SimRobot

Ein Programm zur Simulation sensorbestückter Agenten
in einer dreidimensionalen Umwelt

Uwe Siems, Christoph Herwig, Thomas Röfer

ZKW Bericht Nr. 1/94

Zentrum für Kognitionswissenschaften
Universität Bremen

Herausgeber
Bernd Krieg-Brückner, Gerhard Roth, Helmut Schwegler

ISSN–noch nicht bekannt

Kontakt:
Christoph Herwig
Institut für Formale Methoden der Softwaretechnik
Zentrum für Kognitionswissenschaften
FB 3, Universität Bremen
Postfach 330 440
D–28334 Bremen, Germany
Tel.:+49-421-218-2676
Fax.:+49-421-218-3054
Telex: 245 811 uni D
e-mail: simrobot@informatik.uni-bremen.de

SimRobot

Ein Programm zur Simulation sensorbestückter Agenten
in einer dreidimensionalen Umwelt

Uwe Siems, Christoph Herwig, Thomas Röfer

ZKW Bericht Nr. 1/94

August 1994

Zusammenfassung

Die Forschung auf dem Gebiet autonomer Robotersteuerungen ist oft abhängig von teurer und nur eingeschränkt verfügbarer Roboter-Hardware. Aus diesem Grunde wurde die Simulationssoftware *SimRobot* entwickelt, die eine Alternative zu Experimenten in der realen Welt darstellen soll. Gleichzeitig soll sie den Einsatz der 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. Die Simulation wurde auf leichte Erweiterbarkeit ausgelegt: mit etwas Erfahrung in der Programmiersprache C++ ist es möglich, eigene Objekttypen zu definieren und der Simulationsumgebung hinzuzufügen. Dieser Bericht beschreibt die Möglichkeiten zur Beschreibung und Steuerung einer Simulation und dient als eine Art Handbuch für die Benutzung des Systems.

Zentrum für Kognitionswissenschaften
Universität Bremen

Inhaltsverzeichnis

1 Einführung

Eines der großen Probleme bei der Forschung bezüglich der (Verhaltens-)Steuerung von Robotern ist die Verfügbarkeit der Hardware —jedenfalls soweit es kleinere Forschungseinrichtungen und Universitäten mit begrenztem Etat betrifft. *SimRobot* wurde mit dem Ziel entwickelt, Forschungen im Bereich der experimentellen Robotik von teurer und eingeschränkt verfügbarer Hardware unabhängig zu machen. Die Software stellt dabei ein wichtiges Bindeglied zwischen theoretischer Planung und hardwaremäßiger Implementierung dar. Man umgeht dabei auch viele der typischen Schwierigkeiten bei der Arbeit mit realen Systemen, z.B. Wackelkontakte bei Steckverbindungen oder die „Nabelschnur" zur Verbindung des mobilen Roboters mit einem Steuerrechner.

Gleichzeitig soll mit der Simulation das Experimentieren mit verschiedenen Robotersystemen möglich sein. Diese Flexibilität ist sicher nicht gegeben, wenn man sich einmal für ein reales Robotersystem entschieden hat. Es lassen sich Gegebenheiten simulieren, die in der Realität nicht oder nur sehr schwer erreicht werden können. So läßt sich zum Beispiel ein fliegender Roboter simulieren, ohne daß man bei einer Fehlsteuerung Furcht vor Totalausfällen haben müßte. Es kann ein praktisch unendliches Kontingent von Sensoren verwendet werden. In begrenztem Umfang ist somit auch die Möglichkeit zur Simulation des Verhaltens natürlicher Systeme (des „Vorbilds") gegeben, selbstverständlich als entsprechende Abstraktion. Des weiteren soll möglichst vielen Studenten und Interessierten die Möglichkeit gegeben werden, mit dem Simulator zu experimentieren. Einen Engpaß bei der Verfügbarkeit gibt es nicht.

