aktuellen Daten an die *Views* weitergeleitet wurden. Hierdurch können regelmäßig wiederkehrende Aufgaben kontinuierlich erledigt werden, ohne dass ggf. langwierige Berechnungen in den *Views* das Fortschreiten dieser Aufgaben behindern.
Der Nachteil bei der beschriebenen Vorgehensweise besteht darin, dass, sobald Änderungen der Daten an die *Views* weitergegeben werden, eine Kopie des gesamten Datensatzes erstellt werden muss. Dieses Vorgehen ist für die gewünschte

Anwendung akzeptabel, da die Datenobjekte eine begrenzte Größe haben, die sich aus der Anzahl der Objekte in der Szene ergibt.

4.4 Konkrete Implementierung

In den folgenden Abschnitten werden wichtige Fragen der Implementierung erörtert.

4.4.1 Auswahl der Programmiersprache

Zur Implementierung wird die Programmiersprache C++ (vgl. [Str92]) verwendet. Diese Sprache hat verschiedene Vorteile für die zu lösende Aufgabe:

- C++ ist objektorientiert: Die objektorientierte Programmierung (OOP) erlaubt es, Daten und dazugehörige Funktionen, die im OOP-Kontext als Methoden bezeichnet werden, in Klassen zusammenzufassen. Diese Klassen sind Vorlagen, von denen zur Laufzeit tatsächliche Objekte generiert werden. Zum Behandeln von Klassen gibt es zwei wesentliche Mechanismen: Durch Vererbung ist es möglich, von einer Basisklasse spezielle Unterklassen zu generieren, die die Funktionalität der Basisklasse übernehmen und um spezielle Fähigkeiten erweitern. Durch Polymorphie (Vielgestaltigkeit) ist es möglich, verschiedene Klassen mit gleicher Schnittstelle, aber unterschiedlichen internen Eigenschaften gegeneinander auszutauschen, um so transparent die Funktionalität eines Programmes durch Austauschen einzelner Klassen zu modifizieren.

- C++ wird zu Maschinensprache übersetzt: Daraus ergibt sich eine erhebliche Beschleunigung der Ausführungsgeschwindigkeit gegenüber Sprachen wie Java (vgl. [Fla98]) oder Smalltalk (vgl. [Tuc03]), die zu einem Zwischencode übersetzt werden, der auf jeder Plattform von einem speziellen Interpreter ausgeführt wird. Der Nachteil beim Erzeugen von Maschinenprogrammen ist, dass solche Programme für jede Zielplattform getrennt übersetzt werden müssen, wobei häufig Anpassungen an die jeweilige Plattform vorgenommen werden müssen.

4.4.2 Verkapselung des Multithreading

Um den Programmcode der Software möglichst leicht an verschiedene Plattformen anpassen zu können, wird der stark plattformabhägige Code zur Verwendung paralleler Threads in speziellen Objekten verkapselt. Gleichzeitig werden dabei

die prozeduralen Programmierschnittstellen der Betriebssysteme durch objektorientierte ersetzt.

Zur Verwendung von Multithreading stehen drei Klassen zur Verfügung: `CThread`, `CSemaphore` und `CMutex`.

Die Klasse `CThread` verkapselt dabei einen einzelnen Thread. Mit jedem Objekt wird ein Thread verwaltet, als Operationen stehen „Starten“, „Abbrechen“ und „Warten auf das Ende des Threads“ zur Verfügung. Darüberhinaus existiert eine Methode, die es dem Thread gestattet, eine vorgegebene Zeit zu warten.

Eine Threadklasse, die eine konkrete Aufgabe ausführt, wird erzeugt, indem eine Unterklasse von `CThread` gebildet wird, in der die virtuelle Methode `threadFunction` überschrieben wird. Um die Aufgabe dann in einem eigenständigen Thread auszuführen, muss ein Objekt der Klasse erzeugt und die Ausführung gestartet werden.

Zur Synchronisation mehrerer Threads stehen die Klassen `CSemaphore` und `CMutex` bereit. Dabei verkapselt ein Objekt jeweils ein Semaphor oder Mutex, wie sie von den jeweiligen Betriebssystemen oder Laufzeitumgebungen bereitgestellt werden.

Ein Objekt der Klasse `CMutex` stellt die Methoden `lock` und `unlock` bereit. Mit diesen Methoden können Synchronisationsaufgaben wie der wechselseitige Ausschluss beim Zugriff auf kritische Abschnitte (vgl. [Tan01]) gelöst werden. Dazu kann ein Thread das Mutex-Objekt mittels der Methode `lock` sperren und mittels `unlock` wieder freigeben. Versucht ein zweiter Thread für ein bereits gesperrtes Objekt `lock` aufzurufen, wird er solange blockiert, bis für das Objekt aus einem anderen Thread heraus `unlock` aufgerufen wurde.

Ein Objekt der Klasse `CSemaphor` wird beim Erzeugen mit einem festen Zählerstand initialisiert. Mittels der Methoden `up` und `down` kann dieser Zähler inkrementiert bzw. dekrementiert werden. Wird versucht, einen Zähler, der bereits auf dem Stand 0 ist, weiter zu dekrementieren, wird der aufrufende Thread solange blockiert, bis ein anderer Thread den Zähler einmal inkrementiert.

Dieser Mechanismus erlaubt das Lösen komplizierterer Synchronisationsaufgaben, wie z. B. Erzeuger-Verbraucher-Probleme (vgl. [Tan01]). Generell können Semaphore aber auch für alle Aufgaben verwendet werden, für die Mutexe verwendet werden. Da jedoch manche Thread-Pakete spezielle Mutex-Implementierungen bereitstellen, die weniger Systemressourcen belegen, ist es sinnvoll, beide Synchronisationsobjekte bereitzustellen, so dass es möglich ist, gezielt Mutexe zu verwenden, wo es möglich ist.

Für jede der Zielplattformen müssen die drei Klassen neu implementiert werden, wobei auf die jeweiligen Systemfunktionen der Plattform zurückgegriffen wird. Gleichzeitig steht aber für das eigentliche Programm eine einheitliche Schnittstelle bereit, die systemunabhängig ist.

4.4.3 Aspekte der Speicherverwaltung

In der zu entwickelnden Software werden Datenobjekte zur Laufzeit dynamisch angelegt. Dazu wird dynamisch Speicher vom System angefordert. Der Speicher nicht mehr benötigter Datenobjekte wird von C++ nicht automatisch freigegeben wie etwa unter JAVA über den Mechanismus der „Garbage-Collection“ (vgl. [Fla98]). Stattdessen müssen die Objekte explizit zerstört werden, um den Speicher wieder freizugeben. Dazu stellen Objekte spezielle Methoden, sog. Destruktoren bereit.

Wird ein Datenobjekt von mehreren Threads verwendet, darf der durch das Objekt belegte Speicher erst dann freigegeben werden, wenn der letzte dieser Threads das Objekt nicht mehr benötigt. Damit verbietet es sich, dass einer der Threads den Destruktor aufruft, da ansonsten das Objekt freigegeben und der Speicher ggf. anders wiederverwendet würde, so dass die Daten für die anderen Threads zerstört sind.

Um derartige Probleme zu verhindern, wird für jedes Datenobjekt ein Referenzzähler verwendet. Dieser hat nach dem Erstellen des Objektes den Wert 1, und kann zur Laufzeit inkrementiert und dekrementiert werden. Sinkt der Wert auf 0, wird der Destruktor für das Objekt aufgerufen und damit der belegte Speicher freigegeben.

Wird nun ein Objekt von mehreren Threads verwendet, muss der Referenzzähler erhöht werden, sobald das Objekt an einen weiteren Thread übergeben wird. Jeder Thread, der das Objekt nicht mehr benötigt, dekrementiert den Referenzzähler, so dass das Objekt zerstört wird, sobald es vom letzten Thread freigegeben wird.

Dieses Verhalten wird von der Klasse `CRefCounter` implementiert. Objekte dieser Klasse stellen die Methoden `incrementRefCount` und `decrementRefCount` bereit. Die Methoden inkrementieren und dekrementieren den Referenzzähler. Dabei wird über ein Mutex-Objekt der wechselseitige Ausschluss beim Zugriff auf den Zähler sichergestellt. Wird der Zähler auf den Wert 0 gebracht, wird der Destruktor des Objektes aufgerufen.

Konkrete Datenobjekte der Anwendung können von `CRefCounter` abgeleitet werden, um den beschriebenen Mechanismus zu erben. In der Basisklasse wird der Destruktor als virtuell deklariert. Dadurch wird automatisch der konkrete Destruktor der abgeleiteten Klasse aufgerufen.

Beim Ableiten sollte sichergestellt werden, dass der Destruktor, wie in der Basisklasse auch, `protected` (und nicht `public`) ist, so dass sichergestellt bleibt, dass der Destruktor nirgends explizit aufgerufen werden kann.

4.4.4 Implementierung des MVC-Frameworks

In den Klassen `CModel`, `CView` und `CController` wird der gesamte Interaktionsmechanismus des MVC-Frameworks implementiert. Zusätzlich werden Objekte vom Typ `CDataset` verwendet, um die zwischen *Model*, *View* und *Controller* ausgetauschten Daten aufzunehmen. `CDataset` ist von `CRefCounter` abgeleitet, so dass es den oben beschriebene Mechanismus zum Zählen von Referenzen bereitstellt.
Diese Klassen stellen Basisklassen dar, von denen Unterklassen für die konkrete Anwendung abgeleitet werden können. Dazu stehen virtuelle Methoden bereit, die automatisch durch den Interaktionsmechanismus aufgerufen werden.
Im Folgenden wird beschrieben, wie die einzelnen Klassen des Frameworks interagieren. Dazu wird die Funktion der einzelnen Klassen beschrieben
Die Klassen `CModel` stellt den Kern des Frameworks dar. Objekte dieser Klasse speichern zu jedem Zeitpunkt ein Objekt der Klasse `CDataset`, in dem der aktuelle Stand der Daten gespeichert ist. Sollen diese Daten verändert werden, muss sich ein *Controller* Zugriff auf das Datenobjekt verschaffen. Dazu dient die Methode `mvcGetAndLockCurrentDataset`. Wird sie aufgerufen, erhält der *Controller* einen Zeiger auf das Datenobjekt. Dabei wird über ein Mutex-Objekt sichergestellt, dass immer nur ein *Controller* Zugriff hat. Ist der *Controller* fertig, muss er das Datenobjekt wieder freigeben. Dazu dienen die Methoden `mvcReleaseCurrentDataset` und `mvcReleaseCurrentDatasetAndNotifyChanges`. Beide geben die Sperre des Datenobjektes frei, so dass andere *Controller* Zugriff auf das Datenobjekt erhalten können [3]. Die zweite Methode teilt `CModel` zusätzlich mit, dass Daten verändert wurden.
Wurden Daten verändert, wird eine Kopie des aktuellen Datensatzes erstellt, die ab dann asynchron durch die *Controller* manipuliert werden kann. Der ursprüngliche veränderte Datensatz wird zunächst über die Methode `virtmvcDatasetChanged` verarbeitet. Diese virtuelle Methode ist der Anknüpfungspunkt, an dem abgeleitete Klassen eine eigene Funktionalität einbringen können. Im Anschluss daran wird die Kopie jedem beim *Model* registrierten *View* zur Verfügung gestellt. Dabei wird die Methode `mvcDataChanged` der Reihe nach für jeden *View* aufgerufen; will ein *View* das dabei übergebene Datenobjekt über diese Methode hinaus verwenden, muss er einen Zeiger darauf speichern und den Referenzzähler um eins erhöhen. Zum Abschluss wird der Referenzzähler des Datensatzes um eins dekrementiert, `CModel` darf ab jetzt auf diesen Datensatz nicht mehr zugreifen, da er nun jederzeit durch die *Views* zerstört werden kann, indem diese den Referenzzähler weiter dekrementieren.
Beim Erstellen eines neuen Objektes einer von `CModel` abgeleiteten Klasse muss

[3] Der *Controller* darf im Weiteren den Zeiger nicht mehr verwenden, da nun andere *Controller* auf das Datenobjekt zugreifen können oder das Datenobjekt zwischenzeitlich gelöscht wurde.

ein initialer Datensatz erzeugt werden. Da `CModel` die zu verwendende, von `CDataset` abgeleitete Klasse nicht kennt, wird dieser Datensatz in der virtuellen Methode `virtmvcCreateCurrentDataset` erzeugt. Diese Methode *muss* beim Ableiten von `CModel` überschrieben werden.
Zum Registrieren und Entfernen von *View* stehen die Methoden `mvcAddView` und `mvcRemoveView` bereit. Wird ein neuer *View* registriert, wird ihm über den oben beschriebenen Benachrichtungsmechanismus ein Zeiger auf den aktuellen Datensatz übergeben.
Die Klasse `CView` stellt eine virtuelle Methode `mvcDataChanged` bereit, mit der das *Model* dem *View* einen veränderten Datensatz übergeben kann. In einer abgeleiteten Klasse muss diese Methode überschrieben werden, um zu definieren, wie die Daten dargestellt werden sollen.
Die Klasse `CController` stellt die Methode `mvcSetModel` bereit, um den *Controller* mit dem jeweiligen *Model* zu verbinden. Um den Zugriff auf den im *Model* gespeicherten Datensatz zu erhalten, stellt `CController` die Methode `getCurrentDataset` bereit. Diese liefert einen Zeiger zum exklusiven Zugriff auf den Datensatz. Nachdem der Datensatz durch den *Controller* manipuliert wurde, muss er über die Methoden `releaseCurrentDataset` oder `releaseCurrentDatasetAndNotify` wieder freigegeben werden. Die zweite dieser Methoden veranlasst dabei zusätzlich, dass das *Model* über Veränderungen an den Daten informiert wird.

4.4.5 Abbildung der Anwendung auf das Framework

Um mit den beschriebenen Basisklassen des MVC-Frameworks eine konkrete Anwendung zu erzeugen, müssen Unterklassen von `CModel`, `CView` und `CController` abgeleitet werden, in denen die gewünschte Funktionalität realisiert wird. Weiterhin muss eine Unterklasse von `CDataset` erzeugt werden, mit der die von der Anwendung erzeugten bzw. verarbeiteten Daten verwaltet werden können.
Für die oben beschriebene Anwendung zum Erzeugen von Animationen und Filmen wurden dabei folgenden Klassen erzeugt:
Von `CController` werden Klassen abgeleitet, die Steuerdaten für die Animation erzeugen. Möglichkeiten dafür sind z. B. Klassen, die Stellwerte aus Dateien einlesen und diese in regelmäßigen Zeitabständen weitergeben oder Klassen, die in Echtzeit Stellwerte generieren oder diese aus einem realen oder simulierten Prozess entnehmen.
Die Steuerdaten sowie die von der Anwendung daraus generierten Daten werden in Objekten der Klasse `CFrameData` abgelegt, die von `CDataset` abgeleitet ist.
Die von `CModel` abgeleitete Klasse `CApplicationModel` repräsentiert den Kern der Anwendung. Objekte dieser Klasse verarbeiten die erzeugten Daten und geben sie anschließend an die Ansichtsklassen weiter. Die Verarbeitung besteht aus der

Berechnung der Transformationen der Elemente der Szene und der (optionalen) Durchführung der Kollisionserkennung. Um hier flexibel verschiedene Verfahren einsetzen zu können, wurde eine Schnittstelle definiert, um die Algorithmen leicht austauschen zu können (vgl. Abschnitt 6). Es ist möglich, in `CApplicationModel` weitere Verarbeitungsschritte zu integrieren.
Zur Visualisierung der Daten werden Unterklassen von `CView` verwendet. Dabei wurde von `CView` die Unterklasse `CRoboView` abgeleitet. Diese Klasse bietet spezielle Methoden zum Setzen der Darstellungsoptionen und zum Steuern der Aufnahmefunktionen. Die davon abgeleiteten konkreten Ansichtsklassen werden in Abschnitt 7 ausführlich diskutiert.
Als Ergänzug zu `CRoboView` wurde die Klasse `CRoboViewOptions` definiert. In Objekten dieser Klasse wird gespeichert, welche der Elemente (z. B. Hüllkörper, Zusatzdaten usw.) der Szene dargestellt werden sollen, welche der definierten Kameras (vgl. Abschnitt 5.1.1) verwendet werden soll und welcher Ausschnitt des Fensters zum Zeichnen verwendet werden soll.

4.4.6 Mathematische Datentypen

Innerhalb der Anwendung müssen Berechnungen mit Skalaren, Matrizen und Vektoren durchgeführt werden.
Um sich nicht frühzeitig auf einen Datentyp festlegen zu müssen, wird der Datentyp `FPType` definiert und durchgängig anstelle der von C++ bereitgestellten Datentypen verwendet. In der vorliegenden Implementierung stellt `FPType` einen 32-Bit-Fließkommazahl vom C++-Typ `float` dar. Bei Bedarf lässt sich das gesamte Programm durch Ändern der Definition von `FPType` auf einen anderen Fließkommatype umstellen.
Zur Repräsentation von quadratischen Matrizen und Vektoren werden die Templates `Matrix` und `Vektor` verwendet. Beide sind in Dimension und Datentyp parametrisierbar. Für die benötigten Fälle, nämlich drei- und vierdimensionale Fließkommavektoren und entsprechende Matrizen, werden mit Hilfe dieser Templates die Datentypen `CMatrix3`, `CVector3`, `CMatrix4` und `CVector4` definiert.

Kapitel 5

Repräsentation der Szene

In diesem Abschnitt wird beschrieben, wie die darzustellende Szene und die Steuerdaten in der Software abgebildet werden.

5.1 Interne Repräsentation der Szene

Die darzustellende Szene besteht aus einem oder mehreren Robotern, einer Umgebung und ggf. Zusatzinformationen wie z. B. (Kraft-)Vektoren oder Schwerpunkt. Sowohl die Roboter als auch die Umgebung bestehen dabei aus einer Menge von dreidimensionalen Körpern, die relativ zueinander bewegt werden können und ihre Form nicht ändern. Roboter und Umgebung werden dabei gleichartig beschrieben. Aus Sicht der Software sind beide Mehrkörpersysteme. Für die Zusatzdaten wird eine andere Repräsentation gewählt, da sie zum einen aus ein- bzw. zweidimensionalen Objekten, nämlich Geraden und Polygonen, bestehen und zum anderen über Mengen von Eckpunkten beschrieben werden, so dass sie neben der Lage auch die Form ändern können.
Neben den sichtbaren Elementen enthält die Szene noch eine oder mehrere Kameras und Lichtquellen. Um diese leicht bewegen zu können, werden diese als Teil der Mehrkörpersysteme beschrieben und sind jeweils einem Körper verbunden.

5.1.1 Mehrkörpersysteme

Jedes Mehrkörpersystem wird als Baum von einzelnen Körpern betrachtet, die durch Gelenke miteinander verbunden sind. Jeder dieser Körper ist dabei fest mit einem Koordinatensystem verbunden. Die räumliche Beziehung des Koordinatensystems eines Körpers zu dem des übergeordneten Körpers wird dabei durch zwei Transformationen bestimmt:

Zunächst gibt eine konstante Transformation, die durch eine Matrix $T_{fest} \in \mathbb{R}^{4x4}$ beschrieben wird die Lage des aktuell betrachteten Gelenkes (oder der Gelenke) relativ zum Koordinatensystem des übergeordneten Körpers an. Auf diese Transformation folgt die variable Transformation $T_{gelenk}(p) \in \mathbb{R}^{4x4}$, die sich aus dem Gelenk (oder den Gelenken) ergibt, wobei $p \in \mathbb{R}^n$ der Vektor der Parameter des Gelenkes und n der Freiheitsgrad des Gelenkes ist.
Für die gesamte relative Transformation aus dem Koordinatensystem des übergeordneten Körpers in das Koordinatensystem des Körpers gilt somit

$$T_{relativ} = T_{fest} T_{gelenk}(p).$$

Damit lässt sich die absolute Transformation des Körpers aus der absoluten Transfomation des übergeordneten Körpers berechnen:

$$T_{absolut} = T_{absolut_{Eltern}} T_{fest} T_{gelenk}(p).$$

Somit ist es möglich, von einem beliebigen Körper aus den Pfad zur Wurzel rekursiv zu verfolgen, um die absolute Lage und Orientierung des Körpers zu bestimmen.
Da für manche Berechnungen, z. B. die der relative Lage der Hüllkörper bei der Kollisionserkennung, die Inverse T_{fest}^{-1} der konstanten Transformation T_{fest} benötigt wird, wird diese gespeichert, um erhöhten Aufwand durch wiederholtes Invertieren einzusparen.
Neben der Beschreibung der relativen räumlichen Lage des Körpers sind mit jedem der Körper weitere optionale Informationen verbunden: die Beschreibung der Form und Farbe des Körpers sowie die Form des Hüllkörpers.
Jeder Körper wird durch ein Objekt der Klasse `CSceneElement` beschrieben. Diese Klasse stellt neben Datenelementen für die oben beschriebenen Informationen Methoden zum Aufbauen und Verwalten des Baumes bereit.

Transformationen und Transformationbeschreibungen

Feste Transformationen werden durch 4x4-Matrizen beschrieben. Dies geschieht gekapselt in der Klasse `CTransformation`. Die Klasse stellt verschiedene statische Methoden bereit, mit denen es möglich ist, Objekte für Standardtransformationen (Rotationen um die Hauptachsen, Translationen, ...) zu generieren.
Zur Beschreibung variabler Transformationen werden Unterklassen der Klasse `CTransformationeDescription` verwendet. Jede dieser Unterklassen beschreibt eine bestimmte Art von Transformation, die jeweils von einem oder mehreren Parametern abhängen. Über die virtuellen Methoden `generateTransformation` und `generateInverseTransformation` wird dabei die von den jeweiligen Parametern abhängige Transformationsmatrix bzw. deren Inverse erzeugt. Den Methoden wird jeweils ein Feld mit den Parametern übergeben.

Die Anzahl der Parameter einer Transformationsbeschreibung kann über die Methode `getDofCount` erfragt werden.
Theroretisch könnte auf die Methode `generateInverseTransformation` verzichtet werden, indem statt dessen die mit `generateTransformation` erzeugte Matrix invertiert wird. Das Invertieren einer Matrix erfordert jedoch einen hohen Rechenaufwand[1], während das Erzeugen der Transformation aus dem setzen von drei (im Falle einer Translation) bzw. vier (im Falle einer Rotation) Werten in einer Einheitsmatrix erfordert.
Es ist möglich, mehrere Transformationsbeschreibungen zu verknüpfen. Intern wird dabei eine verkettete Liste der einzelnen Beschreibungen angelegt. Die Methode zum Erzeugen der Transformation wird dabei für das erste Element der Liste aufgerufen und bekommt die Parameter aller Teiltransformationen übergeben. Intern werden die Parameter an die virtuellen Methoden der einzelnen Teiltransformationen übergeben. Dazu wird jeder der virtuellen Methoden ein Zeiger auf den richtigen Bereich des Parameter-Feldes übergeben.
Um die Gesamtzahl der Parameter einer solchen zusammengesetzten Transformation zu bestimmen, wird die Methode `getListDofCount` der ersten Transformation aufgerufen. Diese bestimmt über rekursive Aufrufe von `getListDofCount` die Gesamtsumme der Parameter der Liste der Transformationsbeschreibungen.

