Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Bernd Pohlmann

Aktor-Sensor Simulation für Robotikanwendungen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Bernd Pohlmann

Aktor-Sensor Simulation für Robotikanwendungen

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Andreas Meisel
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 25. März 2013

Bernd Pohlmann

Thema der Arbeit
Aktor-Sensor Simulation für Robotikanwendungen

Stichworte
Simulation, Roboter, 3D-Modell, Rendering, OpenSceneGraph, OpenGL, Kinematik, Kollisionserkennung, Kameradarstellung

Kurzzusammenfassung
Diese Arbeit handelt von der Entwicklung einer 3D-Simulation für Roboter einschließlich ihrer Aktuatoren und Sensoren. Dabei werden zunächst die Rahmenbedingungen festgelegt und eine Architektur ausgewählt. Anschließend wird Schritt für Schritt die Virtualisierung eines Knick-Arm-Roboters vorgenommen und das Ergebnis visualisiert. Es wird die Umsetzung der Bewegungssteuerung erklärt, welche für die Aktor-Simulation benötigt wird. Zudem wird eine Methodik implementiert, welche die Simulation beliebiger realer Kameras erlaubt. Als ein weiterer Aspekt wird eine Kommunikationsschnittstelle via TCP/IP realisiert, um die Simulationsumgebung remote zu steuern und die generierten Bilddaten der Kamera zu empfangen und zu analysieren. Die einzelnen Module werden abschließend zusammengefügt, anhand einiger Beispiele getestet und die Performance analysiert.

Bernd Pohlmann

Title of the paper
Actor-Sensor Simulation for Robotics

Keywords
simulation, robot, 3D model, rendering, OpenSceneGraph, OpenGL, kinematic, collision detection, camera

Abstract
This document describes the development of a 3D simulation for robots with their actors and sensors. At first the general requirements are set and the architecture is chosen. After that a step-by-step virtualisation of an articulated robot is performed. The results are visualized and the kinematics are explained in detail. In addition the possibility to implement various kinds of different cameras is shown. A communication protocol via standard TCP/IP is established to allow remote control of the robot and to add the feature of image transfer to another client for analysis. The unitized software product is being configured and the performance is demonstrated in several examples.

Inhaltsverzeichnis

1 Einleitung

1.1 Stand der Technik

Die Robotik ist heute in vielen Lebenslagen allgegenwärtig, sei es als Haushaltsroboter zum Staubsaugen, als Spielzeug oder in der industriellen Fertigung. In den letzten Jahren ist insbesondere die Sparte der Assistenzroboter stark gewachsen. Ein aktuelles Beispiel dafür ist der Human Support Robot (HSR) aus dem *Partner Robot Programm*[1] von Toyota (Abb. 1.1.1a).

(a) Human Support Robot [Fal12] (b) PR2 [Gar13]

Abbildung 1.1.1: Assistenzroboter

Er wurde Mitte 2012 vorgestellt und sein primäres Aufgabenfeld ist die Unterstützung im Alltag für ältere, kranke und behinderte Menschen. Mit seiner mobilen Plattform, ausfahrbarem Oberkörper und einem Greifarm kann er bei vielen Aufgaben im Haushalt zur Hilfe gehen. Gesteuert wird er entweder über eine GUI oder ein Tablet. Er ist mit zahlreichen Sensoren zur Umgebungsanalyse ausgestattet, die es ihm erlauben, präzise zu navigieren und Gegenstände zu manipulieren. Er wird bereits seit 2011 erfolgreich im Yokohama Rehabilitationszentrum (Japan) getestet. [Fal12; Ack12] Ein weiteres Beispiel ist der PR2 (Abb. 1.1.1b) von [Gar13].

Diese Ausprägung der Robotik ist ein noch recht junges Aufgabengebiet und bringt neue Herausforderungen mit sich. Neben der bloßen Ansteuerung der Gelenkmotoren kommen zusehends mehr Sensoren zum Einsatz, die dem Roboter mehr Autonomie verleihen. Es können diverse Arten von Kameras, Ultraschall, Radar oder Laserscanner verwendet und miteinander kombiniert werden. Dies ermöglicht einerseits ein breiteres Einsatzgebiet, andererseits steigt sowohl die Komplexität als auch den Materialeinsatz. Eine Konsequenz dieser Entwicklung ist, dass durch den Einsatz teurer Roboterhardware in immer sensibleren Umgebungen der Fehlerfall weitreichende Auswirkungen haben kann.

[1] `http://www.toyota-global.com/innovation/partner_robot/`

Die Simulation ist das Mittel der Wahl, wenn es darum geht, solch komplexe Vorgänge zu erforschen und Situationen abzuwägen, bevor reale Experimente durchgeführt werden. Sie kann die Entwicklungszeit verringern und Probleme aufzeigen, bevor teure Hardware gebaut bzw. eingesetzt wird. Trotz oder grade aufgrund dieser Möglichkeit braucht es fortwährend ein Feedback echter Experimente, um sicherzustellen, dass die Simulation nicht missbraucht wird. Zum einen besteht nämlich die Gefahr, das Problem zu stark zu vereinfachen oder zu idealisieren bzw. das reale Rauschen, also die Umgebungseinflüsse, nicht zu beachten. Zum anderen kommt hinzu, dass man eventuell beginnt, die Simulation nach den erwarteten Ergebnissen auszurichten, womit sie ihr Ziel mit hundertprozentiger Sicherheit verfehlen wird. [Bro87]

Beispiele für Umgebungseinflüsse / Rauschen können Dinge wie z.B. Nebel oder Staubpartikel in der Luft sein, welche Kameras oder Ultraschallsensoren beeinträchtigen, oder aber durch Reibung oder Schlupf verursachte Ungenauigkeiten in der Bewegung des Roboters. Es gilt also, das zu simulierende System so zu abstrahieren und zu gestalten, dass diese Fehler unbedingt vermieden werden.

Die Simulation von Robotern hilft, Produktionsabläufe kostengünstig zu testen und zu optimieren. Wenn es jedoch zur Interaktion zwischen Mensch und Roboter kommt, steht in erster Linie die Sicherheit im Vordergrund. Auf der einen Seite muss sichergestellt sein, dass Personen im Arbeitsbereich des Roboters nicht verletzt werden, zum anderen muss jedoch Interaktion, wie das Übergeben eines Gegenstandes möglich sein.

Ein Projekt, welches sich genau dieser Problemstellung annimmt, ist das EXECELL Projekt des Fraunhofer Instituts für Fabrikbetrieb und -automatisierung. In seinem Rahmen wurden Verfahren und Technologien eingesetzt und weiterentwickelt, die zur Arbeitsraumüberwachung verwendet werden können. Die zwei Hauptkomponenten bilden zum einen Leichtbauroboter und zum anderen diverse am Institut für Produktionsanlagen und Konstruktionstechnik (kurz IPK) entwickelte, experimentelle Systeme, die in unterschiedlichen Szenarien den Arbeitsbereich des Roboters überwachen können. So soll eine physisch engere Zusammenarbeit zwischen Mensch und Maschine ermöglicht werden. [Fra12] In Abbildung 1.1.2 deutlich zu erkennen ist der berechnete Arbeits- und Gefahrenbereich (rot). Ein denkbares Szenario ist die Kooperation zwischen Mensch und Roboter in medizinischen Laboren.

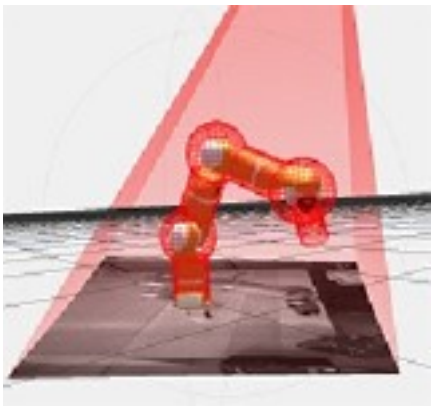


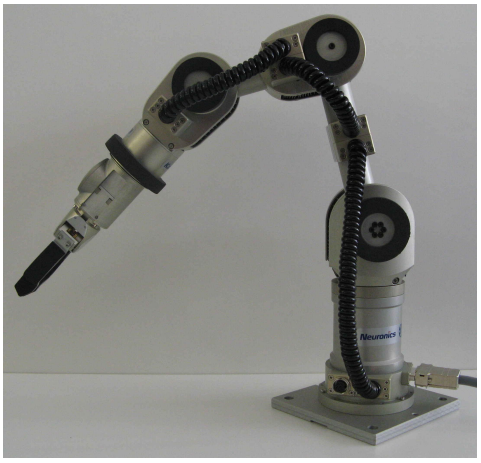
Abbildung 1.1.2: Arbeitsraumüberwachung [Fra12]

Das Thema Assistenzroboter bzw. Mensch-Roboter Interaktion ist noch immer Gegenstand der aktuellen Forschung und noch nicht auf dem Stand angekommen, dass Roboter ein selbstverständlicher Teil unseres Alltags sind, wie dies etwa bei Computern der Fall ist. Mithilfe der Simulation kann diese Arbeit enorm beschleunigt werden. Somit liegt auf der Hand, warum diese Masterarbeit sich mit dem Thema der Robotersimulation beschäftigt.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Simulation für Sensor-geführte Roboter. Diese soll die Möglichkeit bereitstellen, Algorithmen zur Steuerung von Aktoren/ Aktuatoren und Analyse von unterschiedlichen Sensoren prüfen zu können, ohne direkt mit einem physischen Roboter zu interagieren. Als Rahmenbedingungen vorgegeben sind einerseits die Interoperabilität mit Mathworks MATLAB und andererseits die möglichst realitätsnahe Visualisierung. Dies gilt insbesondere für die Simulation der Kameras und der von diesen getätigten Aufnahmen.

Das Einsatzgebiet der Anwendung soll die Forschung und Entwicklung im Bereich der Robotik an der HAW Hamburg sein. Durch sie soll paralleles Arbeiten an unterschiedlichen Projekten ermöglicht werden, da zur Zeit nur ein stationärer und ein mobiler Roboter vorhanden sind, wodurch einzelne Projekte nur nacheinander Experimente durchführen können. Basis für die Entwicklung dieser Arbeit ist der Katana 6M, ein stationärer Greifarm-Roboter des Herstellers Neuronics, in der Ausführung 6M180 mit einem Winkelgreifer als Werkzeug (Abb. 1.2.1a).



(a) Katana 6M180 mit Winkelgreifer [Neu08]

(b) Guppy F-080B/C mit Objektiv [AVT13]

Abbildung 1.2.1: Roboter und Kamera

Als Hauptsensor soll eine auf dem Endeffektor montierte Kamera dienen, welche einschließlich der von ihr aufgenommenen Bilder ebenfalls zu simulieren ist. Die zu diesem Zweck verwendete Kamera ist eine Guppy F-080B/C (Abb. 1.2.1b), welche maximal 30 Bilder pro Sekunde (fps) bei einer Auflösung von 1032 x 778 Pixeln erlaubt. Optional ist die Implementierung eines weiteren Sensors, beispielsweise zur Abstandsmessung oder Kollisionserkennung.

Folgende technische Details sind relevant: Zum einen die max. Bewegungsgeschwindigkeit von 90° pro Gelenk pro Sekunde $\widehat{=}$ 1m/Sekunde und zum anderen der max. Bewegungsradius von $\sim$ 0.6m. Für das virtuelle Greifen von Objekten, solange keine Physik mit simuliert wird, weniger interessant ist die Nutzlast, die mit montiertem Winkelgreifer und Kamera (ohne Objektiv) noch ungefähr 200 Gramm beträgt. Sie wird lediglich für die Verifikation der Simulationsergebnisse am realen Objekt benötigt.

Selbstverständlich müssen virtualisierte Alltagsgegenstände mit in die Simulation eingebracht werden können. Das Gesamtproblem wird in Abbildung 1.2.2 veranschaulicht.

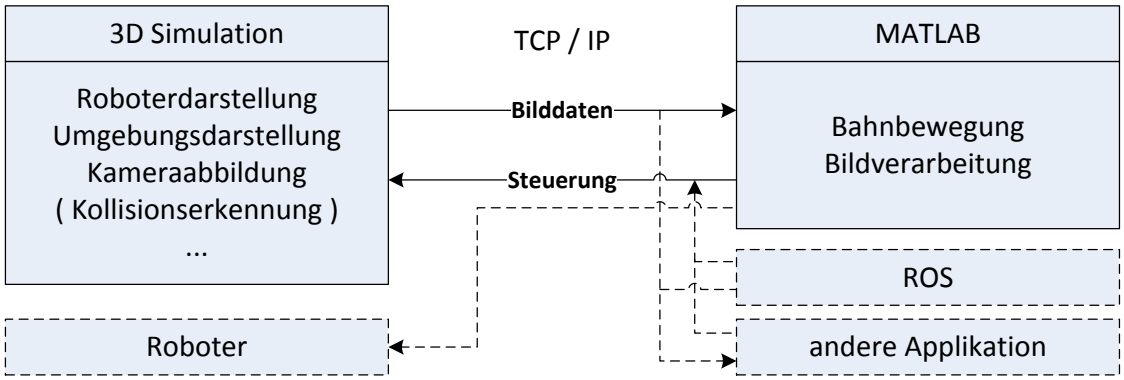


Abbildung 1.2.2: Problemstellung

1.3 Gliederung

Die vorliegende Arbeit ist in folgende Kapitel untergliedert:

Kapitel 1 geht auf den aktuellen Stand der Roboter Entwicklung ein und erläutert die Zielsetzung dieser Arbeit.

Kapitel 2 stellt ähnliche Arbeiten aus dem Bereich der Robotersimulation vor und nimmt eine Abgrenzung zu existierenden Projekten vor.

Kapitel 3 legt den Grundstein für das Verständnis der drei wesentlichen Themengebiete: Roboter Kinematik, Anwendungsdesign und Kameradarstellung.

Kapitel 4 begründet und erläutert die Architektur der entwickelten Software.

Kapitel 5 beschreibt Schritt für Schritt die Entwicklung der Simulationsumgebung von ihren einzelnen Bestandteilen zum Zusammenspiel aller Komponenten.

Kapitel 6 stellt mögliche Anwendungsfälle der entwickelten Applikation vor.

Kapitel 7 beinhaltet die Ergebnisse dieser Arbeit inklusive Variabilität und Performance-Analyse.

Kapitel 8 fasst die Arbeit in ihrer Gesamtheit zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen.

2 Verwandte Arbeiten und Produkte

Dieses Kapitel beschäftigt sich mit aktuellen Entwicklungen im Bereich der Roboter Simulation (2.1). Dies schließt sowohl kommerzielle als auch nicht kommerzielle Projekte ein. Es wird zunächst ein Überblick über deren Anwendungsgebiete gegeben und anschließend einige von ihnen näher erläutert. Schließlich findet eine Analyse (2.2) sowie die Abgrenzung (2.3) zwischen den existierenden Produkten und der in dieser Arbeit vorgestellten Anwendung statt.

2.1 Simulation von Robotern

Moderne Robotersimulation hat im Wesentlichen zwei Kernaufgaben. Zum einen ist dies die (Weiter-) Entwicklung von Robotern durch Verwendung immer neuer Sensoren und Aktoren und in neuen Aufgabengebieten. Der Trend geht hier zunehmend vom reinen Industrie- zum Assistenzroboter, etwa als Werkzeug im Operationssaal. Zum anderen geht es darum, Fertigungsabläufe stärker zu automatisieren und zu optimieren. Dies ist insbesondere in der Automobilbranche der Fall, wo große Fertigungsstraßen z.B. im Karosseriebau ausschließlich unter Zuhilfenahme von Robotern betrieben werden.

Ein bekanntes Projekt aus dem Bereich Fertigungsautomatisierung ist das Virtual Robot Controller (VRC) Interface. Aus dem Bereich der 3D-Simulation und Roboterentwicklung werden die universitären Projekte SimRobot und *Industrierobotersimulation mit VRML und MATLAB*, die kommerzielle Anwendung Microsoft Robotics Developer Studio und das neueste Projekt ROS vorgestellt.

2.1.1 Virtual Robot Controller Interface

Der VRC ist Bestandteil der vom Fraunhofer IPK entwickelten *Realistic Robot Simulation* (RRS) Schnittstelle. Das Projekt wurde Anfang der 90er vom IPK zusammen mit führenden Industrieunternehmen gegründet. Die Schnittstelle ermöglicht es, die originale Bewegungssoftware von Robotersteuerungen in Simulationssysteme zu integrieren. Mittlerweile ist sie zum weltweiten De-facto-Standard zur präzisen Simulation von Roboterbewegungen geworden. [IPK12]

Die Idee für den VRC entstand aus folgendem Problem: Zu Beginn der Offlineprogrammierung von Robotern wichen trotz hoher mathematischer Genauigkeit die simulierten Trajektorien sehr stark von den *realen* Bewegungen der Roboter ab. Es entstanden Bewegungsabläufe, welche bei Messungen am realen Roboter bis zu $300mm$ neben den Sollwerten lagen, für die industrielle Fertigung also unbrauchbar. Die Differenz kam dadurch zustande, dass der Controller welcher die Motoren des Roboters steuert, ein abweichendes Bewegungsprofil hatte. Die logische Folgerung war die Aufnahme des Controllers in die Simulation, die Entstehung des VRC. Im VRC werden alle Schnittstellen abgebildet, über welche der reale Controller verfügt, inklusive aller I/O Devices. Ein vollständiger VRC setzt sich aus den in Abb. 2.1.1 gezeigten (Bau-)Gruppen zusammen. Auf das Element Zeitmanagement möchte ich kurz näher eingehen. *Virtuelle Zeit* ist das Äquivalent zu real-time in der Simulation und erlaubt volle Kontrolle über die zeitlichen Abläufe während der Simulation. Jedes I/O Signal zum Beispiel beinhaltet einen Zeitstempel, welcher es ermöglicht, die Signalisierung zwischen mehreren Controllern präzise anzupassen. Das virtuelle Zeitmanagement ist das Nadelöhr bei der präzisen Simulation von Robotern, unter anderem

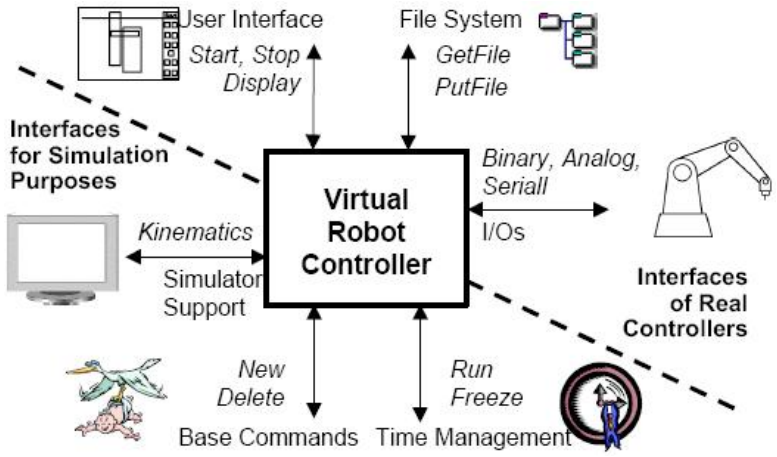


Abbildung 2.1.1: Die Module des VRC [CW02]

dann, wenn die Zeiten für Werkzeug- und Objektwechsel mit eingeplant werden müssen. Da dies in der Realität nicht beliebig schnell passieren kann, wird hier darauf geachtet, dass auch realistische Zeiten in die Simulation mit einfließen. Sollte dies nicht korrekt erfolgen, entsteht wieder die Situation, welche vor dem VRC der Fall war. In der Simulation kann alles beliebig beschleunigt werden und am realen Objekt ergeben sich große Abweichungen. Über den VRC werden ebenfalls Tolleranzen sowie Korrekturfahrten der einzelnen Gelenkmotoren realisiert.

Um von der Simulation im RRS/VRC zum konkreten Anwendungsfall zu kommen, wurde folgender Arbeitsablauf festgelegt:

1. *Graphische Programmierung*: Der Benutzer definiert die Punkte und belegt diese mit Attributen wie Bewegungsart und -geschwindigkeit. Anschließend werden die VRCs festgelegt und der Simulator übersetzt die Controller spezifische Sprache. Dieser Prozess ist für den Anwender transparent und wird meist in früher Planungsphase angewandt. Das Ergebnis bildet die Basis für den nächsten Arbeitsschritt.
2. *Native Programmierung in der Simulation*: Hier wird in der Sprache des Controllers des Roboters der Arbeitsablauf einprogrammiert. In diesem Schritt ist die gesamte Bandbreite der VRC Funktionalität nutzbar. Es wird nun ausführlich das simuliert, was am Ende auf den realen Roboter in der Werkshalle aufgespielt wird.
3. *Native Programmierung am Roboter*: In letzten Arbeitsgang wird das zuvor generierte Programm auf den Roboter geladen und final getestet.

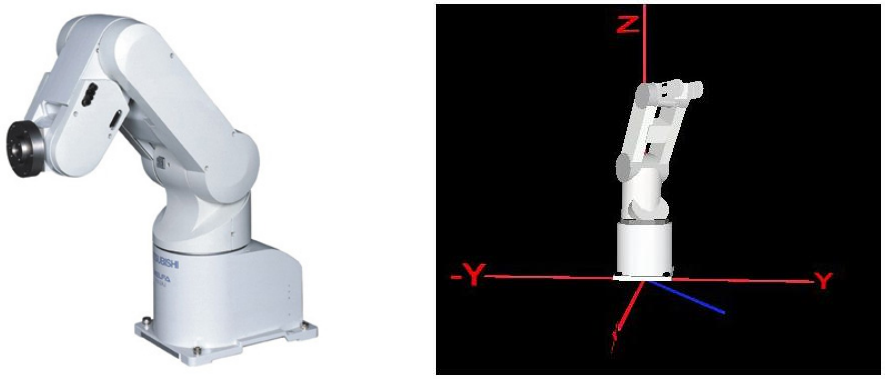
Diese drei Schritte beinhalten jeweils zahlreiche Iterationen des Modifizierens und Testens. Zwischen den einzelnen Schritten gibt es zusätzliche Datenkonsistenz Prüfungen. Die eigentliche Code-Generierung wird von Tools durchgeführt, welche der Roboter Hersteller bereitstellt.

Durch diese Vorgehensweise konnte in der Vergangenheit die Produktivität deutlich gesteigert werden, zum einen durch die Zeitersparnis beim Maschinenstillstand und zum anderen durch die verringerte Fehlerhäufigkeit durch ausgiebiges Testen während des Simulationsprozesses. [CW02]

2.1.2 Industrieroboter Simulation mit MATLAB und VRML

Das Projekt *"INDUSTRIAL ROBOT SIMULATION SOFTWARE DEVELOPMENT USING VIRTUAL REALITY MODELING APPROACH (VRML) AND MATLAB – SIMULINK TOOLBOX"* ist zwar bereits 2004 veröffentlicht worden, hat aber dennoch Relevanz für diese Arbeit, da es sich ebenfalls mit der 3D-Visualisierung eines Roboters mit MATLAB auseinandersetzt, einem Kernaspekt dieser Masterarbeit.

Konkret handelt es von der 3D-Simulation am Beispiel eines Mitsubishi RV-2AJ Roboter Armes (Abb. 2.1.2a), einem stationären Manipulator mit sechs Gelenken und ebenso vielen Freiheitsgraden. Ziel war es, den Roboter abstrahiert zu modellieren und darzustellen, sowie durch Eingabe der Gelenkwinkel bzw. der Position und Orientierung des Endeffektors das 3D-Modell in eine gewünschte Pose zu bringen. Dies wurde in drei Arbeitsschritten verwirklicht: 1. Spezifizierung der Kinematik, 2. Modellierung des Knick-Arm Roboters mit VRML, 3. Entwicklung einer GUI.



(a) in der Realität (b) in der Simulation

Abbildung 2.1.2: Der Mitsubishi RV-2AJ [Har04]

Zu 1: Die geometrische Konfiguration wurde vom Hersteller übernommen und die Gelenkparameter wurden in einer Denavit-Hartenberg Matrix zusammengefasst. Somit war es möglich, die Pose des Endeffektors bei Angabe der einzelnen Gelenkstellungen zu ermitteln. Anschließend wurde eine einfache inverse Kinematik entwickelt, um auch bei Eingabe der Pose im TCP[2] die einzelnen Gelenkstellungen zu bestimmen.

Zu 2: Um den Arm in 3D modellieren zu können, wurde er zerlegt und vermessen. Die VRML war das gewählte Mittel zur Umsetzung; zum einen, da sie einen Standard darstellt und zum anderen, da es eine Simulink Schnittstelle für sie gibt. Im Folgenden wird kurz auf die wesentlichen Merkmale eingegangen.

VRML ist eine Beschreibungssprache für 3D-Szenen, deren Geometrien, Ausleuchtungen, Animationen und Interaktionsmöglichkeiten. Ursprünglich als 3D-Standard für das Internet entwickelt, ist sie *human readable*[3] und wird in vielen 3D-Modellierungswerkzeugen verwendet. Ein VRML Modell besteht aus einer Basis (Group) und einer Anzahl von sog. Transforms und Shapes. Die Basis bildet den Ursprung des Modells, jedes Transform Element hat n Freiheitsgrade und ist mit einer Struktur (Shape) verbunden. Mehrere miteinander Verbundene Transforms ergeben zusammen mit den Shapes eine Baumstruktur, den Szenegraphen, welcher das 3D-Objekt bottom-up darstellt. Siehe hierzu auch 4.2.3. Diese Tatsache macht das Erstellen einer direkten Kinematik sehr einfach.

[2] Tool Center Point (Werkzeugmittelpunkt)
[3] für den Menschen lesbar

Zu 3: Das User Interface besteht aus zwei Teilen. Im ersten Teil, einem Browserfenster, befindet sich das 3D-Modell und im zweiten die Steuerung, welche über Slider und Eingabefelder die Parameter setzt, die anschließend über ein MATLAB Interface die neue Position des Roboters berechnet und mithilfe von Simulink das 3D-Modell aktualisiert. Im Hintergrund arbeitet dabei eine RSS Schnittstelle, welche in vorangegangenen Projekten entwickelt wurde.

Das System ist in dieser Version zwar auf drei bewegliche Gelenke (Hüfte, Schulter und Ellbogen) beschränkt, zeigt aber eine realitätsnahe Simulation des Roboter Armes (Abb. 2.1.2b). [Har04]

2.1.3 SimRobot

SimRobot ist eine frei verfügbare Eigenentwicklung der UNI Bremen. Es ist eine Anwendung zur Simulation sensorbestückter Agenten in einer dreidimensionalen Umwelt. Sie kommt vollkommen ohne Robotik Hardware aus und soll für den Einsatz von Algorithmen in realen Robotersystemen vorbereiten. Die Simulation erlaubt die Definition von hierarchischen Objekten mit Körpern, Dreh- und Teleskopgelenken, Fahr- und Flugzeugen, Abstands-, Farb- und Lichtsensoren sowie zweidimensionalen Kameras und Facettenaugen. Damit wird auch, mit entsprechender Abstraktion, die Modellierung des Verhaltens natürlicher Systeme möglich. Durch den offenen Quellcode in der Programmiersprache C ist die Applikation leicht erweiterbar. [Rö12]

2.1.4 MS Robotics Developer Studio

Microsoft Robotics Developer Studio (RDS), aktuell in der Version 4, stellt eine große Bandbreite an Unterstützung zur Verfügung, um Roboter oder Anwendungen für Roboter zu entwickeln. Microsoft RDS beinhaltet ein Programmiermodell, welches die Entwicklung asynchroner Automaten möglich macht. Die Software ist darauf ausgelegt, ein breites Spektrum verschiedener Roboter zu unterstützen.

Im Paket enthalten ist ebenso ein Set von graphischen Entwicklungswerkzeugen und diverse Simulationstools. Im *Visual Simulation Environment*, kurz VSE, können Robotik-Anwendungen in einer 3D Physik-Engine simuliert werden. [Mic12b]

2.1.5 ROS

ROS, kurz für Robot Operating System, ist ein Open-Source, Meta-Betriebssystem für Roboter. Es stellt alle Dienste bereit, die benötigt werden, einschließlich Hardware Abstraktion, Low-Level Geräte Steuerung und Implementierung von Standard Mechanismen wie Message-Passing zwischen Prozessen und Paketverwaltung. Hinzu kommen Werkzeuge und Bibliotheken für die Softwareentwicklung und Unterstützung für verteilte Systeme. ROS ist vergleichbar mit anderen Roboter Frameworks, wie etwa Player[4] oder Microsoft RDS. [wei12]

ROS ist modular aufgebaut und kann in anderen Frameworks, wie etwa zusammen mit SmartSoft[5] verwendet werden. Mit Gazebo[6] können mehrere, auch unterschiedliche Roboter in 3D simuliert werden. Abb. 2.1.3 zeigt ein Simulationsbeispiel. Die Software ist zurzeit nur auf UNIX/Linux kompilierbar und ist noch fehlerbehaftet.

[4] `http://playerstage.sourceforge.net/doc/Player-cvs/player/index.html`
[5] `http://smart-robotics.sourceforge.net/rosSmartSoft/index.php`
[6] `http://gazebosim.org/`

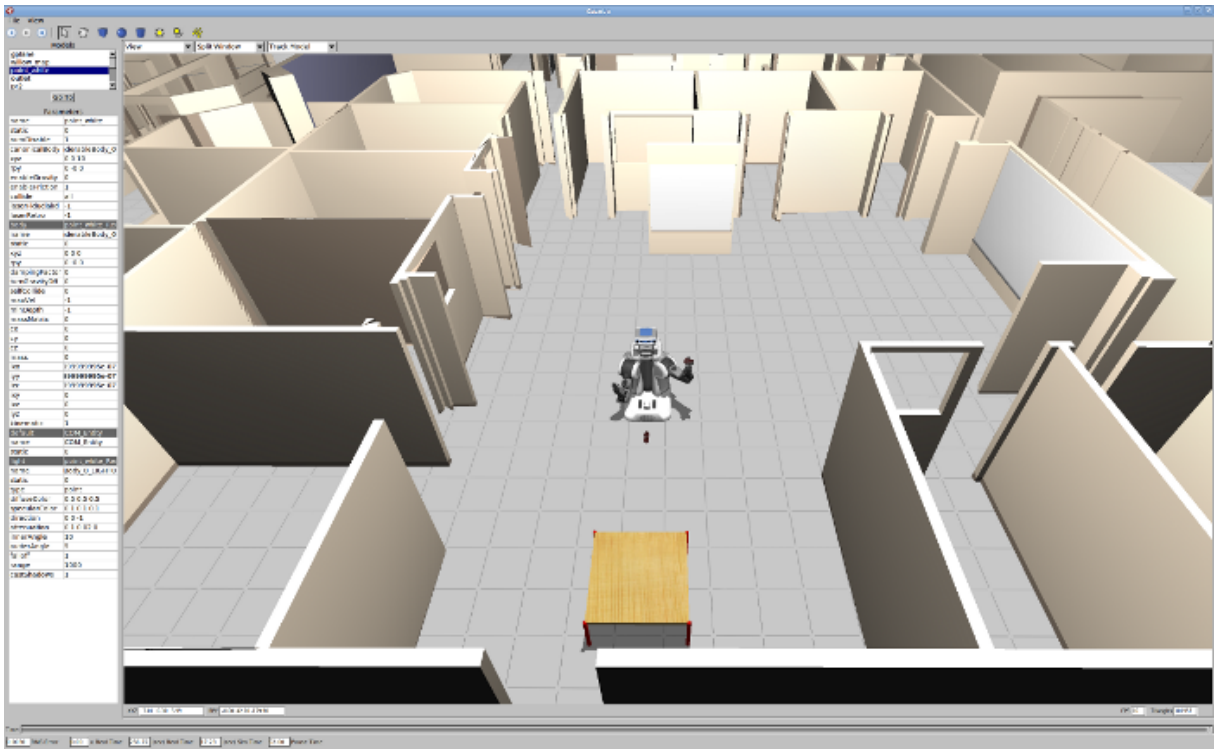


Abbildung 2.1.3: ROS mit Gazebo [Quelle: `http://gazebosim.org`]

2.2 Analyse

Neben den vorgestellten Produkten, gibt es zahlreiche weitere, die hier aufgrund der enormen Vielfalt nicht genannt werden können, da dies den Rahmen dieser Arbeit sprengen würde. Eine Auflistung frei verfügbarer Produkte mit Screenshots und kurzer Beschreibung ist auf der Seite von *Smashing Robotics*[7] einsehbar. Die vorgestellten Systeme bieten in ihrer Gesamtheit die Funktionalität, wie sie in der Zielsetzung (1.2) dieser Arbeit beschrieben ist, jedoch wird keine Anwendung allein den gestellten Anforderung gerecht.

Während RDS und ROS nur auf jeweils einer Betriebssystemarchitektur lauffähig sind, ist VRC speziell auf die präzise Industrieroboter Simulation ausgelegt, wo es in erster Linie auf die Genauigkeit vorprogrammierter Bewegungsabläufe ankommt und nicht um die Simulation von Sensorik, wie Kameras oder Sonar, geht. Die Arbeit am Mitsubishi Roboter zeigt das Potential von 3D-Visualisierung nur sehr vage, beweist aber, dass es funktionieren kann. Sehr ausgereift ist hingegen SimRobot was die Darstellung und die Simulation von Sensoren angeht, jedoch ohne externe Anbindung.

Was nun diese Arbeit von den bisherigen differenziert bzw. an welchen Ansätzen sie sich orientiert, wird im folgenden Abschnitt dargelegt.

[7] `http://www.smashingrobotics.com/most-advanced-and-used-robotics-simulation-software/`

2.3 Abgrenzung

Die mit dieser Arbeit entwickelte Robotersimulation wird in erster Linie zum Testen von Algorithmen, insbesondere zur Bildanalyse und Bewegungssteuerung, verwendet. Sie ist unter Einsatz diverser, bereits existierender Bibliotheken so konstruiert, dass wenig Know-How für den Umgang mit der Simulation erforderlich ist, und sich der Anwender allein auf die Erstellung von Algorithmen konzentrieren kann. Dabei ist es möglich, in einer graphisch realistischen Umgebung das Verhalten einzelner Verfahren zu erproben. Die 3D-Elemente können aus nahezu beliebigen Applikationen stammen und die Umgebung mit wenigen Schritten textuell konfiguriert werden. Dabei ist dem Anwender freigestellt, auf welchem Betriebssystem die Simulation läuft.

Ein wichtiges Unterscheidungsmerkmal zu den vorgestellten Systemen ist neben der einfachen Handhabung die Möglichkeit, vorher definierte Objekte in der 3D-Umgebung via Maus und/oder Tastatur während der Simulation zu bewegen, was eine völlig andere Interaktivität bedeutet. Der Hauptaspekt ist jedoch die Simulation von frei definierbaren Kameras für visuell geführte Roboter. Dies erlaubt, die Simulationsumgebung aus Sicht einer am Roboter montierten Kamera darstellen zu können. So können z.B. bei Einsatz unterschiedlicher Objektive oder Variation in der Auflösung des CCD-Sensors Bildanalysealgorithmen getestet werden.