Aber es gibt auch gewichtige Nachteile, die gegen eine Simulation sprechen:

- Eine Simulation gründet auf ein Modell, das bestimmte Aspekte einer realen Umwelt abbildet – aber nicht alle. Die Ergebnisse einer Simulation können nicht identisch sein mit den Ergebnissen realer Versuche. In der Simulation ist man gezwungen, sich auf bestimmte Gegebenheiten der Umwelt zu beschränken. Die Komplexität von realen Umgebungen läßt sich nur bis zu einem bestimmten Grad in ein Programm übertragen.
- Der Rechenaufwand für eine Simulation wird umso höher, je mehr Aspekte der Umwelt berücksichtigt werden. Eine aufwendige Simulation wird daher jedes Experiment im Ablauf stark verzögern und kann auf den Benutzer so abschreckend wirken, daß er auf wichtige Versuche verzichtet.

Daher ist es wichtig, einen Kompromiß zwischen Naturähnlichkeit und Rechenaufwand zu finden. Natürlich kann es für bestimmte Problematiken auch wünschenswert sein, den Begrenzungen der natürlichen Welt zu entgehen, aber man muß dann bedenken, daß sich diese idealen Bedingungen in der realen Welt nicht (oder nur schwer) reproduzieren lassen.

Die Simulation *SimRobot* entstand 1993 im Rahmen einer Diplomarbeit [1] (auf der auch dieser Bericht basiert) im Umfeld des Graduiertenkollegs „Raumorientierung und Handlungsorganisation Autonomer Systeme" an der Universität Bremen als Nachfolger einer 2D Simulation, die im Laufe des studentischen Projekts „FORANN" (Formalisierung und Anwendung Neuronaler Netze) 1992 im Fachbereich Informatik das Licht der Computerwelt erblickte. Das Programmpaket wird im Rahmen des Graduiertenkollegs gewartet und weiterentwickelt. Die vorliegende Dokumentation beschreibt Version 1.0. Anfragen können an `simrobot@informatik.uni-bremen.de` gerichtet werden.

2 Konzept

2.1 Vergleichende Betrachtung von Robotersimulatoren

2.1.1 Cygnus — Eine Szenen-Beschreibungssprache

Cygnus ist Teil einer Diplomarbeit, die sich mit der Steuerung von Laufmaschinen beschäftigt [1]. Die Sprache Cygnus beschreibt die unbewegten Teile einer Szene, die einer simulierten Laufmaschine als Gelände dient. Es werden keinerlei Oberflächeninformationen definiert, allein die räumlich Anordnung der Flächen ist für den simulierten Laufroboter relevant. Grundelement der Sprache ist das Dreieck, das zu komplexeren Flächen zusammengesetzt und von dem andere Dreiecke geometrisch 'subtrahiert' werden können, um mit geringen Mitteln komplizierte Oberflächen zu erschaffen. Die Laufmaschine selbst ist ein immaterielles Objekt. Es ruht auf sechs Beinen, von denen immer genau drei Beine zur Berechnung einer Bewegung herangezogen werden. Die Lage der beteiligten Beine wird beim Laufen so angepaßt, daß immer eine zulässige Konstellation entsteht und vor allem der Torso eine der Laufsituation möglichst entsprechende Lage einnimmt. Die Unterschiede zwischen angeforderten und realisierten Beinstellungen werden dabei als Rückmeldung an den Benutzer zurückgeliefert. In dieser Simulation wird nur ein ganz bestimmtes Szenario simuliert, wodurch die Flexibilität an freier Definierbarkeit gering ausfällt und nur die feste Umgebung einer Laufmaschine betrifft.

2.1.2 Simderella — Simulation eines Robotarmes

Simderella ist ein Simulator, der an der Universität von Amsterdam entstanden ist [2]. Simuliert werden hier ein virtueller Roboterarm und zwei virtuelle Kameras. Der Roboterarm selbst ist in der Simulation körperlos und wird nur in der Anzeige als Drahtgittermodell dargestellt. Die Kameras arbeiten in der Weise, daß sie die Position eines (virtuellen) Objektes in räumlichen Koordinaten und in Relation zur Position des Endeffektors liefern. Die Simulation ist mit einem Modul zur Berechnung der inversen Kinematik ausgestattet, d.h. zu einer angegebenen Endposition werden die erforderlichen Bewegungsschritte ausgerechnet. Diese Simulation läßt sich nur in der Lage der Kameras und der Länge der Gelenkabstände variieren.