Form

Zur Repräsentation der Form eines Körpers und dessen Hüllkörper werden Objekte der Klasse `CShape` verwendet. Die Unterklassen dieser Klasse beschreiben verschiedene geometrische Formen. Derzeit sind Quader, Ellipsoid, Zylinder und Kegel implementiert.
Jedes Objekt der Klasse `CShape` speichert neben der Form die Abmessung des geometrischen Objektes und dessen relative Lage in Bezug auf das Koordinatensystem des Körpers.
Die Abmessungen werden dabei als Länge, Höhe und Breite des umschriebenen Quaders abgelegt.
Die relative Lage wird durch eine Transformation beschrieben. Zusätzlich wird auch hier die Inverse abgespeichert, um diese für Berechnungen (z. B. zur Kollisionsbestimmung) ohne zusätzlichen Rechenaufwand verfügbar zu haben.
Der Mittelpunkt des umschriebenen Quaders liegt dabei im Ursprung des durch die Transformation beschriebenen Koordinatensystems. Die Seiten des Quaders sind parallel zu den Achsen des Koordinatensystems. Die einzelnen Formen werden wie folgt dargestellt:

[1] Der Aufwand hängt stark von der Beschaffenheit der Matrix ab. Im Falle einer Transformation, bestehend aus Rotation und Translation, reicht es aus, die Rotationsmatrix zu invertieren und anschliessend die umgekehrte Translation mit der neuen Rotation zu verknüpfen.

- Quader sind genau der beschriebene umhüllende Quader.
- Ellipsoide liegen so, dass ihre Hauptachsen auf den Koordinatenachsen liegen.
- Zylinder sind so orientiert, dass ihre Rotationsachse auf der x-Achse liegt.
- Kegel liegen ebenfalls so, dass ihre Rotationsachse auf der x-Achse liegt, die Spitze zeigt in positive Richtung.

Farbe und Materialeigenschaften

Die Gestaltung der Oberfläche der Körper wird durch ein Element vom Typ `CMaterial` beschrieben. Objekte dieser Klasse speichern Farbe und Transparenz der Objekte. Die Farbwerte werden als RGBA-Tupel in Objekten der Klasse `CColor` gespeichert. Darüber hinaus ist es möglich, anzugeben, ob der Körper mit einem Karomuster oder einer beliebigen Textur überzogen werden soll. Für Karomuster wird zusätzlich angegeben, wie viele Karos entlang der beiden Hauptachsen der Textur erzeugt werden sollen.
In einem weiteren Datenfeld wird festgehalten, ob der Körper eine eigene Materialbeschreibung hat oder die des übergeordneten Körpers verwendet werden soll.

Lichtquellen

Es ist möglich, mit jedem Körper des Mehrkörpersystems eineLichtquelle zu verbinden. Lichtquellen werden durch drei RGB-Tripel für ambient, diffus und spekulär reflektiertes Licht beschrieben. Derzeit werden nur Punktlichtquellen unterstützt. Punktlichtquellen befinden sich an einer Position im Raum, die durch den Ursprung des Koordinatensystem des Körpers definiert ist, und strahlen in alle Richtungen gleich stark ab. Die Daten der Lichtquelle werden in der Klasse `CLightDescription` zusammengefasst. Um weitere Typen von Lichtquellen, z. B. für gerichtetes Licht, zu unterstützen, muss diese Klasse erweitert werden.

Kameras

Mit jedem einzelnen Körper jedes Mehrkörpersystems kann jeweils eine Kamera verbunden sein. Die Kamera ist fest mit dem Urprung des Koordinatensystems des Körpers verbunden. Für die Blickrichtung gilt die in OpenSG und OpenGL verwendete Konvention: Die Kamera ist entlang der negativen z-Achse ausgerichtet, die positive x-Achse zeigt im Bild der Kamera nach rechts, die positive y-Achse nach oben. Kameras werden durch Objekte der Klasse `CCameraDescription` beschrieben. Derzeit werden nur Kamers mit perspektivischer Projektion dargestellt,

sollen weitere Typen hinzugefügt werden, muss die Klasse entsprechend erweitert werden.

Zusammenfassung `CSceneElement`

In jedem Objekt vom Type `CSceneElement` stehen folgende Datenelemente zur Verwaltung der Baumstruktur der Szene bereit:

Name	Inhalt
`seParent`	Zeiger auf das übergeordnete Element in der Szene
`seNextSibling`	Zeiger auf den nächsten Bruderknoten
`seFirstChild`	Zeiger auf den ersten Kindknoten

Tabelle 5.1: Baumrepräsentation in `CSceneElement`

Diese Elemente erlauben es, einen Baum mit beliebig vielen Kindknoten pro Knoten (sog. „n-ärer Baum“) in einem Binärbaum darzustellen (vgl. [Sed93]). Dabei bilden die Kindknoten eines Knotens eine verkettete Liste, auf deren erstes Element der Elternknoten verweist. Jeder Knoten verweist zusätzlich auf seinen nächsten Bruderknoten.
Über den Verweis zum Elternknoten ist es möglich, von dort den Verweis auf den ersten Knoten in der Liste der Kindknoten zu erhalten, so dass es beim Traversieren des Baumes von jedem Kindknoten aus möglich ist, alle Kindknoten zu erreichen.

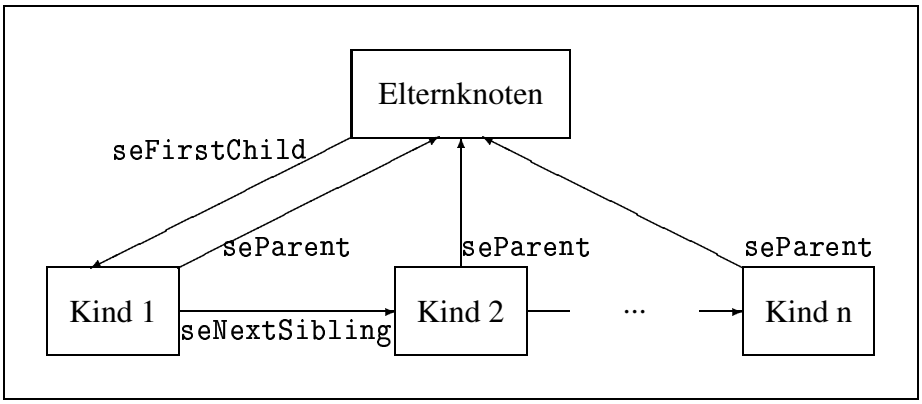

Abbildung 5.1: Struktur des Baumes in `CSceneElement`

Zur Verwaltung der Lage des Körpers in der Szene werden folgende Datenelemente verwendet:

Name	Inhalt
`mainTransformation`	Unveränderlicher Anteil der Transformation vom übergeordneten Körper zu diesem Körper
`mainTransformation-Inverse`	Inverse von `mainTransformation`
`subTransformation`	Beschreibung des parametrisierten Anteils der Transformation vom übergeordneten Körper zu diesem Körper

Tabelle 5.2: Repräsentation der räumlichen Lage in `CSceneElement`

Zur Beschreibung der Form des Körpers werden folgende Datenelemente verwendet:

Name	Inhalt
`shape`	Form und Lage des Körpers
`material`	Material des Körpers
`hasMaterial`	Schalter der angibt, ob Körper eigene Materialbeschreibung hat, oder die des übergeordneten Körpers verwendet wird.
`envelopeShape`	Form und Lage des Hüllkörpers
`hideEnvelope`	Schalter der angibt, ob der Hüllkörper angezeigt werden soll.

Tabelle 5.3: Repräsentation der Form in `CSceneElement`

5.1.2 Zusatzdaten

Neben den Mehrkörpersystemen soll es möglich sein, beliebige Zusatzinformationen in Form von Punkten, Geraden, Pfeilen, Polygonen und Kreisen zu visualisieren. Jedes einzelne darzustellende graphische Objekt wird dabei durch ein Objekt der Klasse `CAuxDataElement` beschrieben. Diese Klasse enthält einen Wert, der den genauen Typ des graphischen Objektes anbietet sowie die Information, wie viele Eckpunkte das graphische Objekt haben soll [2]. Neben diesen Informationen über die Form wird zusätzlich die Farbe und Transparenz des Objektes als RGBA-Quadrupel in einem Objekt vom Typ `CColor` abgelegt [3]. Weiterhin wird

[2] Die explizite Unterscheidung zwischen den Typen ist nur deshalb notwendig, weil Pfeil und Linie beide zwei Eckpunkte haben, sonst würde die Information über die Anzahl der Eckpunkte implizit den Typ festlegen!

[3] Für die Speicherung der Farbe wird hier weniger Aufwand getrieben als beim Material der Elemente der Mehrkörpersysteme. Aus dem Farbwert werden in den jeweiligen Visualisierungskomponenten die Werte für diffuse, ambiente und spekulare Reflexion generiert. Hierdurch wird

festgehalten, ob das Objekt mit der festgelegten Farbe dargestellt werden soll oder ob die Farbwerte für jeden Zeichenvorgang als Parameter angegeben werden.

Informationen über die Lage werden nicht abgelegt. Jedes Objekt speichert lediglich, durch wie viele Parameter seine Lage dargestellt wird. Dabei gibt es drei Klassen von Parametern: Skalare, Vektoren (bestehend aus drei Skalaren) und Farbwerte (bestehend aus 4 Skalaren). In der folgenden Tabelle wird widergegeben, wie viele und welche Parameter die einzelnen Zusatzinformationen haben:

Typ	Skalare	Vektoren	Farbwert
Punkt	keine	Mittelpunkt	optional
Linie	keine	Anfang und Ende	optional
Pfeil	keine	Anfang und Ende	optional
Polygon	keine	beliebige feste Anzahl an Eckpunkten	optional
Ellipse	Länge und Breite	Mittelpunkt und Normale	optional

Tabelle 5.4: Parameter der Zusatzdaten

5.1.3 Gesamtszene

Die Gesamtszene wird in einem Objekt der Klasse `CSceneDescription` abgelegt. Objekte dieser Klassen verfügen über Methoden zum Einlesen von XML-Beschreibungen für Mehrkörpersysteme und Zusatzdaten. Von beiden Dateitypen können beliebig viele Dateien eingelesen werden. Die Inhalte der Dateien werden in der Reihenfolge abgelegt und durchnummeriert, in der die Dateien eingelesen werden, wobei Mehrkörpersysteme und Zusatzdaten getrennt verwaltet werden.

Zum Einlesen der XML-Dateien werden dabei die Methoden `initParseRobotDescription` und `initParseAuxDescription` verwendet, die wiederum auf eine Menge interner Methoden zurückgreifen, um die Beschreibungen rekursiv zu verarbeiten. Dazu wird von der XML-Bibliothek *libXML* (vgl. 3.5) Gebraucht gemacht, die ein komfortables Verabeiten von XML-Dateien gestattet.

Die Daten werden Dabei in fünf dynamischen Feldern, die mit Hilfe des `vector`-Templates der STL (vgl. Abschnitt 3.6) realisiert sind gespeichert. Diese Felder sind:

implizit erzwungen, dass alle Zusatzdaten mit den gleichen Eigenschaften dargestellt werden und lediglich die Farbe beeinflusst werden kann.

Name	Inhalt
`vRootElement`	Wurzelknoten der Mehrkörpersysteme
`vSceneElement`	Elemente der Mehrkörpersysteme
`vCameraDescription`	Beschreibung der Kameras
`vLightDescription`	Beschreibung der Lichtquellen
`vAuxDataElement`	Zusatzdaten

Tabelle 5.5: Datenelemente der Szenebeschreibung `CSceneDescription`

Beim Ablegen aller Elemente in den Arrays wird die Reihenfolge verwendet, in der die Objekte in den Dateien beschrieben werden. Diese Reihenfolge ist wichtig, da sie auch die Zuordnung der Steuerparameter zu den Objekten festlegt.

5.2 XML-Beschreibung der Szene

Alle in der Szene darzustellenden Objekte werden über XML-Dateien beschrieben. Zur Beschreibung der Mehrkörpersysteme lag bereits der Entwurf einer Beschreibungssprache vor, der noch an einigen Stellen angepasst werden musste. Zur Beschreibung von Umgebung und Zusatzdaten wurden neue Beschreibungssprachen entwickelt.

5.2.1 Mehrkörpersysteme und Umgebung

Die Mehrkörpersysteme werden in XML-Dateien beschrieben. Jede dieser Dateien enthält die Beschreibung eines Mehrkörpersystems. Die XML-Beschreibung orientiert sich an der Beschreibung die von Alexander Helm für sein IDP verwendet wurde (vgl. [Hel03]).
Die Umgebung wird ebenfalls als ein Mehrkörpersystem repräsentiert.
Zunächst wird als Überblick die syntaktische Beschreibung in Backus-Naur-Form (BNF) angegeben. Im Anschluss werden die Grobstrukur der Datei sowie die verwendeten Knotentype und deren Bedeutung kurz beschrieben.

BNF

Im Folgenden wird die Syntax der Roboterbeschreibung in BNF angegeben (vgl. [Sch97]), wobei □ das leere Wort repräsentiert:

```
ROBO ::= <robo> BRANCH <robo/>
BRANCH ::= <branch> ORIENTATION LINK_LIST BRANCH_LIST <branch/>
BRANCH_LIST ::= BRANCH BRANCH_LIST | □
LINK_LIST ::= LINK LINK_LIST | LINK
```

```
LINK ::= <link> TRANSLATION JOINT_LIST OPT_SHAPE OPT_COLOR
   OPT_CHECKERED OPT_TEXTURE OPT_CAMERA OPT_LIGHT
   OPT_HIDEBOUNDINGVOLUME <link/>
JOINT_LIST ::= JOINT JOINT_LIST | □
JOINT ::= <trans XYZ/> | <rot XYZ/>
OPT_ORIENTATION ::= ORIENTATION | □
ORIENTATION ::= <orietation w=''NUMBER'' x=''NUMBER'' y=''NUMBER''
   z=''NUMBER''/>
OPT_TRANSLATION ::= TRANSLATION | □
TRANSLATION ::= <translation x=''NUMBER'' y=''NUMBER'' z=''NUMBER''/>
OPT_SHAPE ::= SHAPE | □
SHAPE ::= <shape type=''SHAPETYPE'' x=''NUMBER'' y=''NUMBER''
  z=''NUMBER''> OPT_ORIENTATION OPT_TRANSLATION </>
SHAPETYPE ::= cuboid | ellipsoid | cylinder | cone
OPT_COLOR ::= COLOR | □
COLOR ::= <color r=''NUMBER'' g=''NUMBER'' b=''NUMBER'' a=''NUMBER />
OPT_TEXTURE ::= TEXTURE | □
TEXTURE ::= <texture filename=''CHARSTRING'' />
OPT_CHECKERED ::= CHECKERED | □
CHECKERED ::= <checkered x=''NUMBER'' y=''NUMBER''>
OPT_CAMERA ::= CAMERA | □
CAMERA ::= <camera/>
OPT_LIGHT ::= LIGHT | □
LIGHT ::= <light/>
OPT_HIDEBOUNDINGVOLUME ::= HIDEBOUNDINGVOLUME | □
HIDEBOUNDINGVOLUME ::= <hideBoundingVolume/>
XYZ ::= x | y | z
```

Die Nichtterminale NUMBER und CHARSTRING repräsentieren eine Zahl bzw. eine Zeichenkette.

Grobstruktur

Die Dateien enthalten genau einen Knoten vom Typ *robo*. Dieser wiederum enthält genau einen Knoten vom Typ *branch*.
Mit einem *branch*-Knoten wird ein Kette von Körpern beschrieben, deren Koordinatensysteme in Ruhelage die gleiche Orientierung haben. Ein *branch*-Knoten enthält mindestens einen *link*-Knoten.
Die einzelnen Körper werden durch *link*-Knoten beschrieben. In jedem *link*-Knoten kann sich die Struktur des Mehrkörpersystems beliebig oft verzweigen; hierzu werden entsprechende *branch*-Knoten eingefügt.

branch-Knoten

Der *branch*-Knoten enthält einen *orietation*-Knoten, mit dem die Orientierung der Koordinatensysteme der im *branch*-Knoten enthaltenen Körper beschrieben wird. Dieser Knoten beschreibt die Orientierung als Quaternion (vgl. Abschnitt 3.1).
Auf den *orientation*-Knoten folgt mindestens ein *link*-Knoten. Die *link*-Knoten beschreiben die einzelnen Körper der durch den *branch*-Knoten beschriebenen Kette. Die Reihenfolge der *link*-Knoten bestimmt dabei die Reihenfolge der Körper in der Kette.

link-Knoten

Mit *link*-Knoten werden die einzelnen Körper des Mehrkörpersystems beschrieben. Sie enthalten sowohl Informationen über Lage und Gelenke des Körper als auch über dessen Farbe, Gestalt und Hüllkörper.

Lage und Gelenke Die Lage des Körpers wird durch einen *translation*-Knoten beschrieben. Dieser gibt die Verschiebung des Ursprungs des Körpers (in Ruhelage) zum Ursprung seines Vorgängers an. Die Translation erfolgt dabei in Bezug auf das Koordinatensystem des übergeordneten Körpers. Für den ersten Körper eines *branch*-Knoten bedeutet dies, dass erst die Translation im Koordinatensystem des übergeordneten Körpers ausgeführt wird, bevor die Orientierung des Koordinatensystems verändert wird.
Die Orientierung und Lage des Koordinatensystems des Körpers kann weiterhin durch Gelenke beeinflusst werden. Dazu können beliebig viele Gelenk-Knoten eingefügt werden. Da die Verkettung von Rotationen untereinander und mit Translationen nicht kommutativ ist, muss hier eine Reihenfolge für die Ausführung der Gelenktransformationen festgelegt werden: Die Transformationen werden in der Reihenfolge der Gelenk-Knoten ausgeführt.
Als Gelenk-Knoten sind translatorische (entlang der Hauptachsen) und rotatorische (um die Hauptachen) Gelenke definiert. Sie werden mittel *trans*- und *rot*-Knoten beschrieben.

Farbe und Gestalt Die Farbe des Körpers wird über einen *color*-Knoten beschrieben. Dieser Knoten enthält RGB-Werte für die Farbe sowie einen alpha-Wert für die Transparenz[4]. Aus diesen Werten wird intern eine komplette Materialbeschreibung generiert. Über einen *checkered*-Knoten kann der Körper mit einer

[4]Die bezeichnung „alpha-Wert“kommt vom sog. „Alpha-Blending“, einem Verfahren, bei dem zwei Farbwerte abhängig von einem Parameter, dem alpha-Wert, gemischt werden (vgl. [WNDS99]).

Karotextur überzogen werden, mit einem *texture*-Knoten kann eine beliebige Textur auf dem Körper aufgebracht werden.
Der Schwerpunkt des Körpers kann mit einem *center*-Knoten angegeben werden. Dieser gibt gleichzeitig den Mittelpunkt für die Darstellung der Form des Körpers vor.
Die Gestalt wird über einen *shape*-Knoten festgelegt. In diesem Knoten werden auch die Abmessungen angegeben. Zusätzlich kann der *shape*-Knoten jeweils einen *orietation*- und *translation*-Knoten enthalten, um die Lage des geometrischen Objektes relativ zum Schwerpunkt des Körpers zu beschreiben. Ohne diese Angaben wird das geometrische Objekt parallel zum Koordinatensystem des Körpers dargestellt[5].

Weitere Informationen Um eine Kamera mit dem Körper zu verknüpfen wird, ein *camera*-Knoten eingefügt.
Eine Lichtquelle wird über einen *light*-Knoten eingefügt.
Die Knoten für Lichtquellen und Kameras haben derzeit keine Parameter. Werden verschiedene Typen von Lichtquellen und Kameras in die Szenenbeschreibung integriert, müssen hier entsprechende Parameter ergänzt werden.
Um zu erzwingen, dass der Hüllkörper eines Körpers nicht angezeigt wird, wird ein *hideBoundingVolume*-Knoten eingefügt, der ebenfalls keine Parameter hat.

5.2.2 Zusatzdaten

Die Beschreibungssprache für die Zusatzdaten wird zunächst in BNF angegeben:

```
AUXDATA ::= <aux> ELEMENT_LIST <aux/>
ELEMENT_LIST ::= □| ELEMENT | ELEMENT ELEMENT_LIST
ELEMENT ::= POINT | LINE | ARROW | TRIANGLE | POLYGON
POINT ::= <point> OPTCOLOR <point/> | <point/>
LINE ::= <line> OPTCOLOR <line/> | <line/>
ARROW ::= <arrow> OPTCOLOR <arrow/> | <arrow/>
TRIANGLE ::= <triangle> OPTCOLOR <triangle/> | <triangle/>
POLYGON ::= <polygon n=''INTNUMBER''> OPT_COLOR <polygon/> |
  <polygon n=''NUMBER''/>
OPT_COLOR ::= □ | COLOR
COLOR ::= <color r=''NUMBER'' g=''NUMBER'' b=''NUMBER'' a=''NUMBER />
```

Auch hier repräsentiert `NUMBER` eine reelle Zahl, `INTNUMBER` ist eine natürliche Zahl.

[5]Die Beschreibung der Lage relativ zum Schwerpunkt wird verwendet, um alte Beschreibungsdateien, die diese Semantik verwenden, ohne Änderungen weiterbenutzen zu können.

Die Beschreibung von Zusatzdaten ist sehr einfach. Für jedes anzuzeigende Objekt wird ein entsprechender Knoten in der Datei angelegt. Punkte, Linien, Pfeile und Dreiecke haben als einzigen, optionalen Unterknoten einen Farbwert. Polygone erfordern zusätzlich zwingend die Angabe der Anzahl der Eckpunkte über den Parameter `n`.

5.3 Repräsentation der Steuerdaten

Die Steurdaten zur Animation der Szene werden in Objekten der Klasse `CFrameData` abgelegt, die von `CDataset` abgeleitet ist (vgl. 4.4.4).
Folgende Daten werden in einem solchen Objekt gespeichert:

Name	Inhalt
`sceneParameters`	Feld mit den Steuerdaten für die Mehrkörpersysteme der Szene (z. B. Gelenkwinkel)
`auxParameters`	Feld mit den Steuerdaten für die Zusatzinformationen (z. B. Eckpunkte von Polygonen, Enpunkte von Geraden, etc.)
`trfRel`	Feld mit den relativen Transformationen der Körper zu ihrem jeweiligen Vorgänger
`trfAbs`	Feld mit den absoluten Transformationen der Körper bezogen auf das Haupt-Koordinatensystem
`trfInvRel`	Feld mit den Inversen von `trfRel`
`trfInvAbs`	Feld mit den Inversen von `trfAbs`
`envelpeState`	Feld, das für jeden Körper der Szene den Zustand des Hüllkörpers angibt
`sceneDescription`	Zeiger auf die Beschreibung der aktuellen Szene
`sceneChanged`	Wahrheitswert, der angibt, ob die Szenenbeschreibung geändert wurde
`frameNumber`	Die Nummer des aus den Steuerdaten generierten Bildes

Tabelle 5.6: Datenelemente des Steuerdatenobjekts `CFrameData`

Alle Felder werden dabei mit Hilfe des `vector`-Templates der STL erzeugt.
Zum Speichern der Zustände der Hüllkörper werden Objekte der Klasse `CEnvelopeState` eingesetzt. Diese Objekte verwalten ein Bitfeld, in dem die Zustände abgelegt werden können. Derzeit ist nur der Zustand „kollidiert“ implementiert; weitere können hier leicht integriert werden.