Die Weiterentwicklung der Anwendung ist ebenfalls mit einfachen Mitteln möglich, da es zu den verwendeten Bibliotheken umfangreiche Dokumentation und zahlreiche Beispiele gibt. So können auch eigene Sensoren kreiert und eingebunden werden.

Aufgrund der eingesetzten Technologien ist die Anwendung nicht auf die Anbindung an MATLAB beschränkt, sondern kann durchaus auch zusammen mit anderen Projekten wie beispielsweise ROS eingesetzt werden.

3 Grundlagen

In diesem Kapitel werden die Grundlagen für das Verständnis der Thematik dieser Arbeit gelegt. Da es sich zu einem großen Teil um ein Softwareprojekt handelt, werden in 3.1 die Grundbegriffe für das Applikationsdesign beschrieben. Anschließend folgt eine Einführung in die Roboterkinematik (3.2). Dort wird erklärt, wie ein Roboter gesteuert und die Position seiner Gelenke bestimmt werden kann. Im Abschnitt 3.3 wird die (virtuelle) Kamera betrachtet, die es ebenfalls zu simulieren gilt.

3.1 Objektorientierte Anwendungsentwicklung

Ein großer Teil dieser Arbeit, der nicht direkt aus dieser Dokumentation zu sehen ist, steckt in der Softwareentwicklung. Um verständlich zu machen, warum welches Designmuster verwendet wurde und warum die Anwendung exakt so umgesetzt wurde, wie das Ergebnis zeigt, werden im Folgenden einige Züge der objektorientierten Anwendungsentwicklung beschrieben.

Es gibt diverse objektorientierte Sprachen, für die Entwicklung von 3D-Grafikanwendungen führt jedoch kein Weg an C++ vorbei. Sie ist zwar eine alte, jedoch keine veraltete Programmiersprache und was sie an Komfort gegenüber ihren jüngeren Konkurrenten einbüßt, wird durch die enorme Flexibilität und Performance ausgeglichen.

Der Schlüssel zu der erfolgreichen Umsetzung der im Abschnitt 1.2 beschriebenen Anforderungen liegt im Verständnis der hier beschriebenen Architekturen und Muster (3.1.2). Je umfangreicher und komplexer ein gefordertes System ist, desto mehr muss sich mit der Architektur befasst werden. Es gilt alle Eingangs- und Ausgangsparameter zu kennen, da die Anwendung nicht autark steht sondern mit ihrer Umgebung interagiert. Wenn mehrere Komponenten separiert werden, um gemeinsam eine Aufgabe auszuführen, spricht man von einem verteilten System, oder einer verteilten Anwendung.

3.1.1 Verteilte Anwendung

Genau betrachtet, ist heutzutage beinahe jedes computerbasierte System auch ein verteiltes System. Während früher ein einzelner Computer genau einen Prozessorkern hatte, haben heute bereits Smartphones mehrere Kerne, die unabhängig voneinander arbeiten können. Die Aufgaben eines Systems werden auf mehrere Recheneinheiten verteilt, um effizienter und schneller ein Ergebnis liefern oder Dienste bereitstellen zu können. Dies findet sowohl innerhalb eines Computersystems als auch zwischen verschiedenen Systemen, beispielsweise über ein Netzwerk, statt.

Diese Arbeitsweise bringt viele Vorteile mit sich. Ressourcen können geteilt, Abläufe parallelisiert und Systeme einfach skaliert werden. In der Regel sind verteilte Systeme ebenfalls *offen*, das heißt, sie benutzen Standard Protokolle, so dass sie einfach kombiniert werden können, und durch die Verteilung auf mehrere Komponenten kann ein System fehlertoleranter werden. Auf der anderen Seite kommen all diese Eigenschaften nicht ohne eine erhöhte Komplexität und dadurch erhöhten Verwaltungsaufwand aus. Ziel für die Architektur eines verteilten Systems ist es, eine Balance zwischen den genannten Faktoren zu finden. [Som07]

Eine mögliche Architektur ist der Client-Server Ansatz. Dabei stellt der Server-Teil der Anwendung einen Dienst bereit und der Client-Teil nutzt diesen. Ein einfaches Beispiel hierfür ist die Präsentation einer Webseite. Der Server hält die Daten bereit und bietet eine Verbindungsmöglichkeit an und ein Client Computer verbindet sich via Browser auf diesen. Dabei wird vom Client eine Anfrage geschickt, diese vom Server bearbeitet und anschließend das Ergebnis zur Präsentation zurückgesendet. Dies ist ein Beispiel für das *Thin*-Client Modell. Der Server Prozess verwaltet die Applikationslogik und das Datenmanagement und der Client zeigt lediglich die empfangenen Informationen an. Das Gegenteil hierzu ist der *Fat*-Client, welcher die gesamte Logik und Präsentation beinhaltet und der Server nur für die Datenverwaltung genutzt wird. Zudem können Logik und Datenverwaltung getrennt werden, wodurch eine sog. *dreiteilige* Client-Server Architektur entsteht. Ein Beispiel dafür gibt die Abb. 3.1.1

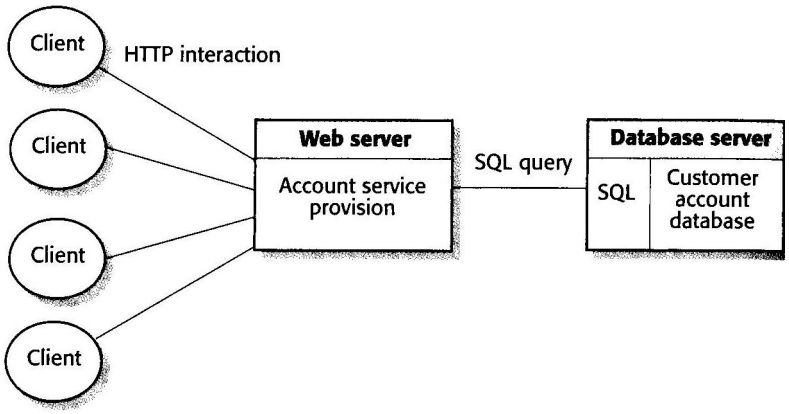


Abbildung 3.1.1: Verteilte Architektur eines Online Banking Systems [Som07]

Es gibt zahlreiche Abwandlungen wie etwa das Peer-to-Peer Modell oder auch serviceorientierte Ansätze mit verteilten Objekten. Diese sind jedoch nicht Gegenstand dieser Arbeit.

Für die Umsetzung einer Software, die gleichzeitig Netzwerkdienste bereitstellt und graphische Berechnungen durchführen und deren Ergebnisse bereitstellen soll, sind weitere Aspekte zu berücksichtigen. Diese beinhalten Multi-Processing und -Threading für parallele Ausführung von Aufgaben sowie Synchronisierung von Abläufen und das Schützen von gemeinsam genutzten Ressourcen.

Jedes moderne Betriebssystem besitzt die Fähigkeit, mehrere Prozesse parallel auszuführen, d.h. *schein*-parallel, falls nur eine Recheneinheit/ ein CPU Kern verfügbar ist und echt parallel, wenn mehrere Kerne verfügbar sind. Zudem kann ein Prozess mehrere Kind-Prozesse, sog. Threads ausführen, welche die Berechnungen durchführen. Dabei gehören die Ressourcen wie Arbeitsspeicher oder Ein- und Ausgabe-Geräte dem Prozess und werden unter den Threads verteilt.

Wenn nun nicht jeder Thread völlig autonom agiert, sondern die Aufgaben, welche parallel ablaufen, Ressourcen, wie etwa bestimmte Variablen, oder eine Netzwerkkarte teilen, muss dafür gesorgt werden, dass dies nicht zu unvorhergesehenen Zuständen führt. Zwei Threads, die sich eine Ressource lesend teilen, werden dabei keine Probleme verursachen. Greift jedoch ein Thread lesend und einer schreibend auf eine Variable zu, so muss sichergestellt werden, dass diese nicht während des Lesevorgangs verändert wird. Sonst kann der Inhalt verfälscht oder sogar unbrauchbar sein. Der Teil eines Threads, welcher auf eine gemeinsam genutzte Ressource zugreift, wird auch *kritischer Bereich (critical section)* genannt. Innerhalb dieses kritischen Bereichs muss die Ressource vor anderweitigem Zugriff geschützt werden. Gegenseitiger Ausschluss, oder auch Mutual Exclusion, verhindert, dass zwei oder mehr Threads gleichzeitig dieselbe Ressource verwenden. Folgende Regeln gelten für gegenseitigen Ausschluss:

1. Nur ein Prozess/ Thread zur Zeit darf sich in dem kritischen Abschnitt derselben Ressource befinden.

2. Ein Prozess/ Thread, welcher in einer nicht kritischen Sektion wartet, darf dadurch keine anderen Prozesse beeinflussen.
3. Es muss sichergestellt sein, dass kein Prozess unbegrenzte Zeit auf Zugang zum kritischen Abschnitt wartet.
4. Wenn sich kein Prozess im kritischen Abschnitt befindet, darf auch kein anderer am Zugang gehindert werden.
5. Es werden keine Annahmen über Anzahl oder Geschwindigkeit von Prozessoren gemacht.
6. Kein Prozess darf für unbegrenzte Zeit in der kritischen Sektion verharren.

Es gibt mehrere Mechanismen, welche für diese Umsetzung verwendet werden können. Zur Synchronisation von Prozessen und gegenseitigem Ausschluss werden Semaphore, Mutexe (binäre Semaphore), Monitore, Message Passing und weitere Techniken verwendet. Die einfachste Umsetzung zum schützen einer gemeinsamen Ressource ist der Mutex, weshalb er in dieser Arbeit Verwendung findet.

Ein Mutex wird initialisiert, ähnlich wie eine Variable. Vor Eintritt in den kritischen Bereich wird dieser blockiert und bei Austritt wieder freigegeben. Sollte, während sich ein Thread im kritischen Abschnitt findet, ein weiterer Zutritt über den Mutex verschaffen wollen, wird dieser geblockt. Nachdem der erste Thread die kritische Sektion verlassen hat, wird der Mutex wieder freigegeben, so dass der zweite Prozess Zugang bekommt. Unter Einhaltung der festgelegten Regeln, sollten nie beide Threads blockieren. [Sta05] Die Anwendung dieser Technik findet sich im Code im Anhang und im Kapitel 5 wieder.

3.1.2 Designmuster

Eines der wichtigsten Designmuster (Pattern) für eine objektorientierte C++ Anwendung ist der Smart Pointer als eine Umsetzung des Proxy Pattern. Ganz ähnlich wie ein HTTP-Proxy Zugriff auf Internetdienste in einem Netzwerk regelt, wird hier ein Platzhalter benutzt, um die Zugriffskontrolle zu einem anderen Objekt bereitzustellen. Der Smart Pointer oder die Smart Referenz ersetzt den Standard-Pointer und führt zusätzliche Operationen aus, wenn das Objekt verwendet wird. Typische Verwendungen sind das Zählen der Anzahl Referenzen, die auf das eigentliche Objekt verweisen, um automatisch den von ihm verwendeten Speicher freizugeben, wenn keine Referenz mehr existiert sowie das Laden eines persistenten Objektes in den Speicher, wenn es das erste mal referenziert wird. [JV07] Dies ist wichtig, da C++ im Gegensatz zu Sprachen wie JAVA keine eigene automatische Speicherbereinigung (Garbage Collection) implementiert hat. Angewandt wird diese Technik mithilfe sog. Klassen-Templates (Generics in JAVA).

Ein weiteres wichtiges Designmuster ist das Command Pattern. Ziel dieses Musters ist es, einen Aufruf in ein Objekt zu kapseln. Dieses Command Objekt ist vollständig vom eigentlichen Aufruf getrennt und hat das Hauptziel, die Abhängigkeit zwischen dem Aufrufer und dem Empfänger zu verringern. Dabei sind zwei Dinge markant: Der Aufrufende weiß weder, wie die angefragte Aktion von statten geht noch, welche Aktion ausgeführt wird. Auf der anderen Seite weiß der Empfänger des Aufrufs nicht unbedingt, wie dieser abgesetzt wurde. Ein typischer Ablauf ist:

1. Ein Client legt ein konkretes Command Objekt an und übergibt die notwendigen Anweisungen zur Erfüllung der Aufgabe.
2. Der Client, die Applikation, übergibt das Interface des konkreten Command Objektes an den Aufrufer (Invoker) und dieser speichert es.

3. Irgendwann entscheidet der Aufrufer, die Execute Methode des Interfaces aufzurufen, welches den Aufruf an das konkrete Command Objekt übergibt und dieses seine Arbeit verrichtet. Dieses wiederum verwendet das Empfänger- (Receiver) Objekt, welches eine Aktion Member Funktion verwendet, um die eigentliche Arbeit zu verrichten. Dieser Schritt kann jedoch auch direkt vom konkreten Command Objekt ausgeführt werden, wodurch der Receiver entfällt.

Abbildung 3.1.2 veranschaulicht dies.

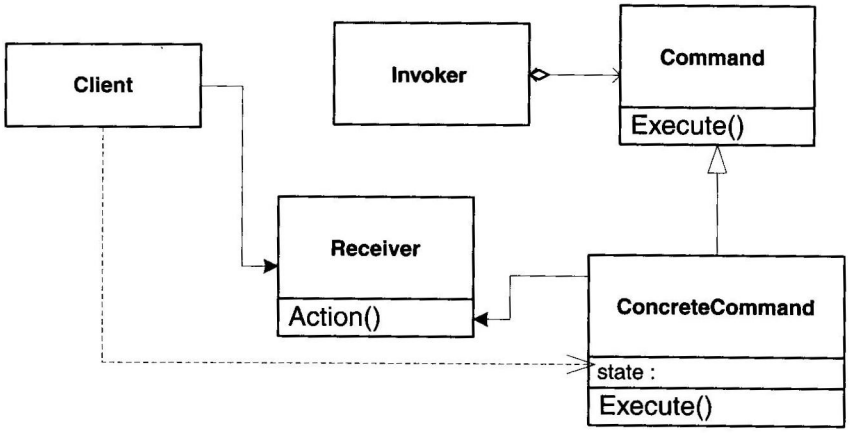


Abbildung 3.1.2: Command Design Pattern [Ale07]

Eine typische Anwendung für das Command Designmuster ist der Callback (Rückruf). Der Callback ist ein Zeiger auf eine Funktion, die übergeben wird und jederzeit aufgerufen werden kann. Tatsächlich wird dieses Muster in vielen Window Systemen wie X-Window für UNIX/Linux verwendet. Jeder Menüeintrag, Button oder jedes andere Widget legt einen Callback ab, der bei einer Benutzer-Aktion, wie das Anklicken einer Schaltfläche, aufgerufen wird. Das Widget selbst weiß dabei nicht, was der Callback macht. [Ale07] Ein solcher, asynchroner, durch Benutzer Interaktion hervorgerufener Callback wird auch als EventHandler bezeichnet.

Es gibt noch weitere Möglichkeiten der Verwendung von Callbacks und andere Implementierungen des Command Musters, in dieser Arbeit spielt der Callback jedoch eine besondere Rolle, wie im weiteren Verlauf gezeigt wird.

3.2 Kinematik

Kinematik beschreibt die Bewegung von Körpern in einem *Roboter-Mechanismus*. Dabei werden die Kräfte und Momente nicht berücksichtigt. Die Tatsache, dass Roboter von Beginn an auf Bewegung ausgelegt sind, macht Kinematik zum fundamentalen Aspekt vom Roboterdesign über die Analyse und Kontrolle bis hin zur Simulation. Der besondere Fokus liegt dabei auf der Repräsentation von Position und Orientierung.

In der Kinematik wird von einer Reihe idealer Bedingungen ausgegangen:

- Die einzelnen Festkörper sind geometrisch perfekt in Position und Form.
- Die einzelnen Elemente haben über ihre Verbindungen idealen Kontakt und den Abstand ‘0’.
- Die einzelnen Körper sind in ihrer Bewegungsfreiheit uneingeschränkt. [Kha08]

Um Position und Orientierung des Roboters bzw. seiner Armglieder exakt bestimmen zu können und den Roboter korrekt steuern zu können, muss zunächst ein Bezugssystem gewählt werden.

3.2.1 Koordinatensysteme

Das Bezugssystem, in welchem sich der Roboter bewegt, ist sein Welt-Koordinatensystem. Dieses ist im Falle eines stationären Roboters sein Arbeitsbereich, also die Umgebung, die er mit seinem Endeffektor bei maximaler Auslenkung aller Armglieder erreichen kann. Bei einem mobilen Roboter ist der Arbeitsbereich theoretisch unbegrenzt, jedoch praktisch durch seine Bewegungsreichweite beschränkt. Zusätzlich wird in jedes Gelenk ein Koordinatensystem gelegt, welches durch die Maße bzw. die Auslenkung des jeweiligen Gelenkes begrenzt wird. Am Ende der sog. kinematischen Kette aus Armgliedern und Gelenken befindet sich der Endeffektor. In der Regel ist dieser ein Werkzeug (z.B. ein Greifer), welches mit Werkstücken interagiert. Dieser hat ebenfalls ein Koordinatensystem, dessen Zentrum der Tool-Center-Point bildet. Je nach Betrachtung kann man den Roboter auch relativ zum Werkstück-Koordinatensystem betrachten. Dies ist insbesondere dann relevant, wenn sich das Werkstück ebenfalls bewegt. [Wü05]

Für diese Arbeit ist noch ein weiteres Koordinatensystem von Bedeutung. Da es sich um einen visuell geführten Roboter handelt, ist dies das Kamera-Koordinatensystem. Die Kamera ist in unmittelbarer Nähe des Endeffektors angebracht und *sieht* das Werkstück aus ihrem Center-Point.

All diese Koordinatensysteme (es handelt sich um kartesische Koordinaten) sind von Bedeutung, um die Position und Orientierung aller beteiligten Elemente bestimmen und letztendlich verändern zu können.

3.2.2 Position und Orientierung

Im Allgemeinen ist ein Roboter ein System aus Festkörpern mit Verbindungspunkten. Die Position und Orientierung der Gesamtheit der Elemente des Roboters bezeichnet man als *Pose*. Innerhalb seines Arbeitsbereiches kann ein Roboter je nach Konstruktion bestimmte Posen annehmen. Welche dies sind, wird durch die Freiheitsgrade (Degree Of Freedom - DOF) bestimmt.

Ein freier starrer Körper hat sechs Freiheitsgrade, drei translatorische und drei rotatorische. Für einen Roboter heißt das: Damit das Werkzeug in jeder Position jede beliebige Orientierung annehmen kann, muss der Roboter sechs Freiheitsgrade besitzen. Durch sinnvolle mechanische Konstruktion besitzt ein Roboter mit 6 Achsen auch 6 Freiheitsgrade. [Rö08] Ein mobiler Assistenzroboter wie der HSR besitzt zusätzlich noch zwei weitere translatorische DOF, da er sich in der Ebene frei bewegen kann. Er ist dadurch kinematisch redundant. Dies bedeutet, dass es in diesem Fall unendlich viele Möglichkeiten gibt, mit dem Endeffektor einen Punkt im Arbeitsraum zu erreichen.

Um nun die Pose des Roboters zu bestimmen, gibt es zwei grundsätzliche Ansätze. Zum einen die direkte Kinematik: Die Winkel aller Gelenke werden angegeben und die daraus resultierende Pose errechnet. Im Gegensatz dazu geht die indirekte Kinematik wie folgt vor: Die zu erreichende Endeffektor Pose wird vorgegeben und die Gelenkwinkel werden entsprechend berechnet. Letzteres Verfahren ist mathematisch herausfordernder und im Falle kinematischer Unbestimmtheit (Redundanz) teilweise nur mit iterativen Methoden durchführbar. Eine Einführung in dieses Thema findet sich im Anschluss an die folgende Sektion unter 3.2.4. Im Folgenden wird ein gängiges Verfahren für die direkte Kinematik vorgestellt.

3.2.3 Denavit-Hartenberg Transformation

Die Denavit-Hartenberg (DH) Transformation ist ein mathematisches Verfahren, mir dessen Hilfe die Koordinatensysteme innerhalb einer kinematischen Kette in einander überführt werden können. Bei einem z.B. 5-gelenkigen Manipulator wie dem Katana beschreibt je eine homogene Matrix jedes der Gelenke in Bezug auf das vorhergehende Gelenk. Dabei beschreibt die Matrix A_1 die Position und Orientierung des ersten Gliedes (Fuß), A_2 die Position und Orientierung des zweiten Gliedes bezüglich Glied 1 usw. Die Kette zieht sich so bis zum fünften Glied, dem Endeffektor, fort, so dass als Produkt die Gleichung 3.2.1 entsteht.

$$T = A_1 A_2 A_3 A_4 A_5 \tag{3.2.1}$$

Die Koordinatensysteme liegen fest in den Bewegungsachsen und müssen dabei nach der Denavit-Hartenberg-Konvention festgelegt werden:

- Die z_i-Achse wird entlang der Bewegungsachse des (i+1)-ten Gelenks gelegt.
- Die x_i-Achse ist senkrecht zur z_{i-1}-Achse und zeigt von ihr weg.
- Die y_i-Achse wird so festgelegt, dass sich ein rechtshändiges Koordinatensystem ergibt.

Um nun die Beziehung zwischen den Koordinatensystemen herzustellen, müssen mehrere Translationen und Rotationen durchgeführt werden. Mit folgenden Schritten wird das Koordinatensystem *i* in *i-1* überführt:

- Drehe um z_{i-1} mit dem Winkel θ_i, damit die x_{i-1}-Achse || zu der x_i-Achse liegt
- Verschiebe entlang z_{i-1} um d_i bis zu dem Punkt, wo sich z_{i-1} und x_i schneiden
- Verschiebe entlang gedrehtem $x_{i-1} = x_i$ um eine Länge a_i, um die Ursprünge der Koordinatensysteme in Deckung zu bringen
- Drehe um x_i mit dem Winkel α_i, um die z_{i-1}-Achse in die z_i-Achse zu überführen

Fasst man alle vier homogenen Transformationen, so kann die Matrix A_i wie folgt berechnet werden:

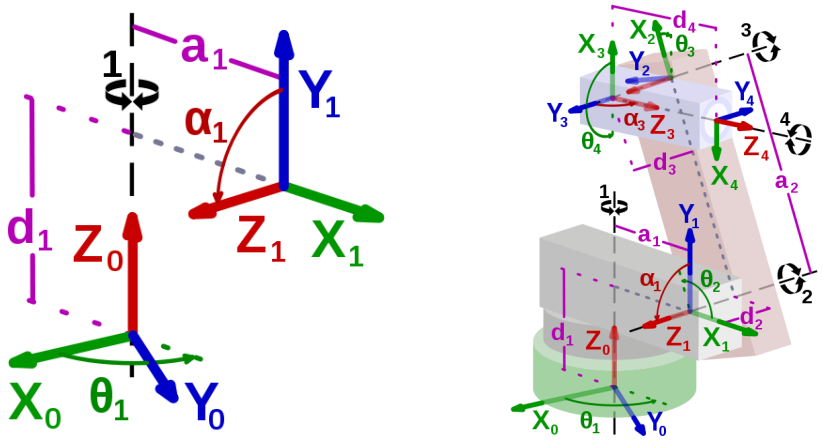
$$A_i = Rot_{z_{i-1},\theta_i} Trans_{(0,0,d_i)} Trans_{(a_i,0,0)} Rot_{x_i,\alpha_i} \tag{3.2.2}$$

Die Gesamttransformation zwischen zwei Gelenken ist also eine einfache Matrizenmultiplikation mit:

$$\mathbf{T}_i^{i+1}(\alpha_i, a_i, d_i, \theta_i) = \begin{pmatrix} cos\theta_i & -sin\theta_i cos\alpha_i & sin\theta_i sin\alpha_i & a_i cos\theta_i \\ sin\theta_i & cos\theta_i cos\alpha_i & -cos\theta_i sin\alpha_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2.3}$$

Sie wird auch als kinematische Kette bezeichnet. Um nun von TCP- in Welt-Koordinaten umzurechnen, wird die Matrizenmultiplikation i-1 mal ausgeführt. [Zha09] Abb. 3.2.1 veranschaulicht dies.

Bei solchen offenen kinematischen Ketten sind θ_n und d_n während der Bewegung des Roboters variabel. Im Falle eines Rotationsgelenks ist d_n konstruktiv bedingt, also konstant, während θ_n variabel ist. Für Linear- / Schubgelenke ist dies umgekehrt, θ_n ist fixiert und d_n ist frei. In beiden Fällen sind α_n und a_n dagegen invariante Größen und müssen nur einmalig zu Beginn initialisiert werden. Bei solchen offenen kinematischen Ketten sind θ_n und d_n während der Bewegung des Roboters variabel. Im Falle eines Rotationsgelenks ist d_n konstruktiv bedingt, also konstant, während θ_n variabel ist. Für Linear-/ Schubgelenke ist dies umgekehrt, θ_n ist fixiert und d_n ist frei. In beiden Fällen sind α_n und a_n dagegen invariante Größen und müssen nur einmalig zu Beginn initialisiert werden.



(a) DH-Transformation (einfach)

(b) DH-Transformation (5 DOF)

Abbildung 3.2.1: Die Denavit-Hartenberg Transformation [Wik12]

3.2.4 Inverse Kinematik

Mithilfe der DH-Transformation lässt sich die Pose zu gegebenen Gelenkwinkeln φ_i berechnen. Der Regelfall ist jedoch, dass nur bekannt ist, in welcher Zielpose sich der Endeffektor, also das Ende der kinematischen Kette befindet bzw. befinden soll. Die passenden Gelenkwinkel dazu müssen mithilfe der inversen Kinematik iterativ berechnet werden. Anders ausgedrückt: Aus der $3x4$ Transformationsmatrix $\mathbf{T}_{base}^{targeteff}$, welche die Abbildung des Basis- auf das Endeffektor Koordinatensystem der Zielpose beschreibt, werden die Ziel-Gelenkwinkel φ_{t_i} berechnet.

$$\mathbf{T}_{base}^{targeteff} \mapsto \varphi_{t_i} \text{ für } i = \text{Anzahl der Gelenke} \tag{3.2.4}$$

Iterativ deshalb, da erstens die Gelenke des Roboters nicht von einer zur nächsten Position *springen* können, sondern in einer Bahn bewegt werden, und zweitens da es oft mehrere Möglichkeiten gibt, ein und dieselbe Endeffektor Pose zu erreichen. Es existieren unterschiedliche Verfahren für die Berechnung der inversen Kinematik. Im Allgemeinen lösen sie das Nullstellenproblem des nichtlinearen Gleichungssystems, welches sich aus den Robotergelenken ableitet. Ein verbreitetes ist der Levenberg-Marquardt-Algorithmus, der im folgenden kurz vorgestellt wird.

Jede Iteration besteht aus folgenden Schritten (siehe [Fri11]):

1. Bestimmung der Vorwärtskinematik $\mathbf{T}_{base}^{eff}$ der aktuellen Gelenkwinkel
2. Berechnung der Abweichung e der Soll/Ist-Transformation
3. Ermittlung der Jakobimatrix $\mathbf{J}$ der Gelenke des Roboters
4. Berechnung der neuen Lösung für die neuen Gelenkwinkel
5. Überprüfung der Abbruchkriterien

Eine sehr detaillierte Erklärung des Algorithmus findet sich u.a. in [OT04] während sich ein detailliertes Anwendungsbeispiel in [Fri11] befindet.

Das Thema der inversen Kinematik wird hier der Vollständigkeit halber erwähnt, jedoch nicht in aller Ausführlichkeit dargelegt, da für den Katana eine solche bereits existiert und im Verlaufe dieser Arbeit eingesetzt, jedoch nicht selbst entwickelt wird.

3.3 Kamera Projektion

Die Projektion beschreibt die Abbildung eines dreidimensionalen Objektes auf ein zweidimensionales Kamerabild. Dabei gehen die Informationen über die Tiefe verloren. Jeder Raumpunkt $(X, Y, Z)^T$ wird so in einen Bildpunkt $(x, y)^T$ projiziert. Die Gleichung

$$x = \begin{pmatrix} x, & y \end{pmatrix}^T = \begin{pmatrix} f \cdot \frac{X}{Z}, & f \cdot \frac{X}{Z} \end{pmatrix}^T (f = Brennweite) \tag{3.3.1}$$

stellt dabei den optimalen Fall dar, nämlich dass der Bildhauptpunkt im Projektionszentrum der Kamera liegt. In der Regel ist jedoch der Bildhauptp unkt verschoben. Diese Bildhauptpunktverschiebung $(p(x, y)^T)$ wird in folgender Transformation mit in die Berechnung des Bildpunktes einbezogen. [Mei08]

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} +f & 0 & p_x & 0 \\ 0 & -f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{pmatrix} \tag{3.3.2}$$

Hier wird davon ausgegangen dass es sich um eine perfekte, lineare Kamera, auch Lochkamera genannt, handelt. Die in der Realität durch physikalische Beschränkung der Kamera entstehende optische Verzerrung spielt somit keine Rolle. Diese Verzeichnung wird durch eine Kalibrierung der Kamera am Computer herausgerechnet. Dies ist jedoch nicht Gegenstand dieser Arbeit.

Wichtig für die Kameradarstellung im Bereich Computergrafik sind die intrinsischen Kameraparameter, insbesondere die Brennweite (*f*) und die Bildwinkel. Sie entscheiden darüber, welcher Ausschnitt einer Szene von der Kamera aufgenommen wird. Die Brennweite beschreibt dabei den Abstand zwischen der Aufnahmeebene (im Falle einer Digitalkamera der CCD-Sensor) und der Objektiv-Hauptebene (vgl. [Mei08]). Zusammen mit dem Bildformat (Diagonale *d*) kann so der Bildwinkel berechnet werden (Gl. 3.3.3).

$$\alpha = 2 \cdot \arctan(\frac{d}{2 \cdot f}) \tag{3.3.3}$$

Dabei ergibt sich *d* aus Länge und Breite des Bildformats. Über den Satz des Pythagoras können sowohl der vertikale als auch der horizontale Bildwinkel bestimmt werden. Abb. 3.3.1 veranschaulicht dies. Hier wird der englische Begriff FOV (field of view) verwendet.

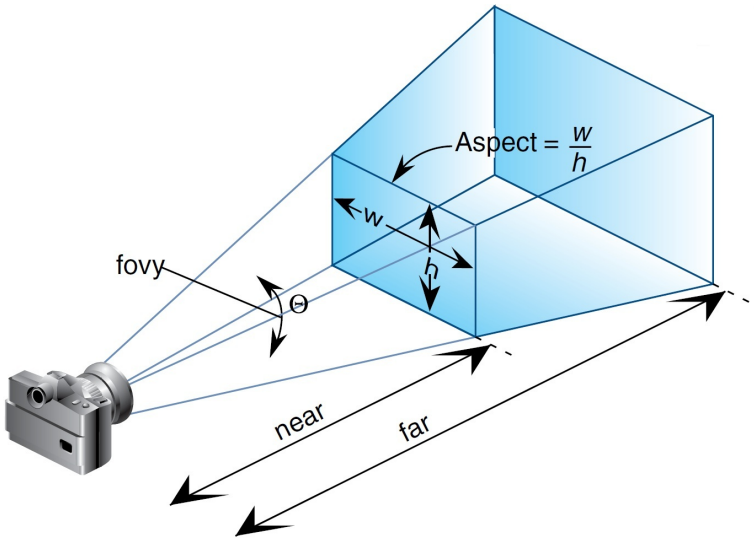


Abbildung 3.3.1: Kamera [Shr09]

Das Seitenverhältnis (Aspect Ratio) kann auch wie folgt beschrieben werden:

$$AspectRatio = \frac{x}{y} = \frac{\tan(horizontalFOV/2)}{\tan(verticalFOV/2)} \tag{3.3.4}$$

Im Bereich der Computergraphik nicht zu vernachlässigen ist die Bedeutung der Werte für die nahe und ferne Bildebene. Zwischen diesen beiden Ebenen müssen Objekte liegen, die erfasst werden sollen. Während in der Realität eine Kamera theoretisch unendlich weit sehen kann, wenn auch nicht mehr scharf, würde dies in der Computergraphik einen sehr hohen Rechenaufwand bedeuten.

3.3.1 Bildausschnitt

Der Pyramidenstumpf, das Frustrum, zwischen der nahen und fernen Bildebene bildet den Bereich, in dem die Welt dargestellt / gerendert wird - hier dargestellt in Abbildung 3.3.1 (blau). Dieser muss nicht zwangsläufig symmetrisch sein, ist so aber einfacher zu definieren. Er wird durch die sog. Clipping Ebenen im Abstand *near* und *far* begrenzt.

Punkte, Kanten oder Polygone die außerhalb dieses Raums liegen, werden abgeschnitten. Dieser Vorgang des Clipping bzw. Culling für dreidimensionale Objekte wird ermöglicht durch das Bilden einer Bounding Box bzw. Bounding Sphere für jedes Objekt einer Szene. Die Bounding Sphere beschreibt das Volumen der kleinstmöglichen Kugel, die ein Objekt einschließt. Sie wird gebildet, indem vom Objektmittelpunkt der Abstand zum weit entferntesten Eckpunkt als Radius angenommen wird und in Formel 3.3.5 eingesetzt wird.

$$V = \frac{4}{3}\pi R^3 \tag{3.3.5}$$

Die Bouding Box (Abb. 3.3.2), oder auch minimal umgebender Quader, hingegen ergibt sich durch das Bestimmen der Raum-Diagonalen d der beiden am weitesten voneinander entfernten Punkte eines Objektes, also den minimalen sowie maximalen (x, y, z) Koordinaten. [Shr09]

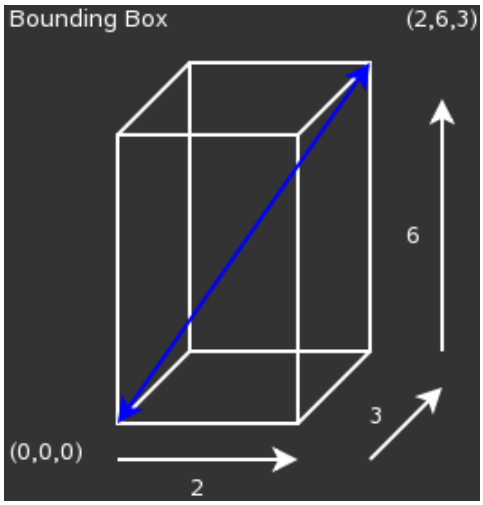


Abbildung 3.3.2: Bounding Box [BM13]

So kann berechnet werden, ob und wie sich zwei Objekte, in diesem Fall das Frustrum der Kamera und ein beliebiges Objekt der Szene schneiden und das Kamerabild entsprechend gerendert werden. Dieser Aspekt wird u.a. auch für die Kollisionserkennung (6.3) genutzt.