2.1.3 SimSalaBIM — Eine flache Welt für flache Käfer

Auf die Simulation *SimSalaBIM* aus dem Projekt FORANN möchten wir etwas näher eingehen, weil sie der Vorgänger dieser Simuation ist. SimSalaBIM, was für „Simulierter Salat beweglicher intelligenter Mobile" steht, entstand aus dem Bedürfnis heraus, fahrende, mit Sensoren ausgestattete Roboter mit wenig Aufwand zu realisieren. Da fahrende Roboter sich meist nur in einer Ebene bewegen, beschränkte man sich auf zwei-dimensionale „Welten", die hauptsächlich aus Kanten bestehen: Sie sind die Grundelemente eines jeden Objektes. Die Roboter können beliebige polygonale Umrißlinien haben und zwei fiktive Antriebsräder werden über zwei Werte in ihrer Drehrichtung und -geschwindigkeit gesteuert. Ein Bewegungsschritt wird dabei ausgeführt, wenn die neue Position ohne Überschneidung mit anderen Teilen der Welt eingenommen werden kann, ansonsten bleibt der Roboter an seiner alten Position. — Überschneidung bezieht sich dabei nur auf die Umrißlinien, ein Roboter kann sich innerhalb eines Objektes frei bewegen, solange sich die Kanten nicht überschneiden. Daher ist es sinnvoll, die Bewegungen so klein zu wählen, daß kein Objekt „übersprungen" werden kann. Die Sensorik umfaßte „Tasthaare", die als Berührungsund Entfernungssensor dienten, und „Rezeptoren“, die das Licht, das von sogenannten „Emittern“ ausging, empfingen. Die Emitter, die von Robotern getragen wurden, konnten an- und ausgeschaltet werden, so daß im Prinzip die Kommunikation von „Leuchtkäfern“ simuliert werden konnte. Zur Beschreibung der Umwelt wurde eine einfache Beschreibungssprache definiert [3].

2.1.4 Vergleich

Vergleichend kann man sagen, daß alle drei Simulationen stark auf bestimmte Versuchsanordnungen zugeschnitten und kaum für Erweiterungen vorgesehen sind. Die größte Breite in der Art der Versuchsanordnungen hat *SimSalaBIM* aufgrund der definierbaren Lichtquellen. Zu den großen Nachteilen von *SimSalaBIM* zählt seine Beschränkung auf die Ebene und die Unmöglichkeit, Roboterarme zu definieren.

2.2 Anforderungsdefinition

Die Defizite bei den oben vorgestellten Simulatoren sollten durch den Entwurf und die Implementierung von *SimRobot* behoben werden. Die Anforderungen an die Simulation gliedern sich in solche an die Art und Weise der Simulation und allgemeine softwaretechnische. Die Funktionsanforderungen haben sich aus eigenen Überlegungen und Erfahrungen im Umgang mit *SimSalaBIM*, aber auch aus der Befragung anderer potentieller Benutzer, vor allem im Kreis des Graduiertenkollegs, ergeben.

- Die Simulation sollte alle mögliche Arten von Robotern nachbilden. Dies sind insbesondere.
  - Fahrzeuge auf Rädern
  - Roboterarme
  - Laufmaschinen
- Diese sollten sich auf ebenen ebenso wie auf unebenen Flächen bewegen können.
- Es sollte eine möglichst große Auswahl an Sensoren in der Simulation zur Verfügung gestellt werden:
  - Entfernungssensoren
  - Lagesensoren
  - Helligkeitssensoren
  - Kameras, Facettenaugen
- Zu Helligkeitssensoren und Kameras gehört natürlich ein vollständiges Lichtmodell mit Lampen und Oberflächeneigenschaften.
- Ein Simulationsaufbau sollte mit Hilfe einer Beschreibungssprache oder interaktiv am Bildschirm frei entworfen werden können.
- Die Simulation sollte erweiterbar sein.
- Die Simulation sollte schnell sein (am besten in Echtzeit laufen).