Neben der Speicherung der Daten stellen Objekte der Klasse `CFrameData` die Möglichkeit bereit, aus der Szenenbeschreibung und den Steuerdaten die absoluten und relativen Transformationen und deren Inverse zu berechnen. Dabei werden die Baumbeschreibungen der einzelnen Mehrkörpersysteme rekursiv durchlaufen und dabei die Transformationen berechnet (vgl. 5.1.1 und 3.1).
Ebenso wie bei den Transformationsbeschreibungen (vgl. 5.1.1) könnte auch hier auf das Berechnen und Speichern der Inversen der Transformationen verzichtet werden. Da jedoch die Inversen unter Umständen mehrfach benötigt werden, erscheint es sinnvoller, sie einmal zu berechnen, so dass sie für alle weiteren Operationen bereitstehen.
Über die Klasse `CFrameData` wird die Beschreibung von Szene und Steuedaten in das in Abschnitt 4.4.5 beschriebene MVC-Konzept integriert.

Kapitel 6

Kollisionserkennung

In diesem Abschnitt wird die Bibliothek um eine Komponente zur Kollisionserkennung erweitert. Dazu wird zunächst die Schnittstelle zum Einbinden entsprechender Verfahren definiert und an einem einfachen Beispiel erläutert. Im Anschluss daran werden zwei Verfahren entwickelt, um die Kollisionserkennung für verschiedene Klassen von Hüllkörpern durchzuführen.

6.1 Schnittstelle zur Kollisionserkennung

Verfahren zur Kollisionserkennung werden in Unterklassen der Klasse `CCollisionDetector` implementiert. `CCollisionDetector` übernimmt folgende Aufgaben:

- Verwalten einer Liste von Paaren von Körpern, die auf Kollision getestet werden sollen.
- Verwalten einer Liste der Körper, für die mindestens ein Kollisionstest durchgeführt werden soll. Diese Liste wird verwendet, um den Rechenaufwand zu verringern, indem die Vorbereitung für den Test nur für die zu testenden Körper durchgeführt wird.
- Durchführen der Kollisionserkennung:
  - Vorbereiten der Kollisionserkennung für die betroffenen Körper
  - Durchführen der Kollisionserkennung für die angegebenen Paare von Körpern
  - Setzen bzw. Löschen der „Kollisionsbits“ (vgl. 5.3) für die einzelnen Körper im aktuellen Steuerdatensatz

Zum Bestimmen der Paare von Hüllkörpern, für die eine Kollisionerkennung durchgeführt werden soll, wird die Methode `addPairOfElements` verwendet, mit der es möglich ist, die Nummern zweier Hüllkörper, die auf Kollision getestet werden sollen zu setzen. Über die Methode `loadPairs` kann eine Liste von Paaren aus einer Datei eingelesen werden.
Um ein konkretes Kollisionsverfahren zu implementieren, muss eine Unterklasse von `CCollisionDetector` erzeugt werden, in der die folgenden Aufgaben übernimmt:

- Erzeugen von Hüllkörpern für die einzelnen Elemente der Mehrkörpersysteme.
- Generieren und Verwalten der für die Kollisionserkennung notwendigen Daten zur Lage der Körper, die sich aus den konkreten Steuerdaten ergeben.
- Erkennen von Kollisionen zwischen je zwei Körpern.

Dazu müssen vier virtuelle Methoden überschrieben werden:

- `createEnvelope` erzeugt aus einer Körperschreibung in `CShape` die Beschreibung eines passenden Hüllkörpers. Dabei wird der Typ von Hüllkörper erzeugt, den das Verfahren verarbeiten kann.
- `prepareDetetection` wird von der Basisklasse einmal für jeden Hüllkörper aufgerufen, für den die Kollisionserkennung durchgeführt werden soll, um die aktuelle Lage des Hüllkörpers und weitere für das jeweilige Verfahren benötigte, zeitlich veränderliche Daten zu setzen. Je nach Verfahren sind dazu verschiedene Berechnungen notwendig. Um den Rechenaufwand für den Fall, dass ein Hüllkörper auf mehrere Kollisionen getestet werden soll, gering zu halten, werden die Berechnungen mit dieser Methode vor den eigentlichen Kollisionstests ausgeführt und die Ergebnisse gespeichert.
- `invalidateVolumes` wird aufgerufen, um alle von der Unterklasse angelegten Daten über Hüllkörper zu löschen. Diese Methode ist dann wichtig, wenn in `prepareDetection` Berechnungen zur Effizienzsteigerung nur einmalig für alle Durchgänge angestellt werden und sich die Szene ändert.
- `detetectCollision` erhält die Nummern zweier Hüllkörper, die auf Kollision getestet werden sollen. Im Falle einer Kollision gibt die Methode den Wert 1 zurück, sonst 0.

Die Verbindung der Kollisionserkennung mit den anderen Modulen der Software erfolgt über die Klasse `CApplicationModel` (vgl 4.4.5). Diese Klasse verwaltet

einen Zeiger auf ein Objekt der Klasse `CCollisionDetector`. Werden die Daten von einem der *Controller* verändert, wird mit dem geänderten Datensatz zunächst die Kollisionserkennung durchgeführt, bevor der Datensatz an die *Views* weitergeleitet wird.

6.2 Kollisionserkennung für Kugeln

In diesem Abschnitt soll anhand eines sehr einfachen Kollisionserkennungsverfahrens die Erzeugung einer Unterklasse von `CCollisionDetector` demonstriert werden.
Als Verfahren wird die Erkennung von Kollisionen kugelförmiger Hüllkörper gewählt. Dazu wird jeder Hüllkörper durch seinen Mittelpunkt und seinen Radius beschrieben. Zwei kugelförmige Hüllkörper kollidieren offensichtlich dann, wenn der Abstand ihrer Mittelpunkte kleiner als die Summe ihrer Radien ist.
Das Verfahren wird in der Klasse `CSphereCollisionDetector` implementiert. Diese Klasse verwaltet als interne Datenstruktur ein Feld von Objekten, die jeweils Mittelpunkt und Radius eines Hüllkörpers und ein Gültigkeitsflag für den Radius[1] enthalten.
Im Folgenden wird beschrieben, welche Funktionen die vier virtuellen Methoden erfüllen:

- `createEnvelope` erzeugt aus der Beschreibung des zu umhüllenen Körpers einen passenden Hüllkörper. Dazu werden die in `CShape` gespeicherte Länge, Breite und Höhe des den Körper umhüllenden Quaders ausgelesen. Aus diesen wird der Durchmesser einer den Quader umhüllenden Kugel berechnet und ein entsprechendes `CShape`-Objekt erzeugt. In diesem Objekt werden zusätzlich die feste Transformation (und deren Inverse) zur Beschreibung der Lage des Hüllkörpers gespeichert.

- `invalidateVolumes` löscht das Feld mit den Körperbeschreibungen.

- `prepareDetection` berechnet den Mittelpunkt für die angegebenen Kugel. Dazu wird die an die Funktion übergebene, zeitlich veränderliche, absolute Position des Hauptkoordinatensystem mit der feste Transformation des Hüllkörpers verknüpft. Der translatorische Anteil dieser Transformation ist der Mittelpunkt der Kugel. Ist das Gültigkeitsflag für den Radius nicht gesetzt, wird dieser berechnet und das Flag gesetzt.

- `detectCollision` prüft für die angegebene Hüllkörper, ob sie kollidieren. Dazu wird lediglich auf die Daten zurückgegriffen, die in `prepareDetection` erzeugt werden.

[1] Das Gültigkeitsflag wird verwendet, um die mehrfache Berechnung des Radius zu vermeiden.

6.3 Kollisionserkennung für Quader

In diesem Abschnitt wird ein Verfahren entwickelt, das es gestattet zu überprüfen, ob zwei beliebig orientierte Quader kollidieren. Im Anschluss daran wird beschrieben, wie dieses Verfahren mit Hilfe der im vorigen Abschnitt beschriebenen Schnittstelle implementiert wird.

6.3.1 Grundgedanke für das Verfahren

Dem Verfahren liegt ein einfacher Gedanke zugrunde:
Betrachtet man zwei Quader als Mengen von Punkten $Q_1, Q_2 \in \mathbb{R}^3$, so kollidieren die beiden Quader genau dann, wenn ihre Schnittmenge $S := Q_1 \cap Q_2$ nicht leer ist. Die Schnittmenge dieser beiden Quader ist dabei immer ein konvexes Polyeder.
Der Schnitt zweier Quader ist ein Sonderfall des Schnittes zweier konvexer Polygone, der in den beiden folgenden Abschritten beschrieben wird.

6.3.2 Schnitt zweier konvexer Polyeder

Die beiden zu schneidenden Polyeder P_1 und P_2 seien auf verschiednen Arten beschrieben:
P_1 sei durch die Menge G der es begrenzenden Ebenen, die im Folgenden als *Grenzebenen* bezeichnet werden, beschriebenen. Jede Grenzebene unterteilt eine beliebige Teilmenge des $\mathbb{R}^3$ in eine Menge, die das Polyeder enthält und eine, die das Polyeder nicht enthält. Auf der Grenzebene liegt eines der das Polyeder begrenzenden Polygone. Wird beginnend mit dem ganzen $\mathbb{R}^3$ der Raum sukzessive unterteilt, wobei immer der Teil des Raumes weiter unterteilt wird, der das Polyeder enthält, bleibt am Ende genau das Polyeder übrig[2].
Werden die Grenzebenen in Hesse-Normalform derart beschrieben, dass die Normale der Ebene in Richtung des Teils des $\mathbb{R}^3$ zeigt, der das Polyeder enthält, so kann für einen beliebigen Punkt $p \in \mathbb{R}^3$ leicht geprüft werden, ob dieser Teil des Polyeders ist: Mit Hilfe der Hesse-Normalform wird für den Punkt der Abstand zu allen Grenzebenen bestimmt. Dabei ergeben sich folgende Fälle:

- Der Abstand ist für eine der Ebenen kleiner als 0: p liegt außerhalb des Polyeders.
- Der Abstand ist für alle Ebenen größer als 0: p liegt im Innern des Polyeders.

[2]Diese Beschreibung ist äquivalent zur Beschreibung eines konvexen Polyeders als Schnittmenge von Halbräumen.

- Der Abstand ist für genau eine Ebenen gleich 0 und für alle anderen kleiner als 0: p liegt auf einer der Flächen des Polyeders.
- Der Abstand ist für genau zwei Ebenen gleich 0 und für alle anderen kleiner als 0: p liegt auf einer der Kanten des Polyeders.
- Der Abstand ist für mehr als zwei Ebenen gleich 0 und für alle anderen kleiner als 0: p liegt auf einer der Ecken des Polyeders.

P_2 sei durch die Menge seiner Eckpunkte E und die Menge der diese verbindenden Kanten K beschrieben. Dabei soll K genau die Menge der Kanten enthalten, die Teil der Oberfläche des Polyeders sind. Jede Kante $k \in K$ ist dabei durch einen Tupel von Eckpunkten (e_i, e_j) mit $e_i, e_j \in E$ beschrieben, also gilt $k = (e_i, e_j) \Rightarrow e_1, e_2 \in E$.
Um zu prüfen, ob P_1 und P_2 kollidieren, wird der Schnittpolyeder der beiden berechnet. Dazu wird P_2 sukzessive an den Ebenen aus G geteilt, wobei das Verfahren immer mit dem Restpolyeders fortgeführt wird, das Teil von P_1 ist.
Dieses Verfahren lässt sich wie folgt als Pseudocode ausdrücken:

```
P_schnitt := P2;
mit jeder Ebene E aus der Beschreibung von P1 {
    P_schnitt := Rest von P_schitt, getrennt an E;
}
```

Im folgenden Abschnitt wird geklärt, wie das Trennen durchgeführt wird.

6.3.3 Teilen eines konvexen Polyeders an einer Ebene

Um ein Polyeder, das in der angegebenen Form beschrieben ist, an einer Ebene zu trennen, müssen mehrere Schritte durchgeführt werden:

1. Prüfen, welche der Eckpunkte innerhalb des Restpolyeders sind und welche nicht.
2. Prüfen, welche Kanten komplett im Restpolyeder sind, welche komplett wegfallen und welche getrennt werden.
3. Erzeugen neuer Eckpunkte und Kanten für die getrennten Kanten.
4. Erzeugen neuer Kanten an der entstandenen Schnittfläche.

Schritt 1 ist trivial: Mit Hilfe der Hesse-Normalform der Ebene ist es leicht, zu entscheiden, welche Eckpunkte im Restpolyeder enthalten sind.
Schritt 2 ist ebenfalls einfach: Für jede Kante wird geprüft, wie viele ihrer Endpunkte im Restpolyeder enthalten sind. Entprechend dieser Anzahl gibt es drei Fälle:

- Beide Endpunkte liegen im Restpolyeder: Die Kante ist Teil der Beschreibung des Restpolyeders.
- Ein Endpunkt liegt im Restpolyeder, einer außerhalb: Die Kante muss an der Grenzebene getrennt werden.
- Beide Endpunkte liegen außerhalb des Restpolyeders: Die Kante wird verworfen.

Schritt 3 kann direkt in Schritt 2 integriert werden: Wird eine Kante getrennt, wird der Schnittpunkt der Kante mit der Ebene berechnet. Dieser Schnittpunkt und der im Restpolyeder enthaltene Endpunkt der Kante bilden eine neue Kante des Restpolyeders. Der Schnittpunkt ist ebenfalls Teil des Restpolyeders.
Schritt 4 benötigt die in Schritt 3 berechneten neuen Eckpunkte. Diese beschreiben ein konvexes Polygon, das die Schnittfläche des ursprünglichen Polyeders mit der Grenzebene beschreibt. Um das Verfahren fortsetzen zu können, müssen die Kanten, die dieses Polygon begrenzen, der Beschreibung des Restpolyeders hinzugefügt werden. Dazu muss untersucht werden, wie die Ecken durch Kanten zu verbinden sind, so dass keine der Kanten eine andere schneidet.
Zum Berechnen der neuen Kanten wird wie folgt vorgegangen: Zunächst wird der Schwerpunkt des Polyons durch Bilden des arithmetischen Mittels der Eckpunkte bestimmt. Der Schwerpunkt liegt immer im Inneren eines konvexen Polygons. Nun werden die Richtungsvektoren des Schwerpunktes zu den Eckpunkten berechnet. Einer der Richtungsvektoren wird willkürlich ausgewählt, und der Winkel aller Richtungsvektoren zu diesem berechnet. Nun werden die Eckpunkte nach den Winkeln sortiert und benachbarte Eckpunkte durch Kanten verbunden.

6.3.4 Implentierung für Quader

Ist einer der beiden Polyeder ein Quader, so lässt sich das oben angegebene Verfahren besonders leicht implementieren. Dazu werden beide Polyeder in einem Koordinatensystem dargestellt, dessen Achsen parallel zu den Seiten des Quaders sind. Durch dieses Vorgehen wird der Schnitttest an den Grenzebenen des Quaders stark vereinfacht, da jeweils nur eine Komponente des Ortsvektors der Eckpunkte des Polyeders betrachtet werden muss, um den Abstand von der Ebene zu bestimmen.
Auf diese Art wird das Verfahren in der Klasse *CCuboidCollisionDetector* implementiert, wobei beide Polyeder Quader sein müssen.
Um den Rechenaufwand zu veringern, wird vor dem eigentlichen Kollisionstest mit einem einfachen Verfahren untersucht, ob eine Kollision auszuschließen ist. Dazu werden die die Quader umhüllenden Kugeln betrachtet (vgl. 6.2). Nur wenn sich diese überschneiden, können sich die Quader selbst überschneiden.

6.4 Verwendung von Superquadriken als Hüllkörper

Mit den in Abschnitt 3.4 beschriebenen Superquadriken ist es möglich, durch verändern der Parameter e_1 und e_2 Funktionen zur Beschreibung verschiedener geometrischer Funktionen anzugeben.
Die allgemeine Form der Superquadriken ist

$$f_{(e_1,e_2)}(x,y,z) = \left(x^{\frac{2}{e_2}} + y^{\frac{2}{e_2}}\right)^{\frac{e_2}{e_1}} + z^{\frac{2}{e_1}}, \;\; e_1,e_2,x,y,z \in \mathbb{R}.$$

Im Folgenden wird gezeigt, wie durch Superquadriken Quader und Zylinder angenähert werden können. Daraufhin wird eine allgemeine Strategie zur Kollisionsbestimmung für implizit definierte Körper angegeben. Es wird gezeigt, dass sich diese Strategie gleichermassen für die Kollisionserkennung zweier Quader, zweier Zylinder und eines Quaders und eines Zylinders anwenden lässt.

6.4.1 Annähern von Quadern und Zylindern durch Superquadriken

Vewendet man Superquadriken zur Annähern von Quadern und Zylindern, so lassen sich durch geignete Wahl der Parameter Funktionen finden, die stetig differenzierbar sind und die über ein globales Minimum verfügen, welches sich im Mittelpunkt des beschriebenen Körpers befindet.
Zum Annähern von Quadern werden die Parameter $e_1 = e_2 = \frac{1}{n}$, $n \in \mathbb{N}$ gewählt, so dass sich die Funktion $q_n(x,y,z)$ ergibt:

$$q_n(x,y,z) := f_{\left(\frac{1}{n},\frac{1}{n}\right)}(x,y,z) = x^{2n} + y^{2n} + z^{2n},\ n \in \mathbb{N}.$$

Zum Annähern von Zylindern werden $e_1 = \frac{1}{n}$, $n \in \mathbb{N}$ und $e_2 = 1$ verwendet, was zu

$$z_n(x,y,z) := f_{\left(\frac{1}{n},1\right)}(x,y,z) = \left(x^2 + y^2\right)^n + z^{2n},\ n \in \mathbb{N}$$

führt.
Für

$$q_1(x,y,z) = z_1(x,y,z) = x^2 + y^2 + z^2$$

ergibt sich das Quadrat des Abstands vom Urprug, der entstehende Körper ist eine Kugel.
Die folgenden Abbildungen zeigen die Körper, die sich ergeben, wenn man die impliziten Flächen, die sich für $q_n(x,y,z) - 1 = 0$ bzw. $z_n(x,y,z) - 1 = 0$ ergeben, betrachtet:

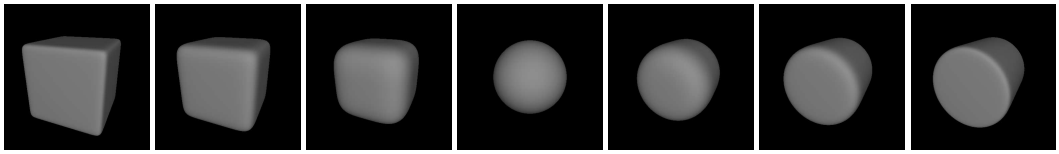

Abbildung 6.1: Die Abbildungen zeigen von links nach rechts die Quader bzw. Zylinder, die sich für die Funktionen q_8 , q_4 , q_2 , $q_1 = z_1$, z_2 , z_4 , z_8 ergeben.

Durch affin-lineare Transformation der Koordinaten $(x,y,z)^T \in \mathbb{R}^3$ lassen sich die Körper beliebig skalieren, drehen und verschieben.
Superquadriken erscheinen als geignetes Mittel verschiedene Arten von Körpern auf einheitliche Art und Weise anzunähern.
Die Vermutung lieg nahe, dass sich die entstehenden Funktionen zur Entwicklung eines Kollisionserkennungsverfahrens eignen. Ein solches Verfahren wurde im Rahmen dieser Arbeit nicht entwickelt.

6.4.2 Strategie zur Kollisionserkennung für implizit definierte Körper

Es wird angenommen, dass zwei Körper implizit durch Funktionen $k_1, k_2 : \mathbb{R}^3 \to \mathbb{R}_0^+$ derart beschrieben sind, dass ein Punkt $p \in \mathbb{R}^3$ genau dann Teil des Körpers i ist, wenn gilt $k_i(p) \leq 1,\ i = 1,2$. Die Körper sind somit durch die Punktmengen

$$K_i = \{p \in \mathbb{R}^3 \,|\, k_i(p) \leq 1\},\ i = 1,2$$

beschrieben.
Offensichtlich kollidieren die beiden Körper genau dann, wenn mindestens ein Punkt $p \in \mathbb{R}^3$ existiert, der Teil beider Körper ist, d.h. sowohl $k_1(p) \leq 1$ als auch $k_2(p) \leq 1$ ist erfüllt. Um die Kollision der beiden Körper festzustellen, muss also ein entsprechender Punkt p aufgefunden werden. Ein einfaches Durchsuchen des $\mathbb{R}^3$ nach einem passenden Punkt ist offensichtlich keine Lösung, da keine weiteren Eigenschaften über die Funktionen k_1 und k_2 bekannt sind.
Anstelle direkt ein Verfahren zu entwickeln, das einen passenden Punkt, sofern er existiert, findet, wurde folgende Überlegung angestellt:
Betrachtet man die Summe $s(p) = k_1(p) + k_2(p)$ der beiden Funktionen k_1 und k_2, ergeben sich drei Fälle:

1. $s(p) \leq 1$: p liegt in beiden Körpern.

2. $1 < s(p) \leq 2$: p liegt in mindestens einem Körper und kann, aber muss nicht, in beiden Körpern liegen.

3. $2 < s(p)$: p liegt in maximal einem Körper, kann aber auch außerhalb beider Körper liegen.

Nun wird das globale Minimum m von $s(p)$ bestimmt und einem der drei Fälle zugeordnet:

1. $m \leq 1$: Es existiert ein Punkt p, der Teil beider Körper ist, also kollidieren die Körper.

2. $1 < m \leq 2$: Da m das Minimum ist, existiert kein Punkt, für den mit dem gegebenen Verfahren entscheidbar ist, ob er Teil beider Körper ist.

3. $2 < m$: Der Punkt p mit $s(p) = m$ ist nicht Teil beider Körper. Da m das Minimum von s ist, kann auch kein anderer Punkt existieren, der Teil beider Körper ist, also kollidieren die beiden Körper nicht.