4 Architektur

Das Ziel, MATLAB an eine realistische, graphische Simulationsumgebung anzubinden, stellt mehrere unterschiedliche Anforderungen. Zum einen ist MATLAB mit seinen Toolboxen sehr modular aufgebaut, wodurch es möglich ist, mit geringem Aufwand eine Anwendung in hohem Maße zu verändern. Somit muss die Schnittstelle zu einer anderen (externen) Anwendung ebenfalls äußerst flexibel sein. Zum anderen gilt es, einen Überblick über die von Mathworks bereitgestellten Funktionen zu haben, um nicht unnötig viel Aufwand in die Erstellung von Funktionalitäten zu stecken, welche bereits existieren. Dies gilt ebenso für die generelle Anwendungsentwicklung. Die einzelnen Architekturen und Ansätze werden in Sektion 4.1 diskutiert. Des Weiteren ist das Feld Computergraphik (4.2) sehr breit aufgestellt, so dass dieses auch mit einbezogen werden muss.

Weitere Aspekte der Architektur werden in den Abschnitten 4.3 bis 4.6 betrachtet. Sie münden schließlich in die konkrete Auswahl der umzusetzenden Architektur (4.7).

4.1 Ansätze

Auf dem Weg zum finalen Design der Applikation wurden verschiedene Ansätze verfolgt. Zu Beginn stand lediglich fest, dass das Ergebnis modular aufgebaut sein muss und sich an der von Rodney A. Brooks beschriebenen Subsumption Architektur orientiert. Diese findet sich seit ihrer Vorstellung 1986 insbesondere im Bereich autonomer Roboter wieder. Kern ist die Aufteilung eines Systems mit komplexem Verhalten in einfache Subsysteme, die aufeinander aufgesetzt werden, kurz Dekomposition. [Bro87] Die folgenden vier Abschnitte beschreiben den Weg bis hin zur gewählten Umsetzung.

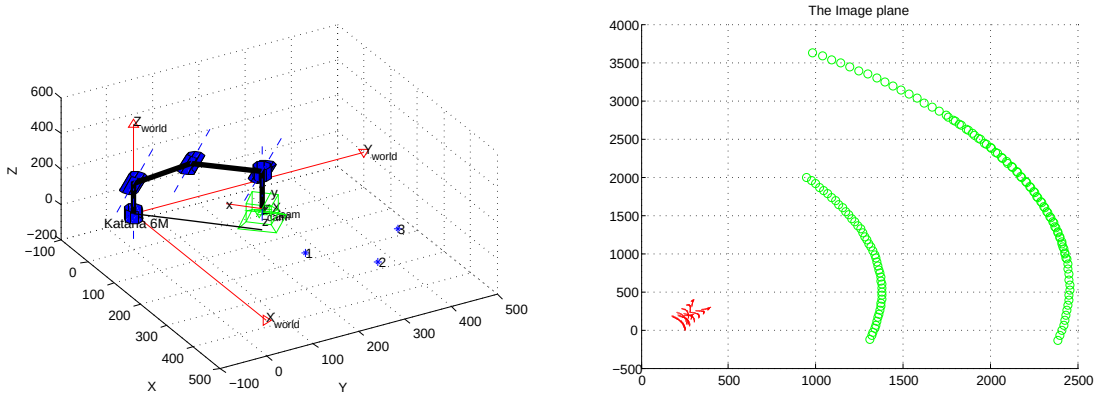
4.1.1 3D-Darstellung mit nativen MATLAB Methoden

Wie bereits erwähnt, verfügt MATLAB, hier eingesetzt in der Version R2012b, über ein breites Spektrum an mitgelieferten Toolboxen. Hinzu kommen unzählige frei verfügbare Add-ons, die sowohl von freien Entwicklern als auch von Universitäten bereitgestellt werden. So war der erste Ansatz, sowohl die Kinematik und Bildverarbeitung als auch die 3D-Simulation direkt in MATLAB umzusetzen. Dies wurde unter Zuhilfenahme der Robotics Toolbox von Peter Corke[8] und der Epipolar-Geometrie Toolbox[9] (EGT) von Gian Luca Mariottini und Domenico Prattichizzo implementiert.

Die Robotics Toolbox abstrahiert die zur Posen- und Winkelberechnung notwendigen Matrixoperationen zu einfachen Funktionen und ermöglicht so einen schnellen Einstieg. Zur Darstellung werden die Kinematik Parameter in eine Matrix geschrieben und eine Startposition für die einzelnen Gelenke gegeben. Über den Befehl `plot(kinematikMatrix,Startposition)` wird der Roboter anschließend dargestellt. [Pet11]

[8] `http://www.petercorke.com`
[9] `http://egt.dii.unisi.it/`

Die Kamera (Details siehe 5.5) wird mithilfe der EGT modelliert. Diese Toolbox stellt eine große Auswahl an Funktionen für Computer Vision bereit, u.a. erlaubt sie die Darstellung von Kameras und Schätz-Algorithmen der Epipolargeometrie (siehe [Mei08]). Hier wird sie dazu verwendet, eine Kamera im TCP zu positionieren, Szene-Punkte zu generieren und diese in der Bildebene darzustellen. Abbildung 4.1.1a und 4.1.1b zeigen die Visualisierung. Der Katana bewegt sich von der y- zur x-Achse und die Kamera nimmt dabei die Szenepunkte auf. Der vollständige Code dazu befindet sich im Anhang A.1.



(a) Roboter mit Kamera

(b) Bildebene der Kamera

Abbildung 4.1.1: Katana mit Kamera im Endeffektor

Für eine rein schematische Darstellung der Roboterbewegung und des Kamerabildes ist diese Funktionalität zwar völlig ausreichend, eine realistische 3D-Simulation mit Objekten, Texturen und Lichtquellen ist so jedoch nicht möglich. Abhilfe schafft hier die Simulink 3D-Animation Toolbox und dem Tool VR Builder. Diese Erweiterung ermöglicht das Erstellen von 3D-Szenen mit VRML, einer Text-Beschreibungssprache für 3D-Elemente. Es können ebenfalls VRML Dokumente aus externen Quellen verwendet werden. VRML arbeitet mit Szenegraphen (4.2.3) und kann von einfachen Objekten wie Kisten bis hin zu komplexer Geometrie in ausgeleuchteten Szenen viele 3D-Elemente darstellen.

Da es vom Katana bereits ein frei verfügbares 3D-Modell im STEP-Format (siehe auch 5.4) gibt, konnte diese mithilfe des Open Source Tools CAD Exchanger[10] einfach in VRML umgewandelt werden und musste nicht, wie in Abschnitt 2.1.2 beschrieben, nachgebaut werden. Bei dieser Konvertierung entstand jedoch ein Modell, welches aus über 1000 3D-Elementen (Shapes) besteht, welche jedoch nicht in einem Baum angeordnet waren, sondern alle in einer Ebene lagen. Die Lösung dieses Problems war die Zerlegung des STEP-Modells mit einer CAD Software in genau so viele Bauteile, wie es drehbare Gelenke gibt, und ihre anschließende Konvertierung. Die so erzeugten Arm-Teile wurden mit dem VR Builder zu einem Szenegraphen zusammengefügt und konnten anschließend animiert werden. Abb. 4.1.2 zeigt einen Sceenshot aus der Simulation. Nach erfolgreichem Test, wurde die Kamera durch einen sog. Viewport, also eine Sicht auf die Szene dargestellt und eine Lichtquelle hinzugefügt.

Die Performance Analyse dieser Lösung ergab jedoch, dass eine Bildrate von 25 Bildern pro Sekunde so nicht zu erreichen ist. Zudem besteht bei der Qualität der Graphik noch ein hoher Verbesserungsbedarf. Die Komplexität des VR Builders im Zusammenhang mit Simulink ist ebenfalls negativ zu bewerten, sieht man die Anwendung im Kontext einer breiten Zielgruppe. Ein schneller Einstieg und leichte Modifikation sind so nicht möglich, wodurch dieser Ansatz verworfen wurde.

[10] `http://cadexchanger.com`

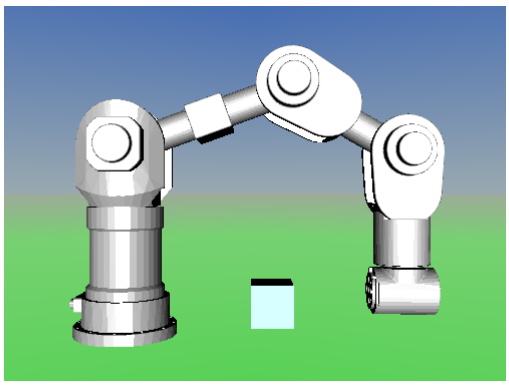


Abbildung 4.1.2: Katana in der 3D-Animations Toolbox von Simulink

4.1.2 Einbinden externer Funktionen

Da Darstellung, Performance und Benutzerfreundlichkeit nicht ausreichend waren, wurde eine weitere Möglichkeit in Betracht gezogen, die ebenfalls MATLAB als zentrale Anwendung hatte. Über MEX (MATLAB executable) ist es möglich, u.a. eigenen C/C++ Code und auch externe Bibliotheken zu verwenden, um die Funktionalität zu erweitern. Dabei generiert MATLAB mithilfe einer Entwicklungsumgebung wie Microsoft Visual Studio aus gegebenem Quellcode eine Bibliothek, deren Funktionen wie bereits existierende MATLAB-Befehle oder als s-Funktion in Simulink ausgeführt werden können.

Zum Test wurden unter Verwendung von OpenSceneGraph Bibliotheken (vgl. 4.2.2) 3D-Objekte erzeugt, die eine höhere Qualität bei gleichzeitig höherer Bildwiederholrate aufweisen, als die mit MATLAB Mitteln generierten. Als Beispiel diente eine einfache s-Funktion, welche nach dem Vorbild von Janusz Goldasz[11] implementiert wurde. Dabei wird in Simulink zunächst ohne Verbindung zu einer Routine in MATLAB eine Stand-Alone Simulation erzeugt (Abb. 4.1.3). Anschließend wird eine präparierte C++ Quelldatei mit MEX zu einer *mexw32* Datei kompiliert und hinter die s-Funktion gelegt. Eine genaue Anleitung, wie dies mit MEX realisiert wird, kann auf der Mathworks Homepage[12] oder auch direkt in der MATLAB / Simulink Hilfe abgerufen werden. Es empfiehlt sich für einen schnellen Einstieg das ebenfalls dort beschriebene Legacy Code Tool zu verwenden.

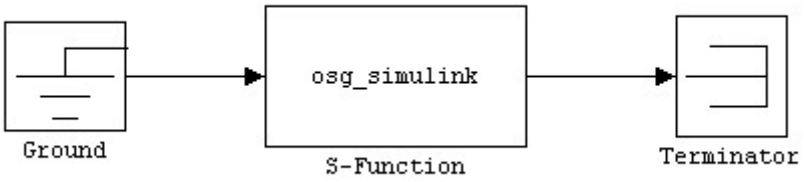


Abbildung 4.1.3: Simulink Aufbau

Die so implementierte Lösung funktionierte zwar, war jedoch sehr instabil, was sich immer wieder in Abstürzen von MATLAB ausdrückte. Sowohl mit 32 als auch mit anfänglich 64 Bit war keine dauerhaft stabile Lösung zu erzielen. Bei genauerer Analyse stellte sich heraus, dass dies im Zusammenhang mit anderen graphischen Simulationen wie OGRE (siehe 4.2.2) schon öfter aufgetreten ist (vgl. [Mat12b]). Dies hängt u.a. mit der Verwaltung von Threads zusammen. Die mit externen Bibliotheken dargestellte

[11] `http://www.3dcalc.pl`
[12] `http://www.mathworks.de`

Graphik muss in einem eigenem Thread laufen, da sonst keine Interaktion zwischen MATLAB und derselben stattfinden kann. Leider bestätigt Mathworks auf einer seiner Support Seiten, dass MATLAB ab Werk keine multi-threaded MEX Funktionen beherrscht [Mat12a].

So musste dieser Lösungsansatz ebenfalls aufgegeben werden, da der Aufwand der Realisierung in keinem akzeptablen Verhältnis mehr zum Nutzen der Anwendung steht.

4.1.3 Shared Library Ansatz

Eine Möglichkeit, das im vorangegangenem Abschnitt aufgetretene Problem zu umgehen, ist das Auslagern der 3D-Simulation. Eine gemeinsam genutzte Bibliothek (kurz: Shared Library) ist hier eine mögliche Lösung. Dabei werden in der Library gemeinsam verwendete Funktionen definiert. Dies ist im Wesentlichen das Senden von Steuerbefehlen in Form von Koordinaten von MATLAB an die Simulation. Diese wiederum bindet ebenfalls die Bibliothek ein, liest die Befehle und führt diese aus. Die Anwendung besteht so nicht mehr nur aus MATLAB sondern aus insgesamt drei Komponenten. Dabei sind Anpassungen in MATLAB einfach zu realisieren, während die Bibliothek entweder sehr generisch sein muss oder bei Änderungen am Kommunikationsprotokoll immer wieder neu kompiliert wird. Das gleiche gilt für das Programm, das die Simulation darstellt.

Im Kontext betriebssystemunabhängiger und flexibler Anwendung betrachtet, wird klar, dass diese Lösung zwar realisierbar jedoch zu komplex und wahrscheinlich sehr wartungsintensiv sein wird. Sowohl die Library als auch die Simulationsumgebung müssen immer wieder kompiliert werden. Da sowohl die Aktoren des Roboters von MATLAB aus gesteuert werden sollen als auch die Sensordaten der montierten Kamera zurück übertragen werden müssen, und insbesondere beim Hinzufügen neuer Kommandos und Sensoren Anpassungen notwendig sind, wurde auch hier eine weiterführende Analyse abgebrochen. Es entstand ein weiteres Konzept, welches auch in die Architektur der finalen Lösung eingeflossen ist.

4.1.4 Verteilte Anwendung

Anstatt die Simulation durch drei Komponenten abzubilden, wurde die Möglichkeit einer Client-Server Architektur in Betracht gezogen. Auf der einen Seite steht dabei MATLAB als Quelle der Gelenkkoordinaten und für die Bildverarbeitung /-analyse der Kameraaufnahmen und auf der anderen Seite die Simulation, welche die gegebenen Kommandos ausführt und die Bilder an MATLAB sendet. Die Kommunikation kann dabei sowohl über das Dateisystem als auch über das Netzwerk stattfinden. Letzteres stellt die elegantere Lösung dar.

Die Simulation kann weitgehend über eine Textdatei konfiguriert werden, was zusätzlich Flexibilität schafft. Sowohl die Roboter als auch die Umgebung mit Objekten als auch die Kamera(s) sind so leicht austauschbar. Ein weiterer Vorteil einer solchen verteilten Anwendung liegt darin, dass die Simulation für MATLAB transparent ist, da nur das Protokoll bekannt sein muss. Dies gilt genauso anders herum: Es muss nicht zwingend MATLAB sein, welches die Befehle sendet. Diese Aufgabe kann auch ein System wie ROS erfüllen. Zusätzlich bietet diese Lösung die Möglichkeit, beide Applikationsteile auf verschiedenen Systemen mit unterschiedlichen Betriebssystemen laufen zu lassen.

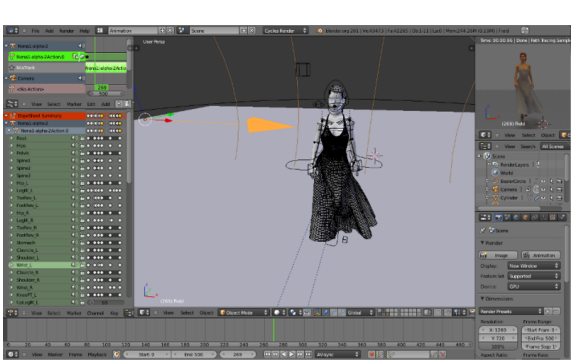
Die Vorteile gegenüber den anderen vorgestellten Ansätzen sind gravierend. All die genannten Engpässe wie die Performance oder nicht vorhandene Interoperabilität sind so nicht mehr von Bedeutung.

Im Folgenden werden nun die eingesetzten Technologien vorgestellt, da diese ebenfalls einen erheblichen Einfluss auf die Architektur haben.

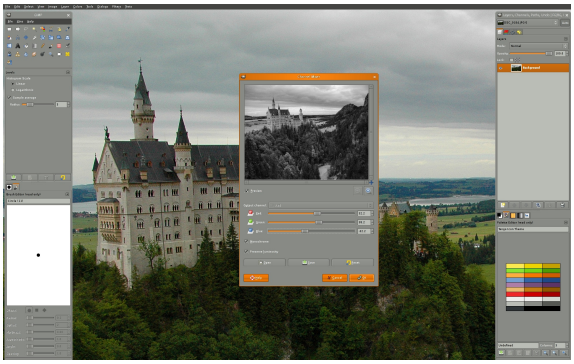
4.2 Grafik

Ein Hauptaspekt dieser Arbeit ist die Generierung und Simulation einer dreidimensionalen Umgebung am Computer. Das Verständnis des Anwendungsgebietes Computergraphik bildet für die Umsetzung die Basis. Der Grundstein für diese Möglichkeit einer graphischen Darstellung wurde 1950 am MIT gelegt; mit der Kathodenstrahlröhre zur Ausgabe von Bilddaten. Nach zunächst rein wissenschaftlicher Nutzung ist diese als Fernsehgerät in den alltäglichen Gebrauch eingegangen. Im Laufe der technischen Weiterentwicklung in den letzten Jahrzehnten nahm die Computergraphik u.a. mit CAD Einfluss auf moderne Produktionsmethoden und es entwickelten sich unzählige Anwendungsgebiete.

Diese lassen sich heute in drei Themenbereiche unterteilen. *Generative Computergraphik* befasst sich mit Visualisierung von Simulation und Animation (Abb. 4.2.1a), atmosphärischen Effekten und Virtual Reality (VR). Daneben gibt es die *Bildverarbeitung*. Sie beinhaltet u.a. 2D Bildrestauration (Abb. 4.2.1b), Filterverfahren und 3D Bearbeitung von Bildsequenzen. Den dritten Bereich stellt die Bildanalyse dar, welche sich mit Bild- und Mustererkennung, Kartographie und auch Augumented Reality[13] beschäftigt. [Kro08]



(a) Animation [ble13]



(b) Bildverarbeitung [gim13]

Abbildung 4.2.1: Computergraphik

Diese Arbeit befasst sich hauptsächlich mit der generativen Computergraphik, also der Simulation von Objekten in 3D. Die Bildverarbeitung wird zur Gewinnung und Konvertierung von Bildern der virtuellen Kamera herangezogen und für die Analyse spielt die Mustererkennung eine Rolle, denn nur mit Objekten, die (wieder-) erkannt werden, kann ein Roboter auch interagieren. Letzteres wird exemplarisch in Kapitel 6.2 behandelt.

4.2.1 Spezifikationen

Heute gibt es im Wesentlichen zwei Standards für generische Computergraphik, OpenGL und Direct3D. Beide sind etabliert und haben einen ähnlichen Funktionsumfang, unterscheiden sich jedoch auch in einigen Punkten.

OpenGL, kurz für *Open Graphics Library*, stellt eine Softwareschnittstelle zur Graphik Hardware dar. Aktuell in der Version 4.3 verfügbar, besteht sie aus einigen hundert Prozeduren und Funktionen, welche es dem Programmierer ermöglichen, die Objekte und Operationen auf diesen zu spezifizieren, welche für interaktive 3D-Anwendungen benötigt werden. [KA12]

[13] computergestützte Erweiterung der Realitätswahrnehmung

OpenGL ist als hardware- und betriebssystemunabhängige Schnittstelle konstruiert um auf einer Vielzahl von Plattformen Verwendung zu finden. Dies wird u.a. dadurch ermöglicht, dass kein Window System[14] bereit gestellt wird, sondern der Entwickler das jeweils vorhandene verwenden muss.

OpenGL stellt keine High-Level Befehle für die Erzeugung komplexer Strukturen wie etwa eines Automobils dar. Es steht lediglich ein Satz von geometrischen Primitiven bereit, bestehend aus Punkten, Linien und Polygonen (Vielecke). Dies bildet jedoch die Basis für jede weitere API (Application Programming Interface), welche komplexere Strukturen bereitstellen möchte. [Shr09]

Microsofts Direct3D stellt den direkten Konkurrenten zu OpenGL dar und ist Teil der DirectX Umgebung. Die API, zurzeit in Version 11 erhältlich, ist rein auf Microsoft Windows zugeschnitten. Sie unterstützt ebenfalls eine große Bandbreite an Graphik Hardware, aber kein anderes Betriebssystem. [Mic12a]

Es ist im Gegensatz zu OpenGL nicht quelloffen und kann somit nicht um eigene Features erweitert werden. Durch den proprietären Status sind jedoch schneller neue Standards geschaffen. Direct3D ist insbesondere im Bereich der Computerspiele angesiedelt; ohne eine aktuelle DirectX Installation sind die meisten heutigen Spiele nicht lauffähig.

Der Funktionsumfang von Direct3D ist dem von OpenGL sehr ähnlich. Es werden alle gängigen Techniken wie Tesselation, Shading und Multi-Threading unterstützt.

4.2.2 Bibliotheken

Jede Form der 3D-Graphik Visualisierung am Computer benötigt 3D-fähige Hardware. Mit der Komplexität und Qualität der Darstellungen steigen die Ansprüche an diese. Die OpenGL und Direct3D Bibliotheken sind daher in einer hardwarenahen Programmiersprache C++ geschrieben. Es besteht zwar die Möglichkeit, mit anderen High-Level Programmier- oder Skriptsprachen, diese Bibliotheken zu nutzen, dies wird jedoch hier nicht angestrebt. Zum einen bleibt so die Möglichkeit erhalten, diese Bibliotheken direkt anzusteuern und zum anderen ist C++, richtig eingesetzt, ein Garant für gute Performance. Da OpenGL aufgrund der Plattformunabhängigkeit die einzusetzende Technik ist, wird im Folgenden kein weiterer Bezug mehr auf Direct3D genommen.

OpenGL liefert bereits alle Bibliotheken, um 3D-Elemente am Computer zu erzeugen. Die Entwicklung mit diesen findet jedoch auf einem niedrigen Level statt, d.h. dass beispielsweise auch einfache geometrische Elemente über eine Folge vieler einzelner Befehle erzeugt werden müssen. Es ist bedeutend komfortabler für die Entwicklung, wenn häufig verwendete Methoden abstrahiert sind. Dies wird in diversen, zumeist frei verfügbaren, APIs realisiert. Die beiden, welche in die nähere Auswahl kommen, sind OpenSceneGraph und OGRE.

OpenSceneGraph, oder kurz OSG, ist 1998 von Don Bruns ins Leben gerufen worden und wie sich aus dem Namen schließen lässt, steht der Szenegraph (4.2.3) im Mittelpunkt. Es ist nicht zu verwechseln mit OpenSG oder OpenProducer, welche zum Teil von den gleichen Entwicklern und zur selben Zeit entstanden sind. OGRE steht für *Object-Oriented Graphics Rendering Engine* und entstand etwa 2001. Sie ist ebenfalls Szenen-orientiert und spezialisiert auf hardwarebeschleunigte 3D-Graphik. Im Gegensatz zu OSG, unterstützt OGRE auch Direct3D.

Obwohl sich auch ansonsten die beiden Schnittstellen sehr ähnlich sind, so gibt es doch ein paar Unterschiede, welche letztlich die Entscheidung zugunsten von OSG beeinflusst haben. OGRE ist vollständiger dokumentiert und für Mainstream Hardware optimiert. Des Weiteren sind mehr Features direkt integriert und müssen nicht selbst implementiert werden, z.B. die Skelett-Animation. OSG hingegen ist ein wenig komplexer, da man sich mehr mit Matrizen und anderen mathematischen

[14] Teil einer graphischen Benutzeroberfläche

Zusammenhängen auseinandersetzen muss. Dies ist jedoch für eine wissenschaftliche Arbeit eher förderlich, zumal die Kamera-Simulation sich ohnehin mit diesem Thema beschäftigt. Den Ausschlag jedoch gibt die eingebaute Simplifizierung, also die Möglichkeit 3D-Geometrie in der Anzahl Eckpunkte (Vertices) zu reduzieren und somit weniger Rechenaufwand zu erzeugen. Da die Performance eine kritische Variable in dieser Arbeit ist, und gleichzeitig der Fokus dieser Arbeit auf der Simulation und nicht auf der 3D-Grafikentwicklung liegt, wodurch eine eigens implementierte Optimierung des Renderings entfällt, ist OSG das eingesetzte Produkt.

4.2.3 Szenegraphen

Eine Szenegraph ist eine Datenstruktur, welche die logische und räumliche Beziehung in einer graphischen Szene beschreibt mit dem Ziel die Daten effizient zu verwalten und graphische Elemente zu rendern. Der Graph ist typischerweise hierarchisch aufgebaut, beginnend mit einem Top-Level Wurzel-Knoten (Root-Node), einer Menge Gruppen-Knoten (Group-Nodes) mit einer unbestimmten Anzahl Kind-Elementen (Child-Nodes), und einem Satz End-Knoten (Leaf-Nodes) als letzte Ebene des Baumes. In der Regel hat ein Szenegraph (Abb. 4.2.2) keine Schleifen und keine freistehenden Elemente ohne Eltern- oder Kind-Beziehung. [XQ10] Es ist jedoch möglich, dass ein Element mehrere Eltern-Elemente hat. Ein solcher gerichteter, azyklischer Graph wird hier jedoch nicht weiter behandelt, da er für die Umsetzung nicht von Bedeutung ist.

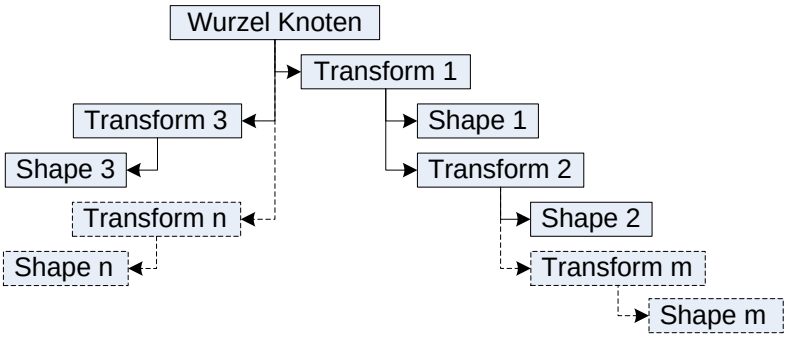


Abbildung 4.2.2: Szenegraph

Innerhalb der Gruppen unterhalb des Root-Node können Eigenschaften einzelner Elemente vererbt werden, ein Umstand welcher insbesondere für Roboter interessant ist, da so die kinematische Kette direkt in der graphischen Szene nachgebildet werden kann.

4.3 MATLAB

Wie bereits erwähnt, verfügt MATLAB über eine große Bandbreite an Funktionen aus vielen Anwendungsbereichen. Dazu gehören u.a. die Signal- und Bildverarbeitung sowie die Simulation. Für diese Arbeit besonders interessant sind drei Toolboxen, welche maßgeblich an der Umsetzung dieser Arbeit beteiligt sind. Dies sind die Robotics, Image Processing und Instrument Control Toolbox. Die mit ihnen bereitgestellten Funktionalitäten umfassen die einfache Darstellung und Simulation von Kinematik, die Bildverarbeitung sowie die Kommunikationsschnittstelle zwischen MATLAB und der 3D-Simulation mit OSG. Erstere Toolbox wurde bereits im Abschnitt 4.1.1 beschrieben.

Die Image Processing Toolbox liefert Methoden für die Gewinnung und Konvertierung von Bilddaten, die den Input für die Analyse mit Algorithmen wie SURF (Speeded Up Robust Feature) bereitstellen.

Dabei geht es um die Definition und Wiedererkennung von markanten Punkten in Bildern. Hat man die Merkmalspunkte eines bekannten Objektes gespeichert, kann man dieses Objekt aufgenommen von einer Kamera anhand dieser Punkte identifizieren. Dies kann verwendet werden, um ein Objekt mit dem Roboter anzufahren und im Falle des Katana, zu greifen.

Zur Kommunikation zwischen externen Geräten kann die Instrument Control Toolbox verwendet werden. Sie stellt neben TCP/IP auch Schnittstellen zu USB oder seriellen Schnittstellen bereit. Im Anwendungsfall einer Client-Server Anwendung, wird mit ihr die Rolle eines MATLAB-Programmes definiert und das Senden und Empfangen von Daten gesteuert.

4.4 Interaktion

Bei der gegebenen Zielsetzung gibt es diverse Möglichkeiten, wie der Anwender bzw. der Roboter mit seiner Umgebung interagiert. Auf der einen Seite, wird der Roboter von MATLAB aus kontrolliert, er kann etwa durch Nutzen seines Greifarmes, Objekte manipulieren. Auf der anderen Seite können sich unter Umständen auch Objekte in der Simulation frei bewegen. Auf seinem Weg zum Objekt können Hindernisse zu überwinden sein. Zu Beginn einer Simulation kann ein zu greifendes Objekt teilweise oder vollständig verdeckt sein oder aber die Umgebung kann sich während des Anfahrens einer Pose verändern. All diese und weitere mögliche Szenarien müssen ebenfalls von der Architektur berücksichtigt werden. Dies erfordert robuste Algorithmen und zuvor definierte Schnittstellen. Die folgenden Unterpunkte beschreiben zwei Interaktions-Schnittstellen, die während der laufenden Simulation angesprochen werden können.

4.4.1 Interaktion via Netzwerk

Der Manipulator kann über Steuerungskommandos, welche er über das Netzwerk empfängt, bewegt werden. Im Standardfall werden Zeichenketten über ein TCP/IP Socket gesendet. Gleichzeitig dient dieses als Sender der Bilddaten der Kamera. Das verwendete Protokoll muss jedoch nicht zwingend über TCP/IP implementiert werden. Es ist ebenso denkbar, die Befehle über eine CAN-Bus Implementierung zu realisieren oder die Bilder als Stream via UDP zu senden. Ein solcher Videostream kann via Multicast an eine Vielzahl Empfänger gesendet werden und so mit unterschiedlichen Algorithmen bzw. Anwendungen analysiert werden. Die Netzwerkschnittstelle, wie auch immer realisiert, soll einfach zu konfigurieren und durch andere Technologien ersetzbar sein.

Neben der Hauptfunktion der Steuerung des Roboters, kann auch die Simulationsumgebung mit ihren Objekten über das Netzwerkprotokoll manipuliert werden. Dies ist mithilfe eines definierten Protokolls ohne Weiteres möglich.

Die Veränderung von Parametern, welche nicht unmittelbar mit der Roboter-Kinematik zu tun haben, kann jedoch auch anders realisiert werden, wie im nächsten Abschnitt dokumentiert ist.

4.4.2 Interaktion via manueller Steuerung

Intuitiver wird die Steuerung über Eingabegeräte (HIDs) wie Maus und Tastatur. Zuvor definierte Objekte der Umgebung können markiert / ausgewählt und bewegt werden. Mithilfe von Tastatur Befehlen kann man z.B. das Licht an oder aus machen. Dies führt automatisch zur Frage nach einem GUI, da eine einfache Bedienoberfläche sofort das Potential der Anwendung erhöht. So kann der Benutzer direkt in die laufende Simulation eingreifen, etwa eine andere Bahnplanung erzwingen,

indem Hindernisse in den Weg gelegt werden, oder die Lichtverhältnisse ändern, um die Bildanalyse-Algorithmen zu testen.

Der Einsatz einer GUI bringt noch weitere Möglichkeiten, der große Nutzen darin liegt jedoch in der erhöhten Usability. Diese ist der Schlüssel, um Einsteiger oder Personen aus anderen Fachbereichen schneller an eine solche Anwendung heranzuführen und so Entwicklungen voranzutreiben.

4.5 Parallelität

Die bisher vorgestellten Elemente der Simulationsumgebung weisen eine große Gemeinsamkeit auf: Sie laufen alle gleichzeitig. Während der Simulation können sowohl Daten empfangen als auch gesendet werden. Parallel zur Berechnung der Bahn kann sich die Umgebung verändern, Bilddaten werden gesendet und analysiert, Steuerungsbefehle werden gesendet und simultan müssen die 3D-Daten gerendert werden. Somit ist ein Multi-Threading Ansatz unabdingbar beim Design der Applikation.

Da auf einige Ressourcen parallel sowohl lesend als auch schreibend zugegriffen wird, müssen die einzelnen Tasks synchronisiert werden. Dies ist erforderlich, da sonst unvorhergesehene Zustände eintreten können, die zum einen die Programmstabilität beeinträchtigen als auch die Korrektheit der Simulation. Die zu schützenden Ressourcen sind:

- *Kinematik-Daten*: Während ein Frame gerendert wird, bzw. der Roboter eine Pose annimmt, müssen alle Gelenkkoordinaten konstant sein.
- *Bild-Daten*: Zwischen zwei berechneten Bildern, muss Zeit genug sein, das vorhergehende zu speichern / senden.
- *Netzwerk*: Das Senden und Empfangen von Daten muss synchronisiert werden, u.a. um Sicherzustellen, dass das zuletzt gesendete/gespeicherte Bild immer dem entspricht, worauf die Bahnplanung basieren soll.

Die detaillierte Implementierung wird im Abschnitt 5.7 beschrieben.

Obwohl die Gelenkkoordinaten eine kritische Ressource sind, so kann doch das Rendering asynchron mit dem Empfang neuer Positionsdaten durchgeführt werden. Auch wenn keine neuen Daten vorhanden sind, darf ein neuer Frame generiert werden. So kann z.B. die Position von Objekten im Simulationsraum oder die Lichtverhältnisse verändert werden und der Benutzer das Ergebnis seiner Handlungen sofort sehen.

4.6 Plattformunabhängigkeit

Obwohl die Simulation auf einer Microsoft Windows Plattform entwickelt und getestet wurde, besteht die Möglichkeit, mit einfachen Mitteln, Teile oder gar die ganze Anwendung auf UNIX/Linux Systeme zu portieren. Es wurde großen Wert darauf gelegt, Bibliotheken und Methoden zu verwenden, die bereits für diverse Plattformen existieren.

Dies ist bei sämtlichen Applikationsmodulen möglich, beim Thema Benutzeroberfläche jedoch nicht. UNIX-basierte und Windows Betriebssysteme haben jeweils völlig unterschiedliche Implementierungen ihrer graphischen Oberflächen. Während Windows vom Kern schon vollkommen auf grafik-basierte Ausgabe ausgelegt ist, ist das X-Window System von UNIX nur ein Aufsatz, der ausgetauscht werden kann. Hier wird daher nur auf die Mittel zurückgegriffen, die OSG liefert. Dies ist zwar etwas weniger

komfortabel, erzeugt jedoch gleichzeitig auch weniger Overhead als eine Implementierung mit Microsoft Foundation Classes (MFC) oder X. Das Thema GUI wird im Abschnitt 5.6 näher behandelt.