- Es muß eine einfache Schnittstelle zu den Steuerprogrammen existieren, die die Roboter bewegen sollen.

Diese Liste ist jedoch nicht an Machbarkeit orientiert, sondern enthält (fast) alle Eigenschaften, die man von einer dreidimensionalen Simulation erwarten könnte. Zu welchen Entwurfsentscheidungen es dabei gekommen ist, soll im folgenden darstellt werden.

2.2.1 Allgemeines, internes Datenmodell

Das bestimmende und ausführende Objekt der Simulation ist die Szene. Eine Szene setzt sich aus Simulationsobjekten zusammen, die miteinander in Wechselwirkung treten (können). Ein Simulationsobjekt hat folgende 'physikalische' Eigenschaften:

- Es hat eine Position und Lage im Raum.
- Es hat eine räumliche Ausdehnung. Diese Ausdehnung definiert den 'Körper' des Objekts. Diese Eigenschaft ist nicht zwingend, aber jedes Objekt sollte in entsprechender Weise reagieren können.
- Zu einem Körper gehört eine Oberfläche. Diese Oberfläche hat eine Oberflächeneigenschaft, die angibt, wie stark es die Strahlung an einem bestimmten Punkt reflektiert.
- Bestimmte Objekte können Strahlung aussenden. Strahlung wird definiert als Strahlungsstärke und -richtung an einem beliebigen Punkt der Szene. Die Strahlung ist in verschiedene 'Sensorklassen' unterteilt. Die Effekte verschiedener Strahlungsquellen addieren sich.

Daneben hat ein Objekt noch logische oder strukturelle Eigenschaften:

- Objekte können über sogenannte 'Ports' angesteuert werden. Dies dient dazu, eine oder mehrere Eigenschaften des Objekts zu verändern. Dies kann die Position und Lage im Raum, die räumliche Ausdehnung, die Oberflächeneigenschaft oder die Strahlung eines Emitters betreffen.
- Objekte können 'Messungen' in der Szene anstellen, die ebenso über Ports gelesen werden können. Diese Messungen können das Objekt selbst betreffen und interne Zustände wiedergeben, oder einen anderen Teilaspekt der Szene betreffen und damit als eine Art externer Sensor wirken.
- Ein Objekt hat einen Namen, über den seine Ports angesprochen werden. Damit ist die Selektion eines Objekts von Adressen und Reihenfolgen unabhängig.
- Ein Objekt kann sich aus Unterobjekten zusammensetzen. Die Eigenschaften des Objekts ergeben sich dann als Summe aus den eigenen Eigenschaften und denen der Unterobjekte. Objekte können aus Oberobjekten entfernt und diesen zugefügt werden. Die Position und Lage der Unterobjekte wird relativ zu dem Oberobjekt definiert.

Dieses Schema ist hinreichend allgemein, um solche Objekte wie Fahrzeuge, Gelenke, Roboterarme, Strahlungsquellen, Kameras, Tastsensoren, usw. zu beschreiben. Es wurde eine Klassenhierarchie (siehe Anhang A) aufgebaut, in der jede Klasse für ganz bestimmte

Eigenschaften eines Objekts zuständig ist. Das gewünschte Objekt wird aus diesen Grundbausteinen zusammengesetzt. Ein Roboterarm z.B. wird aus einem Gelenk zusammengesetzt, das einen Körper (ein Gelenk an sich ist körperlos) und ein weiteres Gelenk enthält, das wiederum einen Körper und ein Gelenk enthält, usw. An diesem Arm können an beliebiger Stelle weitere Objekt (z.B. weitere Gelenke, Sensoren, Lichtquellen) angebracht werden.