Interessant für die weitere Betrachtung ist nur noch der zweite Fall von interesse, da hier keine eindeutige Entscheidung möglich ist. Hierzu wurde folgende Überlegung angestellt:
Durch die Funktionen k_1 und k_2 lassen sich durch vergleich mit anderen Konstanten weitere Körper beschreiben. Im folgenden soll dabei

$$K_i' = \{p \in \mathbb{R}^3 \,|k_i(p) \leq 2\},\ \ i = 1,2$$

gelten. Offensichtlich gilt

$$K_i \subseteq K_i',\ \ i = 1,2.$$

Ist $m \leq 2$ gilt weiterhin $m \in K_1'$ und $m \in K_2'$. Daraus lässt sich schliessen, dass die durch K_1' und K_2' beschriebenen Körper für diesen Fall kollidieren.
Steigen die Funktionen k_1 und k_2 hinreichend steil an, so sind die durch K_i' beschriebene Körper nur wenig größer als die durch K_i beschriebenen. Aus der festgestellten Kollision der Körper K_i' kann man näherungsweise auf eine Kollision der Körper K_i schliessen. Dabei können nicht-Kollisionen als Kollisionen gewertet werden, allerdings müssen dazu die Körper K_i' kollidieren, so dass der Fehler nicht sehr groß ist. Weiterhin ist sichergestellt, dass jede tatsächliche Kollision erkannt wird.
Soll das Verfahren für konkrete, implizit definierte Körper angewendet werden, müssen zwei Fragen geklärt werden:

- Wie groß ist der Fehler bei falsch erkannte Kollisionen?

- Ist es möglich, das absolute Minimum der Summe $s(p) = k_1(p) + k_2(p)$ zu bestimmen?

6.4.3 Kollisionserkennung für angenäherte Quader

Im Folgenden seien zwei angenäherte Quader als Punktmengen Q_1 und Q_2 beschrieben, so dass

$$Q_1 = \{p \in \mathbb{R}^3 \,|\, q_n(Ap+a) \leq 1\}$$

und

$$Q_2 = \{p \in \mathbb{R}^3 \,|\, q_n(Bp+b) \leq 1\}$$

gelten.
Dabei beschreiben $A \in \mathbb{R}^{3x3}$ und $a \in \mathbb{R}^3$ bzw. $B \in \mathbb{R}^{3x3}$ und $b \in \mathbb{R}^3$ affin-lineare Transformationen der Quader, um diese frei rotieren, translieren und skalieren zu können.
Es wird untersucht, ob die im vorigen Abschnitt definierte Strategie für die beschriebenen Quader anwendbar ist.

Fehler durch falsch erkannte Kollision

Es werden die durch

$$Q_1' = \{p \in \mathbb{R}^3 \,|\, q_n(Ap+a) \leq 2\}$$

und

$$Q_2' = \{p \in \mathbb{R}^3 \,|\, q_n(Bp+b) \leq 2\}$$

beschriebene Körper betrachtet.
Diese sind entlang der Hauptachsen um den Faktor $\sqrt[2n]{2}$ größer als die durch Q_1 bzw. Q_2 beschriebenen Körper. Für die falsche Erkennung einer Kollsion ist es notwendig (aber nicht hinreichend), dass sich die durch Q_1' und Q_2' beschriebenen Körper berühren. Durch Vergrößern von n lassen sich diese beliebig nahe an die durch Q_1 und Q_2 beschriebenen Körper annähern, so dass der Bereich in dem eine fälschliche Erkennung möglich ist, beliebig klein werden kann, wenn es die Anwendung erfordert.

Bestimmung des absoluten Minimums

Es wird gezeigt, dass es möglich ist, das absolute Minimum von

$$s_{qq}(p) = q_n(Ap+a) + q_n(Bp+b)$$

zu bestimmen.
Dazu wird zunächst gezeigt, dass die Punktmengen

$$S_c = \{p \in \mathbb{R}^3 \,|\, s_{qq}(p) \leq c\}, \;\; c \in \mathbb{R}_0^+$$

konvexe Körper beschreiben. Im Anschluss daran wird gezeigt, dass ein Minimum existiert und eindeutig ist, so dass es mit einfachen numerischen Methoden bestimmt werden kann.

Konvexität der durch S_c beschriebenen Körper: Ein Körper ist dann konvex, wenn alle Punkte auf der Verbindungslinie zweier beliebiger Punkte des Körpers ebenfalls Teil des Körpers sind. Es muss also gezeigt werden, dass

$$p_1 \in S_c \wedge p_2 \in S_c \Rightarrow \{tp_1 + (1-t)p_2 | t \in [0,1]\} \subseteq S_c$$

gilt.
Aus der Definition von S_c folgt somit $s_{qq}(p_1) \leq c$ und $s_{qq}(p_2) \leq c$.
Die Gerade durch p_1 und p_2 wird nach $t \in \mathbb{R}$ parametrisiert als Funktion

$$p(t) = tp_1 + (1-t)p_2 = p_2 + t(p_1 - p_2)$$

beschrieben und untersucht, ob

$$t \in [0,1] \Rightarrow s_{qq}(p(t)) \leq c$$

gilt.
Zunächst wird dazu $s_{qq}(p(t))$ umgeformt:

$$s_{qq}(p(t)) = q_n(Ap(t) + a) + q_n(Bp(t) + b)$$

Mit der Definition von $p(t)$ ergibt sich:

$$s_{qq}(p(t)) = q_n(A(p_2 + t(p_1 - p_2)) + a) + q_n(B(p_2 + t(p_1 - p_2)) + b)$$

Dies wird umgeformt zu:

$$s_{qq}(p(t)) = q_n(Ap_2 + a + tA(p_1 - p_2)) + q_n(Bp_2 + b + tB(p_1 - p_2))$$

A, a, B, b, p_1 und p_2 sind konstant. Es werden folgenden Substitutionen definiert:

$$c = (c_x\ c_y\ c_z)^T = Ap_2 + a \qquad d = (d_x\ d_y\ d_z)^T = A(p_1 - p_2)$$
$$e = (e_x\ e_y\ e_z)^T = Bp_2 + b \qquad f = (f_x\ f_y\ f_z)^T = B(p_1 - p_2)$$

und eingesetzt:

$$s_{qq}(p(t)) = q_n(c + td) + q_n(e + tf)$$

Damit ergibt sich

$$\begin{aligned} s_{qq}(p(t)) &= (c_x + td_x)^{2n} + (c_y + td_y)^{2n} + (c_z + td_z)^{2n} \\ &+ (e_x + tf_x)^{2n} + (e_y + tf_y)^{2n} + (e_z + tf_z)^{2n} \end{aligned}$$

Jeder der sechs Summanden ist eine nach oben geöffnete Parabel mit einem eindeutigen Minimum. Betrachtet man die erste Ableitung einer solchen Parabel erhält man immer eine streng monoton steigende Funktion, deren Wertebereich ganz

$\mathbb{R}$ umfasst. Die Summe solcher Funktionen ist wieder eine streng monoton steigende Funktion, deren Werteberich ganz $\mathbb{R}$ ist. Also ist die erste Ableitung von $s_{qq}(p(t))$ streng monoton steigend und umfasst ganz $\mathbb{R}$, so dass sie genau eine Nullstelle hat. Daraus folgt, dass $s_{qq}(p(t))$ für $t \in \mathbb{R}$ ein eindeutiges Minimum hat.
Wegen des eindeutigen Minimums kann $s_{qq}(p(t))$ auf dem Intervall $t \in [0,1]$ keine Werte annehmen, die größer als das Maximum der aus $s_{qq}(p(0))$ und $s_{qq}(p(1))$ sind. Da $p(0) = p_2 \in S_c$ und $p(1) = p_2 \in S_c$ gewählt waren, gelten $p(0) \leq c$ und $p(1) \leq c$. Also gilt auf dem Intervall $t \in [0,1]$, dass $s_{qq}(p(t)) \leq c$ ist. Damit sind alle Punkte auf der Verbindunglinie zwischen p_1 und p_2 aus S_c.
Da p_1 und p_2 beliebig aus S_c gewählt sind, ist der durch S_c beschriebene Körper konvex.

Existenz des Minimums von $s_{qq}(p)$: Die Mengen S_c sind für absteigendes c ineinander enthalten, so dass $d \leq c \Rightarrow S_d \subseteq S_c$ gilt. Also existierte eine kleinste nichtleere Menge $S_{c_{min}}$, die den Punkt enthält, in dem $s_{qq}(p)$ ein Minimum c_{min} annimmt.

Eindeutigkeit des Minimums Um die Eindeutigkeit des Minimums zu zeigen wird angenommen, dass zwei Punkte $\tilde{p_1}$ und $\tilde{p_2}$ existieren, in denen s_{qq} ein Extremum annimmt. In den Extrema wird der Gradient der Funktion 0 (vgl. [For84]). Also muss gelten $grad(s_{qq}(\tilde{p_1})) = grad(s_{qq}(\tilde{p_2})) = (0\,0\,0)^T$. Wird der Gradient in einem Punkt 0, so muss auch jede Richtungsableitung in diesem Punkt 0 werden. Nun wird die Gerade durch $\tilde{p_1}$ und $\tilde{p_2}$ betrachtet. Es wurde gezeigt, dass die Funktion s_{qq} entlang jeder Geraden nur ein Minimum annimmt, also kann auch die Ableitung entlang dieser Geraden nur in einem Punkt den Wert 0 annehmen. Da die betrachtete Gerade durch $\tilde{p_1}$ und $\tilde{p_2}$ geht, kann die Ableitung in Richtung der Geraden nur in einem der beiden Punkte 0 sein, so dass in einem der Punkte der Gradient nicht 0 sein kann. Folglich ist mindestens einer der Punkte kein Extremum. Also hat $s_{qq}(p)$ nur ein Extremum. Da die Existenz von Minima bereits gezeigt wurde, muss dieses Extremum ein Minimum sein.

Bestimmen des Minimums Da das Minimum eindeutig ist, lässt es sich auf numerischem Wege bestimmen, z. B. mit der Methode des steilsten Abstiegs.

Zusammenfassung

Es wurde gezeigt, dass es möglich ist für die Funktion $s_{qq}(p)$ das globale Minimum zu bestimmen. Also ist es möglich, die im vorigen Abschnitt beschriebe-

ne Strategie zur Kollisionserkennung anzuwenden. Dabei werden bei hinreichen großem n nur kleine Fehler gemacht.
Das Verfahren eignet sich also, um Kollisionen von Quadern, die durch Superquadriken angenähert sind, zu erkennen.

6.4.4 Übertragung auf angenäherte Zylinder

Um das Verfahren zur Erkennung der Kollision zweier angenäherter Zylinder anzuwenden, muss analog zum vorigen Abschnitt, gezeigt werden, dass die Funktion

$$s_{zz}(p) = z_n(Ap+a) + z_n(Bp+b)$$

ein eindeutiges Minimum hat.
Der Beweis kann dabei völlig analog geführt werden, er unterscheidet sich nur dadurch, dass gezeigt werden muss, dass s_{zz} (und nicht s_{qq} entlang einer beliebigen Geraden nur ein Minimum hat.
Dazu wird s_{zz} entlang einer Geraden $p(t)$ parametrisiert und wie im vorigen Abscnitt umgeformt, so dass sich die Funktion

$$\begin{aligned} s_{zz}(p(t)) &= ((c_x+td_x)^2+(c_y+td_y)^2)^n+(c_z+td_z)^{2n} \\ &+ ((e_x+tf_x)^2+(e_y+tf_y)^2)^n+(e_z+tf_z)^{2n} \end{aligned}$$

ergibt.
Wiederum wird gezeigt, dass die Ableitungen der einzelnen Summanden streng monoton steigend sind. Dies ist für die Summanden $(c_z+td_z)^{2n}$ und $(e_z+tf_z)^{2n}$ bereits bekannt, der erste und dritte Summand sind von der gleichen Gestalt, so dass es nur für einen der beiden gezeigt werden muss.
Im Folgenden wird der erste Summand

$$((c_x+td_x)^2+(c_y+td_y)^2)^n = (v(t))^n$$

untersucht.
Betrachtet man $v(t)$ sieht man leicht, dass die Funktion nie kleiner als 0 wird. Die Ableitung

$$v'(t) = 2d_x(c_x+td_x)+2d_y(c_y+td_y) = 2(d_xc_x+d_yc_y)+t(d_x^2+d_y^2)$$

ist offensichtlich streng monoton steigend, so dass $v(t)$ nur ein Minimum hat.
Da $v(t)$ eine quadratische Parabel ist, läßt es sich in der Form

$$v(t) = u(s+t)^2+w$$

angeben. Dabei nimmt $v(t)$ für $t=-s$ das Minimum an.

Offensichtlich gilt $v(-s+t) = v(-s-t)$. Diese Eigenschaften übertragen sich bei der Exponentiation, so dass $(v(t))^{n-1}$ für $t = -s$ das Minimum annimmt und $(v(-s-t))^{n-1} = (v(-s+t))^{n-1}$ gilt.
Die Ableitung $v'(t)$ hat für $t = -s$ ihren Nullpunkt.
Nun kann das Produkt $(v(t))^{n-1}v'(t)$ betrachtet werden. Für $t = -s$ ergibt sich ein Nullpunkt. für $t > -s$ steigen sind beide Faktoren positiv und steigen streng monoton an, so dass das Produkt ebenfalls streng monoton ansteigt. Für $t < -s$ ist der eine Faktor positiv und fällt streng monoton, während der andere Faktor negativ ist und streng monoton steigt, so dass das Produkt auch hier streng monoton steigt.
Also ist $(v(t))^{n-1}v'(t)$ für $t \in \mathbb{R}$ monoton steigend.
Damit ist die Ableitung von $s_{zz}(p(t))$ streng monoton steigend, so dass $s_{zz}(p(t))$ nur ein Minimum annimmt.
Der weitere Beweis lässt sich analog zum letzten Abschnitt führen.

6.4.5 Übertragung auf angenäherte Zylinder und Quader

Sollen ein angenäherter Zylinder und ein angenäherter Quader auf Kollision untersucht werden, muss das globale Minimum der Funktion

$$s_{qz}(p) = q_n(Ap+a) + z_n(Bp+b)$$

bestimmt werden.
Auch hier wird wieder untersucht, ob $s_{qz}(p)$ entlang beliebiger Geraden nur ein Minimum annimmt.
Analog zum vorherigen Vorgehen erhält man

$$\begin{aligned} s_{qz}(p(t)) &= (c_x+td_x)^{2n} + (c_y+td_y)^{2n} + (c_z+td_z)^{2n} \\ &+ ((e_x+tf_x)^2 + (e_y+tf_y)^2)^n + (e_z+tf_z)^{2n}\,. \end{aligned}$$

Es wurde bereits für alle Summanden gezeigt, dass ihre Ableitungen streng monoton steigend sind. Also ist auch die Ableitung von $s_{qz}(p(t))$ streng monoton steigend. Folglich lässt sich der restliche Beweis analog zum vorletzten Abschnitt führen.

6.4.6 Zusammemfassung

In den letzten Abschnitten wurde ein Verfahren Kollisionserkennung für durch Superquadriken beschriebene Zylinder und Quader beschrieben.
Um zwei dieser Körper auf Kollision zu untersuchen werden die die Körper definierenden Funktionen addiert und das globale Minimum dieser Summe bestimmt.

Über den Wert dieses Minimums lässt sich entscheiden, ob die Körper kollidieren, wobei ein Bereich besteht, in den Kollisionen erkannt werden, die keine sind. Dieser Bereich lässt sich jedoch durch Wahl des Parameters n beliebig verkleinern. Weiterhin ist sichergestellt, dass keine wirklichen Kollisionen nicht erkannt werden.
Es wurde gezeigt, dass die Bestimmung des globalen Minimums durch numerische Verfahren problemlos möglich ist, da die entstehenden Funktionen über ein eindeutiges Minimum verfügen.

Kapitel 7

Graphische Darstellung der Animation

7.1 Auswahl einer 3D-Graphik-Programmierumgebung

In diesem Abschnitt werden verschiedene Programmierumgebungen auf ihre Eignung zur Lösung der Aufgabe untersucht. Dabei sollen die folgenden Kriterien berücksichtigt werden:

- Echtzeitfähigkeit: Echtzeitfähigkeit (vgl. Abschnitt 3.3) bedeutet in diesem Kontext die Einhaltung weicher Echtzeitanforderungen. Wichtig ist die Frage, ob sich die Umgebung generell zur Erzeugung flüssiger Bewegtbilder eignet.
- Plattformunabhängigkeit: Es soll untersucht werden, auf welchen Betriebssystemen die Umgebung verfügbar ist. Von Interesse sind Linux und die 32-Bit-Versionen von Windows.
- Integration in Benutzungsoberfläche: Hier soll geklärt werden, wie leicht oder schwer es ist, die Programmierumgebung in graphische Benutzungsoberflächen zu integrieren. Von besonderem Interesse ist dabei, ob die Ausgabe ausschließlich in speziellen Graphikfenstern erfolgt oder ob es möglich ist, die Ausgabe mit anderen Elementen einer Benutzungsoberfläche zu kombinieren.
- Zukunftstauglichkeit: Es soll geklärt werden, ob die Umgebung noch weiterentwickelt und gepflegt wird.
- Szenenrepräsentation, z. B.

  - interne Verwaltung der Szene
  - Möglichkeiten zur Manipulation der Szene
  - verfügbare graphische Primitive

- graphische Fähigkeiten, z. B. Darstellen von
  - Transparenzeffekten
  - Licht
  - Schatten

- Programmierkonzepte: Hierbei soll geklärt werden, welche Art der Programmierschnittstelle die Umgebung verwendet.

Im Folgenden werden zunächst die Bibliotheken Maverik, OpenSG und Sigma bzgl. der oben genannten Kriterien untersucht. Im Anschluss daran wird eine Entscheidung darüber getroffen, welche der Bibliotheken zur Implementierung verwendet werden soll.

7.1.1 Maverik

Die Beschreibung von Maverik folgt der Darstellung in [CH02].
Maverik, der „**MA**nchester **V**irtual **E**nvi**R**onment **I**nterface **K**ernel“ ist eine Programmierbibliothek für Anwendungen im Bereich virtuelle Realität. Dabei stehen sowohl Funktionen zur Darstellung als auch zur Interaktion mit der dargestellten Umgebung bereit. Maverik wurde an der Universität Manchester entwickelt.
Maverik steht unter verschiedenen Unix- und Linux-System wie auch unter Windows (ab 98) und MacOS zur Verfügung. Zur Darstellung der Graphik verwendet es verschiedene Graphikbibliotheken wie z. B. OpenGL, IrisGL und DirectX.
Die Repräsentation der darzustellenden Umgebung erfolgt nicht innerhalb von Maverik. Statt dessen muss die Szene in der Anwendung aufgebaut und verwaltet werden. Durch einen Registrierungsmechanismus wird Maverik mit den einzelnen Objekten der Szene verbunden. Die Idee hinter diesem Vorgehen ist, dass der Anwendung kein Repräsentationsschema aufgezwungen wird, und dass gleichzeitig die Szene nicht doppelt, nämlich in der Anwendung und in Maverik, verwaltet werden muss. Allerdings erhält Maverik durch dieses Vorgehen keine Information darüber, in welcher Beziehung die einzelnen Objekte der Szene zueinander stehen: Jedes Objekt wird durch seine absolute Lage beschrieben.
Zur Darstellung einer Szene wird diese von der Anwendung aus einzelnen Objekte aufgebaut. Dabei stehen eine Vielzahl geometrischer Grundformen wie Quader, Pyramide, Zylinder, Kugel und Polygon bereit. Zur Darstellung komplexerer

Objekte stehen u. a. Polygongruppen und zusammengesetzte Objekte bereit. Polygongruppen erlauben mehrere Polygone zu einem Objekt zusammenzufügen, zusammengesetzte Objekte gestatten es, mehrere Objekte zu einem Objekt zusammenzufassen, wobei die Lage der Teilobjekte relativ zum Gesamtobjekt angegeben wird. Durch einen speziellen Mechanismus ist es möglich, weitere Klassen bereitzustellen. Neben der Form der Objekte ist es auch möglich, deren Farbe, Material und Textur zu bestimmen. Ebenso wie die Objekte werden auch Lichtquellen der Szene hinzugefügt.
Maverik ist in der Lage, transparente Objekte darzustellen.
Nachdem die Szene von der Anwendung aufgebaut wurde, kann Maverik dazu verwendet werden, die Szene aus verschiedene Perspektiven darzustellen. Dazu öffnet und verwaltet Maverik ein oder mehrere Fenster, in denen die Ausgabe auf verschiedene Arten (z. B. perspektivische und orthogonale Projektion) erfolgen kann. Es existieren verschieden Module, die ein interaktives Navigieren in der Szene mittels Maus oder Tastatur gestatten.
Maverik stellt dem Anwender eine prozedurale C-Schnittstelle bereit. Viele Ansätze, z. B. das Erzeugen neuer Klassen geometrischer Objekte, sind dabei objektorientiert, wobei enstprechende Mechanismen über Callback-Funktionen simuliert werden.
Die letzte veröffentlichte Version von Maverik trägt die Nummer 6.2 und ist vom 29. März 2002. Ob Maverik weiter gepflegt und gewartet wird, ist nicht bekannt.

7.1.2 OpenSG

Die folgende Beschreibung von OpenSG orientiert sich an der Darstellung in [Rei02] und [RBV].
OpenSG ist eine objektorientierte Szenengraphen-Bibliothek, die am Fraunhofer-Institut für Graphische Datenverarbeitung in Darmstadt entwickelt wurde. Sie ist für verschiedene Unix- und Linux-Systeme sowie für Windows verfügbar. Zur Darstellung wird OpenGL verwendet.
Zur Repräsentation der Szene verwendet OpenSG einen Szenengraphen. Der eigentliche Graph ist dabei azyklisch und gerichtet, d.h. jeder Knoten des Graphen (mit Ausnahme der Wurzel) hat genau einen Elternknoten. Um Informationen an verschiedenen Stellen im Graphen verwenden zu können, wird zwischen den Knoten des Graphen und den „Kernen“ der Knoten unterschieden: Die Knoten bilden die Struktur des Graphen und verweisen auf die Kerne, die die eigentliche Information tragen. Dabei können mehrere Knoten auf den gleichen Kern verweisen.
OpenSG stellt eine große Anzahl von Kernen bereit, die alle wesentliche Belange zur Darstellung einer Szene beschreiben. Es existieren unter anderem Kerne zur Beschreibung von Kameras, Lichtquellen, Transformationen, Schaltern und natürlich von Geometriedaten.