4.7 Auswahl der Architektur

Die aufgezeigten Möglichkeiten lassen sich in unterschiedlichen Designs unterbringen. Die gewählte Architektur wird nun im folgenden erläutert.

In Abbildung 4.7.1 wird der Datenfluss dargestellt.

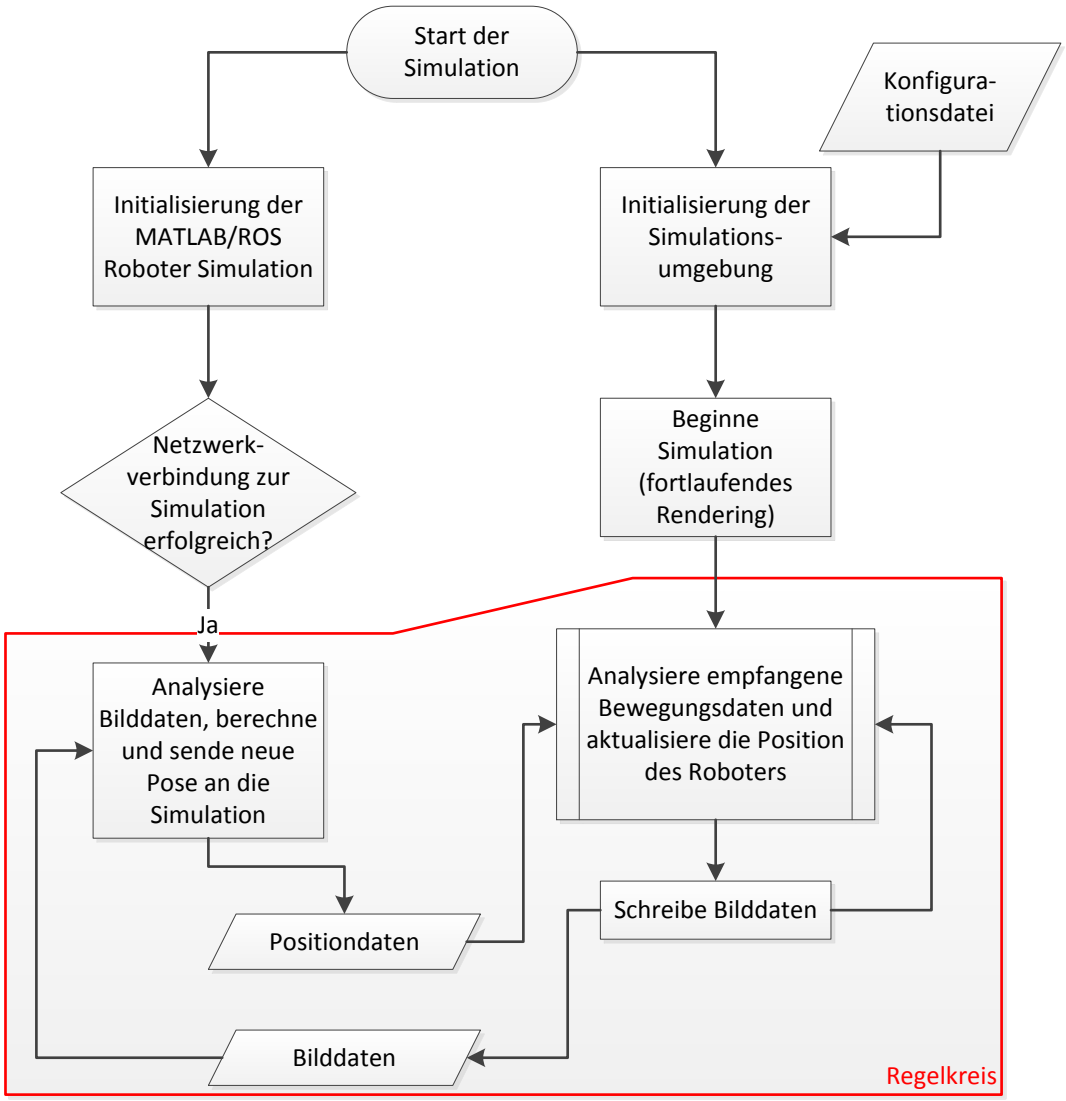


Abbildung 4.7.1: Datenflussdiagramm

Dieser ist dem klassischen Regelkreis der Robotik *Wahrnehmen -> Planen -> Handeln -> Wahrnehmen* nachempfunden. Nach der Initialisierung der Szene über die Konfigurationsdatei, bleibt die Anwendung bis zu ihrer Beendigung in dieser Schleife. Die Kamera sendet ihre Bilder an MATLAB, dort werden sie analysiert und die nächste Pose wird errechnet und zurückgesendet. Nachdem der Manipulator diese eingenommen hat, also die virtuellen Aktoren (Gelenkmotoren) bewegt wurden, werden die nächsten

Bilder gesendet und der Kreislauf beginnt von vorn. Aus dem Datenfluss ergibt sich gleichzeitig der Versuchsaufbau mit MATLAB auf der einen und der Simulation auf der anderen Seite.

Das gewählte Klassendiagramm ist ungleich komplexer. Es muss auf der einen Seite der Client-Server Architektur Rechnung tragen und auf der anderen Seite dem Subsumption Ansatz gerecht werden. Zunächst werden im Fachklassendiagramm (Abb. 4.7.2) die Hauptkomponenten abgebildet. Dabei ist die Simulationsumgebung das zentrale Element, von wo aus alle anderen Klassen angesprochen werden. Die zweitwichtigste Entität ist die Konfiguration, da durch diese bestimmt wird, welche Module die Simulation beinhaltet und wie sich diese verhalten.

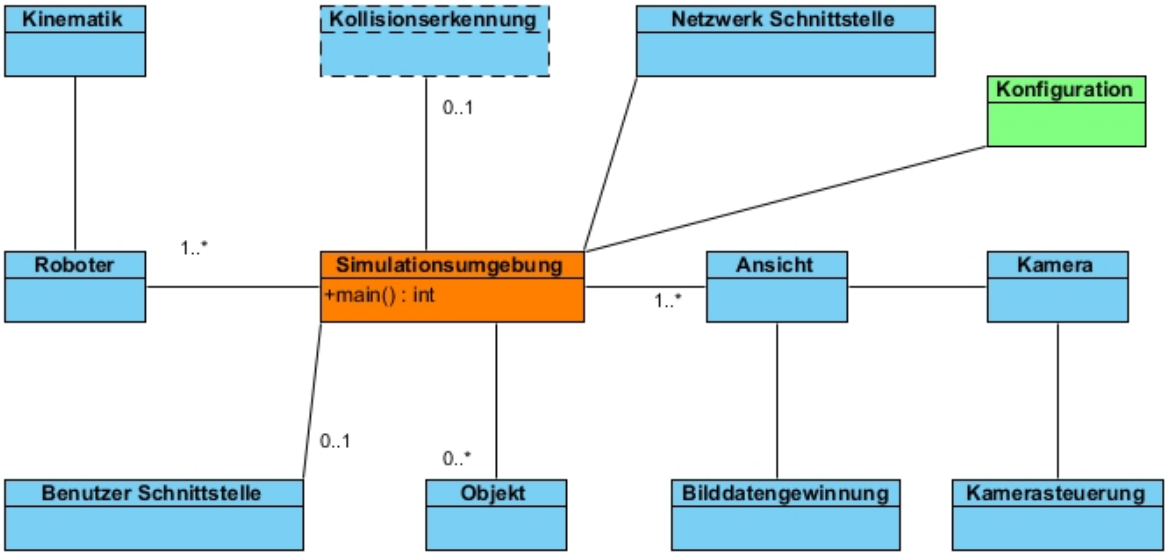


Abbildung 4.7.2: Fachklassen Diagramm

Die Aufgaben der anderen Klassen sind im wesentlichen durch ihre Namen gegeben. Ihre Abhängigkeiten und Ausgestaltung werden zu einem späteren Zeitpunkt, während der Implementierung, näher spezifiziert. Gut zu erkennen ist aber, dass bewusst keine zirkulären Abhängigkeiten entstanden sind, was dem Prinzip der hohen Kohärenz bei geringer Kopplung nachkommt.

Die Roboter- und Bewegungsklasse beinhalten die Aktor-Simulation. Aufgrund ihrer Komplexität werden diese im Folgenden näher erläutert. Abb. 4.7.3 zeigt sowohl die Abhängigkeiten als auch die Methoden. Jeder Roboter hat neben einem eindeutigen Namen zwei Zeiger, einen auf das erste Transform und einen auf das letzte Transform. In der Regel sind dies der Fuß und das Werkzeug. Ein Roboter-Objekt ist analog zur kinematischen Kette aufgebaut. Vom `rootPat_` aus werden die einzelnen Elemente aneinander gehängt und je nach Konfiguration mit Geometrie, Farbe und Koordinatensystem versehen. Falls es sich um ein bewegliches Teil handelt, wird je ein Movement Callback als Update Callback, der bei jedem Frame ausgeführt wird, hinzugefügt. So wird sichergestellt, dass die Gelenke nie manuell, sondern nur über den Callback manipuliert werden, was Sicherheit bringt. In der `main` Klasse kann dem Roboter dann eine Kamera sowie weitere Funktionalität, wie Kollisionserkennung, hinzugefügt werden.

Die drei statischen Methoden `createAxis`, `setPose` und `rgbToVec4` werden zwar hauptsächlich in der Roboter-Klasse verwendet, sie können jedoch auch an anderer Stelle nützlich sein. Für die nächste Architektur-Iteration ist geplant, diese statischen Methoden auszulagern, um unnötige Abhängigkeiten zu vermeiden.

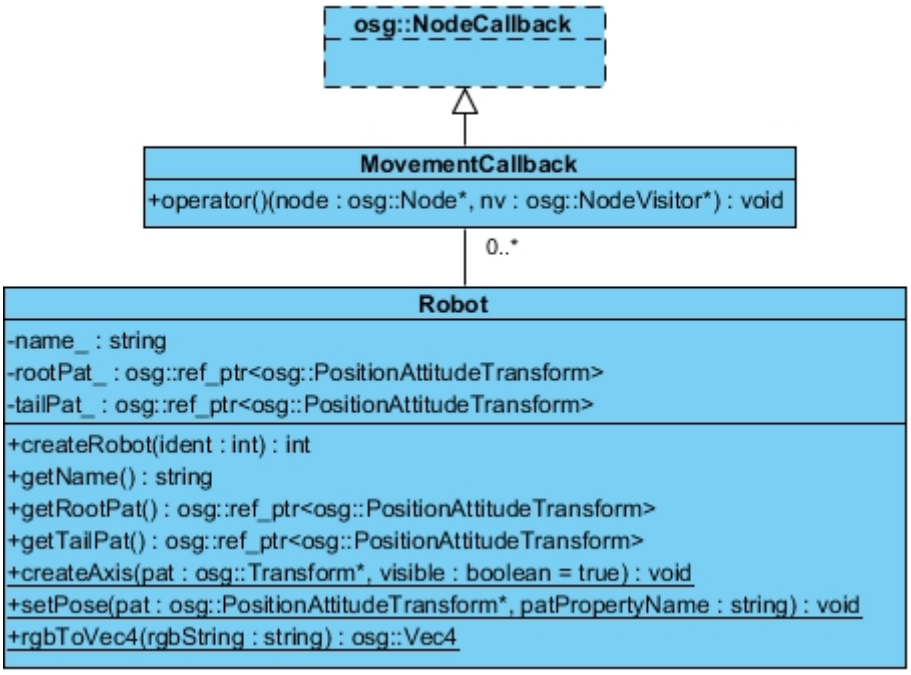


Abbildung 4.7.3: Roboter Klasse

Aufgrund des Umfangs kann an dieser Stelle nicht auf alle Details der Architektur eingegangen werden. Konkretisiert wird das Design im Kapitel 5, wo auch auf die Implementierung näher eingegangen wird. Zuvor werden noch einige getroffene Designentscheidungen des Architekten (4.7.1) erläutert.

4.7.1 Weitere Architekturentscheidungen

Neben den bisher genannten Einflussfaktoren auf die Architektur, gibt es noch weitere Bedingungen, bzw. Entscheidungen des Architekten, die erläutert werden müssen. Dazu gehört die Art des Builds. Es besteht die Möglichkeit, die verwendeten Bibliotheken statisch oder dynamisch zu verlinken. Hier wurde sich für die dynamische Variante entschieden, da zum einen die Bibliotheken fertig kompiliert von OpenSceneGraph und Boost (5.2.2) vorliegen und sie so nicht neu kompiliert werden müssen, und zum anderen, da insbesondere bei der Verwendung von Plug-ins einfach .dll Dateien bekannt gemacht werden können und die Anwendung nicht neu kompiliert werden muss. Zudem ist das Executable mit dynamisch verlinkten Bibliotheken deutlich kleiner.

Aufgrund der fachlichen Abhängigkeit zwischen den Steuerungsdaten, die über Netzwerk empfangen werden, und der Bilddatengenerierung, wurde bewusst eine Kopplung zwischen diesen beiden Modulen geschaffen. Dort findet die Synchronisation über gegenseitigen Ausschluss (siehe. 3.1.1) statt.

Es wurde darauf Wert gelegt, die Architektur primär nach dem K.I.S.S. (Keep it short and simple) Prinzip zu entwickeln.

5 Simulationsumgebung

Dieses Kapitel befasst sich mit den Themen rund um die Implementierung der spezifizierten Architektur und der Vorgehensweise bei der Anwendungsentwicklung. Zu Beginn wird die Entwicklungsumgebung (5.1) beschrieben sowie auf die verwendete Hard- und Software eingegangen. Anschließend entstehen in den Sektionen 5.3 bis 5.8 Schritt für Schritt die Komponenten und in 5.9 werden diese zusammengefügt.

5.1 Entwicklungsumgebung

Als Entwicklungsplattform wurde ein Intel x86 PC mit Microsoft Windows 7 Professional ausgewählt. Diese ist einfach zu konfigurieren und die verwendeten Softwarepakete liegen bereits komfortabel im Binärformat vor. Dies erleichtert die Einrichtung und spart Zeit, die für die Umsetzung der Arbeit benötigt wird. Obwohl beinahe sämtliche verwendete Software im 64-Bit Format vorliegt, wird davon abgesehen, das Endprodukt in dieser Architektur zu entwerfen. Dies hat zwei Gründe:

1. 64-Bit Programme können ausschließlich auf 64-Bit Architektur ausgeführt werden; sie sind nicht abwärtskompatibel.
2. Die 64-Bit Pakete, insbesondere die Bibliotheken, sind zum Teil instabiler und nicht in dem Umfang getestet wie ihre 32-Bit Gegenstücke.

Um das Ergebnis für eine große Bandbreite an Nutzern verfügbar zu machen, wird daher für 32-Bit entwickelt. Nichtsdestotrotz kann die Entwicklungsumgebung voll 64-Bit fähig sein.

Die eingesetzte Hardware besteht aus einem AMD Vier-Kern-Rechner mit je 4.2 GHz und 16 GB Arbeitsspeicher sowie einer 3D-fähigen Grafikkarte (AMD Radeon HD 6800 Serie) mit nochmals 1GB eigenem Speicher. Die eingesetzte integrierte Entwicklungsumgebung (kurz IDE) ist Microsoft Visual Studio 2010 (VS).

5.2 Werkzeuge

Neben Visual Studio kommen zahlreiche Hilfsmittel zum Einsatz. So wird für die Versionsverwaltung Subversion (SVN)[15] eingesetzt. Dies ist notwendig, da eine solche Neuentwicklung diverse Möglichkeiten zur Implementierung bietet, darunter auch solche, die eventuell zu Beginn erzeugt, zwischendurch verworfen und am Ende doch wieder genutzt werden. Um via Netzwerk gesendete Daten auf Fehler hin zu analysieren wird Wireshark[16] verwendet.

Für diese Arbeit spielen noch weitere Werkzeuge eine wichtige Rolle. Dies sind zum einen Werkzeuge für die 3D-Modellierung (5.2.1) sowie die neben OSG eingesetzten Bibliotheken (5.2.2).

[15] `http://subversion.apache.org/`
[16] `http://www.wireshark.org/`

5.2.1 Modellierung

Obwohl es möglich ist, direkt mit OSG 3D-Geometrie zu erstellen, ist es deutlich komfortabler, dies mit externen Applikationen zu tun. Sobald mehr als einfache Geometrie wie Kisten und Kugeln benötigt wird, werden Tools wie Blender, CATIA oder 3D Studio MAX zur Generierung der 3D-Modelle eingesetzt. Das Modell wird direkt in 3D geformt, d.h. es muss nicht explizit über mit Linien und Kurven verbundene Punkte manuell erzeugt werden.

Blender und 3D Studio Max von Autodesk[17] sind für 3D-Modellierung für Animation und Rendering ausgelegt, während Dassault CATIA speziell im CAD Bereich in der Konstruktion eingesetzt wird, um exakte Bauteile, Baugruppen und Zeichnungen zu erzeugen. In dieser Arbeit wird Blender für Freiform-Objekte eingesetzt, da dies im Gegensatz zu 3ds Max Open Source ist. CATIA wird im weiteren Verlauf benötigt, um die 3D Daten des Roboters für die Simulation aufzubereiten. Siehe dazu Abschnitt 5.4.

Neben CATIA gibt es diverse andere CAD Applikationen mit ähnlichen Funktionen, wie beispielsweise ProEngineer oder SolidWorks. Da die Erfahrung des Entwicklers dieser Applikation mit CATIA in der Vergangenheit positiv waren, wurde dieses Produkt gewählt.

5.2.2 Bibliotheken

C++ ist eine sehr vielseitige Sprache, die es ermöglicht, auf Basis einfacher Befehle komplexe Methoden zu erzeugen. Im Auslieferungszustand ist auf einem x86 System in der Regel jede Funktionalität enthalten, die das Betriebssystem zulässt. Diese schließen auch Multi-Threading, Netzwerk und String-Verarbeitung ein. Im Gegensatz zu JAVA, welches plattformunabhängig ist, ist bei C++ jedoch darauf zu achten, dass z.B. auf Windows andere Threading Mechanismen bereitstehen als auf Unix/Linux. Diese Tatsache sorgt dafür, dass entweder ein Cross-Compiler verwendet werden muss, um beispielsweise von Windows aus ein ausführbares Programm für Linux zu generieren oder die Quellen direkt auf dem Zielsystem kompiliert werden müssen. Zudem kann es auch nötig sein, Bibliotheken für die unterschiedlichen OS zu verwenden, auf denen das entwickelte Programm laufen soll. Im schlimmsten Fall kann es sogar erforderlich sein, den Code der Anwendung umzuschreiben.

Um den so entstehenden Aufwand möglichst gering zu halten, können Bibliotheken eingesetzt werden, welche die betriebssystemspezifischen Funktionalitäten abstrahieren und/oder die es allen eingesetzten OS gibt. In dieser Arbeit wird die C++ Library Sammlung *Boost* in der Version 1.51 verwendet. Dafür gibt es folgende Gründe:

- *Vielseitigkeit*: Boost deckt ein sehr großes Spektrum an Funktionalität ab und ist auf vielen x86 OS kompilierbar. Für Windows mit Visual Studio können die .lib und .dll Dateien fertig kompiliert über boostpro[18] heruntergeladen werden. Netzwerk, Threading und reguläre Ausdrücke sind nur einige Punkte, die das Paket beinhaltet, welche in diese Arbeit verwendet werden.
- *Aktualität*: Die Bibliotheken von Boost unterlaufen einen permanenten Verbesserungsprozess. Die Software wird von vielen Projekten und Institutionen verwendet und weiterentwickelt. Die Update Zyklen sind sehr kurz, so dass auf etwaige Bugs schnell reagiert werden kann. Über Apache Subversion[19] ist eine vollständige Versionshistorie abrufbar.

[17] `http://www.autodesk.de/adsk/servlet/pc/index?id=14642267&siteID=403786`
[18] http://www.boostpro.com/download/
[19] ein freies Versionsmanagementsystem für Software

- *Reputation*: Zehn Boost Bibliotheken sind aufgenommen im C++ Standards Committee's Library Technical Report (TR1) und im neuen C++11 Standard. C++11 enthält noch weitere Boost Libraries zusätzlich zu denen aus dem TR1. [Boo12] "...one of the most highly regarded and expertly designed C++ library projects in the world."— [AA04]

Den Nutzen, der unter anderem daraus entsteht, wird nun in drei Exempeln erläutert.

Netzwerk

Die in dieser Arbeit beschriebene Anwendung verwendet TCP/IP zur Kommunikation zwischen der 3D-Simulation und einem Kontroll-Client, beispielsweise MATLAB. Das Netzwerkprotokoll dafür kann unterschiedlich implementiert werden. Unter Windows steht die von Microsoft adaptierte Implementierung der Berkeley Sockets bereit. Diese ist zwar im Wesentlichen vom POSIX[20] Standard abgeleitet, es gibt jedoch diverse Unterschiede, welche die Portierung erschweren. Die Windows Socket API, kurz WSA, benutzt eigene Objekte, welche unter UNIX nicht zur Verfügung stehen. Um die Software auf beiden Systemen verwenden zu können, müsste das Netzwerk-Modul gleich in zwei Varianten implementiert werden.

Die Boost::ASIO Bibliothek macht jedoch diese Differenzen für den Anwender transparent. Durch Angabe eines einzigen DEFINE Makros kann zwischen der UNIX und der Windows Variante gewählt werden, ohne den Code zu verändern und ohne funktionale Einbußen.

Verarbeitung von Zeichenketten

Die über Sockets kommunizierende Anwendung versendet Zeichenketten, die interpretiert werden müssen. Die Kommandos werden zerlegt, beispielsweise in Key-Value Paare, und anschließend weiter verarbeitet. Die unter C++ vorhandenen Low-Level Befehle zum Arbeiten mit Character-Arrays sind ausreichend, jedoch nicht komfortabel genug, um Modifikationen an den übermittelten Daten mit geringem Aufwand durchführen zu können. Die Konvertierung (Cast) von einem zum anderen Datentypen stellt ebenfalls eine Hürde dar, da ein falscher Cast, etwa von String zu Double, die Stabilität der Anwendung beeinflussen kann. Boost stellt auch hier mit einer eigenen *split* Methode zum Zerteilen von Zeichenketten und dem *lexical cast* zur Typkonvertierung Methoden bereit. Diese bieten hohen Komfort und Sicherheit bei der Stringverarbeitung und erlauben schnelle Adaption bei neuen oder veränderten Anforderungen an das Netzwerkprotokoll.

Multi-Threading

Es stehen bei dieser Arbeit gleich vier Möglichkeiten der Realisierung von Threads und Interprozesskommunikation zur Verfügung. Natives Windows Threading widerspricht dabei dem plattformübergreifenden Einsatz der Software. Mit dem POSIX Standard *pthread* steht eine solide Implementierung bereit, die auch für Windows verfügbar ist. Hinzu kommt OpenThread, welches mit OSG geliefert wird. Als beste Lösung wurde jedoch auch hier die Boost Bibliothek bewertet. Sie verwendet, je nach OS, die optimale Thread Implementierung. Neben den Standard Funktionen zum Erzeugen und Synchronisieren von parallelen Abläufen bietet sie sog. *scoped locks*, also kontextabhängigen Schutz von gemeinsam genutzten Ressourcen.

Auch hier vereinfacht die Verwendung von Boost die Programmierung.

[20] Portable Operating System Interface

5.3 Die Welt

Dieser Abschnitt beschreibt die Erstellung der Umgebung, die als Basis für 3D-Elemente wie Roboter und Objekte dient. Hier werden zum einen alle Elemente definiert, die Einfluss auf andere sich in der Simulation befindenden Objekte haben, wie Licht und Schatten. Hinzu kommt der *Boden*, der aus praktischen Gründen aus einem Raster besteht. Des Weiteren ist hier definiert, ob und welche Interaktion in den dargestellten Views möglich ist.

Für eine realitätsnahe Simulation sind Licht und Schatten unbedingt notwendig. Die Bildanalyse muss Objekte auch bei diffusem Licht oder im Halbschatten erkennen können. Diese sind unter OSG mit recht einfachen Mitteln realisierbar. Der Schattenwurf wird durch eine *ShadowMap* generiert und der komplette Szenegraph unter eine *ShadowedScene* gehängt. Dadurch wird der Schatten für alle Objekte im Szenegraphen (Abb. 5.3.1) berechnet und gerendert. Die Anzahl Lichtquellen bestimmt, wie komplex die ShadowMap ist, bzw. wie viel Performance die Schattenberechnung benötigt. Es gilt: Der Rechenaufwand wächst linear mit der Anzahl aktivierter Lichtquellen. Unter OpenGL können diese an beliebiger Stelle unterhalb des Wurzelknotens positioniert werden. Ein einfaches Licht wird durch drei Parameter definiert: Die Position, das Umgebungslicht (Ambient) und die Streuung (Diffuse). Wie die meisten Parameter werden auch diese über Vektoren definiert.

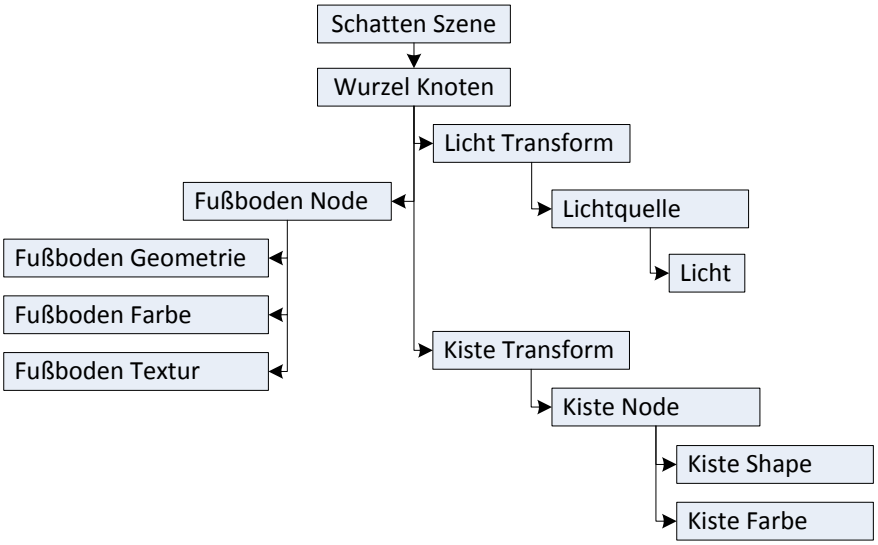


Abbildung 5.3.1: Basis Szenegraph

Der Boden wird vollständig über die Konfigurationsdatei definiert. Er erstreckt sich vom Ursprung (0,0,0) in positive wie negative x- und y-Richtung um eine vorgegebene Ausdehnung. Die Textur kann frei gewählt werden, es wurde jedoch ein Gitternetz gewählt, um Entfernungen besser sehen zu können. Die Implementierung des Basis-Szenegraphen aus Abbildung 5.3.1 wird in der Simulationsumgebung wie folgt dargestellt (Abb. 5.3.2).

Deutlich zu erkennen ist der Schatten vor der Box, welcher auf eine Lichtquelle schräg hinter ihr schließen lässt. Ebenfalls gut sichtbar ist die Gitternetz Textur, welche den quadratischen *Fußboden* überzieht. In diese Umgebung kann nun alles Weitere platziert werden. Der Schatten sowie die unterschiedlichen Lichter werden von der .properties Datei aus konfiguriert. Die Implementierung eines Lichtschalters über CONTROL Befehle ist zu einem späteren Zeitpunkt angedacht.

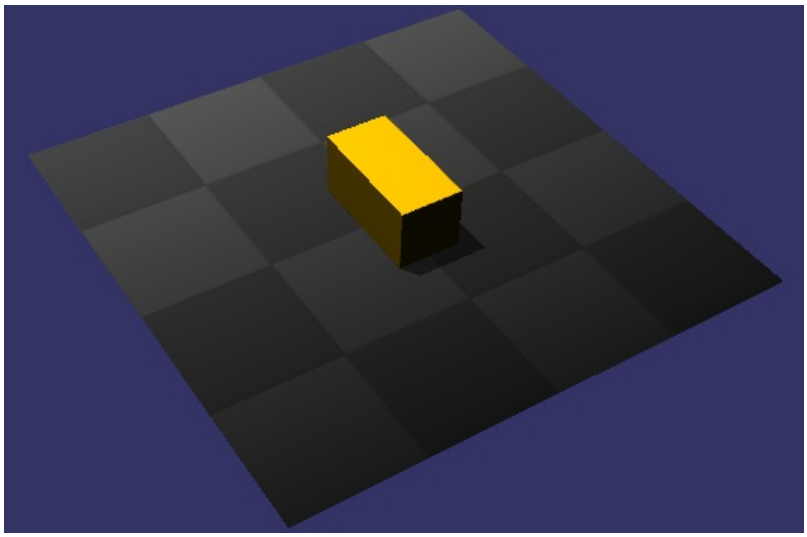


Abbildung 5.3.2: Basis Simulationsumgebung

Um nicht für beliebig große Ebenen die Texturen zu vergrößern oder deren Auflösung zu erhöhen zu müssen, wird dafür gesorgt, dass sich eine Textur vervielfältigen lässt und so die ganze Fläche ausfüllt. Wie oft sie sich wiederholt, kann durch den Parameter `FLOOR_GRAPHICS_DIMENSIONS` in der Konfigurationsdatei festgelegt werden. Dies wird durch folgende Formel beschrieben:

$$\frac{FLOOR\_DIMENSIONS}{FLOOR\_GRAPHICS\_DIMENSIONS} = Anzahl\ der\ Wiederholungen \tag{5.3.1}$$

Neben diesem in Abb. 5.3.2 verwendeten *Rundstrahler* können auch Scheinwerfer, also gerichtete Lichtquellen erzeugt werden. OpenGL lässt maximal acht Lichter (`osg::Light`) parallel zu. Die Anzahl angezeigter Lampen kann jedoch durch Mehrfachverwendung ein und desselben Lichtes an mehreren Positionen erhöht werden. Um für jede Lichtquelle einen eigenen Schatten zu erzeugen, ist jedoch erhöhter Aufwand nötig, der hier nicht getätigt wurde.

Ein einfacher Rundstrahler mit hellem Licht in der Mitte der Szene in einer Höhe von $1.50m$ über dem Boden wird wie folgt definiert:

```
LIGHT_1_ENABLE = 1
LIGHT_1_POSITION = .0;.0;1500.0
LIGHT_1_COLOR = #FFFFF0
LIGHT_1_AMBIENT = #000000
```

Der Einsatz von Scheinwerfern / Spotlights ist etwas aufwändiger und im Beispiel `osgspotlight` aus dem OSG Paket beschrieben. Dafür müssen dem `osg::Light` zusätzlich noch die Richtung und der Abstrahlwinkel mitgegeben werden.

Beim Testen der Robustheit von Algorithmen zur Bildverarbeitung spielen die Lichtverhältnisse eine große Rolle. Mit der Möglichkeit der Definition unterschiedlicher Lichtquellen und in Addition der Veränderung während der Simulation ist diesem Punkt der Anforderung ausdrücklich nachgekommen worden.

5.4 Der Roboter

Wie bereits zu Beginn dieser Arbeit angeführt, wird der Katana von Neuronics als Beispielroboter in die Simulation eingebunden. Neuronics stellt dafür ein STEP-Modell des Knick-Arm-Roboters sowie verschiedene Greifer zur Verfügung. Die nach ISO 10303-21 enthaltenen Produktstrukturdaten müssen zunächst in ein Format gebracht werden, welches mit OSG geladen werden kann. Dies geschieht unter Zuhilfenahme von CATIA.

Im ersten Schritt wird das STEP-Modell analysiert. Unbearbeitet besteht es aus dem Roboter ohne Endeffektor sowie einer Darstellung der maximalen Auslenkungen (Abb. 5.4.1).

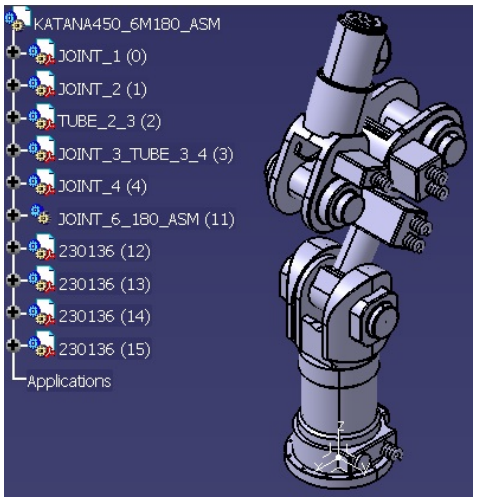
Abbildung 5.4.1: Katana im STEP-Modell

Da die Verfahrwege keine direkte Bedeutung für die Simulation haben, werden diese entfernt, so dass nur die echte Geometrie übrig bleibt, welche aus sechs Armteilen besteht. Es ergeben sich fünf Bauteile (CATParts) sowie ein Zusammenbau oder auch Assembly (CATProduct) für die Aufnahme des Endeffektors. Als Werkzeug wurde ein Winkelgreifer gewählt, der in einer eigenen STEP-Datei geliefert wird. Dieser besteht wiederum aus mehreren Parts und Assemblies.

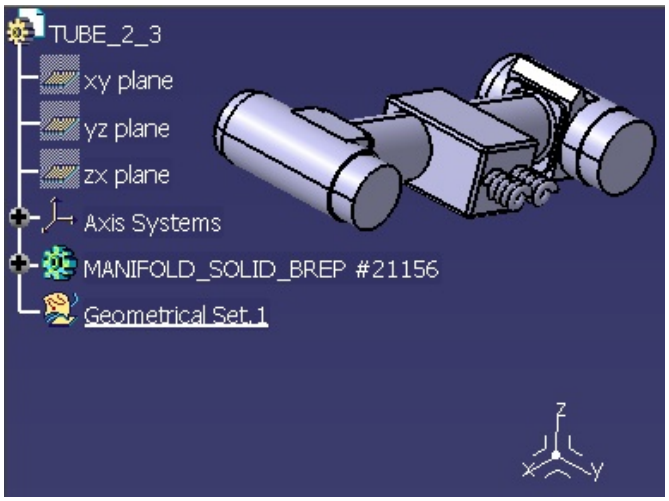
Die einzelnen Bauteile wie Fuß, Drehgelenke und Armteile müssen anschließend in ein Format konvertiert werden, dass von OSG unterstützt wird. Dafür bietet sich das Standardformat STL (kurz für Surface Tesselation Language) an. Wie der Name schon sagt, wird die Geometrie dabei in eine Anzahl Dreiecke zerlegt, also tesseliert. Dadurch lassen sich zwar keine mathematisch korrekten Radien erzeugen, bei einer hohen Anzahl Dreiecke können jedoch sehr ansehnliche runde Flächen und Körper erzeugt werden. [Wik13b]

Zuvor muss jedoch das Problem mit den verschobenen Teil-Koordinatensystemen gelöst werden. Dieses ist nämlich aus dem Assembly des Katana übernommen und liegt somit entweder im Zentrum des Fußes oder an einer anderen Stelle im Raum. Dies ist hinderlich und würde bei nicht Behebung dazu führen, dass für die Gelenkmanipulation eine zusätzliche Matrix-Multiplikation stattfinden müsste. Siehe dazu Abb. 5.4.2a bis 5.4.2c.