2.2.2 Bewegungsmodell

Alle Aktionen in *SimRobot* sollen synchron ablaufen. Dies bedeutet, daß alle Aktorkommandos – die Steuerungsbefehle an die Objekte – erst gesammelt werden und dann in einem Schritt durchgeführt werden. Vor allem wird dieser Schritt erst auf Anforderung des Benutzers durchgeführt. Dies hat den Vorteil, daß man beliebig Zeit hat, Aktionen zu berechnen und diese von der Reihenfolge der Kommandos unabhängig sind. Es soll nur das jeweils letzte Kommando an einen Aktorport gelten, vorige Kommandos werden verworfen.

Die Hauptaktionsmöglichkeit in der Simulation ist die Bewegung. Die Simulation verfolgt ein sehr einfaches Modell, das von seinem Vorgänger *SimSalaBIM* übernommen wurde: Wenn an der neuen Position, die der nächste Bewegungsschritt vorsieht, eine Überschneidung des bewegten Objekts (oder besser von dessen Körper) mit den restlichen Körpern der Szene vorliegt, wird der Schritt nicht durchgeführt. Damit wird eine elementare Eigenschaft der realen Welt nachgebildet: Zwei Körper können sich nicht den gleichen Platz teilen.

Doch wie genau soll dies behandelt werden? Soll, wenn irgendwo eine Kollision vorliegt, gar keine Bewegung durchgeführt werden? Dies hätte den seltsamen Effekt, daß bei räumlich separaten Objekten ein Objekt durch die Kollision des anderen gestoppt würde. Auch wenn ein Roboterarm im Mittelteil gegen ein Hindernis stößt, könnte sich das äußere Ende normalerweise immer noch frei bewegen. Aus diesem Grund wird so verfahren, daß jedes Gelenk einzeln behandelt wird: Führt die Bewegung zur Kollision, wird die Bewegung nicht ausgeführt. Nach dem Schritt des Objektes werden unabhängig davon die Schritte der Unterobjekte der Reihe nach ausgeführt. Eine gegenseitige Hemmung findet nicht statt.

Damit dieses Verfahren nicht zu merklichen Abweichungen von der Realität führt, sollten Bewegungen immer nur im Vergleich zu den Abmessungen der Simulation kleine Entfernungen überbrücken.

2.2.3 Lichtmodell

Die Forderung nach Sensoren wie Kameras und Lichtrezeptoren kann nur mit einem Modell erfüllt werden, das die Ausbreitung und Reflexion von Licht nachbildet. Ein einfaches Raytracing-Modell erschien hier als ausreichend. Raytracing beruht auf dem Verfahren, Sehstrahlen vom fiktiven Auge des Betrachters durch die Bildebene auf die Szene zu schicken, und zu berechnen, welches Objekt an welcher Stelle zuerst von diesem Strahl getroffen wird. Dies ist einerseits einfach zu programmieren, zumal die erforderlichen Routinen schon für die Berechnung des Schnittes zwischen Körpern verwendet werden, andererseits ist es zwar langsamer als andere Verfahren, wird jedoch im allgemeinen nur mit geringen Auflösungen verwendet werden und ermöglicht zudem die einfache Berechnung von Schatten (indem Sehstrahlen von dem gesehenen Punkt zu allen Lichtquellen geschickt werden: Wird ein Körper vor der Lichtquelle vom Strahl getroffen, liegt ein Schatten vor).

Für die Lichtreflexion an Oberflächen wurde ein einfaches Beobachterstandpunktinvariantes Modell gewählt, das als Lambert'sche Reflexion bekannt ist [4].