Lichtquellen und Kameras habe dabei die Eigenschaft, alle Knoten, die sich unterhalb von ihnen im Baum befinden zu beleuchten bzw. abzubilden. Sowohl Lichtquellen als auch Kameras können in der Szene frei bewegt werden. Dazu wird ein Verweis auf einen beliebigen Knoten der Szene angegeben, der die Lage von Lichtquelle bzw. Kamera beschreibt. Es stehen Kameras für verschiedene Darstellungsarten zur Verfügung.
Transformationen werden intern als 4x4-Matrizen repräsentiert und beschreiben die relative Transfomation zum übergeordneten Knoten. Es existieren verschieden spezialisierte Transformationskerne, die für bestimmte Manipulationen optimierte Schnittstellen bieten. Natürlich ist es auch möglich, einfach die Matrix neu zu setzen.
Schalter erlauben es auszuwählen, welcher Unterbaum des Knotens dargestellt werden soll.
Geometriedaten werden als Polygonnetze gespeichert. Es existieren verschiedene Funktionen, die es gestatten, Geometriedaten für Standardformen wie Kugel, Zylinder, Quader und Kegel zu erstellen.
Geometriedaten können mit einer Materialbeschreibung verknüpft werden, die angibt, wie das jeweilige Objekt dargestellt werden soll, wobei eine Materialbeschreibung von mehreren Objekten verwendet werden kann. Zur Beschreibung der Materialien können neben der Farbbeschreibung zusätzlich auch Texturen verwendet werden. OpenSG unterstützt die Darstellung transparenter Objekte.
Bei der Entwicklung von OpenSG wurde viel Wert auf Erweiterbarkeit gelegt. Dabei wurde ein Konzept entwickelt, das es nicht nur gestattet, die Bibliothek um neue Typen von Knotenkernen zu erweitern, sondern auch die verwendeten Darstellungsalgorithmen zu verändern. Dazu kommen verschiedene objektorientierte Techniken wie Ableitung und Templates zum Einsatz. OpenSG wurde in C++ entwickelt und stellt eine entsprechende Programmierschnittstelle bereit.
Da OpenSG für parallele Anwendungen entwickelt wurde, setzt es interne Replikationsmechanismen ein, um die Daten des Szenengraphen konsistent zu halten. Um diese Mechanismen zu verwenden, müssen alle Manipulationen an Objekten des Graphen durch spezielle Befehle eingeleitet und beendet werden. Hierdurch ist es möglich, den Szenegraphen gleichzeitig zu verändern, während aus einem anderen Thread heraus die Bildschirmdarstellung neu generiert wird.
OpenSG erlaubt es, mehrere Ausgaben verschiedener Kameras in verschiedenen Fenstern oder Fensterbereichen zu erzeugen. Zur Verwaltung der Fenster existieren zwei Möglichkeiten: Man kann spezielle Adapterklassen verwenden, die die Fenster und den OpenGL-Kontext vewalten, oder die Anwendung verwaltet die Fenster selbst und lässt vom Szenengraphen zu einem geeigneten Zeitpunkt die OpenGL-Ausgabe erzeugen, so dass die Ausgabe in jedem beliebigen Kontext erfolgen kann.
Die aktuelle Version von OpenSG hat die Nummer 1.2.0 und wurde am 19. März

2003 veröffentlicht.

7.1.3 Sigma

Die folgende Beschreibung von Sigma orientiert sich an der Darstellung in [HHA] und [AHH97].

Sigma, das „**S**ystem for **I**nteractive **G**raphics in **M**athematical **A**pplications“ ist eine Bibliothek zur Programmierung von dreidimensionaler Graphik, die am Institut für höhere Mathematik und Numerische Mathematik der Fakultät Mathematik der TU München entwickelt wurde.

Ziel bei der Entwicklung von Sigma war es, unabhängig von den unterliegenden Darstellungsalgorithmen zu sein. Aus diesem Grund stellt Sigma eine einheitliche Schnittstelle zu verschiedenen tiefer liegenden Graphikbibliotheken wie OpenGL oder PEX bereit, für die jeweils ein eigenes Modul bereitgestellt wird. Dabei ist es möglich, gleichzeitig mehrere Darstellungsverfahren einzusetzen.

Zur Verwaltung der darzustellenden Szenen und der zugehörigen Ressourcen stellt Sigma eine globale Datenbank bereit. Mit dieser Datenbank werden die Ressourcen zentral für alle Darstellungsmodule verwaltet.

Sigma erlaubt die flexible Beschreibung von Materialien und Beleuchtungsmodellen. Dazu ist es möglich, eigene Beleuchtungsalgorithmen über die Registrierung entsprechender Funktionene in das System zu integrieren.

Zur Beschreibung graphischer Objekte stellt Sigma eine Beschreibungssprache bereit. In dieser Sprache definierte Objekte werden zur Laufzeit in das System eingebunden, so dass es möglich ist, die Objekte zu verändern, ohne das System neu übersetzen zu müssen.

Da Sigma seit längerem nicht gewartet und weiterentwickelt wird, treten zunehmend Schwierigkeiten bei der Verwendung von Sigma auf neueren Systemen auf (vgl. [Hel03]).

7.1.4 Vergleich und Auswahl

Die beschriebenen Bibliotheken verfolgen sehr unterschiedlichen Ansätze zur Repräsentation und Darstellung einer Szene. Im Folgenden werden die vergleichbaren Eigenschaften tabellarisch aufgeführt, im Anschluss daran die Unterschiede diskutiert.

	Maverik	OpenSG	Sigma
Unterstützte Plattformen	Unix, Linux, Windows, Mac-OS	Unix, Linux, Windows	im Sourcecode vorhanden, Übersetzen auf neueren Systemen problematisch
Unterstützte Graphikbibliotheken	OpenGL, DirectX	OpenGL	OpenGL, PEX, RenderMan
Programmierschnittstelle	prozedural, in C	objektorientiert, in C++	prozedural, in C
Repräsentation der Szene	keine interne; greift auf Anwendungsdaten zurück	Szenengraph	war aus der vorhandenen Literatur nicht zu entnehmen
Graphische Primitive	viele Grundformen; durch Polygonnetze und Komposition beliebig erweiterbar	Polygonnetze; Funktionen zum erzeugen von Grundformen	verschiedene Primitive wie Kugel, Kegel, Prisma, Polygonnetze, Rotationskörper etc.

Tabelle 7.1: Vergleich zwischen Maverik, OpenSG und Sigma

Durch die Verwendung von OpenGL sind alle angegebenen Bibliotheken prinzipiell in der Lage, bewegte Bilder in Echtzeit zu erzeugen. Auch bei den graphischen Fähigkeiten zeigen sich keine schwerwiegenden Unterschiede, lediglich für Sigma fehlt die Information, ob die Bibliothek mit transparenten Objekten umgehen kann.
Betrachtet man die Verfügbarkeit der Bibliotheken für verschiedene Plattformen, treten die ersten Unterschiede zu Tage: Sigma wird seit längerem nicht gewartet, so dass hier Probleme beim Verwenden der Bibliothek abzusehen sind. Maverik ist zwar für viele Plattformen verfügbar, die letzte veröffentlichte Version ist aber fast eineinhalb Jahr alt, so dass auch hier die Weiterentwicklung fraglich erscheint. Die Internetseite der Entwickler macht leider keine Angaben über weitere geplante Entwicklungen oder die Einstellung des Projekts. OpenSG ist die neueste der betrachteten Bibliotheken, die aktuelle Version ist nicht ganz sechs Monate in Betrieb, aber auch hier schweigt sich die Internetseite der Entwickler über weitere Entwicklungen aus.
Aufgrund der Wartungssituation wird Sigma von der weiteren Betrachtung ausgeschlossen.

Vergleicht man die Art der Szenenrepräsentation von Maverik und OpenSG, zeigen sich sehr unterschiedliche Konzepte: Maverik versucht, dem Anweder keinerlei Vorgaben zu machen und greift auf dessen Datenstrukturen zurück. OpenSG hingegen verwendet eine Baum zur internen Repräsentation der darzustellenden Szene. Durch diese Strukturierung verwaltet OpenSG nicht nur die darzustellenden Objekte, sondern ermöglicht ein leichtes Manipulieren ganzer zusammengehöriger Teilbäume.

Da sich die darzustellenden Mehrkörpersysteme gut auf die Szenenrepräsentation von OpenSG abbilden lassen, bietet die anwendungsbezogene Repräsentation von Maverik keine Vorteile. Stattdessen lassen sich sogar die internen Mechanismen von OpenSG optimal einsetzen, um der Anwendung, z. B. bei der selektiven Darstellung von Elementen der Szene, Arbeit abzunehmen.

Aus diesem Grund wird für die Implementierung der Visualisierung OpenSG verwendet.

Bei der Untersuchung ist aufgefallen, dass keiner der Entwickler Aussagen über die weitere Pflege der Bibliotheken macht. Dies unterstreicht die Wichtigkeit der in dieser Arbeit vorgestellten modularen Architektur, da dadurch ein leichtes Austauschen der Visualisierungskomponenten möglich wird.

7.2 Implementierung der Darstellung

Zur Implementierung der Darstellung wird OpenSG verwendet. Dazu wird eine Unterklasse `COSGGLView` über `CGLRoboView` von `CRoboView` abgeleitet, die alle Aufrufe an OpenSG verkapselt und in das in Abschnitt 4.4.5 beschriebene MVC-Framework integriert ist.

7.2.1 Die Basisklasse `CRoboView`

Die Klasse `CRoboView` verwaltet eine Menge von Objekte des Typs `CRoboViewOptions`, in denen jeweils festgelegt ist, auf welche Art die Szene dargestellt werden soll. Zur Verwaltung dieser Elemente wird das `vector`-Template der STL eingesetzt. Zum Setzten der Optionen wird die Methode `setRoboViewOptions` verwendet.

In einem Objekt vom Typ `CRoboViewOptions` werden folgende Daten gespeichert:

Name	Inhalt
`camera`	Nummer der zu verwendenden Kamera
`showShape`, `showEnvelope`, `showKOO`	Schalter die festlegen, welche Elemente der Szene angezeigt werden sollen
`rightBorder`, `leftBorder`, `topBorder`, `bottomBorder`	Größe des anzuzeigenden Bildes relativ zur Fenstergröße und auf die linke untere Ecke bezogen

Tabelle 7.2: Darstellungsoptionen für die Roboteransicht in `CRoboViewOptions`

Weiterhin stellt `CRoboView` die Methoden `enableRecording` und `disableRecording` bereit, durch die das Aufnehmen der erzeugten Bilder an- und abgeschaltet werden kann. Beide Methoden sind virtuell und müssen in abgeleiteten Klassen überschrieben werden. Beim Aktivieren der Aufnahme mit `enableRecording` werden dabei die Nummern von Start- und Endbild sowie ein Namensanfang für die abzuspeichernden Bilder übergeben. In der abgeleiteten Klasse müssen dann aus dem Namensanfang die kompletten Bildnamen (z. B. mit Nummern) generiert werden.

7.2.2 Integration der OpenGL-Ausgabe

OpenSG verwendet OpenGL zum Erzeugen der graphischen Ausgabe. Dazu werden Folgen von OpenGL-Funktionsaufrufen von der Software generiert. Diese müssen jedoch in einem speziellen Kontext erfolgen, in dem z. B. das Ausgabefenster festgelegt ist. Da das MVC-Framework unabhängig vom der Oberfläche des Gesamtprogrammes ist, dürfen aus `COSGGLView` nicht zu beliebigen Zeitpunkten OpenGL-Funktionsaufrufe ausgeführt werden. Stattdessen sendet das Objekt eine Nachricht an den Programmcode für die Benutzungsoberfläche und dieser ruft seinerseits die OpenGL-Methoden von `COSGGLView` auf, sobald die notwendigen Vorbereitungen wie z. B. das Aktivieren des richtigen OpenGL-Kontextes erfolgt sind.
Hierzu wird ein Callback-Mechanismus verwendet. Dieser wird in der von `CRoboView` abgeleiteten Klasse `CGLRoboView` implementiert. Dieser Callback-Mechanismus schlägt eine Brücke zwischen MVC-Framework und Benutzungsoberfläche. Dazu wird die Klasse `CGLWindowAdapter` verwendet. In dieser Klasse wird die virtuelle Methode `initRedisplay` definiert. Für eine konkrete Anwendung muss eine Unterklasse von `CGLWindowAdapter` abgeleitet werden, in der diese Methode überschrieben wird. In der überschriebenen Methode muss das Neuzeichnen des Fensters mit der Ansicht initialisiert werden. Wie dies genau zu geschehen hat, ergibt sich aus der verwendeten Benutzungsoberflächen-Bibliothek (vgl. Abschnitt 8.3).

Nachdem die Anwendung das Neuzeichnen der Ansicht vorbereitet hat, muss sie die Ausgabemethoden von `CGLRoboView` aufrufen. Das Zeichnen erfolgt dabei in zwei Schritten. Zunächst muss der Zeichenvorgang mit `prepareGenerateGL` vorbereitet werden. In dieser Methode sollen alle Berechnungen durchgeführt werden, die von der eigentlichen Ausgabe unabhängig sind. Im Anschluss daran wird die GL-Ausgabe mit `generateGL` tatsächlich generiert. Die beiden Methoden wurden deswegen getrennt, um es zu erlauben, die gleiche Szene mehrfach auszugeben (z. B. in verschiedenen Fenstern mit unterschiedlichen Ausgabeoptionen), ohne jedesmal die vorbereitenden Berechnungen ausführen zu müssen. Sowohl `prepareGenerateGL` als auch `generateGL` sind virtuelle Methoden und müssen beim Ableiten einer neuen Ansichtsklasse überschrieben werden.
`COSGGLView` wird nicht direkt von `CRoboView`, sondern von `CGLRoboView` abgeleitet.

7.2.3 Repräsentation der Szene in OpenSG

OpenSG verwendet einen azyklischen, gerichteten Graphen zur Repräsentation der Szene. Die inneren Knoten enthalten dabei üblicherweise Transformationen und andere Daten, die die Darstellung der Szene beeinflussen (z. B. Schalter, zum Aktivieren und Deaktivieren einzelner Teilgraphen), die Blätter enthalten die eigentlichen Geometriedaten und deren Materialeigenschaften (z. B. Farbe und Textur). In OpenSG sind die eigentlichen Knoten, die immer nur einmal im Graphen auftauchen, und ihr Inhalt (als „Kern“ bezeichnet) getrennt. Hierdurch wird es möglich, in einem azyklischen Graphen die gleichen Geometriedaten an mehreren Stellen zu verwenden: Das gleiche Geometrieobjek bildet den Kern verschiedener Knoten. Eine andere Anwendung dieses Konzeptes ist es, mehrere Teilgraphen gleichzeitig zu schalten: Der Wurzelknoten jedes dieser Teilgraphen enthält einen Verweis auf das gleiche Schalterobjekt, durch dessen Manipulation sich alle Teilgraphen gleichzeitig beeinflussen lassen.
Im Folgenden wird beschrieben, wie der Szenengraph in `COSGGLView` aufgebaut ist.

Elemente der Mehrkörpersysteme

Für jedes Element eines Mehrkörpersystems wird ein eigener Teilgraph generiert. Dieser beginnt mit einem Transformationknoten, dessen Inhalt sich aus den Steuerdaten der Szene ergibt. Dieser Inhalt wird immer angepasst, wenn sich die Lage des Elements verändert. Da auf den Inhalt dieses Knotens zur Laufzeit zugegriffen werden muss, wird er nicht nur innerhalb des Graphen gespeichert, sondern zusätzlich ein Zeiger in `COSGGLView` abgelegt, um direkte Manipulationen zur Laufzeit zu ermöglichen.

Unterhalb dieses Transformationknotens und einem darauf folgenden Gruppierungsknoten befinden sich drei Teilbäume für die Darstellung des Körpers, des Hüllkörpers und des Hauptkoordinatensytems des Elements. Jeder dieser Teilbäume besteht dabei aus einem Schalter, einer Transformation und dem eigentlichen Geometrieobjekt. Die Schalterknoten aller Körper verweisen auf einen gemeinsamen Schalterkern, über den es möglich ist, die Anzeige aller Körper global an- oder abzuschalten. Analog wird mit den Schaltern für die Hüllkörper und Koordinatensysteme verfahren. Die Transformationsknoten enthalten die in der Szenenbeschreibung angegebene festen Transformationen der Körper bzw. Hüllkörper, ihr Inhalt verändert sich nach dem Aufbauen des Graphen nicht mehr. Die Inhalte der Geometrieknoten werden ebenfalls entsprechend der Szenenbeschreibung erzeugt und nach dem Aufbauen es Graphen nicht mehr verändert. Lediglich die Geometriobjekte der Hüllkörper werden auch außerhalb des Graphen gespeichert, um zur Laufzeit die Farbe des ihnen zugeordnete Materials verändern zu können, wenn Kollisionen angezeigt werden sollen.
Um neu Darstellungsoptionen zu realisieren können weitere Teilgraphen unterhalb des Gruppierungsknotens eingehängt werden.
Die folgende Abbildung zeigt den Teilgraphen für ein Element eines Mehrkörpersystems.

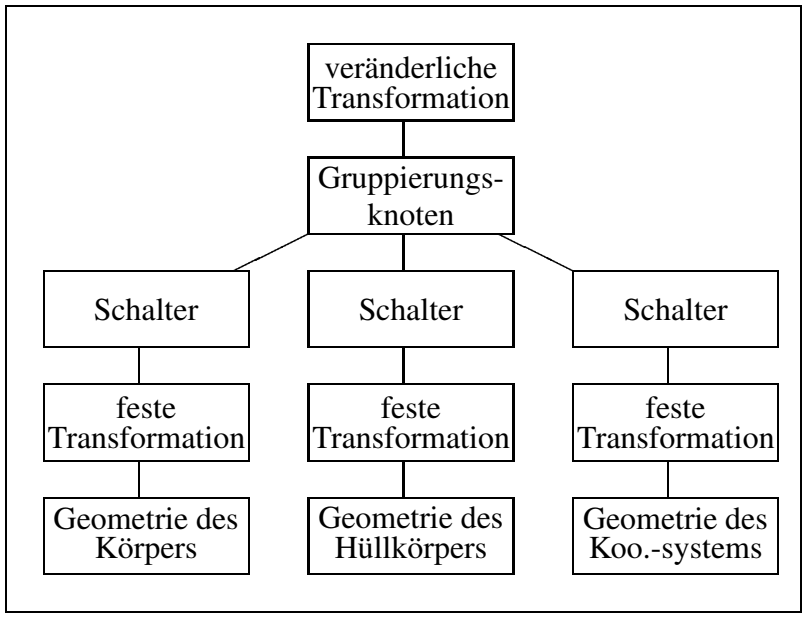

Abbildung 7.1: Teilgraph für ein Element eines Mehrkörpersystemes

Die Teilbäume werden von der Methode `generateSceneElementRek` erzeugt, die die Szenenbeschreibung rekursiv durchläuft. Zum Erzeugen der Äste für die

einzelnen Körper wird die Methode `shapeToNode` verwendet, die aus der Formbeschreibung in `CShape`, wie sie in der Szenenbeschreibung für Körper und Hüllkörper vorliegt, eine Folge von OpenSG-Knoten erzeugt. Ebenso wird mit Hilfe der Methode `makeOSGMaterial` aus der Materialbeschreibung in einem `CMaterial`-Objekt ein OpenSG-Materialobjekt erzeugt.
Folgende Datenobjekte stehen innerhalb von `COSGGLView` zur Verfügung, um die von einem einmal aufgebbauten Szenengraphen generierte Ansicht zu manipulieren:

`elementTrf`	Feld mit Objekten vom Typ `osg::TransformPtr`, die auf die veränderlichen Transformationen der einzelnen Körper verweisen
`switchShowShape`	Schalter zum globalen An- und Abschalten aller Körper in der Ansicht
`switchShowEnvelope`	Schalter zum globalen An- und Abschalten aller Hüllkörper
`switchShowKoo`	Schalter zum globalen An- und Abschalten der Koordinatensysteme
`switchShowAux`	Schalter zum globalen An- und Abschalten der Zusatzdaten
`envelopeGeometry`	Array mit Objekten vom Typ `osg::GeometryPtr`, die auf die Geometrieobjekte der einzelnen Hüllkörper verweisen

Tabelle 7.3: Datenobjekte zum Manipulieren bestehender Szenen

Alle Schalter sind dabei vom Typ `osg::SwitchPtr`, die Arrays werden mit Hilfe des `vector`-Templates der STL realisiert.

Geometrie der Körper

Zur Darstellung der geometrischen Objekte der Körper werden die von OpenSG bereitgestellten Knotenkerne vom Type `OSGGeometry` verwendet. Ein solches Objekt kann beliebige Geometrieinformationen aufnehmen. OpenSG stellt einen Satz von Hilfsfunktionen bereit, die Standardformen wie Ellipsoide, Quader, Zylinder, Kegel und Kegelstümpfe erzeugen.
Beim Erzeugen der Geometrieknoten werden dabei nur Objekte, die genau in einen Würfel der Seitenlänge 1 passen, erzeugt. Die tatsächliche Größe der Objekte in der Darstellung wird dadurch erreicht, dass die feste Transformation des Körpers mit einer entsprechenden Skalierung verknüpft wird.

Material der Körper

Das Material der Körper wird in OpenSG durch drei Farbwerte (jeweils als RGB-Tripel) für diffuse, ambiente und spekuläre Reflexionen, einen alpha-Wert für die Transparenz sowie eine optionale Textur beschrieben. Alle diese Informationen werden im Graphen durch Objekten vom Typ `OSGSimpleTexturedMaterial` dargestellt.
Die Erzeugung der OpenSG-Materialbeschreibung erfolgt in der Methode `make-OSGMaterial`. In dieser Funktion werden die Werte aus dem ursprünglichen RGBA-Tupel der Materialbeschreibung (vgl. 5.1.1) wie folgt generiert: Die Farbwerte für ambient und diffus reflektiertes Licht werden mit dem vorgegebene RGB-Wert gesetzt, als Farbwert für spekulär reflektiertes Licht wird Weiss (1,1,1) verwendet, um natürlich aussehende Glanzeffekte zu erzeugen. Für den alpha-Wert wird der in der A-Anteil des RGBA-Tupels der Materialbeschreibung verwendet. Wird der Methode explizit ein alpha-Wert übergeben, wird eine Materialbeschreibung generiert, die die gewünschte Transparenz hat und nicht texturiert ist.
Wird einem solchen Objekt keine Textur zugewiesen, wird der zugehörige Körper mit der vorgegebenen Farbe dargestellt.
Wird eine Textur angegeben, bietet OpenSG mehrere Möglichkeiten, wie diese mit dem Farbwert verrechnet werden soll. Für die gegebene Anwendung wurde dabei der Modus ausgewählt, in dem zunächst die Beleuchtungsrechung für den zu zeichnenden Bildpunkt mit dem reinen Farbwert ausgeführt wird und im Anschluss daran dieser Farbwert elementweise die Farbwerte der Textur moduliert. Durch dieses Vorgehen wird die Textur der Beleuchtungsrechnung unterworfen, was bei der Alternative, nämlich die Textur direkt auf dem Körper aufzubringen, nicht der Fall ist.
Die Texturen selbst müssen rechteckige Bilder sein, deren Seitenlängen Potenzen von 2 sind. Dabei können verschiedene Werte für die Seiten verwendet werden.
Um Texturen auf einem graphischen Objekt aufzubringen, muss für jedes Polygon des Objekts dessen Lage in der Textur angegeben werden. Dies geschieht dadurch, dass jedem Eckpunkt des Polygons eine Position innerhalb der Textur zugewiesen wird. Diese Abbildung ist bei den verwendeten Körpern durch OpenSG bereits fest vorgegeben. Die folgenden Abbildungen zeigen, wie eine Beispieltextur auf die einzelnen Körper abgebildet wird:

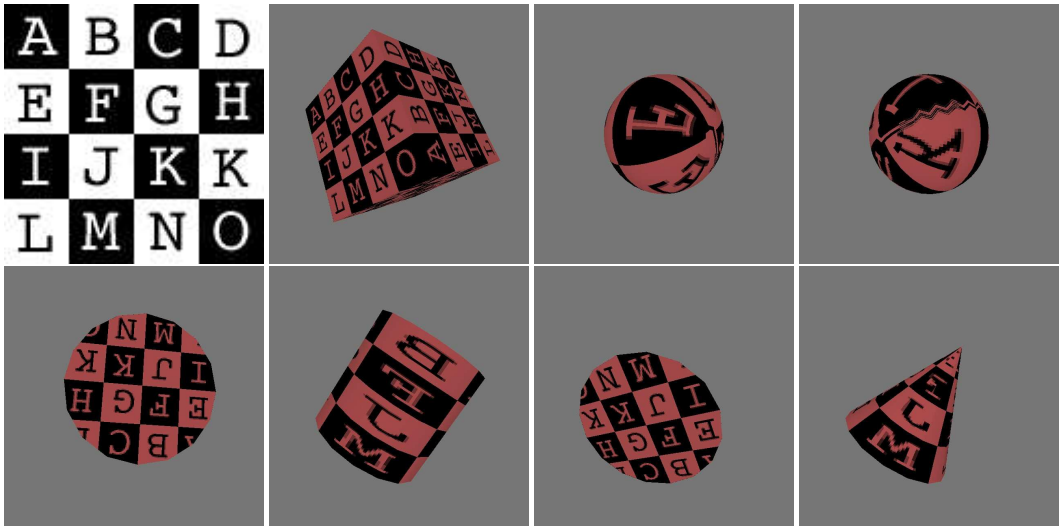

Abbildung 7.2: Die Abbildung oben links zeigt eine Textur, die auf die auf verschiedene Körper mit der Grundfarbe rot aufgetragen wird. Bei der Kugel (obere Reihe, 3. u. 4. Bild von links) wird deutlich, dass die vorgegebene Texturkoordinaten nicht die gesamte Textur abbilden, gleichzeitig entsteht ein unschöner Saum an der einen Seite der Kugel. Bei Zylinder (untere Reihe, 1. u. 2. Bild von links) und Kegel (untere Reihe, 3. u. 4. Bild von links) wird ein kreisförmiger Ausschnitt der Textur auf Boden und Deckel aufgebracht, der restliche Körper wird in die Textur „eingewickelt“, wobei die Textur seitenverkehrt widergegeben wird, was an den Buchstaben B, F und J deutlich wird.