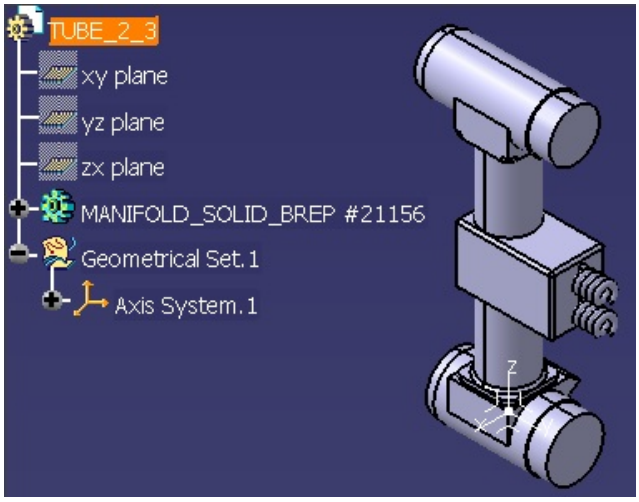
CATIA unterstützt jedoch die Funktion, die Position eines Bauteils zu seinem Koordinatensystem zu verändern. So ist es möglich, das Koordinatensystem direkt in das jeweilige Gelenk zu legen, so dass die



(a) Katana Zusammenbau



(b) einzelnes Armteil (vorher)



(c) einzelnes Armteil (nachher)

Abbildung 5.4.2: CATIA Modelle

zukünftige Anwendung der Denavit-Hartenberg Parameter vereinfacht wird. Nach diesem Schritt kann das jeweilige Bauteil nach STL exportiert werden.

Für die Aufnahme des Tools und das Tool selbst, in diesem Fall der Winkelgreifer, müssen jedoch zwei weitere Schritte vorab geschehen. Da es unnötiger Aufwand ist, jede Schraube und jedes Plättchen einzeln in den Szenegraphen einzubauen, werden diese Assemblies nicht in ihre Einzelteile zerlegt und konvertiert, sondern *zusammengeschweißt*, so dass je ein Teil für die Werkzeug-Aufnahme und das Werkzeug entstehen. Dies geschieht über die Funktion *CATPart aus Produkt generieren*. Der Winkelgreifer könnte zwar ein Transform bilden, da er schließlich auf und zu geht, dies aber ist für das Anfahren von Objekten nicht relevant. Daher wurde die offene Stellung des Greifers für die Konvertierung gewählt. In einer zukünftigen Version könnte ebenfalls die Greifbewegung des Werkzeugs dargestellt werden.

Beim Schritt der Verwendung der 3D-Daten im Code der Anwendung, müssen noch einige Dinge geklärt werden. Vorweg wird festgelegt, dass die Grenzen der Auslenkung der einzelnen Glieder des Roboters in der Steuerung, also in MATLAB, verankert werden. So wird vermieden, dass der Roboter in der Simulation Posen erreicht, die physikalisch nicht möglich sind. Außerdem wird so verhindert, dass der

Roboter im realen Betrieb sich selbst beschädigt. Des Weiteren werden in dieser Version der Anwendung nur lineare kinematische Ketten abgebildet, d.h. ein Gelenk hat maximal ein nachfolgendes.

Der Roboter, welcher in einer eigenen Klasse definiert ist, wird aus den einzelnen Gelenkteilen zusammengefügt. Zu einem 3D-Modell gehören noch die Startposition sowie die Freiheitsgrade. All dies zusammen bildet ein Gelenk (Joint). In der *createRobot* Methode wird aus den einzelnen Joints ein animierbares Robotermodell generiert. Die Gelenke sind in der Konfigurationsdatei mit Nummern versehen. Sie können so in aufsteigender Reihenfolge zu einem Roboter zusammengesetzt werden. Es entsteht ein Szenegraph, bei dem vom Standfuß des Manipulators eine Kette von Kind-Elementen mit je einem Transform (`osg::PositionAttitudeTransform`) und einem 3D-Modell (Shape) bzw. Node (`osg::Node`) bis hin zum Endeffektor entsteht. Hier werden auch die in der .properties Datei angegebenen DOFs festgelegt und der Callback (5.4.1) definiert, welcher später für die Bewegung des Roboters sorgt. Alle beweglichen Teile werden in einer global deklarierten Map `g_movement_instructions` gespeichert, damit sie später extern angesprochen werden können (siehe 5.7).

Neben der *createRobot* Methode ist in der Roboter Klasse eine statische Methode zur Erzeugung von Achsenkreuzen implementiert. Diese ermöglicht es, jedes beliebige Transform mit einem rechtshändigen, kartesischen Koordinatensystem zu versehen. So kann der Anwender auch bei komplizierten Vorgängen stets sehen, in welche Richtung die Achsen eines Gelenkes oder andere Objekte im Simulationsraum zeigen. Für bessere Visualisierung wurde eine weitere statische Methode zum Verändern der Farbe von Shapes hinzugefügt.

5.4.1 Bewegungs-Callback

Die Callback Methode für die Bewegung des Roboters, oder kurz Movement Callback, wird bei jedem neuen Render-Vorgang aufgerufen und prüft ob Bewegungs-Befehle für den jeweiligen Transform vorliegen. Falls ja, werden diese auf Gültigkeit überprüft und umgesetzt. Dies geschieht, indem nach Feststellung der Art der Winkelangabe (Grad oder Radiant) auf dem `osg::PositionAttitudeTransform` die Methoden `setPosition` und `setAttitude` aufgerufen werden.

Selbstverständlich wird durch die Verwendung eines Mutex sichergestellt, dass während der Callback aktiv ist, keine neuen Daten in den Datensatz der Bewegungskommandos geschrieben werden. So wird die Datenintegrität der neuen Pose des Gelenks sichergestellt.

Die Callback Methode selbst ist abgeleitet von der `osg::NodeCallback` Klasse. Sie implementiert das Überschreiben des *()* Operators, in welchem der beschriebene Ablauf stattfindet. Vor dem Verlassen des Operators wird mit `traverse` die Anwendung der gesetzten Parameter auf das Transform sichergestellt. Hier kommt das Kommando Design Pattern (siehe 3.1) zum Einsatz. Der vollständige Code befindet sich im Anhang A.2. Dieser Callback kann ebenso für Objekte, die keine Roboter sind, verwendet werden.

5.5 Die Kamera

In OpenSceneGraph wird eine Kamera genau wie in der realen Welt über ihre extrinsischen und intrinsischen Parameter definiert. Die Ausrichtung der Kamera kann auf verschiedene Arten festgelegt werden. Dies kann entweder über das Setzen einer ViewMatrix über Roll-, Anstell- und Gierwinkel geschehen oder aber mithilfe der Methode `setViewMatrixAsLookAt`, welche das Auge der Kamera, die Blickrichtung und den *Oben* Vektor verwendet. Bei der auf dem Roboter montierten Kamera wurde sich für die erstgenannte Methode entschieden. Für die Festlegung des Bildhauptpunktes wurde ein nicht sichtbares `osg::Transform` angelegt, welches als Teil der kinematischen Kette relativ zum Endeffektor positioniert ist und dessen Pose frei konfiguriert werden kann. So ist die Positionsveränderung der

Kamera einfach an die Position des Roboters geknüpft. Die Aktualisierung der Positionsdaten wird vom `CameraUpdateCallback` übernommen. Der Code dazu befindet sich im Anhang A.3.

Die intrinsischen Kameraparameter, also die Projektionsmatrix, können ebenfalls auf unterschiedlichem Wege definiert werden. In Anlehnung an die beschriebene Darstellung aus Abschnitt 3.3 wird dafür die Methode `setProjectionMatrixAsPerspective` ausgeführt. Über Angabe des horizontalen Bildwinkels sowie des Seitenverhältnisses und der Parameter für *nah* und *fern* wird hier das Objektiv und das Bildformat festgelegt (vgl. 3.3.1).

Mit OSG sind eine Vielzahl verschiedener Kameras realisierbar, das Spektrum reicht von der einfachen Handykamera bis hin zur Stereokamera. Es können in Abhängigkeit der Hard- und Software Performance Aufnahmen mit beliebig vielen Bildern pro Sekunde getätigt werden.

5.5.1 Views

Aus Sicht von OSG, bzw. OpenGL, ist die Kamera nur der Teil einer Ansicht/ eines Views, der beeinflusst, welcher Ausschnitt der gerenderten Szene angezeigt wird. Welche Objekte aus welcher Szene überhaupt für einen View dargestellt werden, wird bei der Erstellung desselben festgelegt. So kann etwa die Szene von der Wurzel an oder aber etwa ohne Schatten oder auch nur ein bestimmter Teil des Roboters sichtbar sein. Dieses wird über die `setSceneData` Methode eines Views festgelegt. Ein Viewer wie der `CompositeViewer` kann eine beliebige Anzahl Views enthalten, die mit `setUpViewInWindow` frei über die Anzeige verteilt werden können.

Pro View können EventHandler festgelegt sowie sog. Kamera Manipulatoren definiert werden. Letztere erlauben es, z.B. mit dem `osgGA::TrackballManipulator` die Kamera frei im Raum zu bewegen. Über EventHandler können GUI Elemente realisiert werden. Mithilfe von Tastatur Events lassen sich Bewegungsbefehle an einzelne Objekte senden (z.B. Pfeiltastensteuerung) oder die Umgebung beeinflussen, etwa das Licht an und ausmachen. Die implementierten GUI Elemente werden in 5.6 näher erläutert.

Für die Simulationsumgebung wurden zunächst drei Ansichten angelegt: Einer als Beobachter (Spectator), ein über die Maus frei navigierbarer sowie einer für die auf dem Roboter montierte Kamera. Der Spectator View ist eine im Simulationsraum fest installierte Überwachungskamera, die keine EventHandler hat und auch sonst nicht manipulierbar ist. Der frei navigierbare, im Weiteren als FreeView geführte, View hat beim Start eine orthogonale Draufsicht auf die Szene und kann um alle drei Achsen verschoben und rotiert werden. Hinzu kommt die Möglichkeit, die Statistiken des Renderings anzuzeigen. Dazu gehören die Bildrate sowie die CPU und GPU Auslastung und zugleich die aktuelle Anzahl gerenderter Polygone. Der FreeView dient dem Anwender dazu, die Bewegungen im Raum genau zu verfolgen und den Fokus bei Bedarf auf einzelne Ausschnitte der Szene zu setzten.

Der Roboterkamera-View zeigt exakt den Bildausschnitt, den auch die reale Kamera auf dem Endeffektor aufnimmt. Damit die Kamera dieser Ansicht immer korrekt zur Roboterposition ausgerichtet ist, wurde der `CameraUpdateCallback` implementiert. Um Bilder vom View zu machen und diese verschicken zu können, gibt es zusätzlich noch einen `PostDrawCallback`. Dieser wird immer ausgeführt, nachdem der View fertig gerendert ist, um einwandfreies Bildmaterial zu bekommen. Der `CameraUpdateCallback` hingegen wird während des Updates der Szene ausgeführt.

Wie der Name schon andeutet, wird der `CameraUpdateCallback` eingesetzt, um nach jedem Frame die Bildebene zu aktualisieren. Dabei wird der Callback Instanz das Transform der Kamera, welche Teil des Roboter-Szenegraphen ist, mitgegeben. Über die Methode `computeLocalToWorld` wird daraus die Kameramatrix in Weltkoordinaten ermittelt. Diese wird anschließend um 180° um ihre y-Achse gedreht, um den UP-Vektor so auszurichten, dass das Bild nicht länger auf dem Kopf steht, und zuletzt um 90° in

z-Richtung, was der Montage der Kamera Rechnung trägt. Die so entstandene *View-Matrix* wird dann auf die Kamera des `robotCamView` angewendet.

Der Callback, welcher nach dem Rendering des Views ausgeführt wird (`CameraTakePictureCallback`) erzeugt ein Bild im konfigurierten Format (JPEG oder PNG) und speichert dieses binär in einem Objekt, welches via TCP/IP an die Zielanwendung übertragen werden kann. Auf diesen Punkt wird in Abschnitt 5.5.3 näher eingegangen.

5.5.2 Darstellungsqualität

Im Zusammenhang mit dem Rendering möchte ich an dieser Stelle auf die Möglichkeit der 3D-Datenvereinfachung zum Performancegewinn eingehen. Das Verfahren basiert auf der Tesselierung und kann entweder auf einzelne Objekte oder auch ganze Szenen angewandt werden. Es ist ebenfalls möglich, diese Vereinfachung in Abhängigkeit des Abstandes der Kamera zum Objekt vorzunehmen.

Die Tesselierung, hier im Spezialfall Triangulation, zerlegt eine Fläche in Dreiecke. Aus je weniger Dreiecken die Polygone eines 3D-Objekt bestehen, desto gröber ist dieses. Für die Tesselierung stehen spezielle Algorithmen zur Verfügung, u.a. der DelaunyWall (DeWall) Algorithmus. (vgl. [Wie04])

OpenGl, und damit OSG, kann nur einfache konvexe Polygone darstellen. Dies sind solche, deren Kanten sich nur an Ecken schneiden und die keine redundanten Ecken besitzen. Zudem bildet der Schnittpunkt zwischen zwei Kanten immer exakt eine Ecke. Wenn die Anwendung dennoch konkave Polygone, welche mit Löchern oder Polygone mit sich schneidenden Kanten benötigt, müssen diese in Dreiecke zerlegt werden. [Shr09] Abbildung 5.5.1 veranschaulicht dies. Teile der Grafik wurden von [Ahn13] übernommen und neu angeordnet.

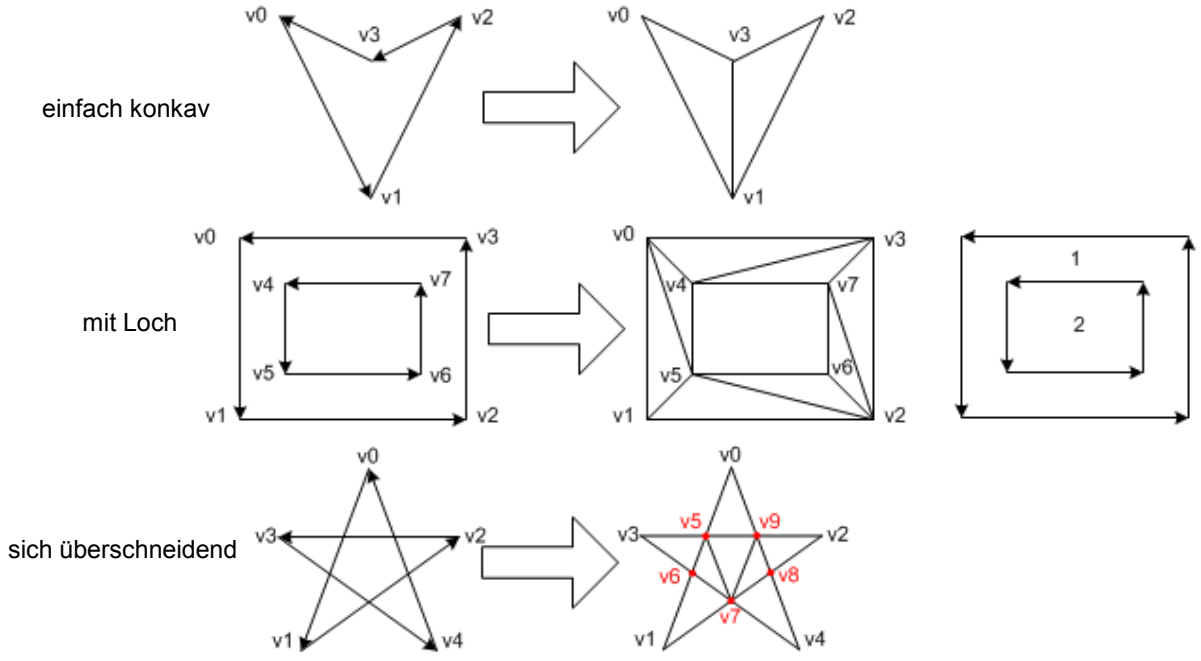


Abbildung 5.5.1: Tesselierung von Polygonen

Dabei spielt die Reihenfolge der Ecken-Nummerierung der Polygone eine Rolle. Je nachdem, ob die Flächen mit oder gegen den Uhrzeigersinn und von innen nach außen oder umgekehrt nummeriert

werden, variieren die Ergebnisse. Im Falle des Polygons mit Loch entscheidet die Nummerierung der Flächen darüber, was angezeigt wird. Diese sog. *Winding Rules* bestimmen die Darstellung und geben in diesem Beispiel vor, dass das Innere des Polygons unsichtbar ist.

Jedes dreidimensionale Objekt in der Computergrafik bildet ein Polygonnetz oder auch Mesh. Dieses wird über die Triangulation in Dreiecke zerlegt. Es gibt nun die Möglichkeit, die Anzahl der Dreiecke kontrolliert zu reduzieren. Je nach Einsatzgebiet stehen verschiedene Algorithmen zur Auswahl. In der Regel verwenden diese eine Kombination aus den vier Grundarten zur Reduzierung der Polygone. Dies sind:

1. *Sampling*: Es wird eine Auswahl aus einer Menge an (Oberflächen-) Punkten, welche die initiale Geometrie darstellen, gebildet und mit dieser Auswahl ein neues Mesh generiert. Dies funktioniert sehr gut bei organischen Objekten ohne scharfe Ecken und Kanten.
2. *Adaptive Subdivision*: Diese Algortihmen finden ein Basis-Poygonnetz, welches rekursiv unterteilt wird, um es an das Original anzunähern. So bleibt die Oberflächenstruktur eines 3D-Modells gut erhalten.
3. *Decimation*: Die Eckpunkte der Geometrie werden iterativ reduziert und nach jedem Schritt die Lücken durch erneute Triangulation gefüllt. Dieses Vorgehen ist sehr schnell und kann redundanter Geometrie sehr effektiv vereinfachen.
4. *Vertex-Merging*: Wie der Name sagt, werden Ecken zusammengefasst. Aus je zwei Ecken, entsteht eine Neue. So bleibt die Grundtopologie des Meshes erhalten.

Beim Einsatz der Methoden spielt die Beschaffenheit des Polygonnetzes eine große Rolle, so dass nicht jedes Verfahren für jedes Modell zufriedenstellende Ergebnisse liefern kann. Bei sehr guter Auswahl können die Anzahl Ecken deutlich reduziert werden, ohne dass die Form verloren geht. Eine solche kombinierte Vereinfachung ist in Abbildung 5.5.2 dargestellt. [Lue01] Dort ist deutlich erkennbar, dass die Reduzierung auf knapp 4% der ursprünglichen Polygonanzahl kaum Auswirkungen auf die Qualität des Modells hat. Erst die weitere Vereinfachung hinterläßt deutliche Spuren am Objekt.

Abbildung 5.5.2: Vereinfachung eines Meshes [Lue01]

Die Technik, ein Objekt in Abhängigkeit der Entfernung mehr oder weniger detailliert darzustellen, kann insbesondere bei großen Szenen sinnvoll sein, denn Dinge, die das Auge/ die Kamera nicht erkennen kann, müssen auch nicht unbedingt gerendert werden. Dafür wird mit OSG ein 3D-Objekt, `osg::Node`, erzeugt und von diesem sog. *tiefe* Kopien (deep copy) erstellt. Im Gegensatz zur *flachen* Kopie (shallow copy) wird dabei das Objekt inklusive all seiner Attribute im Speicher dupliziert und nicht nur die Referenzen. Dies ist notwendig, um wirklich voneinander getrennte Modelle zu erhalten. Diese werden einzeln mithilfe des `osgUtil::Simplifier` jeweils mit einer unterschiedliche Detaillierung versehen. Anschließend werden die Objekte als Kinder einem `osg::LOD`[21] Node zugeordnet, und zwar jeweils mit der Angabe zwischen welchen Entfernungen des Viewers zum Objekt welcher kopierte Node sichtbar

[21] **L**evel **O**f **D**etail

sein soll. Dabei dürfen sich die Parameter nicht überlappen, da sonst zwei Nodes zur gleichen Zeit gezeigt werden, was nach der Spezifikation nicht gewünscht ist. [XQ10]

OSG unterstützt sowohl das generelle Tesselieren von Polygonen als auch die Verwendung von verschiedenen Detailleveln in Abhängigkeit der Entfernung zwischen Betrachter und Objekt. Bei Bedarf kann dies zur Performance-Steigerung eingesetzt werden. Dies wurde zwar in der Vorbereitung getestet, ist jedoch in den in dieser Arbeit beschriebenen Szenarien und unter Einsatz der beschriebenen Hardware bisher nicht nötig.

5.5.3 Datenübertragung

Wie bereits zuvor erwähnt, gibt es verschiedene Möglichkeiten, die Bilddaten der Kamera an eine Ziel-Applikation zu übertragen. Dies kann in Bildern via Dateisystem oder Netzwerk, als Video-Stream, z.B. mit ffmpeg, oder auch über ein externes Tool wie VideoLAN VLC geschehen.

Die eingesetzte Technik ist neben der Vorliebe des Anwenders auch davon abhängig, wie viele Bilder pro Sekunde die Kamera schießen soll. Grundsätzlich gilt, dass keine Kamera simuliert werden kann, die eine höhere Bildrate hat, als die Simulation auf der eingesetzten Hardware leisten kann. Pro fertig gerenderter Szene ist nur ein Bild sinnvoll, da sich auf weiteren keine Veränderung zeigen würde. Auch muss die Verarbeitungsgeschwindigkeit des Zielsystems, welches die Bilddaten analysiert, mit in Betracht gezogen werden. Ist ein Algorithmus nur fähig, fünf Bilder pro Sekunde zu analysieren und die nächste Pose zu bestimmen, sind höhere Bildraten nicht sinnvoll. Für die später aufgeführten Beispiele (siehe 6) wird davon ausgegangen dass diese Bedingungen erfüllt sind. Anderenfalls müssen die Anwendungen über die Konfiguration synchronisiert werden. Dies kann etwa über das Limitieren der Bildwiederholrate geschehen.

Für das Senden der Bilder via TCP/IP bzw. das Schreiben auf ein eingebundenes Speichermedium wird, wie in 5.5.1 bereits erwähnt, ein Callback auf der Kamera des Tools (`robotCamView`) eingesetzt. Dieser `CameraTakePictureCallback` kann auf jedem beliebigen View ausgeführt werden und erstellt, vereinfacht gesagt, einen Sceenshot. Dazu fragt er die x- und y-Koordinaten der oberen, linken Ecke des Views sowie die Länge und Breite ab und erzeugt daraus mit der Methode `readPixels` ein `osg::Image` Objekt. Anschließend wird dieses u.a. für das Netzwerk-Modul global bereitgestellt. Hinzu kommt die Möglichkeit, über ein Flag sowie die Angabe der Qualität und des Speicherortes das Bild auf einem eingebundenen Datenträger zu speichern. Der Code dazu kann in Anhang A.4 eingesehen werden.

Neben dieser internen Möglichkeit der Bilddatengewinnung stehen eine Vielzahl von nicht kommerziellen Werkzeugen bereit, welche diese Aufgabe übernehmen können. Eine der einfachsten Lösungen ist das Streaming via Desktop Capturing. Dabei wird unter Zuhilfenahme eines Streaming Servers ein Ausschnitt oder der komplette sichtbare Bildschirminhalt n-mal pro Sekunde abfotografiert, in ein gängiges Format wie H.264 konvertiert und via Netzwerk (UDP) bereitgestellt. Ein Client wie MATLAB oder jede Wiedergabesoftware für Streams kann den Inhalt empfangen und weiterverarbeiten. Zwei frei verfügbare Applikationen dafür sind VLC[22] oder ffserver[23].

Es ist ebenso möglich, die OpenSource Bibliotheken von ffmpeg in OSG zu verwenden und die Anwendung direkt den Stream generieren zu lassen. Dies ist im Rahmen dieser Arbeit jedoch nicht umgesetzt worden, wird aber für die Zukunft in Betracht gezogen.

[22] `http://wiki.videolan.org/Documentation:Modules/screen`
[23] `http://ffmpeg.org/trac/ffmpeg/wiki/Streaming%20media%20with%20ffserver`

5.6 User Interface

Der Benutzer kann über Maus- und Tastatureingaben mit der Simulation interagieren und so Parameter oder auch 3D-Objekte manipulieren (4.4.2). Andere Eingabegeräte werden zum jetzigen Zeitpunkt nicht unterstützt. Ziel ist es, den Anwender die Ansicht auf die Szene frei wählen zu lassen und in einem festen Rahmen Objekte zu manipulieren, um die Robustheit und Reaktionsgeschwindigkeit von Algorithmen besser testen zu können, als dies einem reinen Zuschauer möglich wäre.

Um die Anwendung nicht noch komplexer zu gestalten, wurde darauf verzichtet, ein weiteres Framework wie, MFC, GTK oder Ultimate++ zur GUI Gestaltung zu verwenden und stattdessen auf die von OSG mitgelieferten Mittel zurückgegriffen. So können Elemente recht einfach mit EventHandlern versehen werden, die etwa beim Mausklick eine vorher festgelegte Aktion auslösen. Ein solcher EventHandler ist der mitgelieferte StatsHandler, ein Handler, der statistische Daten zum aktuellen Rendering-Prozess anzeigt und über die s-Taste an- und ausgeschaltet wird. Er kann einfach mit einem View verknüpft werden und zeigt neben der aktuellen Bildwiederholrate auch Angaben zur CPU und GPU Auslastung sowie Details zu den Geometriedaten wie Anzahl Eckpunkte (Vertices) und Polygone.

Zusätzlich wird der für die Benutzer Interaktion essenzielle Bewegungs-Handler (im Weiteren als ObjectController bezeichnet) implementiert. Dieser stellt Methoden bereit, um durch die vorhandenen Objekte zu *laufen* und das jeweils ausgewählte anschließend zu bewegen. Ein ähnlicher Handler ist für den Kamera Screenshot vorgesehen.

Der ObjectController besteht aus zwei Teilen, der Logik für das Erkennen des ausgewählten Objektes und der Umsetzung der Tastatureingaben in Bewegung. Beides ist über eine abgeleitete Klasse des `GUIEventAdapters` und anschließendem Überschreiben der `handle` Methode von OSG realisiert worden. Dieser Handler wird auf dem FreeView etabliert und von da an bei jedem GUI Event dieses Views aufgerufen. Der ObjectController reagiert auf gedrückte Tasten, prüft, welche Taste gedrückt wurde und führt anschließend eine Aktion aus (Abb. 5.6.1).

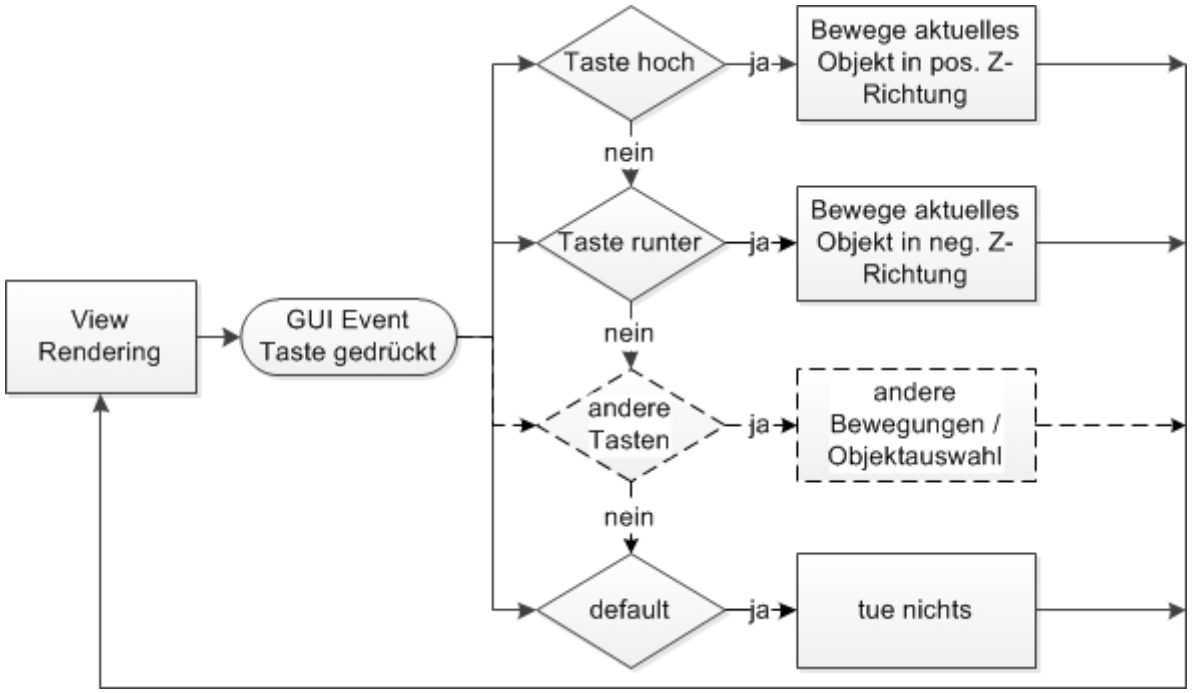


Abbildung 5.6.1: Objekt Controller Datenfluss

Die Steuerung wurde ausschließlich über Tastaturbefehle realisiert, da sie so deutlich einfacher umzusetzen war, als die Interpretation der Mouse-Daten. Für die nächste Version ist dies jedoch in Planung.

5.7 Kommunikation

Die Interaktion zwischen der Simulation und der Steuer-/ Bildanalyse Anwendung findet via Netzwerk statt (4.4.1). Als Protokoll wird Standard TCP/IP verwendet. Dieses arbeitet verbindungsorientiert und sorgt so dafür, dass verlorene Pakete erkannt und nochmals angefordert werden. Die Implementierung wird mit Boost ASIO (siehe 5.2.2) durchgeführt.

Bei jeder Kommunikation muss eine Sprache definiert werden, in diesem Fall das Kommunikationsprotokoll, welches die Anwendungsschicht auf TCP/IP bildet. Es entscheidet, wie die Daten aussehen müssen, die versendet und empfangen und verwendet werden können. Die einfachste Art zu kommunizieren, ist über Textnachrichten, also das Versenden von Zeichenketten. Die maximale Länge dieser Nachrichten wird, ebenso wie der Port, über den kommuniziert wird, über die .properties Datei festgelegt.

Die Simulationsumgebung stellt aus verschiedenen Gründen den Server dieser Client-Server Architektur dar. Zum einen liegt damit die Verwaltung der Verbindungen lokal, wodurch die Kommunikation unabhängig von anderer Software gesteuert werden kann. Ein oder auch mehrere unterschiedliche Clients können sich verbinden und müssen außer dem Protokoll keine Kenntnis über die Nachrichtenverarbeitung haben. So wird sichergestellt, dass Anwendungen, die z.B. mit ROS implementiert wurden oder selbst ein einfacher TELNET Client sich verbinden können. Ein weiterer Grund für die Simulationsumgebung als Server ist, dass andere Applikationen nicht unbedingt mehrere Verbindungen parallel verwalten können und eventuell nicht offene Protokolle zum Einsatz kommen, die nicht so flexibel sind, wie das hier implementierte. Die Instrument Control Toolbox von MATLAB beispielsweise ist starken Einschränkungen unterworfen und es ist erheblicher Aufwand erforderlich, um ein flexibles, auf der Übertragung von Zeichenketten basierendes, Protokoll umzusetzen.

Das Format der gesendeten und empfangenen Nachrichten (Messages) wurde möglichst einfach gehalten. Es handelt sich um eine Zeichenkette, die mehrere, durch Semikolon (oder ein anderes zuvor festgelegtes Zeichen) getrennte Befehle mit einer per Konfiguration festgelegten Länge enthalten kann. Jede Nachricht wird zerlegt, validiert und bei Gültigkeit verarbeitet. Es wurden zunächst zwei Typen von Messages implementiert, CONTROL und MOVEMENT Messages. Wie die Namen vermuten lassen, dient die CONTROL Nachricht dazu, das Verhalten oder Eigenschaften der Simulation, wie z.B. Licht und Schatten zu steuern, während die MOVEMENT Message alle beweglichen Transforms steuern kann. Eine mögliche valide Nachricht bildet sich wie folgt:

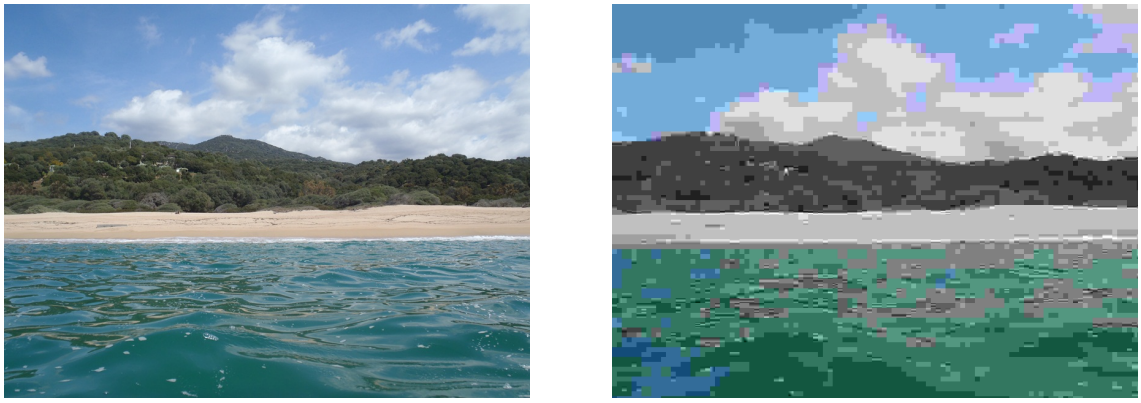
```
Robotername_Gelenk_DOF:Wert\n
Katana_joint2_ROTZ:25.0\n
```

Die Dekodierung wird vom Kommunikationsserver (`CommServer`) übernommen. Dieser prüft, ob die Nachricht gültig ist. In diesem Fall ist dies die Prüfung auf den Doppelpunkt als Trennung zwischen Key und Value und ob der gesendete String mit einem Zeilenumbruch (\n) endet. Zudem wird verifiziert, ob der Joint überhaupt beweglich ist bzw. den angegebenen DOF hat, d.h. in der globalen Variable `g_movement_instructions` enthalten ist. Ist dies der Fall, wird die Variable mit den neuen Gelenkkoordinaten aktualisiert.

Das Spektrum an Nachrichten kann beliebig erweitert werden. Es sind auch binäre Messages oder Protokolle über CAN oder andere Technologien denkbar.