2.2.4 Schnittstelle

Da die Simulation hauptsächlich von anderen Programmen gesteuert werden soll, ist es wichtig, eine Schnittstelle zu beschreiben, die einen einfachen und flexiblen Zugriff auf (fast) alle Funktionen der Simulation zuläßt. Wichtig sind die Funktionen, die für eine Versuchsdurchführung wichtig sind. Insbesondere sollten folgende Aktionen möglich sein:

- Steuerung aller Aktoren (Gelenke, Fahrzeuge, Emitter)
- Lesen aller Sensoren
- Entfernen und Zufügen vordefinierter Teilobjekte sollte ermöglicht werden, um den Versuchsaufbau während eines Programmlaufs in Grenzen modifizieren zu können.
- Positionsbestimmung und Positionierung existierender Objekte sollte aus dem gleichen Grund möglich sein.
- Eine hardware-unabhängige Ausgabe der Szene an einen abstrakten Graphikport sollte dem Benutzer zur Verfügung gestellt werden, damit dieser die Szene von seinem Programm darstellen lassen kann.

Andere Funktionen wie das Entwerfen neuer Objekte sind dagegen für ein Steuerprogramm nicht so wichtig, und brauchen deshalb in diese Schnittstelle nicht aufgenommen werden.

2.2.5 Beschreibungssprache

Dem Benutzer sollte natürlich eine Möglichkeit gegeben werden, seine Szenarien möglichst einfach zu definieren. Dieses geschieht am einfachsten interaktiv am Bildschirm. Diese Möglichkeit ist jedoch teilweise an bestimmte technische Voraussetzungen (Graphikfähigkeiten, Interaktionsmöglichkeiten, Betriebssystem) gebunden. Zudem ist die Implementierung eines *bequemen*, interaktiven graphischen Editors keine triviale Aufgabe, zumal die Klassenhierarchie der Simulationsobjekte erweiterbar bleiben sollte. Ein Editor müßte also die Objekte in einer allgemeinen, erweiterbaren Weise behandeln. Um diesen Beschränkungen zu entgehen, wurde eine Beschreibungssprache entworfen, die den besonderen Eigenschaften des Simulators gerecht wurde. Insbesondere mußte die Sprache dem hierarchischen Datenmodell, dem objektorientierten Entwurf und der gewünschten Erweiterbarkeit des Systems Rechnung tragen. Diese Sprache wird in Kapitel 3.2 besprochen.

2.2.6 Wahl der Programmiersprache

Die Wahl der Programmiersprache wurde von zwei wesentlichen Anforderungen beeinflußt: Die verwendbaren Simulationselemente wie Sensoren, Körper, Gelenke, usw. sollten in einer objektorientierten Vererbungshierarchie untergebracht werden. Und diese Elemente sollten von einem halbwegs erfahrenen Programmierer einfach um eigene Kreationen ergänzt werden können. Insbesondere erforderte dies, daß eine relativ weit verbreitete Programmiersprache zum Einsatz kam. Eine wenig bekannte Sprache wäre ein Hindernis für die Benutzung durch andere, die eigene Objekte entwerfen wollen. Die Sprache C++ kommt dieser Anforderung am ehesten nach, und so wurde sie für die Implementation gewählt, obwohl sie auch starke Schwächen aufweist (Pointerkonzept, separate Header-Files, Komplexität der Sprache, Gefahr der stark verkürzenden Schreibweise von Ausdrücken).

2.2.7 Graphische Oberfläche

Für Versuche bezüglich Steuerung von Robotern und „Agenten" ist es unverzichtbar, einen Überblick über die Geschehnisse in der simulierten Umwelt zu haben – schon allein, um den Erfolg einer Aktion besser abschätzen zu können. Man möchte die Szene vor Augen haben, ausgewählte Sensoren kontrollieren und eventuell korrigierend in die Steuerung eingreifen können. Wie schon im Absatz „Beschreibungssprache" dargestellt, ist aber eine interaktive, graphische Schnittstelle stark von den technischen Gegebenheiten abhängig. Eine Plattform, die am ehesten im technisch-wissenschaftlichen Bereich anzutreffen sein sollte und genügend Rechenleistung zur Verfügung stellt, ist die der Rechner mit UNIX-Betriebssystem und einer X-Windows Oberfläche für die graphische Ausgabe am Bildschirm.