Lichtquellen und Kameras

In OpenSG stellen Kameras nur diejenigen Objekte dar, die unterhalb ihres Knotens im Graphen abgespeichert sind. Genauso beleuchten Lichtquellen nur diejenigen Objekte der Szene, die sich unterhalb von ihnen im Graphen befinden. Aus diesem Grund bilden die Kamera- und Lichtquellenknoten die obersten Objekte im erstellten Graphen: Auf die Wurzel folgt eine Kette von Kameraknoten, gefolgt von einer Kette von Lichtquellenknoten. Erst nach dem letzten Element dieser Kette spaltet sich der Graph in eine Menge von Teilgraphen für die einzelnen Elemente der Mehrkörpersysteme und die Zusatzdaten auf.

Da sowohl Lichtquellen als auch Kameras in der Szene beweglich sein sollen, wird jedes dieser Objekte mit einem Knoten des Graphen verknüpft, der die Position und Orientierung angibt. Diese Knoten können sich auch unterhalb der Kamera bzw. des Lichts im Graphen befinden.

Zusatzdaten

Für jedes graphische Objekt der Zusatzdaten wird ein Teilgraph erzeugt. Die Art des Teilgraphen variiert dabei je nach Objekttyp:

Objekt	Repräsentation
Gerade	Transformation für Anfangspunkt und Richtung gefolgt von Skalierung für die Länge, danach Geometriedaten für einen Zylinder fester Länge
Pfeil	wie Gerade; zusätzlich zweiter Ast für Spitze: Transformation für Anfang und Richtung der Spitze, danach Geometriedaten
Polygon	Geometriedaten, die das Polygon in Weltkoordinaten beschreiben

Tabelle 7.4: Repräsentation der Zusatzdaten

Da die einzelnen Typen von Zusatzdaten auf verschiedene Weisen repräsentiert werden, müssen für jeden Typ eigene Routinen zum Erzeugen und Manipulieren bereitgestellt werden. Dies geschieht auf objektorientiertem Weg über die Klasse `COSGGLViewAuxDataElement`. Diese Klasse stellt virtuelle Methoden zum Erzeugen und Manipulieren eines entsprechenden Teilgraphen bereit, die in den Unterklassen für die einzelnen Typen überschrieben werden müssen.
`COSGGLView` verwaltet ein Feld, in dem für jedes darzustellenden Zusatzdatum ein entsprechendes Objekt abgelegt wird. Beim Aufbauen des Szenengraphen werden in diesen Objekten die notwendigen Verweise auf die Elemente des Graphen gespeichert, die später manipuliert werden müssen.

7.2.4 Erzeugen der graphischen Ausgabe

Das Erzeugen der graphischen Ausgabe erfolgt in zwei Schritten: Zunächst werden aus den Steuerdaten für den aktuellen Zeitpunkt die Transformationen und andere veränderliche Daten berechnet und im Szenengraphen an den entsprechenden Stellen gespeichert. Im Anschluss daran wird der Szenengraph verwendet, um die OpenGL-Ausgabe zu generieren.
Eng mit der graphischen Ausgabe verknüpft ist die Funktion, die fertig dargestellte Szene als Bild abzuspeichern, um diese Einzelbilder später zu einem Film zu montieren.

Verändern des Szenengraphen

Um die Darstellung der Szene für einen bestimmten Zeitpunkt zu generieren, müssen im Szenengraphen folgende Daten aktualisiert werden:

- die Transformationen, die die räumliche Lage der Elemente der Mehrkörpersysteme bestimmen
- die Transformationen, die Materialien und ggf. auch die Form der graphischen Repräsentation der Zusatzdaten
- die Materialien der Hüllkörper zur Repräsentation verschiedener Zustände

Um die gewünschten Änderungen vornehmen zu können, werden Zeiger auf die entsprechenden Knoten, Knotenkerne und Materialobjekte des Szenengraphen außerhalb des Graphen gespeichert. Für jeden darzustellenden Zeitschritt werden die entsprechenden Informationen neu eingetragen. Alle Manipulationen am Szenegraphen werden in der Methode `prepareGenerateGL` ausgeführt.

Erzeugen der graphischen Darstellung und Abspeichern von Bildern

Um die graphische Darstellung zu erzeugen, wird die Methode `generateGL` verwendet. Diese Methode durchläuft die Liste der Darstellungsoptionen und erzeugt für jedes Optionsobjekt die entsprechende OpenGL-Ausgabe. Dazu wird für jedes Optionsobjekt die Methode `progGenerateGL` aufgerufen, die die jeweils notwendigen Szenenteile im Graphen an- bzw. abschaltet, den Fensterausschnitt und die gewünschte Kamera setzt und schließlich den Szenegraphen die Ausgabe generieren lässt.
Soll das so erzeugte Bild abgespeichert werden, wird anschließend noch die Methode `grabImage` aufgerufen, die den Szenengraphen veranlasst, den erzeugten Bildschirminhalt abzuspeichern.
Dazu wird ein spezieller Bildvordergrund verwendet, den OpenSG mit der Klasse `OSGGrabForeground` bereitstellt. Dieser Vordergrund veranlasst den Szenengraphen, nach dem Erstellen der Ausgabe das erzeugte Bild abzuspeichern. Da der Szenengraph mehrfach aufgerufen werden kann, wird der `OSGGrabForeground` erst nach dem Zeichnen der letzten Ansicht aktiviert, wobei der gesamte Fensterinhalt als Zeichenbereich angegeben wird. Um das Speichern auszulösen, muss eine Zeichenoperation erfolgen. Dazu werden alle sichtbaren Objekte des Szenengraphen abgeschaltet, um keine der vorher generierten Bildinhalte zu zerstören.
Der zum Abspeichern der Einzelbilder verwendete Name wird dabei aus dem in `enableRecording` (vgl. Abschnitt 7.2.1) angegebene Namensanfang wie folgt erzeugt: Dem Namensanfang folgen die fünfstellige Nummer des Bildes, der Name der Ansichtsklasse und schließlich die Dateiendung für den erzeugten Dateitypen. Für den Namensanfang *XYZ* entsteht somit folgender Dateiname für das JPEG-Bild mit der Nummer 3: *XYZ* `_frame`*00003*`.osgglview.`*jpg*`.`
Nachdem alle Einzelbilder einer Sequenz erzeugt wurden, werden zusätzlich zwei Dateien erzeugt, um mit Hilfe von `transcode` aus den Einzelbildern eine Filmda-

tei zu erzeugen (vgl. Abschnitt 3.8). Die Dateien sind *XYZ*`.osgglview.imlist`, welche eine Liste aller Einzelbilder enthält, und *XYZ*`_makemovie.osgglview.sh`, welche den Aufruf für `transcode` mit allen notwendigen Parametern enthält.

7.3 POV-Ray-Export

Als Alternative zur echtzeitfähigen Bilderzeugung mit Hilfe von OpenSG und OpenGL steht ein weiteres Ausgabemodul bereit, das Steuerdateien für POV-Ray erzeugt. POV-Ray, der **P**ersistence **o**f **V**ision **Ray**tracer, ist ein frei verfügbares Raytracing-Programm (vgl. [PovDoc] und [Car96]). Durch den Einsatz von Raytracing lassen sich Bilder mit einer sehr viel höheren Qualität erzeugen, als dies mit OpenGL möglich ist (vgl. 3.7.1). Allerdings eignet sich Raytracing noch nicht für den Einsatz in Echtzeit[1].
Der POV-Ray-Export wird in der Klasse `CPRView` implementiert. Diese ist eine Unterklasse von `CRoboView`.
Ist die Aufnahem aktiviert, erzeugt `CPRView` eine Menge von Dateien, aus denen später mit POV-Ray die gewünschten Bilder erzeugt werden können. Die Dateinamen ergeben sich dabei aus dem Namensanfang, der in `enableRecording` angegeben wurde. Für den Namensanfang *XYZ* werden dabei, analog zum Vorgehen in `COSGGLView` (vgl. 7.2.4), folgende Dateien erzeugt:

- *XYZ*`_scene.prview.pov` enthält die POV-Ray-Beschreibung der Szene.
- *XYZ*`_frame`*nnnnn*`.prview.pow` enthält die Steuerparameter für das Bild *nnnnn* und wird für jedes der Bilder der Sequenz generiert.
- *XYZ*`.prview.imlist` enthält eine Liste mit den von POV-Ray erzeugten Bilddateien, die benötigt wird, um aus den Einzelbildern eine Filmdatei zu erzeugen.
- *XYZ*`_makemovie.prview.sh` enthält ein Shell-Script, das für jedes Bild der Sequenz POV-Ray aufruft und anschliessend die Einzelbilder zu einem Film montiert.

7.3.1 Zusammenwirken von Szenenbeschreibung und Bildparameterdateien

Werden mit `PRView` die POV-Ray-Dateien für eine Bildfolge generiert, so wird die eigentliche Beschreibung der Szene nur einmal erzeugt. Diese Beschreibungsdatei

[1] Diese Aussage gilt zumindest dann, wenn die Anwendung auf einem einzelnen PC ausgeführt werden soll. Es gibt jedoch erfolgreiche Versuche, Raytracing auf PC-Clustern in Echtzeit zu betreiben (vgl. [WBS02]).

kann jedoch von POV-Ray nicht direkt in ein Bild übersetzt werden, da an allen Stellen, an denen die Szene variabel ist (z. B. parametriesierte Transformationen bei Gelenken oder Farben bei Hüllkörpern) Variablen eingetragen sind, die in der Datei selbst nicht definert sind.
Diese Variablen werden für jedes einzelne Bild der Sequenz in der entsprechenden Parameterdatei abgelegt. Zusätzlich bindet die Parameterdatei die Szenenbeschreibung ein, so dass die Parameterdatei letztendlich von POV-Ray übersetzt werden kann.
Der Ansatz, Szenenbeschreibung und Parameter aufzutrennen, erlaubt es, die erzeugte Szenenbeschreibung vor dem Erzeugen der Bilder auf einfache Weise für alle Bilder zu verändern. Dies ist sinnvoll, wenn z. B. Materialien einzelner Elemente der Szene verändert werden sollen oder die Beleuchtung angepasst werden muss.

7.3.2 Aufbau der POV-Ray-Szenenbeschreibung

Die POV-Ray-Szenenbeschreibungsdatei wird aus der in `CSceneDescription` enthaltenen Szenenbeschreibung generiert. Sie enthält der Reihe nach folgende POV-Ray-Objekte:

- ein `camera`-Objekt, dessen Lage der ersten in der Szene definierten Kamera entspricht
- für jedes in der Szene definierte Licht ein `light_source`-Objekt
- für jeden in der Szene definierten Körper ein entsprechendes geometrisches Objekt

Die Orientierung und Position aller Objekte ist dabei nicht festgelegt, sondern wird durch eine Variablen pro Objekt beschrieben. Diese Variablen tragen die Bezeichnung `Trf`*X*`()`, wobei *X* die laufende Nummer des zugehörigen Körpers in der Szene ist. Sie beschreiben die absolute Lage des Hauptkoordinatensystems des jeweiligen Körpers.
Im Folgenden wird kurz beschrieben, wie die entsprechenden Objekte in der Datei repräsentiert werden. Es werden auch Hinweise gegeben, wie die Datei angepasst werden kann, um weitere optische Effekte zu erzeugen. Für die genaue Vorgehensweise beim Verändern und Erstellen von POV-Ray-Dateien sei hier auf die entsprechende Dokumentation verwiesen (vgl.[PovDoc])!

Kamera-Objekte

Kameras werden mit Hilfe des `camera`-Objekts von POV-Ray definiert. Die Kameras werden so definiert, dass sie sich im Zentrum eins rechtshändigen Koor-

dinatensystems befinden, entlang der negativen z-Achse gerichtet sind und die y-Achse im Bild nach oben zeigt.
Die erzeugten Kameraobjekte bilden die Szene mit einer perspektivischen Zentralprojektion ab. Für spezielle Anwendungen stellt POV-Ray eine Vielzahl von weiteren Optionen bereit, die an dieser Stelle manuell in die Datei eingetragen werden können.

Licht-Objekte

Lichtquellen werden mit Hilfe des `light_source`-Objekts definiert. Es werden die Standardlichtquellen von PovRay verwendet. Für spezielle Effekte ist es möglich, entsprechende Optionen manuell einzutragen.

Körper

Zur Darstellung der Körper werden die entsprechenden geometrischen Objekte verwendet, die POV-Ray bereitstellt. Da die Körper nicht zwingend im Ursprung des durch die Variablen definierten Koordinatensystems liegen, enthält die Beschreibung weitere Matrizen, die die zeitlich unveränderliche Transformation der Körper relativ zur durch die Variable angegebenen veränderlichen Lage bestimmen.
Zur Beschreibung der Oberfläche wird das `rgbf`-Objekt verwendet, das einfarbige, optional transparente Oberflächen beschreibt. Sollen andere Oberflächenstrukturen, z. B. Texturen, verwendet werden, können diese hier manuell eingefügt werden.

Erweitern der PovRay-Szenenbeschreibung

Für die dargestellten Körper, die Kameras und die Lichtquellen werden jeweils die Standardoptionen von PovRay verwendet. PovRay bietet sehr viele Möglichkeiten, die Darstellung zu beeinflussen. Hierzu existieren eine Vielzahl von Optionen, z. B. Kameras mit speziellen Abbildungseigenschaften, gerichtete Lichtquellen, spezielle Materialien zur Beschreibung der Oberflächen etc., die in der Dokumentation zu PovRay (vgl. [PovDoc]) ausführlich dargestellt werden.
Durch das Konzept, die gleich Beschreibungsdatei für den gesamten Film zu verwenden, ist es leicht, die Szene mit den gewünschten Optionen zu versehen.

7.3.3 Shell-Script zur Filmerzeugung

Um aus der Menge von Parameterdateien und der Szenenbeschreibungsdatei mit Hilfe von POV-Ray einen Film zu generieren, sind zwei Schritte notwendig:

- Erzeugen eines Bildes für jede Parameterdatei
- Montieren der Bilder zu einem Film

Zur Erledigung beider Aufgaben wird das Script *XYZ*`_makemovie.prview.sh` verwendet (vgl. Abschnitt 7.3). Dieses Script erzeugt zunächst die Bilder, indem es für jede Parameterdatei POV-Ray aufruft. Im Anschluss daran wird das Programm `transcode` zur Filmgenerierung verwendet, um die Menge von Bildern zu einem Film zusammenzufassen (vgl. Abschnitt 3.8 und 7.2.4).
Um Filme verschiedener Auflösungen erzeugen zu können, ist die Bildgröße im Script nicht fest kodiert, sondern wird in den ersten beiden Zeilen in den Variablen `width` und `height` festgelegt. Als Standardwerte sind hier 320 und 240 eingetragen. Diese Werte können frei variiert werden.

Kapitel 8

Erzeugen von Filmen

Um mit den oben beschriebenen Klassen Filme zu erzeugen oder in Echtzeit abzuspielen, müssen die einzelnen Klassen entsprechend der in Abschnitt 4.4.5 beschriebenen Architektur zusammengefügt werden. Um die Architektur zu vervollständigen, muss darüber hinaus noch eine konkrete Unterklasse von `CController` erzeugt werden, die für jedes Bild des zu erzeugenden Films die notwendigen Steuerdaten erzeugt. Die komplette Architektur wird dann in einer Klasse zusammengefasst, die unabhängig von einer umgebenden Anwendung ist. Diese Klasse wird dann mit einer graphischen Benutzungsoberfläche verbunden, um eine sinnvoll verwendbare Anwendung zu erhalten.

8.1 Steuerung der Animation

Zur Steuerung der Animation wird die von `CController` abgeleitete Klasse `CMovie` verwendet. Objekte dieser Klasse verwalten eine Menge von Steuerdaten, die sie aus einer Datei einlesen, und generieren auf Anfrage einen neuen Datensatz vom Typ `CFrameData`, der die Steuerdaten für den gewünschten Zeitpunkt enthält. Zum Manipulieren von Steuerdaten-Datensätzen stehen zusätzlich verschiedene Werkzeuge bereit.

8.1.1 Verwendetes Dateiformat

Die Steuerdaten werden im Klartext in einer Datei abgelegt. Die Datei enthält ausschließlich Fließkomma- und Integerzahlen. Zur Strukturierung der Datei ist es möglich, an beliebigen Stellen Zeilenumbrüche einzufügen, diese tragen jedoch keinerlei Information für das Programm.
Die Datei hat folgenden Aufbau:

- ein Fließkommawert, der die Länge eines Zeitschrittes in Millisekunden angibt
- drei ganzzahlige Werte n_m, n_z und n_h, die die Anzahl der Steuerwerte für die Mehrkörpersysteme, die Zusatzdaten und die Hüllkörper für jeden Zeitschritt angeben
- pro Zeitschritt $n_m + n_z$ Fließkommawerte, gefolgt von n_h ganzzahligen Werten

Die Anzahl der Zeitschritte ist beliebig. Die Fließkommawerte des Zeitschrittes geben die Steuerdaten für die Mehrkörpersysteme in der Reihenfolge der Gelenkdefinitionen an, gefolgt von den Steuerdaten für die Zusatzdaten. Die Integerwerte geben die Zustandsdaten für die Hüllkörper in der Reihenfolge der Körperdefinitionen an.
Die ganzzahligen Werte zur Steuerung des Aussehens der Hüllkörper werden dabei als Bitfelder interpretiert, deren niederwertigstes Bit bestimmt, ob der Hüllkörper den Zustand „kollidiert“ anzeigen soll. Alle weiteren Bits werden derzeit nicht ausgewertet und stehen für Erweiterungen zur Verfügung.

8.1.2 Die Klasse `CMovie`

Intern werden die Steuerdaten für die Animation in drei eindimensionalen Feldern verwaltet, wobei zusammengehörige Daten für einen Zeitschritt eine zusammenhängende Folge bilden. Zwei der Felder enthalten die Fließkommazahlen für die Steuerwerte der Mehrkörpersysteme und der Zusatzdaten. Das dritte Feld enthält die Integerwerte mit den Zustandsdaten der Hüllkörper.
Alle Felder werden mit Hilfe des `vector`-Templates der STL (vgl. 3.6) realisiert. Aus der Steuerdatei ist bekannt, wie viele Datenelemente für einen Zeitschritt benötigt werden. Benötigt ein Zeitschritt n Datenelemente, so bilden die Elemente 0 bis $n-1$ des Feldes die Daten für den ersten Zeitschritt, die Elemente n bis $2n-1$ die für den zweiten usw.
Die Zugriffszeit innerhalb des `Vector` ist dabei unabhängig von der Position, d.h. ein Auslesen der Daten ist unabhängig von der Position in konstanter Zeit möglich, so dass es möglich ist, in der Bildfolge hin und her zu springen.
Zusätzlich ist es möglich, beliebig viele Zwischenpositionen einer Animation zu erzeugen. Dazu wird im Datenelement `frameMultiplier` festgehalten, wie viele Zwischenschritte erzeugt werden sollen. Die Stuerwerte der Zwischenschritte werden jedoch nicht in den Datenstrukturen abgelegt, sondern erzeugt, wenn der entsprechende Zeitschritt dargestellt wird. Dadurch ist es möglich, die Anzahl der Zwischenschritte während dem Abspielen zu verändern.

Um die Animationsdaten eines beliebigen Zeitschrittes erzeugen zu lassen, wird die Methode `setFrame` verwendet. Diese leitet die gewünschte Position an die Methode `generateFrame` weiter, in der aus den Daten in `data` die Steuerwerte für den gewünschten Zeitschritt generiert werden. Die Steuerdaten für die Lage der einzelnen Elemente der Mehrkörpersysteme werden dabei linear zwischen den benachbarten vorgegebenen Werten interpoliert. Für Zusatzdaten und Hüllkörper wird hingegen immer der letzte eingetragene Wert verwendet, da Zusatzdaten nicht zwingend kontinuierliche Bewegungen beinhalten, so dass ein Interpolieren hier zu unerwünschten Effekten führen kann.
Stimmt die Anzahl der für jeden Zeitschritt gespeicherten Werte nicht mit der für den Datensatz benötigten Anzahl überein, so werden überschüssige Werte ignoriert bzw. fehlende mit Null angenommen. In beiden Fällen wird eine Warnmeldung am Bildschirm ausgegeben.
Zum Laden der Steuerdaten wird die Methode `loadMovie` verwendet. Die Methode erwartet die Daten im Format, das im vorigen Abschnitt definiert wurde.