Nachdem nun die Kommunikation von einer Steuerungssoftware zur Simulation beschrieben wurde, muss noch kurz auf die Gegenrichtung eingegangen werden, also das Senden von der Simulation zur Steuerungseinheit. In dieser Evolutionsstufe kann die Simulation entweder Textnachrichten oder binäre Bilddaten übertragen. Implementiert ist zum einen die Antwort auf den Steuerungsbefehl `GETPIC`, der ein Bild der aktuell gerenderten Szene anfordert sowie textuelle Informationen über die Bildgröße in Byte `GETPICSIZE`. Natürlich sind auch andere Nachrichten möglich.

Je nach Anwendung, ist es erforderlich, wenn ein Bild angefordert wird, zunächst die Bildgröße zu kennen, um entsprechend Speicher im Input-Buffer reservieren zu können. Die Anzahl der Bytes ist nämlich neben der Kompression auch von den Dimensionen des Bildes abhängig und kann, je nachdem wie der Anwender die Fenstergröße des Kamerabildes in seinem Windowsystem wählt, stark variieren. Da im binären Format leere Felder im Buffer als Inhalt gewertet werden, würde ein zu großer Buffer, eine unlesbare Bilddatei zur Folge haben, während ein zu kleiner Buffer überlaufen würde. Auch für MATLAB gilt daher, dass bevor das Bild empfangen werden kann, erst die Größe angefragt werden muss.



(a) JPEG Bild bei hoher Qualität (100) $\approx 189kB$

(b) JPEG Bild bei niedriger Qualität (2) $\approx 20kB$

Abbildung 5.7.1: JPEG Bildqualität - Vergleich

Die Kompression und das Dateiformat können in einem gewählten Rahmen via Konfigurationsdatei (`PICTURE_FILETYPE` und `PICTURE_QUALITY`) gesetzt werden. Es sind JPEG und PNG implementiert. Bei erstgenanntem Format entspricht eine Qualität von 100 der niedrigsten Kompressionsrate ($2.6 : 1$), während eine von 1 die Bilddaten auf $\frac{1}{144}$ reduziert, was eine entsprechend schlechte Bildqualität bedeutet (siehe [Wik13a]). Abbildung 5.7.1 macht dies sichtbar. Für die Zukunft ist geplant, ebenfalls die Farbtiefe modifizierbar zu machen. So könnte etwa zwischen RGB und 8-Bit Monochrom gewählt werden, was im zweiten Fall die Bildgröße deutlich reduziert.

Hinweis: Im Falle eines Bildes im RAW-Format mit festgelegter Skalierung ist die Bildgröße konstant. In diesem Fall muss die Bildgröße nur einmal initial und nicht für jedes angeforderte Bild neu angefragt werden.

5.8 Konfiguration

Um einen hohen Grad an Flexibilität zu gewährleisten, ist ein großer Teil der Anwendung über eine Textdatei konfigurierbar. Der Name dieser sog. Property Datei wird beim Start der Anwendung als Parameter übergeben. Darin lässt sich sowohl die Umgebung mit ihren Objekten beeinflussen, als auch der Manipulator selbst. Features wie die Netzwerkkommunikation werden ebenfalls hier definiert. Das Protokoll zur Steuerung des Manipulators sowie die möglichen Befehle zur Online-Modifikation der Simulationsumgebung sind zusätzlich dort angegeben.

Das Format ist an das unter JAVA hauptsächlich verwendete .properties Format[24] angelehnt, in dem Key-Value-Paare aufgelistet werden. Hierfür wurde eine eigene rudimentäre Klasse implementiert. Unter Windows ist dies zwar nicht nötig, da mit `getprivateprofilestring` eine eigene Methode zur Verfügung steht, dies steht jedoch im Widerspruch zur leichten Portierbarkeit.

Zu den Parametern zählen einerseits generelle Einstellungen wie der verwendete Netzwerk Port und die Länge des Datagramms sowie die vollständige Definition der 3D-Umgebung, insbesondere der Roboter mit ihren einzelnen Freiheitsgraden. Die nachfolgende Tabelle (5.8.1) gibt einen Überblick über die implementierten Befehle.

Schlüssel	erlaubte Werte	Anwendung
FILEPATH	*string*	Pfad im Dateisystem, wo sich die 3D-Daten befinden
ALL_ANGLES_IN	DEG; RAD	setzt die globale Art der Winkelangabe, Grad oder Gradient
SHADOW_ENABLE	1,0	Schattengenerierung EIN und AUS
NETWORK_SERVER_PORT	*uint*	Listen-Port des Kommunikationsservers
ROBOT_VIEW_CAM_1_POSITION	OSX;POSY;POSZ; ROTX;ROTY;ROTZ	Position der Roboterkamera relativ zum Endeffektor
ROBOT_?_NUMBER_OF_JOINTS	*uint*	Anzahl der Gelenke für den Roboter mit der Nummer ?
ROBOT_?_JOINT_?_DOF	POSX;POSY;POSZ; ROTX;ROTY;ROTZ	gibt die DOF des jeweiligen Gelenks an

Tabelle 5.8.1: Konfigurationsparameter (Ausschnitt)

Neben diesen Parametern gibt es viele weitere. Die vollständige Liste befindet sich auf der beigelegten CD. An dieser Stelle soll betont werden, dass so viel wie möglich in die Datei ausgelagert wurde. Selbst die Objekte und auch die Kameras können auf diese Weise spezifiziert werden. Für jede mögliche Konfiguration ist in der mitgelieferten Datei ein auskommentiertes Beispiel vorhanden, so dass jeder neue Nutzer sich sofort mit der Anwendung vertraut machen kann, ohne den Code zu studieren. Im folgenden Unterpunkt wird die vollständige Definition eines Roboters gezeigt.

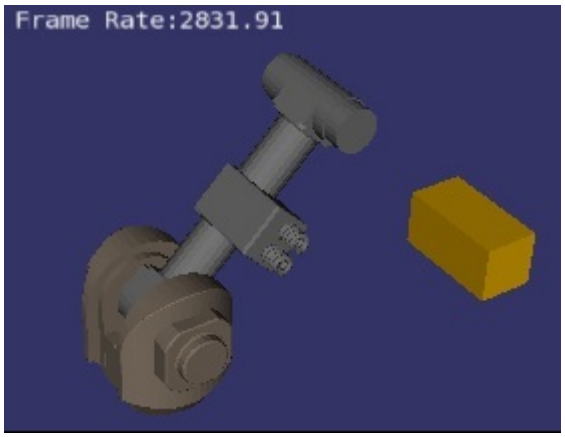
[24] `http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html`

5.8.1 Definition eines Roboters

Ein einfacher Roboter in einer minimalen Umgebung kann in wenigen Schritten definiert werden (Abb. 5.8.1 f.). Für einen funktionsfähigen Roboter sind mindestens zwei durch ein Gelenk verbundene Armteile nötig. Ein Objekt zur Interaktion, die OpenGL Default-Lichtquelle und eine Kameraansicht komplettieren die Simulation. Mit aktiviertem Netzwerkserver kann dieser remote bewegt werden.

```
FILEPATH = .\data\
LOGLEVEL = NOTICE
ALL_ANGLES_IN = DEG
SHADOW = 0
LIGHT_1_ENABLE = 0
NETWORK_ENABLE = 1
NETWORK_SERVER_PORT = 6099
ROBOT_2_NAME = Dummy
ROBOT_2_ENABLE = 1
ROBOT_2_NUMBER_OF_JOINTS = 2
ROBOT_2_JOINT_1_START_POSE=.0;.0;.0;.0;.0;.0
ROBOT_2_JOINT_1_NAME = Foot
ROBOT_2_JOINT_1_FILENAME = Katana_Joint1.stl
ROBOT_2_JOINT_1_COLOR = #908070
ROBOT_2_JOINT_2_START_POSE=.0;.0;68.5;...
   ... .0;45.0;.0
ROBOT_2_JOINT_2_DOF = ROTY
ROBOT_2_JOINT_2_NAME = Joint
ROBOT_2_JOINT_2_FILENAME = Katana_Joint2.stl
ROBOT_2_JOINT_2_COLOR = #808080
OBJECT_1_TYPE = REGULAR_BOX
OBJECT_1_ENABLE = 1
OBJECT_1_DIMENSIONS = 50.0;100.0;50.0
OBJECT_1_COLOR = #FFBF00
OBJECT_1_START_POSE = 300.0;.0;25.0;.0;.0;.0
FREE_VIEW_CAM_ENABLE = 1
```

Abbildung 5.8.1: Beispiel Properties




```
[WARN]: DOF of Dummy_Foot not valid. ...
[WARN]: Property ROBOT_2_JOINT_1_AXIS missing
[WARN]: Invalid value for ROBOT_2_JOINT_1_AXIS.
[NOTICE]: Adding movable joint Dummy_Joint_ROTY
[NOTICE]: Creating regular Box
[NOTICE]: Creating Floor...DISABLED.
[NOTICE]: Creating Lights...DISABLED.
[NOTICE]: Creating Shadow...DISABLED.
[NOTICE]: Creating Cameras / Views...
[NOTICE]: Building Robots...Dummy.
...
```

Abbildung 5.8.2: Beispiel 3D & Log-Auszug

Wenn kein Licht definiert/enabled wird, ist bei aktueller Implementierung das `GL_LIGHT0` aktiv, was bedeutet, dass alle gerenderten Objekte *selbst leuchten*. Neben den offensichtlichen Key-Value Paaren gibt es zahlreiche, die sich gegenseitig beeinflussen können. Diese sind in der Schnellstartanleitung beschrieben. Es verändert sich zum Beispiel die Anzahl der Werte für die Dimension eines Objektes. Während eine Kiste über Länge, Breite und Tiefe definiert wird, ist der Parameter für eine Kugel eindimensional, also der Radius r. Weitere Beispiele liegen dieser Dokumentation anbei.

Es wurde viel Wert auf aussagekräftige Hinweise, Warnungen und Fehlermeldungen gelegt, so dass beim Vergessen oder Falschdeklaration eines Parameters ein daraus resultierendes Programmende bzw. ein fehlendes Element schnell analysiert werden kann. An dieser Stelle sei auf das konfigurierbare Loglevel hingewiesen. Es sind vier mögliche Fehlergewichtungen implementiert:

1. *INFO*: Gibt Informationen zu jedem Methodenaufruf in OSG an.
2. *NOTICE*: Gibt nur wichtige Informationen aus. Alle Nachrichten der Simulationsumgebung selbst haben mindestens diesen Loglevel.
3. *WARN*: Gibt Warnungen aus, wie fehlende/ falsche Konfigurationsparameter oder falsch eingesetzte OSG Befehle.

4. *FATAL*: Gibt Fehler aus, bevor das Programm aufgrund dieser beendet wird. Bsp.: Keine Property-Datei vorhanden.

Das DEBUG Level wurde bewusst weggelassen, da es nur benötigt wird, wenn die Quellen der OSG Bibliotheken verändert werden. Wenn ein `LOGLEVEL` gesetzt ist, werden immer die Nachrichten dieses Levels zuzüglich aller schwerwiegender Fehler ausgegeben.

5.9 Zusammenspiel der Komponenten

Eine Konfiguration, wie sie im vorhergehenden Abschnitt beschrieben ist, (5.8.1) wird in einem festen Ablauf abgearbeitet (Abb. 5.9.1).

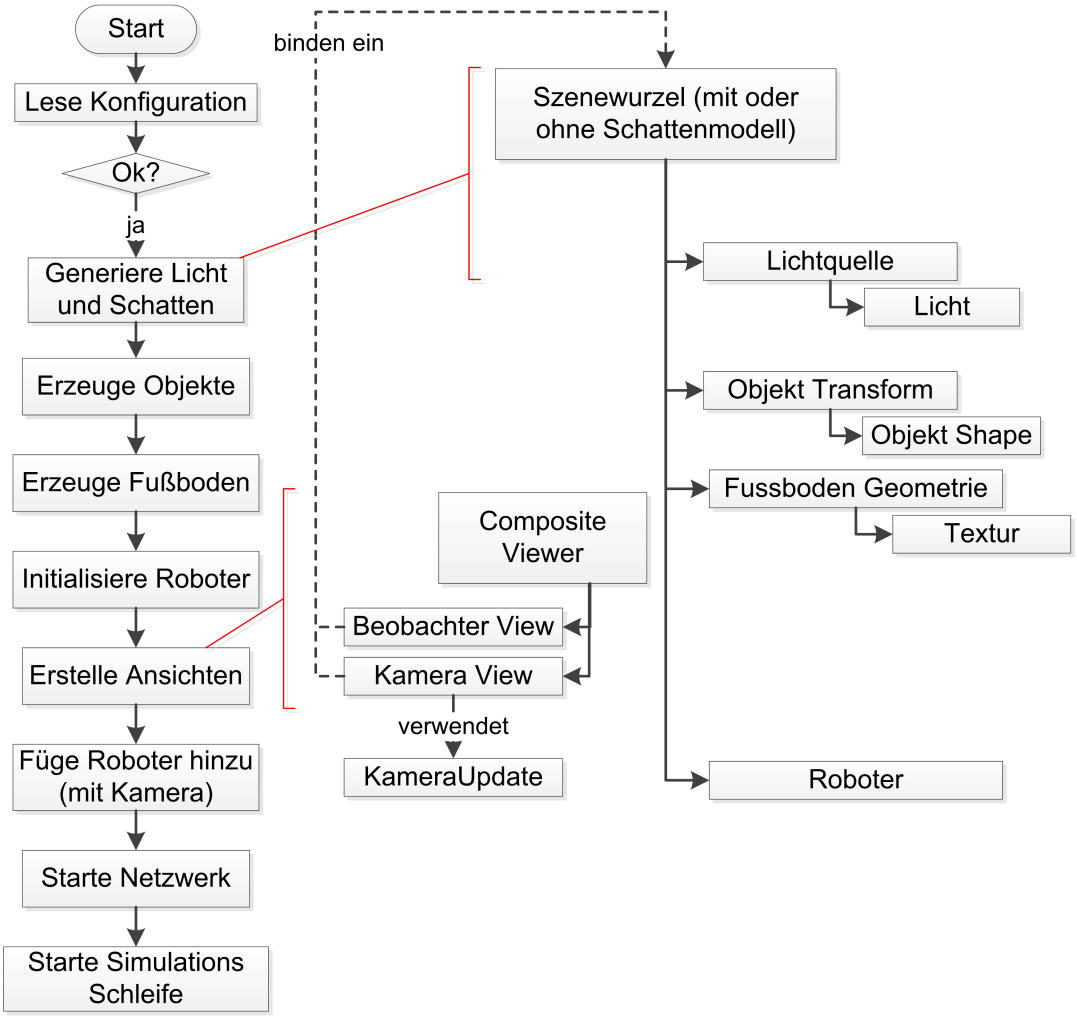


Abbildung 5.9.1: Programmablauf

Beim Schließen der Applikation wird zuerst der Netzwerkserver heruntergefahren und anschließend die Simulation beendet. So entstehen keine *Zombie*-Prozesse[25] und dadurch auch keine Fehler bei einem Neustart auf demselben Port.

[25] ein Prozess, der beendet ist, aber trotzdem noch in der Prozesstabelle auftaucht und Systemressourcen belegt

6 Anwendungsfälle

Nach den vorangegangenen Kapiteln, welche die Analyse, das Design sowie die Implementierung beschreiben, wird hier nun an konkreten Beispielen die Funktionsweise der Applikation dargestellt. Abschnitt 6.1 beschreibt die einfache Steuerung der Simulation via MATLAB, mit und ohne Kamerabildübertragung und -darstellung. Anschließend wird in 6.2 über eine Analyse der gewonnenen Bilddaten eine Rückkopplung geschaffen, so dass ein Aktor-Sensor Zusammenspiel entsteht. Nach der Einführung eines weiteren Sensors (6.3) werden abschließend weitere mögliche Szenarien vorgestellt (6.4).

6.1 Roboter Steuerung

Im ersten Schritt soll MATLAB zunächst mithilfe der Instrument Control Toolbox Befehle an die Simulationsumgebung schicken und so den virtuellen Katana-Roboter steuern. Die von der onboard-Kamera des Roboters geschossenen Bilder sollen von MATLAB via Netzwerkverbindung empfangen und angezeigt werden.

Zur Verifikation der Bewegungen wird die Robotics Toolbox (RTB) herangezogen, die bereits zuvor in diversen Projekten, u.a. an der HAW Hamburg, getestet und für gut befunden wurde. So kann sichergestellt werden, dass die Bewegungen der Simulation der RTB mit denen der entwickelten Anwendung übereinstimmen. Da die RTB in zahlreichen Experimenten bewiesen hat, dass ihre Darstellungen sich den Gelenkstellungen des realen Manipulators decken, wird diese indirekte Validierung der Simulation verwendet und nicht direkt am realen Roboter getestet.

Das Setup mit der RTB besteht allein aus der Definition der DH-Parameter sowie dem Erstellen einer einfachen 3D-Ansicht des Roboters aus diesen. Der Programmablauf besteht weitgehend aus einer Schleife, welche von der Start- bis zur Zielpose folgende Schritte iterativ ausführt:

1. Setze neue Parameter für die Winkel und aktualisiere die lokale 3D-Ansicht.
2. Sende Winkel an die Simulationsumgebung.
3. Empfange Bilddaten und zeige diese lokal in einem Fenster an.
4. Gehe zu 1.

6.1.1 Winkeldefinitionen

Die Winkeldefinitionen in der Simulationsumgebung (HAW-Sim) unterscheiden sich, ähnlich wie die der Katana4D Software, von jenen Winkeln, die nach der DH-Konvention (Kin) berechnet werden. Das liegt daran, dass die Koordinatensysteme aller Gelenke wie das des Basiskoordinatensystems ausgerichtet sind, und zwar in vertikaler Maximalauslenkung, aufrecht stehend mit dem Endeffektor Richtung Himmel zeigend. Daraus ergibt sich die Umrechnungstabelle 6.1.1.

Winkel [Typ]	HAW-Sim → Kin	Kin → HAW-Sim
θ_1	$\theta_1^{Kin} = \theta_1^{HAW} - \pi$	$\theta_1^{HAW} = \theta_1^{Kin} + \pi$
θ_2	$\theta_2^{Kin} = \theta_2^{HAW} + \frac{\pi}{2}$	$\theta_2^{HAW} = \theta_2^{Kin} - \frac{\pi}{2}$
θ_3	$\theta_3^{Kin} = \theta_3^{HAW}$	$\theta_3^{HAW} = \theta_3^{Kin}$
θ_4	$\theta_4^{Kin} = \theta_4^{HAW} + \frac{3}{2}\pi$	$\theta_4^{HAW} = \theta_4^{Kin} - \frac{3}{2}\pi$
θ_5	$\theta_5^{Kin} = \theta_5^{HAW} - \pi$	$\theta_5^{HAW} = \theta_5^{Kin} + \pi$

Tabelle 6.1.1: Umrechnungstabelle für Winkel

6.1.2 Der Katana Roboter in der Simulation

Bevor mit der eigentlichen Simulation begonnen werden kann, muss sichergestellt sein, dass die Kommunikation zwischen Client und Server einwandfrei funktioniert. Um Fehler seitens der Clientanwendung auszuschließen, wurden zwei einfache Methoden verwendet.

Nach dem Start der Umgebung samt Server wurde via telnet[26] eine Verbindung hergestellt und ebenso erfolgreich wieder terminiert. Im weiteren wurde mit einem 10 Zeilen PERL[27] Skript die Ansteuerung der einzelnen Gelenke über das implementierte TCP/IP Protokoll getestet. Näheres zu PERL befindet sich u.a. in [Chr03] und das Skript befindet sich im Anhang A.5.

Nach erfolgreichem Test wurde eine iterative Gelenksteuerung sowie Anzeige der Bilddaten mit MATLAB implementiert. Die DH-Parameter (Tab. 6.1.2) dafür ergeben sich aus den Längen der einzelnen Arm-Abschnitte dargestellt in Abb. 6.1.1 sowie der Anwendung der DH-Konventionen 3.2.3.

In dieser Variante haben sowohl der Motor des Flansches als auch der Motor des Tools die gleiche Ausrichtung. Bei Einbeziehung beider Gelenke in die DH-Konventionen, würde dies eine Redundanz erzeugen, was zu unendlich vielen Lösungen in der inversen Kinematik führen würde. Um dies zu vermeiden, wird nur ein Motor mit in die DH-Parameter einbezogen. Das fünfte Gelenk wird folglich aus den Längen vom Flansch und Endeffektor gebildet, wobei der Wert auch davon abhängig ist, wo man den TCP annimmt(*).

Gelenk	α_i	a_i	θ_i	d_i
1	$\frac{\pi}{2}$	0	0	201.5
2	0	190	0	0
3	0	139	0	0
4	$-\frac{\pi}{2}$	0	0	0
5	0	0	0	188.3*

Tabelle 6.1.2: Denavit-Hartenberg Parameter des Katana 6M180

Das eingesetzte Objektiv für die Guppy Kamera ist das C60402KP - H416 (KP) des Herstellers [PEN13]. Dabei handelt es sich um ein Weitwinkelobjektiv mit einer Brennweite von $4.2mm$ und einem horizontalen Blickwinkel von 86.8° bei einem minimalen Objektabstand von $200mm$. Das Format der Kamera ist 4:3. Daraus ergibt sich der View des Kamerabildes sowie die Parameter für die Projektion (6.1.1).

$$\text{setProjectionMatrixAsPerspective}(70.0°, 4.0/3.0, 200\ mm, \infty\ mm) \tag{6.1.1}$$

[26] `https://tools.ietf.org/html/rfc854`
[27] `http://www.perl.org/`

Um den vertikalen Blickwinkel ($verticalFOV$) für die Methode zu bekommen, muss die Gleichung 3.3.4 entsprechend umgestellt werden (6.1.2). Da ∞ kein gültiger Wert für die Simulation ist, wird $1m$ angenommen.

$$fovY = 2 \cdot \arctan \frac{\tan(fovX/2) \cdot y}{x} \tag{6.1.2}$$

Mit dem Startvektor

$$qStart_{Kin} = \begin{pmatrix} 0 & \pi/4 & -\pi/4 & \pi & 0 \end{pmatrix} \tag{6.1.3}$$

für die fünf Gelenke ist das Werkzeug des Katana orthogonal zur xy-Ebene ausgerichtet. Damit *sieht* die montierte Kamera ebenfalls in negative z-Richtung. Unter Berücksichtigung der Umrechnungstabelle 6.1.1 ergibt sich folglich für die Simulation die identische Pose des Roboters. Der Zusammenbau (Abb. 6.1.1) wird mit einem 48° geöffneten Winkelgreifer dargestellt, hinzu kommt das Kameragehäuse der Guppy, welches wie beim realen Roboter auf den Camera Mount gesetzt wird.

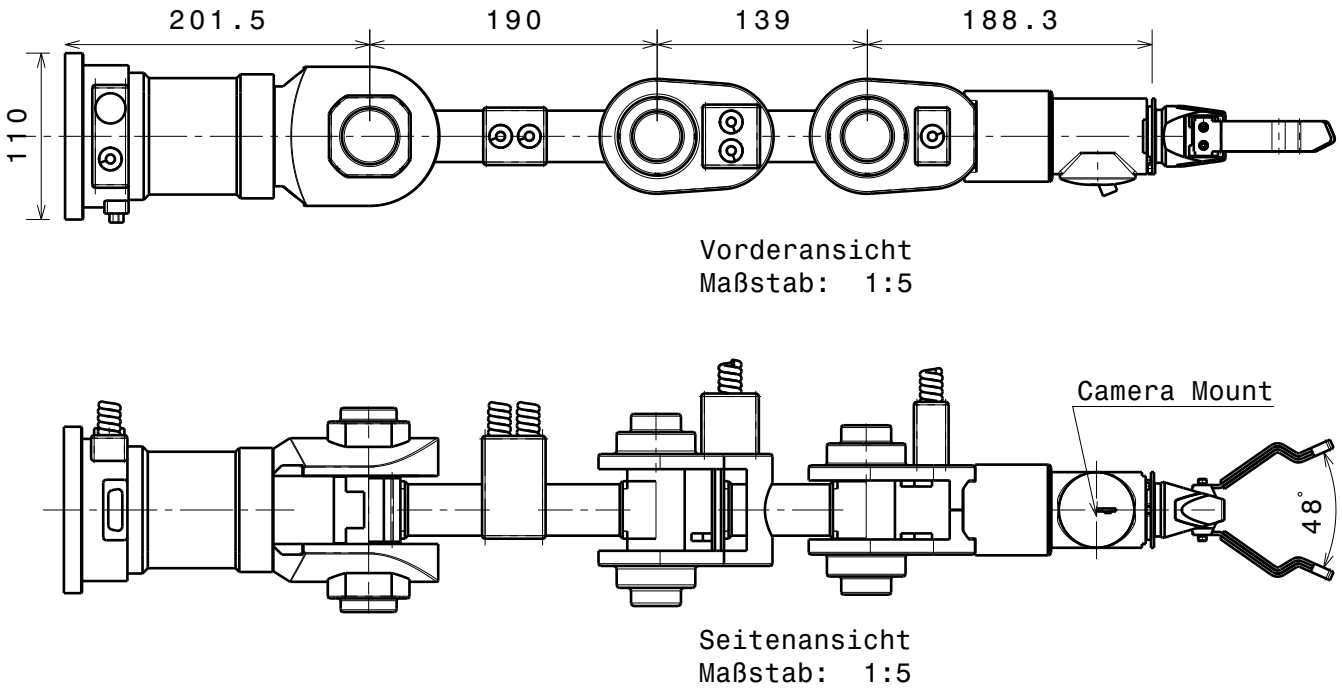
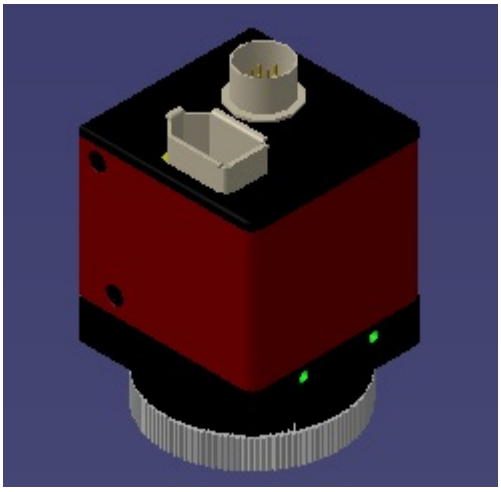


Abbildung 6.1.1: Katana Zeichnung

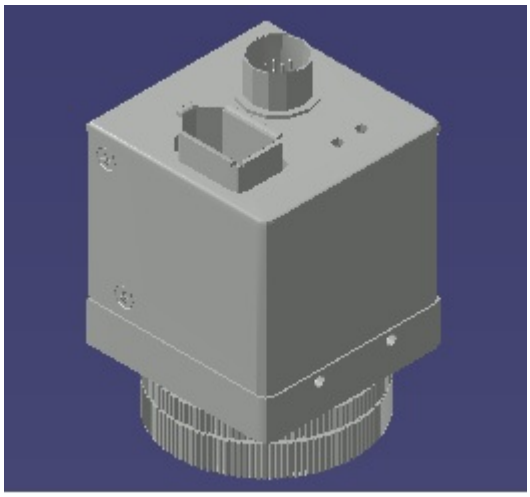
Für eine realitätsnahe Darstellung wurde die Kamera, wie der Roboter zuvor (5.4), aus der STEP Datei, welche der Hersteller frei zur Verfügung stellt, in das STL Format konvertiert und eingebunden (Abb. 6.1.2b). Hier ist gut zu erkennen, dass aus den STEP Produktdaten ausschließlich Geometrie exportiert wird; die Farben (Abb. 6.1.2a) verschwinden. Die Kameraposition wird stets relativ zum Endeffektor angegeben. Es ist also darauf zu achten, wo sich das Koordinatensystem des STL Modells befindet. Im hier beschriebenen Fall ergibt sich folgende Einstellung:

```
ROBOT_VIEW_CAM_1_START_POSE=-40.0;0.0;-20.0;0.0;180.0;90.0
```

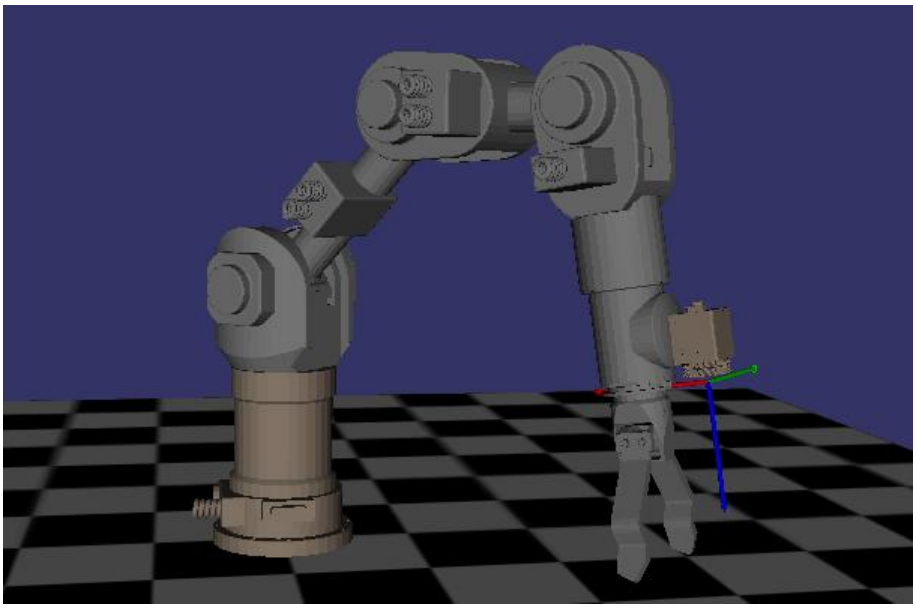
Fertig montiert resultiert daraus Abbildung 6.1.2c. In dieser Ansicht wird ebenfalls das Kamera-Koordinatensystem angezeigt, welches für die Bilddatengewinnung herangezogen wird. Dabei wird dir x-Achse rot, die y-Achse grün und die z-Achse blau dargestellt.



(a) Guppy im STEP Format



(b) Guppy im STL Format



(c) Katana mit Guppy und eingeblendetem Kamera Koordinatensystem

Abbildung 6.1.2: Guppy Kamera in der Simulation

Es ist bewusst von dem des 3D-Kameramodells, welches abweichend liegen kann, entkoppelt, so dass der Kamera-View von der Modellgeometrie unabhängig ist. Ansonsten müsste je nach 3D-Modell der View angepasst werden.

Für die Kommunikation zwischen MATLAB und der Robotersimulation wird mithilfe der Instrument Control Toolbox eine Verbindung hergestellt.

```
tcpc=tcpip('localhost', 6099, 'NetworkRole', 'client',
     'InputBufferSize',100000);
fopen(tcpc);
```

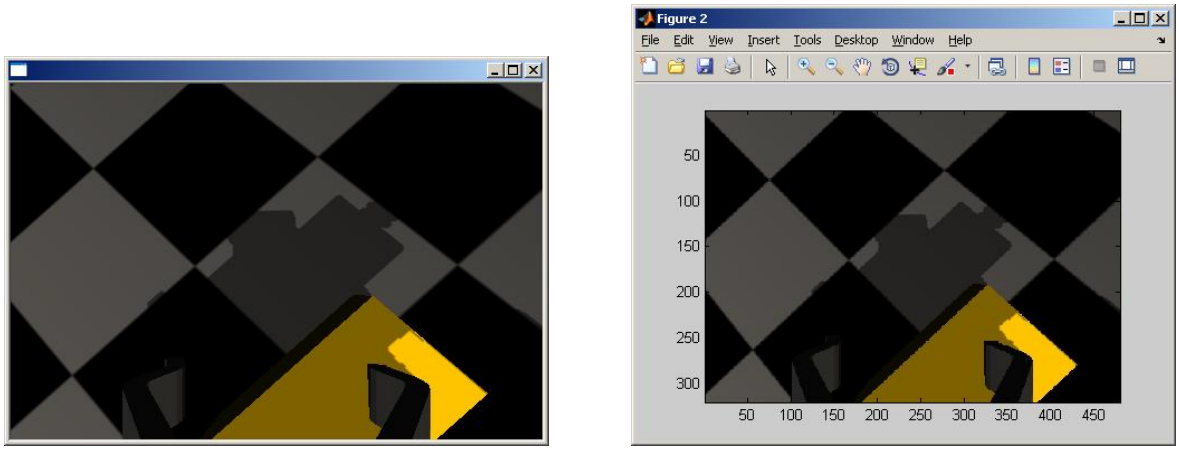
Als Parameter benötigt die Funktion neben dem Hostnamen und Port, auf der der Kommunikationsserver läuft, die Größe des Buffers für die empfangenen Bilddaten. Dabei handelt es sich um die Angabe des

Maximums. Die Bilddaten können sehr wohl kleiner sein. Dies ist abhängig von den Dimensionen des Bildes und wurde sowohl berechnet (Gleichung 6.1.4) als auch experimentell nachgewiesen.

$$\text{Länge} \times \text{Breite} \times 3 = \text{Rohdaten Bildgröße in Bytes} \tag{6.1.4}$$

Für ein 320×240 Pixel RGB JPEG sind die angegebenen 100.000 Bytes ausreichend. Nichtsdestotrotz variiert die Datenmenge in Abhängigkeit des Bildinhaltes. Bei einer durchschnittlichen Kompression der Rohdaten lagen die Messwerte zwischen 25 und 50kB pro Bild. Man kann sich so schnell ausrechnen, dass ohne Kompression bei *großen* Bildern in einem Netzwerk mit einer Bandbreite von 100Mbit/s hohe Bildraten schnell unmöglich werden. Detaillierte Informationen zum Thema JPEG und Kompression sind unter [Cla13] verfügbar.

Da MATLAB die Binärdaten nicht direkt in ein Bild (`figure`) umwandeln kann, muss ein Umweg über das Speichern auf der Festplatte genommen werden. Die per Stream empfangenen Daten werden ohne weitere Bearbeitung gespeichert und sofort wieder mit Methoden der Image Processing Toolbox eingelesen und so im Falle eines RGB Bildes in eine $m \times n \times 3$ Matrix gewandelt, die abschließend in einer `figure`, wie in Abb. 6.1.3b angezeigt werden.