8.2 Kern der Applikation

Der Kern der Applikation besteht aus Objekten der Klassen `CMovie`, `CApplicationModel`, `COSGGLView` und `CPRView`. Der zur Initialisierung notwendige Programmcode und je eine Instanz der vier Objekte werden zum Objekt `CMoviePlayer` zusammengefügt. Diese Klasse stellt Methoden zur Initialisierung, zur Ablaufsteuerung und zur Anbindung einer Benutzungsoberfläche bereit.
`CMoviePlayer` hat folgende schematische Struktur:

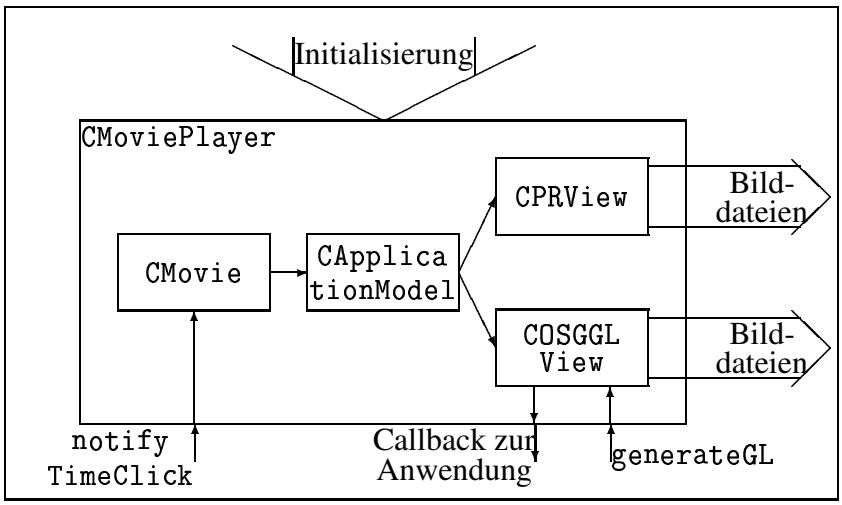

Abbildung 8.1: Aufbau und Kommunikation von `CMoviePlayer`

8.2.1 Initialisierung

Das Objekt `CMoviePlayer` stellt Methoden bereit, um die Anwendung mit den zur Filmererzeugung benötigten Daten zu initialisieren. Diese Daten sind die Szenenbeschreibung, repräsentiert durch ein Objekt der Klasse `CSceneDescription`, der Name der Datei mit den Steuerdaten und der Name mit der Datei mit den Darstellungsparametern.
Beim Initialisieren werden die Szenenbeschreibung und der Dateiname für die Steuerdaten an `CMovie` bzw. `CApplicationModel` weitergeleitet. Die Darstellungsparameter werden von `CMoviePlayer` selbst verarbeitet.
Zum Einlesen der Darstellungsparameter wird die Methode `loadRenderOptions` verwendet. Diese Methode liest aus einer Datei eine Menge von Darstellungsparametern ein. Für jede Zeile der Datei wird dabei ein Objekt vom Typ `CRenderOptions` erzeugt. Eine Zeile der Datei hat folgenden Aufbau:

- eine positive Ganzzahl für die Nummer der verwendeten Kamera
- vier Fließkommazahlen zwischen 0 und 1 zur Beschreibung der linke untere und rechte obere Ecke des Bildschirmausschnittes
- eine positive Ganzzahl, die die darzustellenden Elemente der Szene als Bitfeld kodiert

Im Bitfeld der Darstellungsoptionen haben die einzelnen Bits folgende Bedeutung:

Wertigkeit	Bedeutung
1	Anzeigen der Körper
2	Anzeigen der Hüllkörper
4	Anzeigen der Hauptkoordinatensysteme
8	Körper ohne Textur und semitransparent anzeigen
16	Zusatzdaten anzeigen

Tabelle 8.1: Bedeutung der Bits in den Darstellungsoptionen

8.2.2 Einbindung in eine Anwendung

`CMoviePlayer` muss in eine umgebende Anwendung integriert werden, die Bedienelemente und ein Graphikfenster bereitstellt. Um Graphikausgaben darstellen zu können, muss die Anwendung ein Objekt vom Typ `CGLWindowAdapter` bereitstellen, über das von `CMoviePlayer` ein Callback zur Anwendung ausgeführt werden kann, um die OpenGL-Ausgabe zu initialisieren (vgl. 7.2.2). Um dieses Objekt zu setzen, steht die Methode `setWindowAdapter` zur Verfügung. Ist die

OpenGL-Ausgabe initialisiert, muss die Anwendung ihrerseits `generateGL` aufrufen, um die OpenGL-Ausgabe durchzuführen.
Um `CMoviePlayer` frei von plattformabhängigem Code zu halten, enthält die Klasse keinen Code, der das Verstreichen der Zeit überwacht und automatisch neue Steuerdatensätze erzeugt. Statt dessen muss die Anwendung über die Methode `notifyTimeClick` das Objekt der Klasse `CMoviePlayer` über das Verstreichen der Zeit informieren. Durch dieses Vorgehen enthält `CMoviePlayer` keinen asynchronen Code, alle Aktivitäten laufen synchron innerhalb der Aufrufe durch die Anwendung ab.

8.2.3 Ablaufsteuerung des Films

Über die Methode `setTicksPerFrame` wird festgelegt, wie häufig ein neues Bild erzeugt werden soll. Intern werden die Aufrufe von `notifyTimeClick` gezählt und sobald die vorgegebene Anzahl von Aufrufen erfolgt ist, wird zum nächsten Bild weitergeschaltet und über das `CMovie` Objekt ein neuer Steuerdatensatz erzeugt. Die umgebende Anwendung kann jederzeit über `getCurrentFramenumber` feststellen, welches Bild gerade angezeigt wird.
Wie in Abschnitt 8.1.2 beschrieben, ist es möglich, Zwischenschritte im Film zu erzeugen. Die Anzahl der Zwischenschritte wird über die Methode `setFrameMultiplier` gesetzt, die diesen Aufruf an `CMovie` weiterleitet. Da die umgebende Anwendung keine Informationen über die geladene Steuerdatei hat, kann mit der Methode `getFrameCount` die Anzahl der Bilder erfragt werden.
Zur Steuerung des Filmablaufs durch die umgebende Anwendung stellt `CMoviePlayer` folgende Methoden bereit:

`play`	Starten der Filmwiedergabe
`stop`	Anhalten der Wiedergabe
`record`	Starten der Aufnahme unter Angabe von Dateinamen, Start- und Endbild
`setFrame`	Springen zu einem bestimmten Bild

Tabelle 8.2: Methoden von `CMoviePlayer` zur Kommunikation mit der Anwendung

8.3 Graphische Benutzungsoberfläche

Um eine vollständige Anwendung zu erzeugen, wird die im vorigen Abschnitt beschriebene Klasse in eine graphische Benutzungsoberfläche integriert. Die Benutzungsoberfläche hat dabei folgende Aufgaben:

- Starten und Stoppen der Wiedergabe
- Auswahl von Start- und Endbild eines zu erstellenden Films
- Auswahl des Bildformats
- Erzeugen von Bilddateien zur Filmerzeugung

Um das Programm einfach zu halten, wird darauf verzichtet, Szenenbeschreibungen, Steuerdaten und Darstellungsoptionen interaktiv zu laden. Die entsprechenden Dateien werden statt dessen als Parameter über die Kommandozeile angegeben. Dazu stehen folgende Parameter zur Verfügung:

- `R`*`Dateiname`* lädt die Beschreibung eines Mehrkörpersystems zur Szene. Diese Option kann mehrfach verwendet werden, die Systeme werden in der Reihenfolge der Kommandozeile geladen.
- `A`*`Dateiname`* lädt analog zu `R` die Beschreibung von Zusatzdaten.
- `M`*`Dateiname`* lädt die für den Film zu verwendende Steuerdatei.
- `O`*`Dateiname`* lädt die zu verwendende Datei mit Darstellungsoptionen.
- `C`*`Dateiname`* bzw. `S`*`Datiname`*) aktiviert die Kollisionserkennung für quaderförmige (vgl. Abschnitt 6.3) bzw. kugelförmige (vgl. Abschnitt 6.2) Hüllkörper und läd die auf Kollision zu testenden Paare von Körpern (vgl. Abschnitt 6.1).
- `H`*`Höhe`* und `W`*`Breite`* bestimmen Breite und Höhe der Bildausgabe. Diese Werte gelten auch für die erzeugten Bilder.

8.3.1 Aufbau der Benutzungsoberfläche

Die Benutzungsoberfläche wird komplett in einem Fenster dargestellt. Der obere Teil des Fensters dient der Darstellung der graphischen Ausgabe, im unteren Teil sind die Bedienelemente angeordnet.
Zur Steuerung des Ablaufs der Animation stehen Schaltflächen zum Starten, Anhalten und Rücksetzen bereit. Über einen Schieberegler ist es möglich, beliebige Bilder der Folge direkt anzufahren. Der Schieberegler wie auch ein Zahlenfeld zeigen die Nummer des aktuell gezeigten Bildes an.
Über zwei weitere Regler ist es möglich, die Bildrate und die Anzahl der interpolierten Zwischenbilder einzustellen.
Zum Erzeugen von Filme stehen weitere Regler bereit: Start- und Endbild des zu erzeugenden Filmes werden über zwei Schaltflächen festgelegt, die die Nummer

des jeweils aktuell angezeigten Bildes übernehmen. Über eine weitere Schaltfläche wird die Aufnahme gestartet. In einem Textfeld ist es möglich, den Namen für die zu erzeugenden Dateien anzugeben. Über vier Schalter wird ausgewählt, von welchem Typ die erzeugten Bilder sein sollen, wobei das Speichern in den Formaten jpeg, png und tiff sowie der Export nach POV-Ray zur Auswahl stehen.

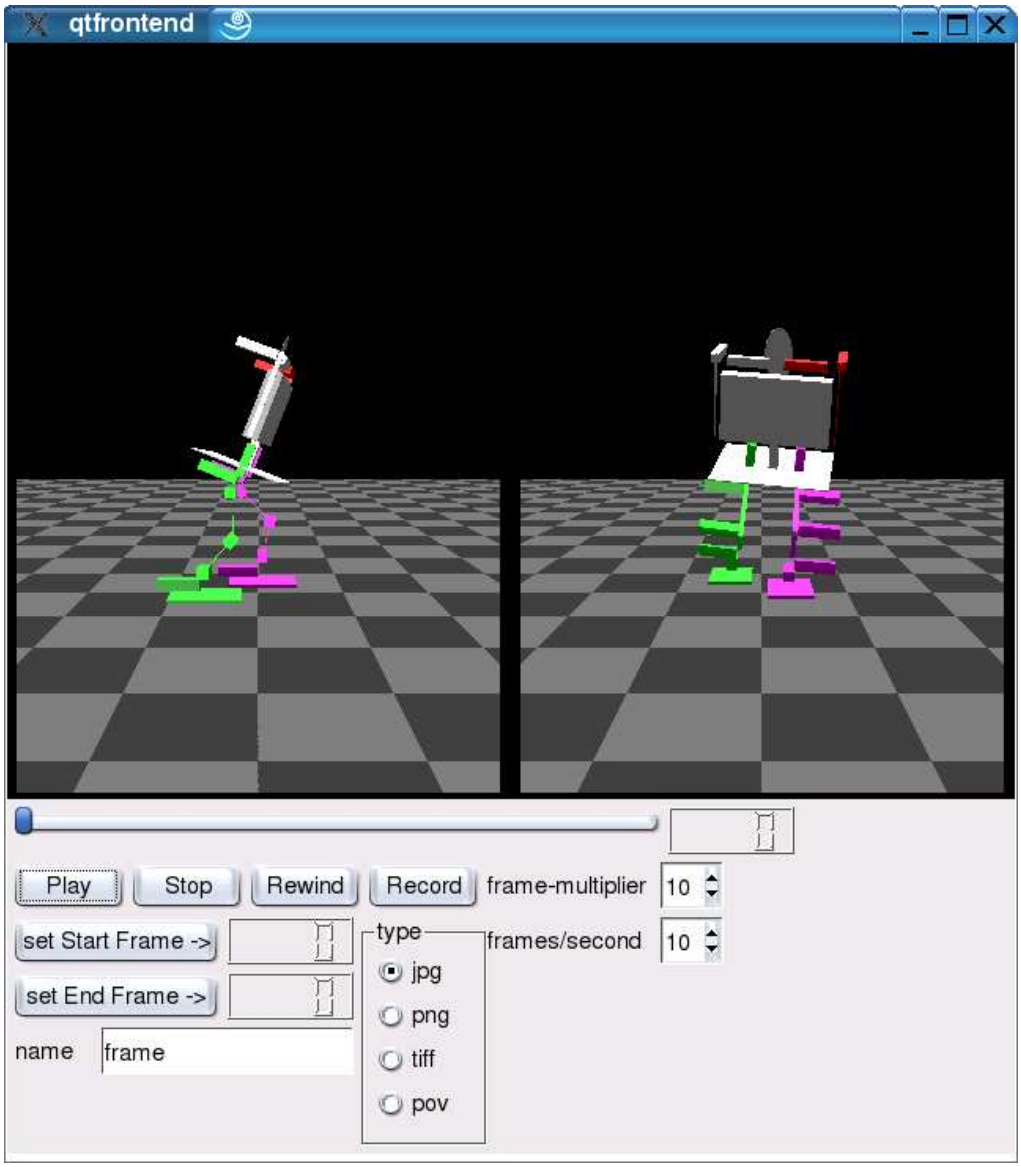

Abbildung 8.2: Die Abbildung zeigt die Benutzungsoberfläche im Einsatz. Das dargestellte Bild ist eine Visualisierung des humanoiden Roboters, der als Gemeinschaftsprojekt der Informatik der TU Darmtadt, Fachgebiets Simulation und Systemoptimierung und der Elektrotechnik der TU Berlin, Fachgebiet Regelungstechnik, entsteht.

8.3.2 Implementierung der Benutzungsoberfläche

Zur Implementierung der Benutzungsoberfläche wird die QT-Bibliothek (vgl. [QT] und [Die00]) verwendet. Diese stellt eine Vielzahl von Klassen zur Darstellung verschiedener, als „Widgets“ bezeichneter Komponenten graphischer Benutzungsoberflächen bereit.
Zur Darstellung der graphischen Ausgabe wird die Komponente `QGLWidget` verwendet. Diese erlaubt es, in einem rechteckigen Fensterbereich eine Graphikausgabe mit OpenGL-Befehlen zu generieren. Dazu müssen lediglich die beiden virtuelle Methoden `resizeGL` und `paintGL` überschrieben werden, in denen auf Größenänderungen des Ausgabebereichs reagiert bzw. die OpenGL-Ausgabe ausgeführt wird. Diese Methoden werden von `QGLWidget` aufgerufen, wenn die entsprechenden Ereignisse eintreten. `QGLWidget` verbirgt dabei alle Aufgaben, die mit dem Intialisieren des OpenGL-Kontextes zusammenhängen.
Von `QGLWidget` wird die Klasse `mpglWidget` abgeleitet, die `CMoviePlayer` (vgl. Abschnitt 8.2) zum Generieren der OpenGL-Ausgabe verwendet.
Zur Darstellung der Bedienelemente werden verschiedene weitere Widgets wie z. B. `QRadioButton` für Schaltflächen oder `QSlider` für den Schieberegler verwendet. Die Bedienelemente werden in der Klasse `movieControllerWidget` zu einem großen Bedienelement zusammengefügt und mit Methoden dieser Klasse verknüpft. Wird ein Bedienelement zur Laufzeit verwendet, so werden automatisch die entsprechenden Methoden aufgerufen.
`movieControllerWidget` arbeitet eng mit der Klasse `CMoviePlayer` zusammen: Die durch die Bedienelemente aufgerufenen Methoden geben die entsprechenden Aufrufe und Parameter weiter an das Objekt der Klasse `CMoviePlayer`. Neben den sichtbaren Bedienlementen enthält `movieControllerWidget` zusätzlich ein Zeitgeber-Objekt der Klasse `QTimer`. Mit Hilfe dieses Zeitgebers wird der Ablauf der Filme in `CMoviePlayer` gesteuert.
Aus je einem Objekt der Klassen `mpglWidget` und `movieControllerWidget` wird die Benutzungsoberfläche generiert. Beim Aufbauen der Benutzungsoberfläche werden die beiden Objekte mit einem Objekt der Klasse `CMoviePlayer` verbunden.
Da `CMoviePlayer` über den Callback-Mechanismus von `CGLRoboView` (vgl. Abschnitte 7.2.2 und 8.2.2) die OpenGL-Ausgabe auslöst, wird eine passende Adapterklasse `myWindowAdapter` von `CGLWindowAdapter` abgeleitet. Diese Klasse verbindet `CGLRoboView` mit `mpglWidget` über die Methode `initRedisplay`. Wird diese Methode aus `CGLRoboView` aufgerufen, ergibt sich folgender Ablauf:

1. `CRoboView` teilt der Anwendung durch Aufruf von `initRedisplay` mit, dass eine neue Ausgabe generiert werden muss.

2. In `initRedisplay` wird die Methode `updateGL` von `mpglWidget` aufgeru-

fen, die in der Oberklasse `QGLWidget` implementiert ist.

3. Dort wird die graphische Ausgabe vorbereitet und im Anschluss daran die virtuelle Methode `paintGL`, die in `mpglWidget` implementiert ist, aufgerufen.

4. Von dort wird schließlich die Methode `generateGL` von `CMoviePlayer` aufgerufen, die die OpenGL-Ausgabe ausführt.

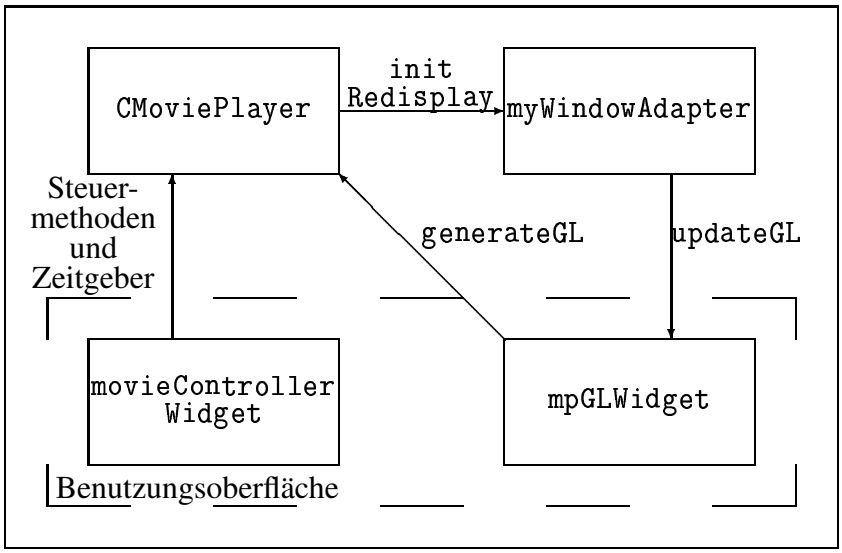

Abbildung 8.3: Interaktion zwischen Benutzungsoberfläche und `CMoviePlayer`

Kapitel 9

Echtzeitvisualisierung von Simulationsdaten

In diesem Abschnitt wird die Anwendung `roboRTView` vorgestellt, die mit Hilfe der Bibliothek `roboViewLib` erstellt wurde. Die Anwendung erlaubt es, ein oder mehrere Systeme in Echtzeit zu animieren. Dazu werden die Mehrkörpersysteme beim Aufruf von `roboRTView` über die Kommandozeile angegeben.
Nach dem Start liest das Programm die Steuerparameter für die Mehrkörpersysteme von der Standardeingabe und animiert die Darstellung entsprechend. Zum Übertragen der Daten wird ein Protokoll (vgl. Abschnitt 9.2)eingesetzt, das es gestattet, beliebig viele Parameter zu ändern, bevor die Darstellung geändert wird.
Durch Umlenken der Ausgabe eines beliebigen Programmes auf die Standardeingabe von `roboRTView` kann somit jedes Programm verwendet werden, um die Animation zu steuern, sofern es das entsprechende Protokoll zur Übermittlung der Steuerdaten einhält.

9.1 Implementierumg von `roboRTView`

Um die oben beschriebene Funktionalität zu implementieren, wird mit `CJoint-Controller` eine neue Unterklasse von `CController` bereitgestellt. Diese erlaubt es, über die Methode `setJoints` die Steuerparameter eines oder mehrerer Gelenke der Mehrkörpersysteme neu zu setzen.
Zur Visualisierung wird eine Instanz der Klasse `COSGGLView` verwendet. Diese wird über eine Instanz der Klasse `CApplicationModel` mit einer Instanz von `CJointController` verbunden.
Zum Einlesen der Steuerparameter von der Kommandozeile wird die Klasse `CKeyboardReaderThread` von `CThread` abgeleitet. Diese Klasse stellt einen Thread bereit, der die Eingabe von der Standardeingabe liest und entsprechend des im

nächsten Abschnitt vorgestellten Protokolls interpretiert. Dabei werden die Werte zunächst in einem Feld zwischengespeichert. Wurden alle zu ändernden Werte übertragen, werden die gesammelten Werte über die Instanz von `CJointController` geändert.
Zur Darstellung der mit OpenGL erzeugten Visualisierung wird die Bibliothek `GLUT` verwendet (vgl. [Kil96]). Diese gestattet es, auf einfache Art unter verschiedenen Betriebssystemen Fenster zur OpenGL-Ausgabe zu öffnen und einfache Anwendungen ohne graphische Benutzungsoberfläche zu erstellen. Die Bibliothek steht für verschiedene Betriebssysteme zur Verfügung, darunter Linux und Windows.
Zum Verwenden der `GLUT` Bibliothek müssen verschiedene Funktionen bereitgestellt und bei der Bibliothek registriert werden. Die Funktionen werden bei bestimmten Ereignissen wie dem Neuzeichnen des Fensters, dem Verstreichen einer bestimmten Zeitspanne oder Mausaktionen im Fenster aufgerufen. Nachdem diese Funktionen bei der Bibliothek registriert sind, wird `GLUT` aktiviert. Damit geht das Programm in eine Endlosschleife über, von der aus die registrierten Funktionen aufgerufen werden, wenn die entsprechenden Ereignisse eintreten.
Da das Einlesen der Standardeingabe in einem eigene Thread durchgeführt wird, besteht zunächst keine Möglichkeit, die Methode zum Neuzeichnen des Fensters aufzurufen, wenn dies notwendig wird. Dazu wird die Klasse `glutWindowAdapter` von `CWindowAdapter` abgeleitet. Über eine Instanz dieser Klasse kann `COSGGLView` das Neuzeichnen des Fensters veranlassen. Dazu wird ein interner Zähler in `glutWindowAdapter` um eins erhöht. Parallel dazu wird über eine bei der Bibliothek registrierte Funktion alle 20 ms überprüft, ob der Zähler seit der letzten Überprüfung erhöht wurde. Ist dies der Fall, wird ein Neuzeichnen ausgelöst.

9.2 Protokoll zum Steuern von `roboRTView`

Die Anwendung verarbeitet im Betrieb drei Befehlstypen, von der Standardeingabe eingelesen werden. Diese sind:

- `s` g w: Setze den Steuerwert von Gelenk g auf den Wert w.
- `d`: Führe die mit der letzten Folge von `s`-Befehlen angegebenen Änderungen aus.
- `q`: Beende die Anwendung.

Alle Befehle müssen mit einem „carriage-return“[1] abgeschlossen werden.

[1] Das ASCII-Symbol 13, auf der Tastatur die Eingabetaste.

9.3 Betrieb von roboRTView

Beim Starten von roboRTView werden die gleichen Kommandozeilenparameter verwendet wie beim roboMovieMaker (vgl. Abschnitt 8.3). Lediglich der Parameter M steht nicht zur Verfügung, da zur Visualisierung keine Steuerdateien sondern Echtzeitdaten verwendet werden.
Die Steuerdaten werden über die Standardeingabe des Programmes bereitgestellt. Dies ist beim Start von der Kommandozeile die Tastatur. Wird roboRTView aus einem anderen Programm heraus aufgerufen, kann das aufrufende Programm direkt auf die Standardeingabe von roboRTView zugreifen, um Steuerdaten zu übertragen.

Kapitel 10

Zusammenfassung

10.1 Ergebnisse

Die im Rahmen dieser Arbeit entwickelte Bibliothek `roboViewLib` dient der Visualisierung, Animation und Kollisionserkennung von Mehrkörpersysteme. Durch die gezielte Verwendung aktueller und plattformunabhängiger Bibliotheken soll ein möglichst langer und problemloser Einsatz von `roboViewLib` sichergestellt werden.
Zur Entwicklung kamen objektorientierter Programmiertechniken zum Einsatz. Dadurch ist ein flexibler, modularer Aufbau entstanden, der ein einfaches Austauschen einzelner Komponenten gestattet.
Die Flexibilität zeigt sich z. B. darin, dass es leicht möglich ist, verschiedene Komponenten zur Visualisierung zu implementieren. Die entstandenen Komponenten unterstützen so verschiedene Ansätze wie Echtzeitdarstellung mit OpenGL (vgl. `COSGGLView`, Abschnitt 7.2) und Export zu einer Raytracing-Software (vgl. `CPRView`, Abschnitt 7.3).
Zur Beschreibung der zu visualisierenden Systeme wird eine XML-Sprache eingesetzt. Diese Beschreibungssprache eignet sich besonders gut für laufende Roboter, ist aber auch für andere Mehrkörpersysteme geeignet. Die interne Repräsentation der Mehrkörpersysteme macht keine Annahmen über die Beschaffenheit der Systeme, so dass hier eine größtmögliche Flexibilität erreicht wird.
Neben der Visualisierung der Mehrkörpersysteme ist es möglich zusätzliche Informationen in Form von Punkten, Pfeilen und Geraden einzublenden.
Mit Hilfe der Bibliothek wurden zwei Anwendungen für verschiedene Aufgaben entwickelt.
Das Programm `roboMovieMaker` dient der Erstellung von Filmen aus vorgegebenen Trajektorien. Es erlaubt es, Animationen in Echtzeit abzuspielen. Zusätlich ist es möglich, die entstehenden Bilder abzuspeichern und daraus eine Filmdatei zu

erzeugen. Zur Erzeugung besonders hochwertiger Grafiken ist es auch möglich, Raytracing zu verwenden, was beim derzeitigen Stand der Technik auf einem einzelnen PC aber nicht in Echtzeit möglich ist.

Das Programm `roboRTView` dient der Visualisierung von Steuerdaten für Roboter und andere Mehrkörpersysteme in Echtzeit. Es kann z. B. dazu eingesetzt werden, Simulationsdaten direkt beim Entstehen zu Visualisieren. Durch die Integration von Komponenten zur Kollisionserkennung ist es dabei möglich, auftretende Kollisionen sofort zu erkennen.

Im Bereich Kollisionserkennung wurden zwei Verfahre implementiert, die es gestatten, Kollisionen von kugel- bzw. quaderförmigen Hüllkörpern zu erkennen.

Darüberhinaus wurde ein neues Verfahren hergeleitet, das es gestattet quader- und zylinderförmige Hüllkörper durch den Einsatz von Superquadriken anzunähern und einheitlich zu behandeln. Durch den Einsatz verschiedenen Formen für die Hüllkörper können diese besser an die Elemente der Mehrskörpersysteme angepasst werden, so dass eine präzisere Kollisionserkennung möglich wird.

10.2 Erzeugte Bilder

Die folgenden Abbildungen zeigen Bilder einer Animation eines humanoiden Roboters, die mit `roboMovieMaker` erzeugt wurden. Jede Folge enthält dabei die gleichen Positionen.

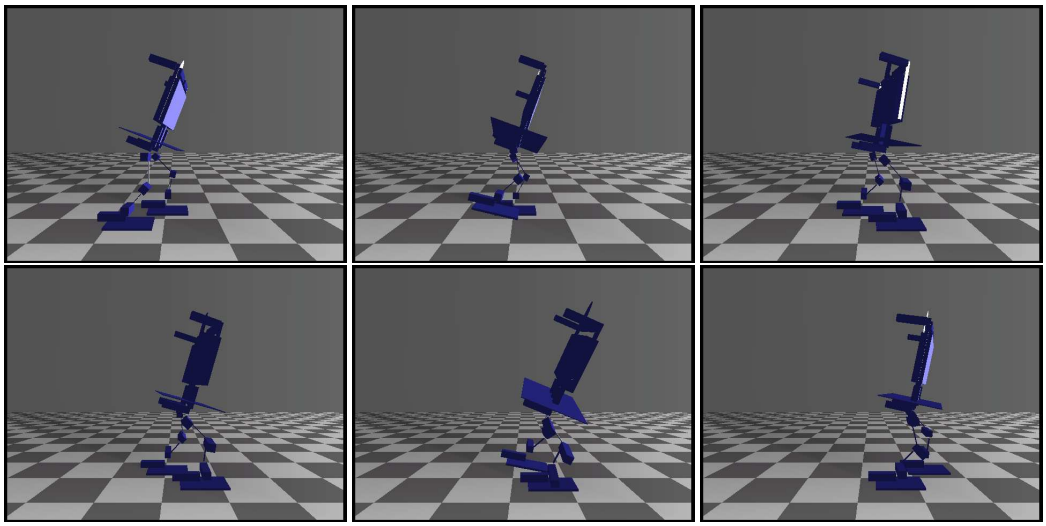

Abbildung 10.1: Humanoider Roboter in Seitenansicht

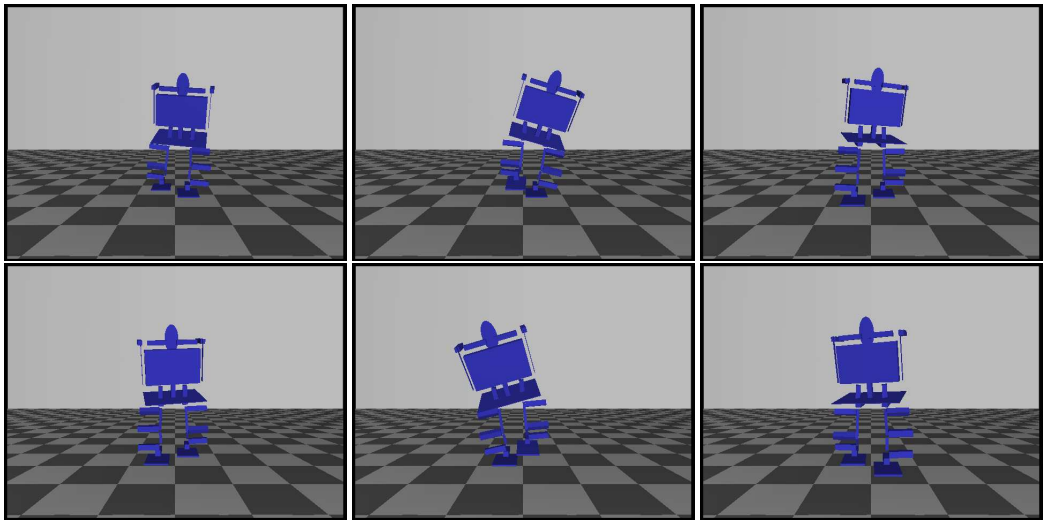

Abbildung 10.2: Humanoider Roboter in Vorderansicht

Die folgenden Abbildungen zeigen den humanoiden Roboter in Seitenansicht, einmal mit OpenGL und einmal mit PovRay erzeugt:

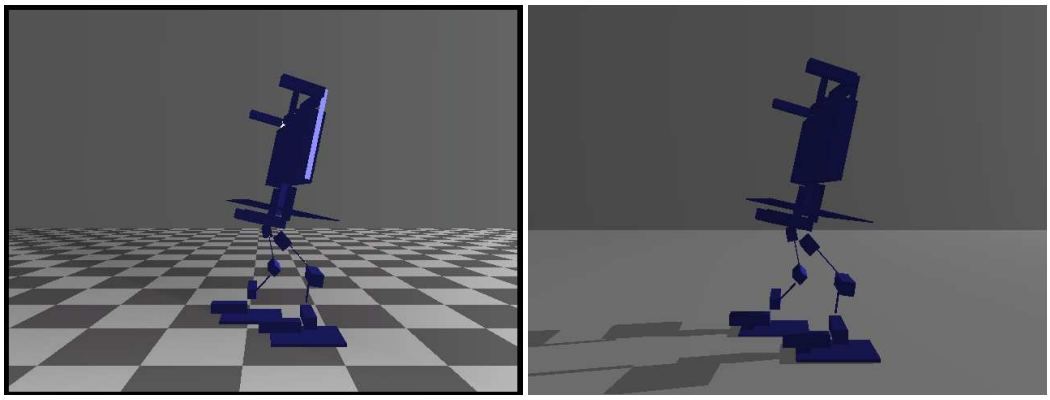

Abbildung 10.3: Die linke Abbildung zeigt eine mit `COSGGLView` und OpenGL erzeugte Visualisierung, die rechte Abbildung die gleiche Szene mit `COSGGLView` und POVRay visualisiert.

Auf den folgenden Abbildungen wird ein einfacher Manipulator mit der Echtzeitansicht `roboRTView` dargestellt. Dabei werden zwei Ansichten auf die gleiche Szene gezeigt. Zusätzlich ist die Kollisionserkennung für quaderförmige Hüllkörper aktiviert, wobei nur zwei Hüllkörper auf Kollision getestet werden[1]:

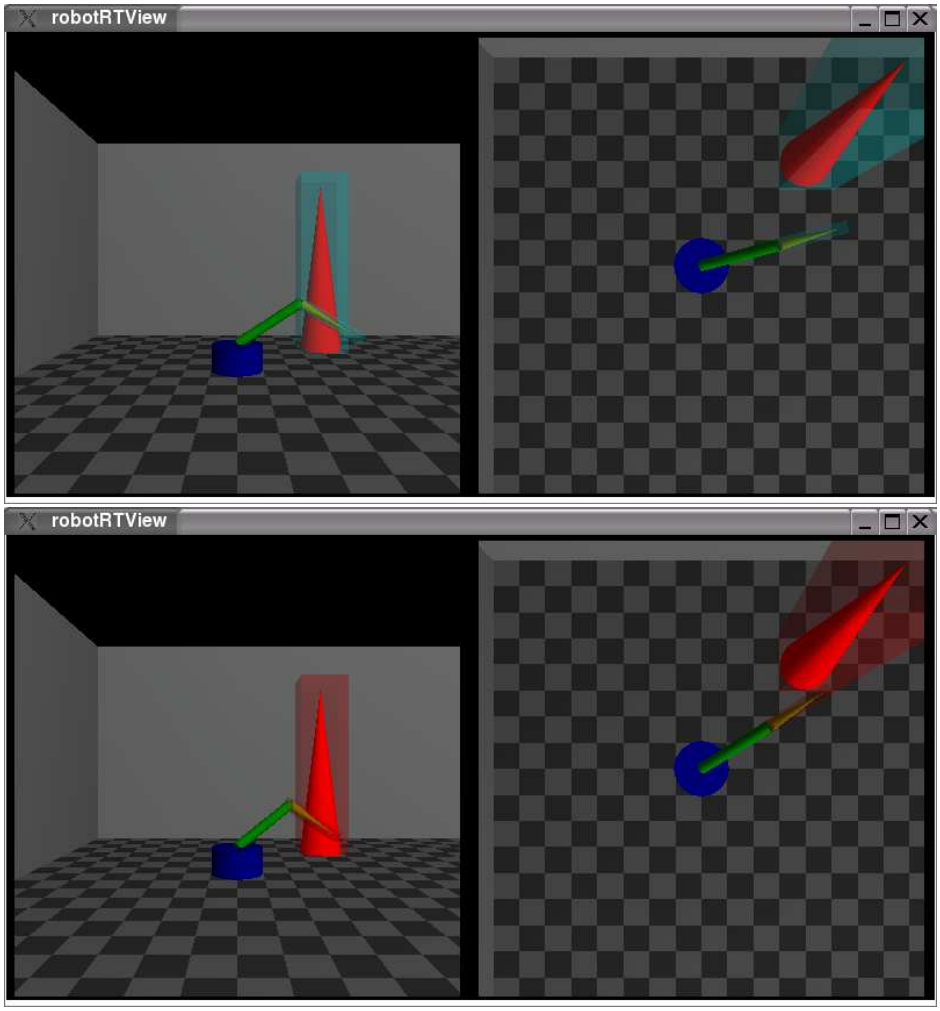

Abbildung 10.4: Die linke Abbildung zeigt eine Szene, in der Hüllkörpers des vorderstens Gliedes des Manipulators (die gelbe Spizte) fast den Hüllkörper des roten Kegels berührt. In der rechten Abbildung hat sich der Manipulater weitergedreht, so dass sich die Hüllkörper berühren. Dies wird durch eine Rotfärbung der Hüllkörper angezeigt.

[1]Zur Verdeutlichung wurden die anderen Hüllkörper ausgeblendet. Dies geschieht über das `hideBoundingVolume`-Tag (vgl. Abschnitt 5.2) in der Beschreibung der Roboters; ansonsten würden auch diejenigen Hüllkörper angezeigt, für die keine Kollisionserkennung durchgeführt wird.

10.3 Ausblick

Im Folgenden werden verschiedene Möglichkeiten gezeigt, die vorliegende Arbeit zu erweitern und forzuführen.

Verbesserung der graphischen Ausgabe

Durch ihre flexible, objektorientierte Architektur ist die Bibliothek `roboViewLib` offen für vielfältige Erweiterungen.
Besonders bei den Komponenten zur Visualisierung eröffnen sich hier vielfältige Möglichkeiten. Denkbar ist es z. B. die Echtzeitvisualisierung in der Klasse `COSGGLView` um weitere Ansichten zu erweitern. Besonders interessant wäre es, hier die Möglichkeit zu schaffen, statt der vorgegebene Körper Polygonnetze zu verwenden, um die Darstellung der Mehrkörpersysteme realistischer zu gestalten.
Ebenfalls von Interesse ist die Frage, ob es möglich ist, Echtzeit-Raytracing, wie es gerade für PC-Cluster entwickelt wird (vgl. [WBS02]), zur Visualisierung einzusetzen. Hierzu ist es leicht möglich, eine weitere Visualisierungkomponente in die Bibliothek einzufügen.

Kollisionserkennung

Das in Abschnitt 6.4 entwickelte Verfahren ist noch nicht in die Bibliothek integriert. Hierzu ist ein geeignetes numerischer Verfahren zur Bestimmung der benötigten Minima auszuwählen und das Verfahren mit dessen Hilfe zu implementieren.
Darüberhinaus stellen sich noch verschiedene Fragen bezüglich des Verfahrens, die im Rahmen dieser Arbeit nicht behandelt werden konnten:
Es ist zu prüfen, ob sich das Verfahren auch zur Kollisionsvermeidung eignet. Dazu muss untersucht werden, ob der Wert der berechneten Minima in einem verwertbaren Zusammenhang mit dem Abstand der Körper steht. Ist dies der Fall, ist es möglich, den Wert als Optimierungskriterium zur Bahnsteuerung heranzuziehen.
Weiterhin ist es von Interesse, ob sich weitere implizit beschriebene Körper finden lassen, die sich ebenfalls zur Kollisionserkennung eignen. Als konkreter Vorschlag seien hier Ellipsoide genannt, die sich mit Hilfe der Funktion

$$e_n(x,y,z) = (x^2 + y^2 + z^2)^n$$

beschreiben lassen.

Literaturverzeichnis

[AHH97] M. Alefeld, J. Haber, A. Heim: *Sigma: Ein System für interaktive Graphiken in mathematischen Anwendungen - Referenz der Funktionen und Datentype*, Institut für Höhere Mathematik und Numerische Mathematik, TU München, 1997

[Ale01] M. Alexa: *Unterlagen zur Vorlesung Graphische Datenverarbeitung 1*, gelesen von M. Alexa an der TU Darmstadt im Wintersemester 2001

[Ale02] M. Alexa: *Unterlagen zur Vorlesung Graphische Datenverarbeitung 2*, gelesen von M. Alexa an der TU Darmstadt im Sommersemester 2002

[Bag97] J. Bager: *HTML++: Style Sheets und XML - die Zukunft des Web?*, c't 08/1997, S. 298-303, Heise-Verlag

[Ber02] M. Bertuch: *Aufklärung in 3D*, c't 15/2002, S. 194-198, Heise-Verlag

[Ber03] M. Bertuch: *Standards für viruelle Welten*, iX 04/2003, S. 105-109, Heise-Verlag

[Beu98] A. Beutelspacher: *Lineare Algebra*, Vieweg, 3. Auflage, 1998

[BSMM95] I. Bronstein, K. Semendjajew, G. Musiol, H. Mühlig: *Taschenbuch der Mathematik*, Harri Deutsch Verlag, 2. Auflage, 1995

[Car96] C. Caroli: *Strahlenforscher*, c't 09/1996, S. 300-304, Heise-Verlag

[CH02] J. Cook, T. Howard: *Marerik Programmer's Guide for GNU Maverik version 6.2.*, http://aig.cs.man.ac.uk/maverik

[Cra89] J. Craig: *Inroduction to Robotics: Mechanics and Control*, Addison-Wesley, 2. Auflage, 1989

[Die00] O. Diedrich *Selbstgebaut: Toolkits zur Programmierung des X Windows Systems*, c't 01/2000, S. 156-163, Heise-Verlag

[ER94] H. Erlenkötter, V. Reher: *Programmiersprache C: Eine strukturierete Einführung*, Rowohlt Taschenbuch Verlag, 1994

[ESK96] J. Encarnação, W. Straßer, R. Klein: *Graphische Datenverarbeitung 1*, Oldenbourg, 4. Auflage, 1996

[ESK97] J. Encarnação, W. Straßer, R. Klein: *Graphische Datenverarbeitung 2*, Oldenbourg, 4. Auflage, 1997

[FDFHP94] J. Foley, A. van Dam, S. Feiner, J. Hughes, R. Phillips: *Grundlagen der Computergraphic: Einführung, Konzepte, Methoden*, Addison-Wesley, 1. Auflage, 1994

[Fla98] D. Flanagan *Java in a Nutshell, Deutsche Ausgabe für Java 1.1*, O'Reilly, 2. Auflage, 1998

[For96] O. Forster: *Analysis 1*, Vieweg, 4. Auflage, 1996

[For84] O. Forster: *Analysis 2*, Vieweg, 5. Auflage, 1996

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Desing Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1. Auflage, 1995

[HHA] J. Haber, A. Heim, M. Alefeld: *A new 3D Graphics Library: Concepts, Implementation, and Examples*, In: H.-C. Hege, K. Polthier (eds.), S. 211-225, 375-376, Springer-Verlag, 1997

[Hel03] A. Helm: *Entwicklung einer Umgebung zur Modellierung, Simulation und Animation von Laufmaschinen*, Projektbericht, Fakultät Informatik der TU München, 2003

[Kil96] M. J. Kilgard: *The OpenGL Utility Toolkit (GLUT) Programming Interface – API Version 3*, 1996, www.opengl.org/documentation/glut/index

[Kir02] T. Kirste: *Anmerkungen zur Rotation mit Quaternionen*, www.gris.informatik.tu-darmstadt.de/~tkirste/GDV-I/quatrot.pdf

[Ost03] T. Östreich. *Dokumentation zu* `transcode`, http://www.theorie.physik.uni-goettingen.de/~ostreich/transcode/

[PovDoc] *Dokumentation zu PovRay 3.5*, www.povray.org

[QT] *Dokumenation zu QT 3.1*, http://www.trolltech.no

[RBV] D. Reiners, J. Behr, G. Voss: *OpenSG Starter Guide Version 1.1.0*, www.opensg.org

[Rei02] D. Reiners: *OpenSG: A Scene Graph System for Flexible and Efficient Reltime Rendering for Virtual and Augmented Reality Applications*, Dissertation, TU Darmstadt, 2002

[Sch97] U. Schöning: *Theoretische Informatik – kurzgefasst*, Spektrum, 3. Auflage, 1997

[Sed93] R. Sedgewick: *Algorithmen in C*, Addison-Wesley, 1. Auflage, 1993

[STL] *Standard Template Library Programmer's Guide*, http://www.sgi.com/tech/stl

[Str92] B. Stroustrup: *Die C++-Programmiersprache*, Addison-Wesley, 2. Auflage, 1992

[Str01] O. v. Stryk: *Skriptum zur Vorlesung Robotik 1*, gelesen von O. v. Stryk im Sommersemester 2001 an der TU Darmstadt.

[SG01] H.-J. Siegert, U. Baumgarten: *Betriebssysteme – Eine Einführung*, 5. Auflage, Oldenbourg, 2001

[SG98] A. Silberschatz, P. Galvin: *Operating System Concepts*, Addison-Wesley, 5th Edition, 1998

[SWS02] J. Schmittler, I. Wald, P. Slusallek: *SaarCOR – A Hardware Architecture For Ray Tracing*, http://www.openrt.de/Publications

[Tan01] A. S. Tanenbaum: *Modern operating systems*, Prentice-Hall, 2. Auflage, 2001

[Tuc03] J. Tuchel: *Lebendes Objekt – Smalltalk: Ein aktueller Klassiker*, c't 02/2003, S. 188-193, Heise-Verlag

[WBS02] I. Wald, C. Benthin, P. Slusallek: *A Simple and Practical Method for Interactive Ray Tracing of Dynamic Scenes*, Technical Report 2002-04, http://www.openrt.de/Publications/index.html

[Wil98] R. Wille und Mitarbeiter: *Unterlagen zur Vorlesung Lineare Algebra*, gelesen von R. Wille an der TU Darmstadt im Wintersemester 1998

[WNDS99] M. Woo, J. Neider, T. Davies, D. Shreiner: *OpenGL Programming Guide*, Addison-Wesley, 3. Auflage, 1999

[XML1] *Dokumentation zu XML*, http://www.w3.org/XML

[XML2] *Dokumentation zur* `libXML`, http://www.xmlsoft.ort

Erklärung zur Diplomarbeit gemäß §19 abs. 6 DPAO/AT

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat ingleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. September 2003

Martin Friedmann