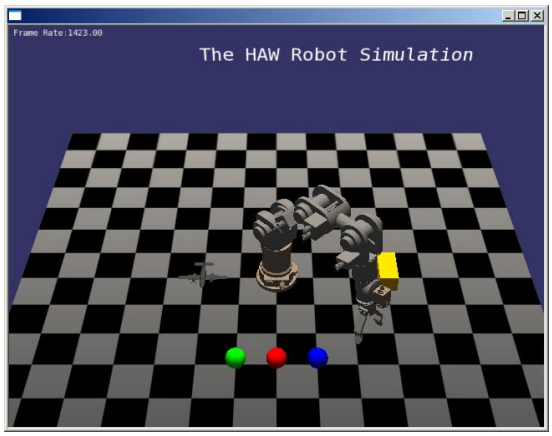
(a) Simulation Kamera View

(b) MATLAB Kamera View

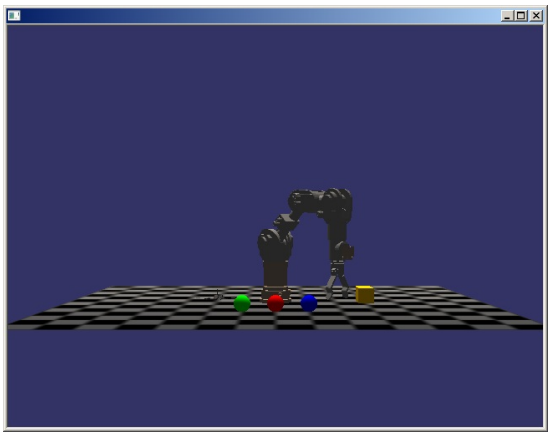
Abbildung 6.1.3: Kamerabild

Die beiden Ansichten der Darstellung 6.1.3 lassen keine Qualitätsunterschiede erkennen. Gut zu sehen ist sowohl die Spitze des Tools mit den beiden Backen des Greifers als auch die gelbe Kiste, welche auf dem Fußboden platziert ist. Im Abschnitt 7.3 wird beschrieben, wie MATLAB und die HAW Simulation sich in Bezug auf die Performance beeinflussen und wo die Engpässe liegen. Der vollständige MATLAB Code des Anwendungsbeispiels (Anhang A.6) sowie die dazugehörige Konfiguration sind zusammen mit anderen Beispielen auf der beigelegten CD einsehbar.

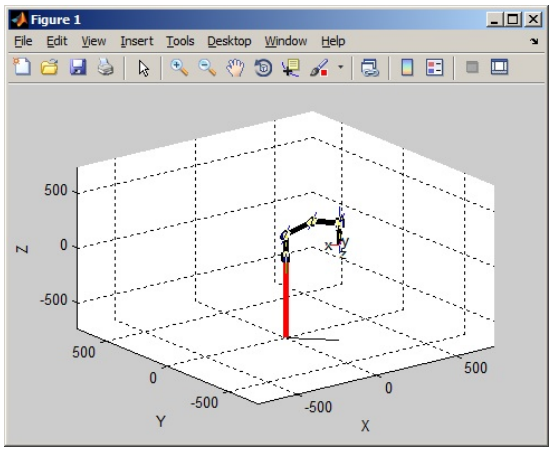
Die korrekte Umsetzung der an die Simulation gesendeten Befehle kann ebenfalls anhand des Vergleiches mit den von MATLAB ausgegebenen Daten analysiert werden. Die drei Views werden zusammen mit den MATLAB Ausgaben in den folgenden Bildern (Abb. 6.1.4) dargestellt. Auch dabei ist zu erkennen, dass die Gelenkstellungen der Simulation exakt denen der über die Robotics Toolbox generierten entspricht.



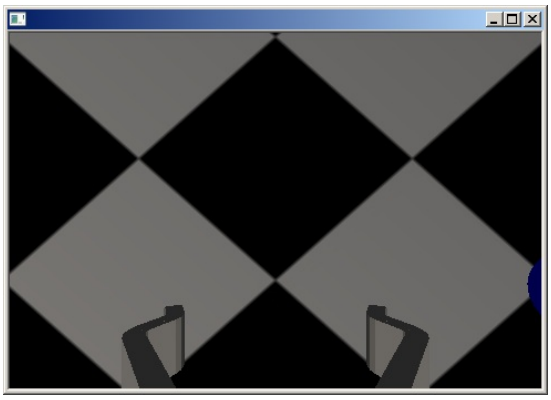
(a) Beobachter View



(b) Free View



(c) MATLAB View



(d) Kamera View

Abbildung 6.1.4: Ansichten der Simulation

6.2 Aktor-Sensor-Regelkreis

Nachdem nun sichergestellt ist, dass die bidirektionale Kommunikation zwischen MATLAB und der Simulationsumgebung funktioniert, ist es Zeit, einen Anwendungsfall zu beschreiben, der dem tatsächlichen Regelkreis (siehe 4.7) entspricht. Dafür wurde folgendes Szenario entwickelt.

Zunächst werden von einem realen Objekt mehrere Kameraaufnahmen geschossen, das Objekt virtualisiert und in die Simulation eingebracht. Es wird so positioniert, dass der Roboter in seiner Ausgangsstellung das Objekt mit seiner Kamera nicht sehen kann. Der Roboter wird dann eine vorgegebene Trajektorie durchlaufen. Wenn die Kamera das gesuchte Objekt gefunden bzw. wiedererkannt hat, sollen dies in Form einer Nachricht sowie der Ausgabe der TCP-Koordinaten in der Konsole ausgegeben und der Roboter wieder in seine Ausgangsposition gefahren werden.

Der Greifvorgang samt iterativer Korrektur der Gelenkstellungen beim Anfahren der Zielpose wird in [Fri11] detailliert beschrieben. Hier wird bewusst ein einfacheres Szenario gewählt.

Als repräsentatives Beispielobjekt wird eine (leere) Verpackung für Teebeutel gewählt. Sie ist mit ihren Dimensionen und Gewicht auch in der Realität ein Objekt, mit welchem der Katana interagieren kann, ohne den Roboter dabei zu überlasten. Nachdem von allen sechs Seiten der Teepackung mit einer Olympus μTough 8010 8MP Photos gemacht wurden, werden diese zugeschnitten, skaliert und als

Textur auf eine zuvor erzeugte virtuelle Box (`ADVANCED_BOX` in der Konfiguration) gelegt. Damit ist die Virtualisierung abgeschlossen (Abb. 6.2.1). Die Qualität der Texturen ist hauptsächlich durch die Eigenschaften der realen Kamera begrenzt. Was eine höhere Auflösung für die Performance bedeutet, wird in 7.3 erläutert.

(a) Vorderseite (b) Rückseite

(c) Deckel (d) Boden

(e) linke Seite (f) rechte Seite

(g) 3D-Teepackung

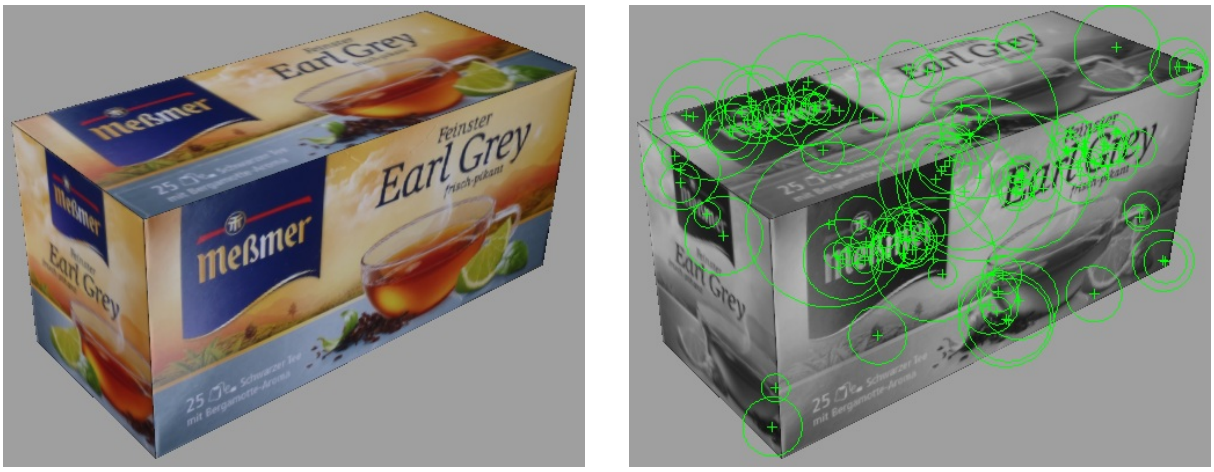
Abbildung 6.2.1: Virtualisierung einer Teepackung

Für das Extrahieren der Merkmale werden mithilfe der Roboterkamera mehrere Aufnahmen aus unterschiedlichen Perspektiven gemacht. Aus diesen kann dann ein 3D-Merkmalsmodell erstellt werden, um später das Objekt aus jeder Kamera-Pose wiedererkennen zu können. Hier wird jedoch ein einfacheres Verfahren gewählt, bei dem nur eine repräsentative Aufnahme zum Vergleichen mit anderen Aufnahmen entnommen wird. Dies wird unter Verwendung der Computer Vision System (CVS) Toolbox durchgeführt. Für die Umsetzung wurden die von Mathworks bereitgestellten Beispiele angepasst, die in der *MATLAB Help* aufgeführt sind.

6.2.1 Merkmalsextraktion und -abgleich

Für die Bestimmung markanter Features können verschiedene Verfahren eingesetzt werden. An dieser Stelle wird das SURF Verfahren als Weiterentwicklung des SIFT Algorithmus verwendet. Die ausführliche Dokumentation dazu findet sich u.a. in [LVG13]. Beide Verfahren finden Punkte in Abbildungen, die invariant gegenüber Skalierung und Rotation sind, d.h. die ein Objekt auch dann noch in einer Aufnahme registrieren, wenn sich die Kameraperspektive und der Fokus geändert haben.

Die CVS Toolbox implementiert diesen Algorithmus und erlaubt, mit wenigen Schritten sowohl die Merkmalsextraktion als auch den Vergleich mit einer zweiten Aufnahme. Beide Bilder müssen dafür im 8bit-Graustufenformat vorliegen. So ist keine eigene Implementierung erforderlich und das Szenario des Regelkreises kann einfach getestet werden. Als Referenzaufnahme wird Abb. 6.2.2a gewählt, da hier gleich drei Seiten der Teepackung sichtbar sind. Mit `detectSURFFeatures` werden die Merkmale bestimmt und in eine $M \times N$-Matrix geschrieben, welche für jeden gefundenen Punkt die xy-Koordinate beinhaltet. Die 100 stabilsten, von insgesamt 245 gefundenen Punkten werden in Abb. 6.2.2b gezeigt.



(a) Teebox Referenz Bild

(b) Teebox SURF Merkmale

Abbildung 6.2.2: Teepackung

Es liegt auf der Hand, dass die Anzahl markanter Punkte im direkten Zusammenhang mit der Texturkomplexität steht. Eine blanke Textur hat außer ihren Kanten keine stabilen Merkmale, während die Textur der Teepackung mehrere hundert SURF-Features aufweist.

Die CVS Toolbox liefert ebenfalls die Funktion mit, die SURF-Features zweier Aufnahmen miteinander zu vergleichen. Diese Bilderkennung erlaubt die Vervollständigung des Regelkreises, welcher in diesem Beispiel wie folgt aufgebaut ist (Abb. 6.2.3).

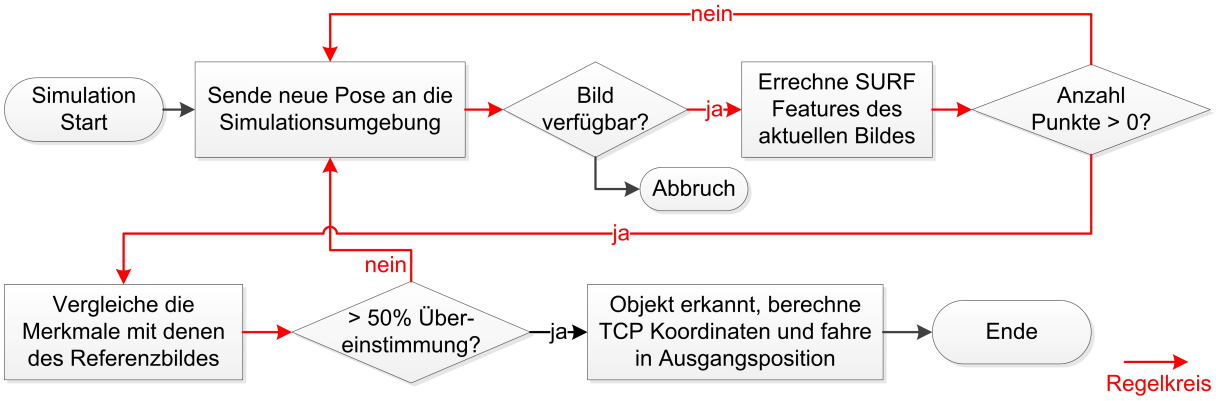


Abbildung 6.2.3: Aktor-Sensor-Regelkreis

Die Steuerung und Bilderkennung wird ausschließlich mit MATLAB realisiert, die Simulationsumgebung dient lediglich der Bilddatenerzeugung. Der Roboter beschreibt aus seiner Ausgangsposition einen Vollkreis und MATLAB vergleicht fortwährend die empfangen Bilddaten mit denen der Referenz. Wenn er das Objekt erkannt hat, endet die Simulation mit der Ausgabe der aktuellen Gelenkwinkel sowie der daraus resultierenden TCP Position. Auch dieses Anwendungsbeispiel befindet sich auf der mitgelieferten CD. Neben SURF können selbstverständlich auch andere Algorithmen implementiert und getestet werden.

Auf der folgenden Seite wird der Ablauf anhand einiger Bilder dargestellt.

Abb. 6.2.4 zeigt von oben vier Aufnahmen, vom Moment nach dem Start (Bild 1) bis hin zum Erkennen des Objektes, hier definiert als die Wiedererkennung von mindestens 10 Merkmalen.

Gut zu erkennen ist sowohl, dass das Kontrollobjekt (Bild 2) nicht fälschlicherweise als Zielobjekt identifiziert wird, es aber dennoch zu Fehlern in der Erkennung kommt (Bild 3). Das Auffinden der gesuchten Teepackung funktioniert erwartungsgemäß (Bild 4).

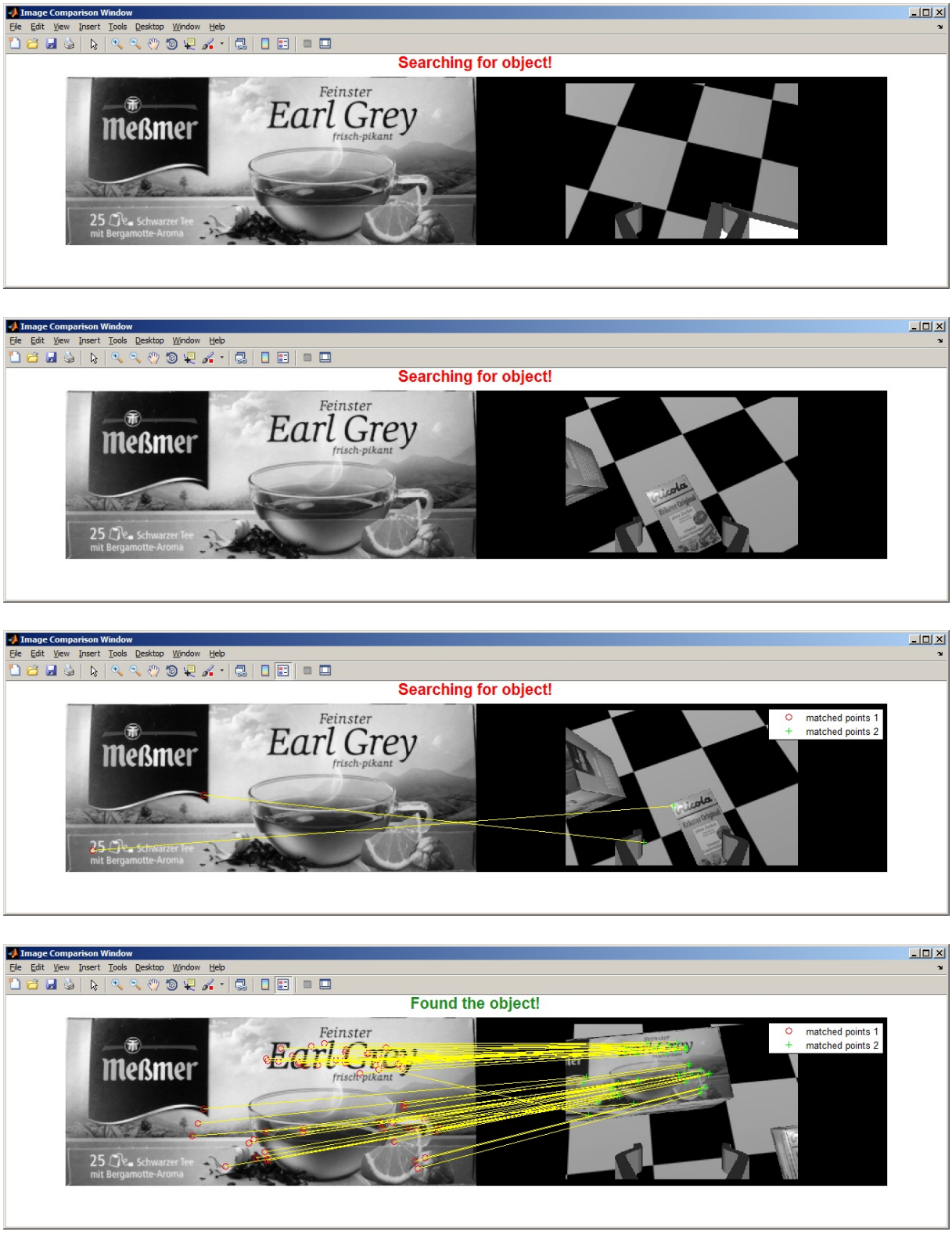


Abbildung 6.2.4: Bildanalyse - Wiedererkennung von Merkmalen

Obwohl kein realer Roboter verwendet wird, *fühlt* es sich für MATLAB an, wie die echte Roboterkamera. Zum Vergleich bieten sich die Bildanalyseergebnisse aus [Fri11] an.

6.3 Kollisionserkennung

Die Kollisionserkennung bzw. -prävention kann maßgeblich dazu beitragen, die Sicherheit im Arbeitsraum eines Roboters zu erhöhen. Dies wird in der Regel mit Ultraschall- und Lasersensoren realisiert. Für eine Simulation können diese Sensoren entweder nachempfunden oder aber durch andere Mechanismen näherungsweise dargestellt werden. Wenn sich die Kanten oder Polygone zweier Objekte schneiden, kommt dies einer Kollision gleich. Wird das jeweilige Objekt mit einer oder mehreren BoundingBox/-Spheres (Abb. 3.3.2) als Sicherheitsbereich umgeben, kann der Abstand zwischen zwei Objekten bestimmt werden. So kann die reale Kollisionserkennung mit der virtuellen verglichen werden, um etwa Sensoren zu kalibrieren.

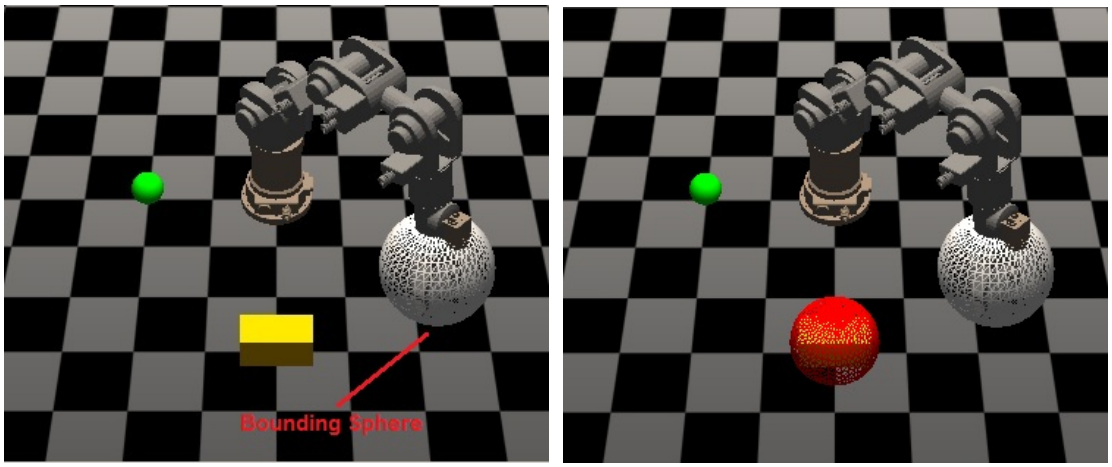
Dieses Anwendungsszenario beschreibt nun, mit welch einfachen Mitteln die Simulationsumgebung um eine Kollisionserkennung erweitert werden kann und wie genau sie funktioniert. Dieses Beispiel gilt analog für viele andere Weiterentwicklungen.

Zunächst wird definiert, welche Objekte/Events involviert sind. In diesem konkreten Fall sind dies der Endeffektor sowie alle Objekte im Raum. Die Kollisionsprüfung soll bei jedem neu berechneten Frame durchgeführt werden, ist also analog zu den bereits bestehenden UpdateCallbacks zu implementieren. Damit steht auch der Punkt der Kopplung fest, nämlich bei der Einbindung des Roboters in der `main` Methode. Der Algorithmus innerhalb des Callbacks ist denkbar einfach:

1. Berechne die Endeffektor Pose in Weltkoordinaten.
2. Generiere die Bounding-Sphere (vgl. 3.3.1) des Tools.
3. Generiere iterativ die Bounding-Sphere für jedes Objekt einzeln und prüfe, ob diese sich mit der des Tools schneidet.
4. Aktualisiere die Ausgabe der Kollisionserkennung.

Die Bounding-Sphere ist eine eher unscharfe, jedoch einfache Form, mit OSG sich durchdringende Geometrien zu erkennen. Eine höhere Genauigkeit kann durch Verfahren auf Polygonebene oder auch durch eigene Definition einer Bounding-Box /-Sphere geringeren Volumens erreicht werden. Ein Tutorial findet sich u.a. in [BM13]. Durch die genannte Unschärfe kommt es vor, dass, obwohl in der 3D-Simulation keine Kollision erkennbar ist, eine solche gemeldet wird. Aus diesem Grund wurde ein Schalter eingebaut, der die Bounding-Sphere des Tools und eines ausgewählten Objektes sichtbar macht (Abb. 6.3.1).

Wenn sich die Bounding Sphere des Endeffektors mit der eines oder mehrerer Objekte schneidet, wird dies in Form einer Textnachricht im Beobachter View ausgegeben (Abb. 6.3.2). Besonders in Bild 6.3.1b gut zu sehen ist die entstehende Ungenauigkeit in den Kollisionsbereichen. Das Volumen um die Objekte ist deutlich größer als das reale. Durch die Verwendung einer Bounding Box im Falle der (gelben) Kiste würde eine exaktere Detektion von Überschneidungen erreicht. Als erste Näherung ist die Sphäre jedoch sehr gut geeignet, da es um Kollisionsvermeidung geht und solch eine zusätzliche Sicherheit durchaus von Vorteil ist.



(a) Bounding Sphere im TCP (b) Kollisionsräume des TCP und Box

Abbildung 6.3.1: Arbeitsraum mit Kollisionserkennung

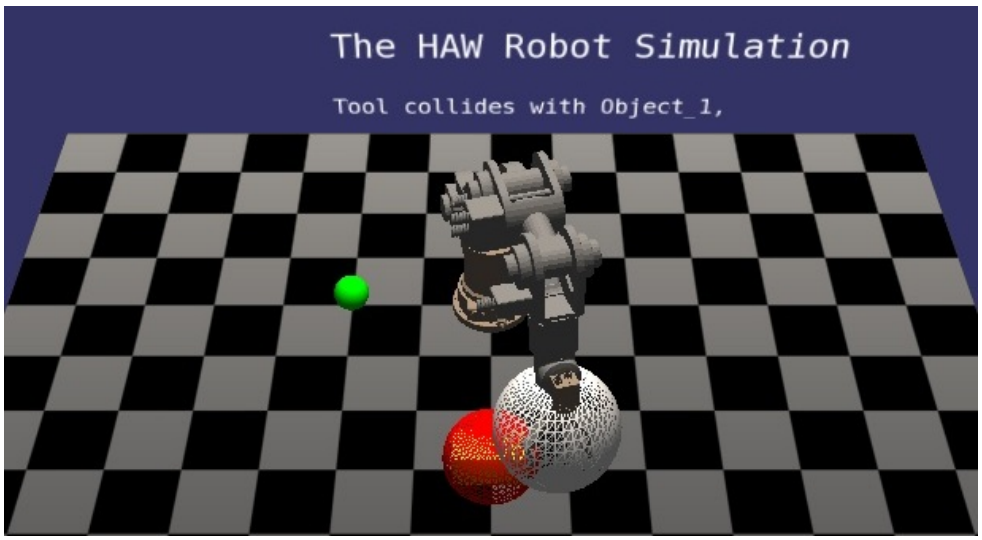


Abbildung 6.3.2: Kollisionserkennung

6.4 Weitere Szenarien

Das Anwendungsgebiet der Software ist nicht auf Robotersimulation beschränkt. Da nahezu jedes 3D-Objekt dargestellt und remote bewegt werden kann, können auch Fahrzeuge wie die des RoboCups[28] simuliert werden. Auch hier könnte dies die Entwicklung beschleunigen bzw. den Materialeinsatz vermindern. Es ist ebenfalls die Einbindung in den Praktikumsbetrieb an der HAW möglich. So könnte der VR-Builder von MATLAB abgelöst werden, wenn es z.B. um die Ausgabe von Bewegungsmustern im Bereich der Physiksimulation geht. Auf weitere Möglichkeiten geht das Kapitel 8.2 ein.

[28] `http://www.robocup.org`

7 Ergebnisse

Mit der hier beschriebenen Arbeit wurde eine Lösung für die 3D-Simulation von Robotern mit ihren Aktoren und unterschiedlichen Sensoren realisiert. Die Ergebnisse der Anwendungsbeispiele aus dem vorhergehenden Kapitel (6) sowie der durchgeführten Experimente werden im Folgenden beschrieben. Das Kapitel stellt die Ergebnisse in vier übergeordneten Punkten zusammen: Darstellungsqualität (7.1), Steuerung und Datenübertragung (7.2), Performance (7.3) und Applikation (7.4).

7.1 Darstellungsqualität

Unter Einsatz von OpenSceneGraph und den Boost Bibliotheken wurde eine C++ Anwendung realisiert, welche die dreidimensionale Darstellung beliebiger linearer Manipulatoren am Computer ermöglicht. Dabei wird die Darstellungsqualität nur durch zwei Faktoren begrenzt:

1. Die Leistung der Hardware, auf dem die Software eingesetzt wird
2. Die Güte der 3D-Modelle, die der Anwendung bereitgestellt werden

Die einzelnen Modelle können dabei unterschiedliche Qualität haben. Neben der reinen Robotersimulation kann eine vollständige Umgebung einschließlich Objekten, Licht und Schatten generiert werden. Die Objekte können frei im Raum bewegt und mit Texturen versehen werden, deren Auflösung theoretisch unbegrenzt sein kann. Obendrein können Kameras simuliert werden, deren ex- wie intrinsische Parameter analog zu einer echten konfiguriert und relativ zum Endeffektor des Manipulators positioniert werden. Dadurch besteht die Möglichkeit, Bildanalyse zu betreiben und Algorithmen für visuell geführte Roboter entwickeln und testen zu können, wie dies im Anwendungsbeispiel 6.2 beschrieben wurde.

(a) Reale Kameraaufnahme (b) Virtuelle Kameraaufnahme

Abbildung 7.1.1: Gegenüberstellung der Kameraaufnahmen

Im Vergleich zu den bestehenden Simulationen, die vorwiegend mit von MATLAB bereitgestellten Methoden umgesetzt wurden, ist hier eine neue Nähe zur Realität erreicht worden, die das Arbeiten mit dem realen Roboter sehr gut unterstützt. Bei der Bilderkennung ist praktisch kein Unterschied sichtbar,

ob ein Objekt nun von einer echten oder einer simulierten Kamera aufgenommen wurde (vgl. Abb. 7.1.1). Auch die 3D-Darstellung des Roboters in seiner Umgebung ist gravierend verbessert worden. (vgl. Abb. 6.1.4a und 6.1.4c). Dabei ist die Performance gegenüber der nativen RTB Visualisierung sogar erhöht worden. Die Verarbeitung erfolgt flüssig und es ist, je nach Hardwareeinsatz kein *Ruckeln* festzustellen. Abschnitt 7.3 geht tiefer auf die Performance ein.

7.2 Steuerung und Datenübertragung

Es wurde eine Vielzahl von Steuerungs- / Interaktionsschnittstellen zur Simulationsumgebung geschaffen. Der Anwender kann sowohl mit Maus und Tastatur die Simulation beeinflussen, als auch über Nachrichten via Standard TCP/IP. Unabhängig von der eingesetzten Steuerungssoftware muss nur der Protokollstandard sowie das Nachrichtenformat beachtet werden. Daraus Resultiert ein hohes Maß an Flexibilität. Es spielt keine Rolle, ob die Befehle von MATLAB, ROS oder etwa einem Perl Skript gesendet werden, das Ergebnis ist das Gleiche.

Das Kommunikationsprotokoll basiert auf Textnachrichten und kann nach Belieben um weitere Kommandos (etwa zur Umgebungssteuerung) erweitert werden. Der Overhead der Kommunikation ist durch die Verwendung von TCP/IP zwar etwas erhöht, doch dies wird durch die gewonnene Sicherheit der verbindungsorientierten Arbeitsweise des Protokolls ausgeglichen.

Auch die Übertragung der Bilddaten ist einfach zu handhaben, da das standardisierte JPEG Format verwendet wird. So können die Bilder auf jedem Computer angezeigt und weiterverarbeitet werden. Durch die JPEG Komprimierung und bei maximaler Qualität der Bilder, sind auch große Kameraansichten immer noch mit über 30 Bildern pro Sekunde über ein 100MBit Netzwerk problemlos übertragbar. Aufgrund der Tatsache, dass der Grad der Kompression ebenfalls konfigurierbar ist, sind auch höhere Bildraten bei gleichzeitig weitgehendem Erhalt des Bildinhaltes möglich. Die Verwendung von weit verbreiteter Software ermöglicht zudem den Einsatz von Streaming für noch einfachere Bild-/Videodatenübertragung in auf dieser Arbeit aufbauenden Projekten.

7.3 Performance

In Abhängigkeit der Hard- und Softwarekonfiguration kann die Performance stark schwanken. Um eine klare Aussage über die Einsatztauglichkeit der Anwendung machen zu können, wurden zahlreiche Testszenarien entwickelt und die Simulation mithilfe dieser hinsichtlich Bildqualität/ -rate und der Anzahl und Güte der dargestellten Modelle untersucht.

Folgende Randbedingungen wurden festgelegt:

- Alle Messungen werden in der Infrastruktur aus 5.1 durchgeführt. Außer der Simulationsumgebung sind nur Standard Windows Prozesse aktiv, soweit das Szenario nichts anderes vorgibt.
- Die vertikale Synchronisation der Grafikkarte ist deaktiviert.
- Die Messungen werden alle aus der Applikation heraus mit OSG Mitteln durchgeführt. (s-Taste für das Anzeigen der Frame Rate)
- Es gibt drei Ansichten. Der Freie View sowie die Beobachteransicht haben eine Fenstergröße von 640×480 Bildpunkten. Die Bildebene der Kamera wird mit 480×320 Pixeln angezeigt.

- Es werden drei Messungen der Bildrate durchgeführt: Nur mit der Beobachter Ansicht View (fps 1), nur mit Roboter-Kamera-Bild (fps 2) und mit allen drei Ansichten (fps 3). Die Werte sind gerundet.
- Die Anordnung der einzelnen Objekte ist in Abb. 7.3.1 festgelegt.
- Die Messtoleranz bewegt sich im Bereich von $+/-10\%$ der Bildrate.

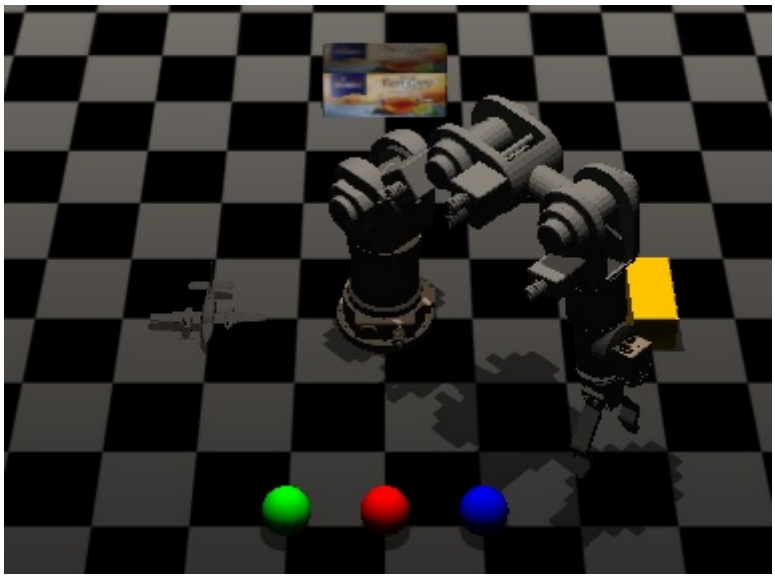


Abbildung 7.3.1: Anordnung der Objekte während der Messungen

Die Szenarien sind zusammen mit den Ergebnissen in Tabelle 7.3.1 zusammengefasst.

Nr.	Szenario	Vertices	fps 1	fps 2	fps 3
01	leere Simulation	0	4.000	N/A	1.000
02	Umgebung (Boden) mit einer Standard Kiste	28	3.150	N/A	850
03	[02], nur mit fünf Standardobjekten	14.652	2.000	N/A	650
04	[03] + einer Teepackung mit 6 Texturen á 1MP	14.676	1.950	N/A	640
05	[04], jedoch mit 6 Texturen á 8MP	14.676	1.950	N/A	640
06	[04], zuzüglich Katana Roboter ohne Cam-Model	108.600	1.700	1.150	550
07	[06], jedoch mit Kamera Modell	159.318	1.700	1.150	550
08	[07], mit einer Lichtquelle	159.318	1.700	1.200	550
09	[08], mit Schatten	159.318	920	700	300
10	[08], mit PERL als Controller (5 Gelenke)	159.318	900	650	290
11	[08], mit MATLAB als Controller (5 Gelenke)	159.318	900	650	290
12	[11], mit Plot des Roboters in MATLAB	159.318	800	620	200
13	[12], mit Plot der Bilddaten in MATLAB	159.318	N/A	500	150

Tabelle 7.3.1: Performance Analyse

Die Anzahl der Eckpunkte variiert nicht nur durch die Menge und den Detaillierungsgrad der Objekte innerhalb der Simulation, sondern auch durch die Perspektive der Ansichten. Im Roboter-Kamera View verändern sich bei Bewegung die sichtbaren Objekte genauso, wie im FreeView bei Veränderung der Kamerapose durch die Maussteuerung. Die Kantenangabe der Tabelle bezieht sich daher auf den Beobachter View, da dort zu jeder Zeit die gesamte Szene sichtbar ist. Die Konfiguration befindet sich

dokumentiert auf der beiliegenden CD. Die Texturgröße und -qualität stellt einen weiteren Faktor dar. Bei großen Bildern, die für ihre Zieloberfläche stark skaliert werden müssen, verringert sich zwar die Framerate nur minimal, aber das Starten der Anwendung nimmt aufgrund der Skalierung mehr Zeit in Anspruch.

Weiterhin ist zu beobachten, dass sich die Anzahl gleichzeitig sichtbarer Views zur erwarteten Bildrate antiproportional verhält. Bei der Bilddatengenerierung findet ein Kopiervorgang im Arbeitsspeicher (mithilfe von `stringstream`) statt, welcher ebenfalls einen negativen Einfluss auf die Bildrate hat. Das parallele Laufen von MATLAB samt Roboter- und Bilddatenplot beansprucht die CPU auch in einem Maße, das sich die Bildrate verringert. Auch gut zu erkennen ist, dass die Netzwerkkommunikation die Bildrate nicht beeinflusst, da sie in einem separaten Thread läuft.

Mit allen gängigen Features aktiviert sind bei dieser Hardware immer noch 150 fps möglich, wenn nur die notwendigen Daten (Beobachterkamera und MATLAB Bildplot) angezeigt werden, noch deutlich höhere. Auch beim Test auf einem vier Jahre alten Notebook wurden noch Bildraten von 60 fps erreicht. Es ist jedoch bei Notwendigkeit von hohen Bildraten, $>$60 fps, darauf zu achten, dass die vertikale Synchronisation für die Anpassung der Bildrate an die Bildwiederholfrequenz des Monitors deaktiviert ist, da diese die Framerate künstlich drosselt. Selbstverständlich sinkt die Bildrate auch mit steigender Anzahl laufender Programme auf dem Simulationsrechner.

7.4 Applikation

Das Resultat dieser Arbeit, die Anwendung *HAW Robot Simulation*, wurde von Grund auf neu entwickelt und setzt ausschließlich auf gültige und erprobte Standards. Dies beginnt bei der Wahl der Programmiersprache und der Bibliotheken und endet beim Kommunikationsprotokoll und dem Dateistandard. Grundlage für die Architektur war eine ausführliche Analyse bestehender Techniken, aus denen die Vorteile übernommen und viele Nachteile ausgeräumt werden konnten.

Die Anwendung wurde vollständig modular aufgebaut und ist auf unterschiedlichen Plattformen einsetzbar. Soweit möglich wurde das Konzept der Austauschbarkeit der einzelnen Features implementiert, so dass Weiterentwicklungen oder Portierungen flexibel durchgeführt werden können. Das Design und die daraus resultierende Implementierung beinhaltet die Verteilung der Anwendung (über mehrere Systeme) genauso wie Multi-Threading.

Die entwickelte Software erlaubt die Simulation unterschiedlicher Roboter und Umgebungen ohne große Vorkenntnisse. Wie in den Anwendungsbeispielen (6) an einigen Szenarien gezeigt wurde, deckt die Funktionalität ein breites Einsatzgebiet ab und erlaubt so die unkomplizierte parallele Entwicklung diverser Projekte bei nur einem oder auch keinem real vorhandenen Roboter.

Da es sich um eine Neuentwicklung handelt, wurde besonders großer Wert auf die Architektur sowie eine detaillierte Beschreibung der Implementation gelegt. Dies gilt u.a. für Verwendung bestehender Designmuster als auch die Codequalität (vgl. [AA04]). Das Ergebnis ist eine leistungsfähige Anwendung für 3D-Robotersimulation und -visualisierung.

8 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit (8.1) zusammengefasst. Abschließend wird ein Ausblick auf mögliche Weiterentwicklungen in Form von Verbesserungen bzw. Erweiterungen (8.2) gegeben.

8.1 Stand der Arbeit

In dieser Arbeit wurde auf OpenGL und C++ Basis eine Aktor-Sensor Simulation für Roboter entwickelt und implementiert.

Als Vorlage diente der Katana 6M180 Greifarmroboter. Dieser ortsgebundene Manipulator mit einem Aktionsradius von etwa $60cm$ ist mit einer Kamera mit Weitwinkelobjektiv ausgestattet, um seine Umgebung wahrnehmen zu können. Anhand dieses Roboters wurde eine 3D-Simulationssoftware entwickelt.

Die Konzeption dafür entstand nach ausführlicher Analyse existierender Lösungen sowie der Berücksichtigung der initialen Anforderungen. Es ergab sich eine Architektur (4), welche sich im Wesentlichen aus einem 3D-Daten Generierungsprozess und einem Netzwerkserver zusammensetzt.

Für die Umsetzung wurde der Katana Roboter virtualisiert. Zudem entstand eine Umgebungssimulation mit zahlreichen definierbaren Objekten, welche durch Texturen real existierenden Gegenständen nachempfunden wurden. Auch Umgebungseinflüsse wie Licht und Schatten wurden umgesetzt. Die Umgebung wurde voll konfigurierbar gestaltet und lässt so schnelle Adaptionen zu. Diese Flexibilität wurde auch im Hinblick auf den Einsatz auf unterschiedlichen Betriebssysteme (Windows, Linux, Mac OS) implementiert, da ausschließlich quelloffene, weit verbreitete Standards verwendet wurden.

Das Resultat ist eine Anwendung, die über Netzwerknachrichten gesteuert wird und so die Verbindung zu externen Applikationen wie MATLAB oder ROS ermöglicht. So kann der Roboter in der Simulation gesteuert werden und die ebenfalls simulierte Kamera als Sensor liefert Bilddaten, die z.B. für die Bahnplanung verwendet werden können. Die 3D- und 2D-Darstellungen (Bilder) sind dabei sehr realitätsnah. In Kapitel 6 werden neben der Robotersteuerung und Bildanalyse zusätzlich eine Kollisionserkennung vorgestellt. Dieses Feature kann insbesondere für die Validierung der Bahnplanung genutzt werden. Ebenso ergänzt sie parallel zur Bildanalyse den Aspekt der Sicherheit bei der Roboterinteraktion.

Die implementierte Lösung ist sehr flexibel und kann ohne weiteres mit anderen Robotern / Kameras arbeiten, auch mit mehreren unterschiedlichen Robotern parallel. Gleichzeitig ist bei entsprechendem Hardwareeinsatz eine Echtzeit-Simulation möglich.

Die Ergebnisse (7) erfüllen die ursprünglichen Erwartung vollständig und doch sind noch viele Erweiterungen und Verbesserungen für die Zukunft angedacht.

8.2 Verbesserung / Erweiterung

Die entwickelte Anwendung stellt eine Vorlage dar, welche einige Möglichkeiten 3D-simulierter Roboter aufzeigt. Als Quasi-Prototyp bietet sie natürlich viel Raum für Verbesserungen bzw. neue Funktionalitäten.

Durch die Erweiterung der Gelenkkonfiguration um einen *Eltern-Gelenk* und eine Anpassung in der *createRobot* Routine können auch komplexere Roboter dargestellt werden. Dies können beispielsweise Roboter mit redundanten Gelenken sein. Eine andere mögliche Anpassung ist die Nutzung der Kollisionserkennung zur Modellierung von Tast-Sensoren. Ebenso kann sie dazu dienen, Objekte mithilfe des Endeffektors zubewegen.

Auch die Darstellungsqualität bietet noch viel Spielraum nach oben. Neben komplexeren Schattenmodellen oder einer einfacheren Methode der Lichtquellendefinition kann auch die Erzeugung der Bilddaten optimiert werden. So ist es denkbar, das Bildformat (RAW, JPG, PNG) auswählbar zu machen oder aber die Farbtiefe (8bit Graustufen, 24Bit RGB) zu wählen. Letzteres könnte eine höhere Performance, also höhere Bildraten via Netzwerk, ermöglichen.

Eine weitere Optimierungsmöglichkeit bietet die Netzwerk-Schnittstelle selbst. Der TCP-Server unterstützt nur eine Verbindung zur Zeit. Mit mehreren Clients jedoch, könnte man die Berechnungen für die nächste Pose und die Bildanalyse auf getrennten Rechnern ausführen, oder aber die Kontrolle über mehrere Roboter verteilen.

Neben der Optimierung der graphischen Darstellung und Erweiterung der Definitionsmöglichkeiten für Objekte und Roboter, bietet die Anwendung Spielraum für die Implementierung realitätsnaher physikalischer Gesetze. Es wäre etwa denkbar, die Gravitation auf die Objekte im Raum wirken zu lassen, so dass sie nicht mehr *schweben*, sondern zu Boden fallen, wenn sie in einer Höhe über dem Grund positioniert werden.

Ein bereits existierendes Beispiel aus der Entwicklung mit OSG ist die ebenfalls frei erhältliche Physiksimulation mit `osgBullet`[29].

Dies sind nur einige Möglichkeiten, wie die Anwendung weiterentwickelt werden kann.

[29] `http://code.google.com/p/osgbullet/`

A Auszüge aus dem Code

Dieses Kapitel beinhaltet die verschiedenen Codeauszüge, die in den Kapiteln 4,5 und 6 textuell beschrieben werden.

Codeauszug A.1: MATLAB Prototyp

```
%%%%%%%%%%%%%%%%%%%%| theCameraModelWidtRobot.m |%%%%%%%%%%%%%%%%%%%%

close all; clear all
%%%%%%%%%%%%%%%%%%%%
%the environment
%%%%%%%%%%%%%%%%%%%%
figure(1); %grid on; axis equal; hold on;
title('The Robot with Kamera ');
figure(2); grid on;
title('The Image plane');
%viewpoint
%view(60,46);
%create scene points
X=[300,400,300];
Y=[200,300,400];
Z=[0,0,0];
scenePoints=[X;Y;Z];

%%%%%%%%%%%%%%%%%%%%
%the camera
%%%%%%%%%%%%%%%%%%%%

%standard intrinsic camera parametres
%Kd=eye(3);
%real intrinsic camera parametres from Carsten (in pixels)
Kd=[923.56072  0.0     527.91664;
      0.0   -923.65235 419.86750;
      0.0      0.0       1.0    ];

%%%%%%%%%%%%%%%%%%%%%%%%
%the Katana robot
%%%%%%%%%%%%%%%%%%%%%%%%

kat6=getRobotKatana();

%%%%%%%%%%%%%%%%%%%%%%%%%%
%katana start position
%%%%%%%%%%%%%%%%%%%%%%%%%%

options=optimset('Display','off','NonlEqnAlgorithm', ...
    'gn', 'MaxFunEvals', 1000);
toolRadius = 250;  toolHeight = 120;  toolPitch=180; %degrees
x0 = [.3; 0.2; 0.6];
%inverse kinematics formula
fkt = @(x)MSL_SolveNonlin_01(x,toolRadius,toolHeight,toolPitch);
%solve inverse kinematics
x = fsolve(fkt, x0, options);  % Call optimizer
```

```
48
49 q2=x(1);
50 q3=x(2);
51 q4=x(3);
52 %calculate angles
53 q1=0;
54 q2=atan2(sin(q2), cos(q2));
55 q3=atan2(sin(q3), cos(q3));
56 q4=atan2(sin(q4), cos(q4));
57 q5=0;
58 %set initial angles
59 qStart=[q1 q2 q3 q4 q5]
60
61
62 %%%%%%%%%%%%%%%%%%%%%%%%
63 %turn the cam 360deg and refresh the figure 1&2
64 for i=1:1:90,
65     figure(1)
66     clf
67     hold on
68     %world frame
69     f_3Dwf('r',500,'_{world}');
70     %get TCP position
71     qCurrent=[q1 q2 q3 q4 q5];
72     t6=fkine(kat6, qCurrent);
73     %get TCP rotation
74     e=tr2rpy(t6);
75     %camera position
76     td=[t6(1,4),t6(2,4),t6(3,4)]';
77     %camera rotation (look down and rotate / look from TCP pose)
78     q1=i*pi/180;
79     %Rd=rotoz(0)*rotoy(q5)*rotox(-pi/2);
80     Rd=rotoz(0)*rotoy(-q1-q5)*rotox(q4+pi/2);
81     %Rd=rotox(e(1))*rotoy(e(3))*rotoz(e(2));
82     %creating the homogenuous matrix
83     Hd=f_Rt2H(Rd,td);
84     %plot the camera frame
85     f_3Dframe(Hd,'g',20,'_{cam}');
86     %plot the 3D pin-hole camera
87     f_3Dcamera(Hd,'g',25);
88     %reset the scene points
89     scenePoints=[X;Y;Z];
90     %plot scene points
91     f_scenepnt(scenePoints,'b*',1);
92     f_3Dwfenum(scenePoints,'k',.5);
93     %the point of view from outside
94     view(60,46);
95     grid on
96     %plot the robot
97     plot(kat6,qCurrent);
98
99     axis auto
100
101    figure(2)
102    %clf
103    hold on
104    [ud,vd]=f_perspproj(scenePoints,Hd,Kd,2);
105    plot(ud,vd,'gO');
106    grid on;
107
108     pause(0.1)
```

```
109     if i<360
110         hold off
111     end
112 end
```

Codeauszug A.2: MovementCallback

```
1  #include "StdAfx.h"
2  #include <osg/PositionAttitudeTransform>
3  #include <boost/algorithm/string.hpp>
4  #include "MovementCallback.h"
5  #include "global.h"
6
7  using namespace osg;
8  using namespace std;
9
10 MovementCallback::MovementCallback(){}
11
12 //overwrite () operator to force movement update on every frame
13 void MovementCallback::operator()(osg::Node* node,osg::NodeVisitor* nv){
14   PositionAttitudeTransform *objectPAT =
15     dynamic_cast<osg::PositionAttitudeTransform*>( node );
16   map<string,double>::iterator it;
17   std::vector<std::string> strings;
18
19   g_mutex_movement_instructions.lock();
20   //for every instruction in the map
21   for(it=g_movement_instructions.begin();it!=g_movement_instructions.end();it++){
22     //check if it is for this joint
23     if(it->first.find(objectPAT->getName()) != string::npos){
24
25       boost::split(strings, it->first, boost::is_any_of("_"));
26       string posAtt= strings[2];
27       double value = it->second;
28
29       if(g_ALL_ANGLES_IN=="RAD" && (posAtt==ROTX || posAtt==ROTY || posAtt==ROTZ))
30         value=RadiansToDegrees(value);
31
32       if(posAtt==POSX)
33         objectPAT->setPosition(Vec3(value,objectPAT->getPosition().y(),
34           objectPAT->getPosition().z()));
35       else if(posAtt==POSY)
36         objectPAT->setPosition(Vec3(objectPAT->getPosition().x(),value,
37           objectPAT->getPosition().z()));
38       else if(posAtt==POSZ)
39         objectPAT->setPosition(Vec3(objectPAT->getPosition().x(),
40           objectPAT->getPosition().y(),value));
41       else if(posAtt==ROTX)
42         objectPAT->setAttitude(Quat(DegreesToRadians(value), X_AXIS,
43           objectPAT->getAttitude().y(), Y_AXIS,
44           objectPAT->getAttitude().z(), Z_AXIS));
45       else if(posAtt==ROTY)
46         objectPAT->setAttitude(Quat(objectPAT->getAttitude().x(),
47           X_AXIS,DegreesToRadians(value), Y_AXIS,
48           objectPAT->getAttitude().z(), Z_AXIS));
49       else if(posAtt==ROTZ)
50         objectPAT->setAttitude(Quat(objectPAT->getAttitude().x(),
51           X_AXIS, objectPAT->getAttitude().y(), Y_AXIS,
52           DegreesToRadians(value), Z_AXIS));
53       else{}
54     }
55   }
```

```
56   g_mutex_movement_instructions.unlock();
57
58   traverse(node, nv);
59 }
60
61 MovementCallback::~MovementCallback(void){}
```

Codeauszug A.3: CameraUpdateCallback

```
1 ...
2 CameraUpdateCallback::CameraUpdateCallback(PositionAttitudeTransform* cam,
3   bool printCamPose)
4   :cam_(cam),
5   printCamPose_(printCamPose){}
6
7 void CameraUpdateCallback::operator()(Node* node,NodeVisitor* nv){
8   Camera* camera = dynamic_cast<Camera*>(node);
9
10  //Get camera matrix
11  Matrix camMatrix=computeLocalToWorld(cam_->getParentalNodePaths()[0]);
12
13  //calculate and set new camera matrix
14  camera->setViewMatrix(camMatrix.inverse(camMatrix)
15    * Matrix::rotate(PI,0,1,0) //UP - Vektor
16    * Matrix::rotate(PI/2,0,0,1));//CAM - Direction
17
18  //Print camera translation and rotation
19  if(printCamPose_){
20    Vec3 camTrans = camMatrix.getTrans();
21    Quat camRot = camMatrix.getRotate();
22
23    OSG_NOTICE<<"trans␣x:␣"<<camTrans.x()
24    <<"␣trans␣y:␣"<<camTrans.y()
25    <<"␣trans␣z:␣"<<camTrans.z()<<endl;
26
27    OSG_NOTICE<<"rot␣x:␣"<<RadiansToDegrees(camRot.asVec3().x())
28    <<"␣rot␣y:␣"<<RadiansToDegrees(camRot.asVec3().y())
29    <<"␣rot␣z:␣"<<RadiansToDegrees(camRot.asVec3().z())<<endl;
30  }
31 }
32 ...
```

Codeauszug A.4: CameraTakePictureCallback

```
1 #include "StdAfx.h"
2 #include "CameraTakePictureCallback.h"
3 #include "global.h"
4 #include <osgDB/WriteFile>
5 #include <osgDB/ReaderWriter>
6 #include <boost/lexical_cast.hpp>
7
8 CameraTakePictureCallback::CameraTakePictureCallback(void){}
9
10 CameraTakePictureCallback::~CameraTakePictureCallback(void){}
11
12 void CameraTakePictureCallback::operator ()
13   (const ::osg::CameraNode& camera) const{
14
15   int x = camera.getViewport()->x();
16   int y = camera.getViewport()->y();
17   int width = camera.getViewport()->width();
18   int height = camera.getViewport()->height();
```

```
19    osg::ref_ptr<osg::Image> image = new osg::Image;
20    osg::ref_ptr<osgDB::ReaderWriter::Options> op =
21            new osgDB::ReaderWriter::Options();
22    image->readPixels(x,y,width,height,GL_RGB,GL_UNSIGNED_BYTE);
23
24    g_mutex_createImage.lock();
25    if(!g_createImage)
26      g_mutex_createImage.unlock();
27    else{
28      g_createImage=false;
29      g_mutex_createImage.unlock();
30      std::stringstream ss;
31      osg::ref_ptr<osgDB::ReaderWriter> writer;
32      string picType=properties->getProperty("PICTURE_FILETYPE");
33
34      //determine if JPG or PNG image is to be written
35      if(picType=="JPG" || picType!="PNG"){
36        writer = osgDB::Registry::instance()->getReaderWriterForExtension("jpg");
37        unsigned int compression = properties->getAsInt("PICTURE_QUALITY");
38        if(compression>0 && compression<=100)
39          op->setOptionString("JPEG_QUALITY␣"
40            + boost::lexical_cast<string>(compression));
41        else
42          op->setOptionString("JPEG_QUALITY␣100");
43        if(picType!="JPG" && picType!="PNG")
44          OSG_WARN<<"[WARN]:␣PICTURE_FILETYPE␣not␣correct,
45 ␣␣␣␣␣␣␣␣␣␣using␣JPG␣(default)"<<endl;
46      }else
47        writer = osgDB::Registry::instance()->getReaderWriterForExtension("png");
48
49      writer->writeImage(*image,ss,op);
50
51      //Write current image to variable
52      g_mutex_camera_image_data.lock();
53      g_camera_image_data = ss.str();
54      g_camera_image_size = g_camera_image_data.size();
55      g_mutex_camera_image_data.unlock();
56
57      //signal waiting process that image data is complete
58      g_mutex_createImageReady.lock();
59      g_createImageReady=true;
60      g_mutex_createImageReady.unlock();
61
62    }
63    //future enhancement (for taking pictures via print-key)
64    g_mutex_takePicture.lock();
65    if(g_takePicture){
66      g_takePicture=false;
67      g_mutex_takePicture.unlock();
68      string fileName=properties->getProperty("FILEPATH")
69                    + properties->getProperty("PICTURE_NAME");
70      if (osgDB::writeImageFile(*image,fileName,op)){
71        OSG_NOTICE  << "Saved␣screen␣image␣to␣"<<fileName<<"␣."<< std::endl;
72      }else{
73        OSG_WARN << "Error␣while␣saving␣image..."<<std::endl;
74      }
75    }else{
76      g_mutex_takePicture.unlock();
77    }
78  }
```

Codeauszug A.5: Perl Network Test Script

```
#!/usr/bin/perl -w
use IO::Socket;

#Create socket
$socket = new IO::Socket::INET (
    PeerAddr  => '127.0.0.1',
    PeerPort  =>  6099,
    Proto => 'tcp') or die "Couldn't connect to Server\n";

#endless loop
while(1){
   #turn joint 2 from 0 to 360 degrees
   for($i=0;$i<360;$i+=0.1){
      $send_data = "Katana_joint2_ROTZ:$i\n";
      if($i % 1 ==0){$j++;}
      $socket->send($send_data);
      #Sleep 0.01 seconds
      select(undef, undef, undef, 0.01);
   }
}
```

Codeauszug A.6: Anwendungsfall 1 - MATLAB Skript

```
%---   example1.m   ---%
%robot definition
l1=201.5;
l2=190;
l3=139;
l4=206.1;

%      link([   alpha    a       theta    d       Sigma], CONVENTION)
L{1} = link([   pi/2     0        0       l1        0], 'standard');
L{2} = link([     0     l2       0       0         0], 'standard');
L{3} = link([     0     l3       0       0         0], 'standard');
L{4} = link([  -pi/2     0        0       0         0], 'standard');
L{5} = link([     0      0       0       l4        0], 'standard');

R=robot(L,'Katana 6M','Neuronics');
qStart=[-pi/4 pi/4 -pi/4 pi 0];   %start position
plot(R,qStart);

%connect with sim
tcpc=tcpip('localhost', 6099, 'NetworkRole', 'client',...
    'InputBufferSize',200000);
fopen(tcpc);

%define temporary picture name
picname='simpic.jpg';

robFig = figure(1);
set(robFig, 'Position', [530 50 480 320]);

q=qStart;
plot(R,q);   %plot robot start position

camFig=figure(2);
set(camFig, 'Position', [50 50 480 320]);

for i=0:0.5:360,
    %update joint 1
```

```
38      q(1)=i*pi/180;
39
40      %update robot figure (makes it more slowly)
41      %robFig;
42      %plot(R,q);
43      camFig;
44
45      %In the first loop, no image will be displayed
46      if i ~= 0
47          picture=imread(picname);
48          figure(2);
49          image(picture);
50      end
51
52      %angle adaption to simulation (see documentation [6.1.1])
53      qSim={(q(1)-pi)*180/pi
54          (q(2)-pi/2)*180/pi
55          q(3)*180/pi
56          (q(4)-3/2*pi)*180/pi
57          (q(5)-pi)*180/pi};
58
59      %prepare command string for sending
60      sendString = [
61          sprintf('Katana_joint2_ROTZ:%f;',qSim{1})...
62          sprintf('Katana_joint3_ROTY:%f;',qSim{2})...
63          sprintf('Katana_joint4_ROTY:%f;',qSim{3})...
64          sprintf('Katana_joint5_ROTY:%f;',qSim{4})...
65          sprintf('Katana_joint6_ROTZ:%f;',qSim{5})...
66          'GETIMG:1.0\n'
67      ];
68      %send command string
69      fprintf(tcpc,sendString);
70
71      %receive image size to determine buffer size
72      imgsize=fread(tcpc,8,'uint8');
73      imgBufferSize=str2double(char(imgsize));
74
75      %read image data from stream and write into file
76      imdata=fread(tcpc,imgBufferSize,'uint8');
77      fid = fopen(picname, 'w');
78      fwrite(fid, imdata);
79      fclose(fid);
80  end
81  fclose(tcpc);
82  delete(tcpc);
83  clear tcpc;
```

B Inhalt der beiliegenden CD

Dieser Arbeit liegt eine CD mit dem nachfolgendem Inhalt bei:

- Dokumentation der Masterarbeit im PDF Format
- Vollständig dokumentierter Quellcode und ausführbare Software
- Dokumentierte Anwendungsbeispiele
- Schnellstartanleitung für den Gebrauch der HAW Robot Simulation Anwendung
- Kommando-Referenz für die Konfigurationsdatei

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abb.	Abbildung
API	Application Programming Interface
bzw.	beziehungsweise
CAD	Computer Aided Design
CAN	Controller Area Network
CCD	Charge-Coupled Device - digitaler Bildsensor
CPU	Central Processing Unit - Prozessor
CVS	Computer Vision System
DH	Denavit-Hartenberg
DOF	Degree Of Freedom - Freiheitsgrad
EGT	Epipolar Geometry Toolbox
FOV	Field Of View - Bildwinkel
fps	frames per second - Bilder pro Sekunde
GB	GigaByte
GHz	GigaHerz
Gl.	Gleichung
GPU	Graphical Processing Unit - Prozessor der Grafikkarte
GUI	Graphical User Interface
HAW	Hochschule für Angewandte Wissenschaften
HID	Human Interface Device
HSR	Human Support Robot
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment - Entwicklungsumgebung
IPK	Institut für Produktionsanlagen und Konstruktionstechnik
MIT	Massachusetts Institute of Technology
OSG	OpenSceneGraph
RDS	Robotics Developer Studio
ROS	Robot Operating System
RRS	Realistic Robot Simulation
RTB	Robotics ToolBox
SIFT	Scale Invariant Feature Transform
sog.	so genannter
STEP	Standard for the Exchange of Product model data
STL	Surface Tesselation Language
SURF	Speeded Up Robust Feature
TCP	Tool Center Point

TCP/IP	Transfer Control Protocol / Internet Protocol
u.a.	unter anderem
UDP	User Datagram Protocol
USB	Universal Serial Bus
vgl.	vergleiche
VR	Virtual Reality
VRC	Virtual Robot Controller
VRML	Virtual Reality Modelling Language
VSE	Visual Simulation Environment
z.B.	zum Beispiel

Quellenverzeichnis

[AA04] Herb Sutter und Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 2004. ISBN: 978-0321113580.

[Ack12] Evan Ackermann. *Toyota's New Human Support Robot Gives Disabled Humans a Hand (and an Arm)*. Sep. 2012. URL: http://spectrum.ieee.org/automaton/robotics/home-robots/toyotas-new-human-support-robot-gives-disabled-humans-a-hand-and-an-arm.

[Ahn13] Song Ho Ahn. *OPENGL Tessellation*. Techn. Ber. Jan. 2013. URL: http://www.songho.ca/opengl/gl_tessellation.html.

[Ale07] Andrei Alexandrescu. *Modern C++ Design*. Addison Wesley, 2007. ISBN: 0-201-70431-5.

[AVT13] AVT. *Guppy F-080 Datasheet*. Jan. 2013. URL: http://www.alliedvisiontec.com/de/support/downloads/produktdokumentation/guppy.html.

[ble13] blender.org. *Blender Screenshot*. März 2013. URL: http://www.blender.org/typo3temp/pics/10301f96da.png.

[BM13] Rob Morefield und Brian Malloy. *3D Game Development Tutorials*. Jan. 2013. URL: http://people.cs.clemson.edu/~malloy/courses/3dgames-2007/tutor/web/collisions/collisions.html.

[Boo12] Boost. Okt. 2012. URL: http://www.boost.org/.

[Bro87] Rodney A. Brooks. *PLANNING IS JUST A WAY OF AVOIDING FIGURING OUT WHAT TO DO NEXT*. Techn. Ber. Massachusetts Institute of Technology (MIT), 1987. URL: http://people.csail.mit.edu/brooks/papers/Planning%20is%20Just.pdf.

[Cla13] Richard Clark. *JPEG*. Jan. 2013. URL: http://www.jpeg.org.

[CW02] Rolf Bernhardt und Gerhard Schreck und Cornelius Willnow. *THE VIRTUAL ROBOT CONTROLLER (VRC) INTERFACE*. Techn. Ber. Fraunhofer IPK, Berlin, 2002. URL: http://www.realistic-robot-simulation.org/7public.pdf.

[Fal12] Jason Falconer. *Toyota unveils helpful Human Support Robot*. Sep. 2012. URL: http://www.gizmag.com/toyota-human-support-robot/24246/.

[Fra12] Fraunhofer. *Experimental Evaluation of Advanced Sensor-Based Supervision and Work Cell Integration Strategies (EXECELL)*. 2012. URL: http://www.iff.fraunhofer.de/de/geschaeftsbereiche/robotersysteme/echord-execell.html.

[Fri11] Carsten Fries. "Kamerabasierte Identifizierung und Lokalisierung von Gegenständen für flexible Roboter". Magisterarb. HAW Hamburg, 2011.

[Gar13] Willow Garage. *PR2*. Jan. 2013. URL: http://www.willowgarage.com/pages/pr2/overview.

[gim13] gimp.org. *GIMP Screenshot*. März 2013. URL: http://www.gimp.org/screenshots/linux_mixer.jpg?rand=701607919.

[Har04] Arya Wirabhuana und Habibollah bin Haron. *INDUSTRIAL ROBOT SIMULATION SOFTWARE DEVELOPMENT USING VIRTUAL REALITY MODELING APPROACH (VRML) AND MATLAB – SIMULINK TOOLBOX*. Techn. Ber. University Teknologi Malaysia, 2004. URL: http://saintek.uin-suka.ac.id/file_ilmiah/Paper\%20RAPI\%20UMS\%20-\%20Habib,\%20Arya,.pdf.

[IPK12] Fraunhofer IPK. *Realistik Robot Simulation*. Techn. Ber. Okt. 2012. URL: http://www.ipk.fhg.de/geschaeftsfelder/automatisierungstechnik/prozessautomatisierung-und-robotik/forschung-und-entwicklung/rrs-realistic-robot-simulation.

[JV07] Erich Gamma und Richard Helm und Ralph Johnson und John Vlissides. *Design Patterns*. Addison Wesley, 2007. ISBN: 978-0-201-6331-0.

[KA12] Mark Segal und Kurt Akeley. *The OpenGL Graphics System: A Specification*. Aug. 2012. URL: http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf.

[Kha08] Siciliano und Khatib. *Handbook of Robotics*. Springer Verlag, 2008. ISBN: 978-3-540-23957-4.

[Kro08] Susanne Kroemker. *Computergraphik I*. Okt. 2008. URL: http://www.iwr.uni-heidelberg.de/groups/ngg/CG200708/Txt/ComputergraphikSkript_I.pdf.

[Lue01] David P. Luebke. *A Developer's Survey of Polygonal Simplification Algorithms*. Techn. Ber. University of Virginia, 2001. URL: http://webdocs.cs.ualberta.ca/~anup/Courses/604_3DTV/Presentation_files/Polygon_Simplification/luebke01developers.pdf.

[LVG13] Herbert Bay und Tinne Tuytelaars und Luc Van Gool. *SURF: Speeded Up Robust Features*. Techn. Ber. ETH Zurich, Katholieke Universiteit Leuven, Jan. 2013. URL: http://www.vision.ee.ethz.ch/~surf/eccv06.pdf.

[Mat12a] Mathworks. *Is it possible to start new threads from a C-MEX file?* 2012. URL: http://www.mathworks.com/support/solutions/en/data/1-V3B5T/?solution=1-V3B5T.

[Mat12b] Mathworks. *MATLAB CENTRAL - 3d graphics with s-function*. 2012. URL: http://www.mathworks.com/matlabcentral/answers/5445-3d-graphics-with-s-function.

[Mei08] Prof. Dr. Andreas Meisel. *3D Bildverarbeitung*. Techn. Ber. HAW Hamburg, 2008. URL: http://www.haw-hamburg.de.

[Mic12a] Microsoft. *MSDN Library*. März 2012. URL: http://msdn.microsoft.com/en-us/library/ff476342%28v=VS.85%29.aspx.

[Mic12b] Microsoft. *Robotics Developer Studio 4*. 2012. URL: http://www.microsoft.com/robotics/\#Product.

[Neu08] Neuronics. *Katana 450 Benutzerhandbuch*. 2008.

[OT04] K. Madsen und H.B. Nielsen und O. Tingleff. *METHODS FOR NON-LINEAR LEAST SQUARES PROBLEMS*. Techn. Ber. Technical University of Denmark, 2004.

[PEN13] PENTAX. *PENTAX*. Jan. 2013. URL: http://security-systems.pentax.de/en/product/C60402KP/body/specifications/ssd_products_image_processing.php.

[Rö08] Prof. Dr. Christof Röhrig. *Kinematik stationärer Roboter*. Wintersemester. 2008.

[Rö12] Dr. Thomas Röfer. *SimRobot*. Okt. 2012. URL: http://www.informatik.uni-bremen.de/simrobot/index_d.htm.

[Shr09] Dave Shreiner. *OpenGL Programming Guide (Seventh Edition)*. Addison Wesley, Juli 2009. ISBN: 978-0321552624.

[Som07] Ian Sommerville. *Software Engineering*. Addison Wesley, 2007. ISBN: 978-0-321-31379-9.

[Sta05] William Stallings. *Operating Systems*. Prentice Hall, 2005. ISBN: 0-13-127837-1.

[wei12] Ken Conley und weitere. *ROS / introduction*. Okt. 2012. URL: http://www.ros.org/wiki/ROS/Introduction.

[Wie04] Hendrik Wiebel. *Tesselierung von Oberflächen*. Techn. Ber. Universität Koblenz Landau, 2004. URL: http://www.uni-koblenz.de/~cg/Studienarbeiten/Tesselation.pdf.

[Wik12] Wikipedia. *Denavit-Hartenberg-Transformation*. Okt. 2012. URL: http://de.wikipedia.org/wiki/Denavit-Hartenberg-Transformation.

[Wik13a] Wikipedia. *JPEG*. März 2013. URL: http://en.wikipedia.org/wiki/JPEG.

[Wik13b] Wikipedia. *STL-Schnittstelle*. Jan. 2013. URL: http://de.wikipedia.org/wiki/STL-Format.

[Wü05] Prof. Dr. Klaus Wüst. *Robotik*. Sommersemester. 2005. URL: http://homepages.thm.de/~hg6458/Robotik.html.

[XQ10] Rui Wang und Xuelei Qian. *OpenSceneGraph 3.0*. PACKT Publishing, Dez. 2010. ISBN: 978-1849512824.

[Zha09] Prof. Dr. Jianwei Zhang. *Einführung in die Robotik*. Apr. 2009. URL: http://tams-www.informatik.uni-hamburg.de/lehre/2009ss/vorlesung/Introduction_to_robotics/folien/Vorlesung1_druck4to1.pdf.

[Chr03] Christiansen, T. und Torkington, N. *Perl Cookbook*. O'Reilly, 2003. ISBN: 978-0596003135.

[Pet11] Peter I. Corke. *Robotics, Vision & Control: Fundamental Algorithms in Matlab*. Springer, 2011. ISBN: 978-3-642-20143-1.

Alle Quellen mit angegebenem Hyperlink waren zum Zeitpunkt der Abgabe erreichbar und auf dem Stand, welcher in dieser Arbeit verwendet wurde.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. März 2013 Bernd Pohlmann