Die graphische Oberfläche soll die aktuelle Szene und Teile davon darstellen können, wobei Blickwinkel und Vergrößerung vom Benutzer wählbar sein sollte. Die verfügbaren Sensorwerte sollen auf Wunsch in einer angemessenen Weise dargestellt werden können. Am schnellsten können dabei graphische Repräsentationen wahrgenommen werden. Da eine Szene sehr viele Sensoren enthalten kann, ist es auch entscheidend, diese platzsparend darzustellen und einen einfachen Zugriff auf gerade nicht dargestellte Sensoren zu haben. Ähnliches gilt für die Aktoren: Es soll bequem in die Steuerung eingegriffen werden können, wobei die anzusteuernden Elemente leicht anzuwählen sind und platzsparend untergebracht werden.

Die graphische Darstellung soll eine automatische Leistung des Programms sein und keinerlei Vorbereitung durch das Steuerprogramm erfordern (wenn man einmal vom Vorhandensein von X-Windows absieht).

2.3 Überblick

SimRobot wurde für den Einsatz unter *Unix*-Betriebssystemen konzipiert, was sich vor allem in der Art der Benutzung von Dateien und der Kontrolle von Speicheranforderungen niederschlägt. Die Simulation *SimRobot* existiert einmal als integrierte Version als eine Reihe von Module, die zu einem Steuerprogramm gebunden werden können. Diese Version hat keine graphische Oberfläche und kommuniziert (bis auf Fehlermeldungen) nur über das ausführende Programm mit dem Benutzer.

Die stand-alone Version setzt zusätzlich das Vorhandensein der Graphiksysteme *XWindows* mitsamt seiner Bibliotheken und des Toolkits *XView* zur Erstellung graphikorientierter Programme voraus. Diese Version ermöglicht dem Benutzer verschiedene Sichten auf die Szene und die interaktive Kontrolle der Sensoren bzw. die Steuerung der Aktoren. Solche Beeinflussungsmöglichkeiten sind ein wichtiges Hilfsmittel zur Kontrolle der laufenden Szenarien. Vor allem die Darstellung der Sensorwerte ist für die Bewertung eines Versuchsverlaufes von großer Bedeutung.

Zur Steuerung der Simulation durch ein anderes Steuerprogramm (Controller) werden in dieser Version die unter *Unix* bekannten Pipes verwendet[1]. Aufgrund dieser Einschränkungen kann es nur unter *Unix* unverändert verwendet werden. Zusätzlich können

[1] Simulation und Controller kommunizieren damit als selbständige Prozesse.

unter anderen Versionen von Betriebssystem und Toolkit geringfügige Anpassungen des Sourcetextes nötig werden.

Die Beeinflussung der Szene und die Überprüfung ihres Zustandes findet für ein Steuerprogramm – genau so wie auch in der Realität – im allgemeinen nur über die definierten Aktoren und Sensoren statt. Schließlich hat auch in wirklichen Anwendungen ein Steuerprogramm keine Möglichkeit, Werkstücke zu manipulieren, wenn nicht durch die Bewegungen eines Robotarmes oder ähnlichem. Genauso kann es die Position eines Werkstückes nur über angeschlossene Sensoren wie z.B. Kameras wahrnehmen (Dies gilt natürlich nicht, wenn ein Mensch der Software durch eigene Aktionen unter die Arme greift).

Die in der Realität abgesendeten Aktorkommmandos und eingehenden Sensorsignale sind nur Spannungen in elektrischen Leitern (bei künstlichen Systemen meist Drähte, in biologischen Systemen Nervenbahnen), die keinen Rückschluß auf die ausgelösten Aktionen und wahrgenommenen Phänomene erlauben. Die Signale erhalten erst durch das verarbeitende Gehirn einen Sinn. Die Semantik eines Sensorsignals genauso wie die eines Aktorsignals sollte daher hauptsächlich im Steuerprogramm enthalten sein oder von diesem festgelegt werden. Aus diesem Grund ist keine direkte Möglichkeit vorgesehen, z.B. den Typ eines Sensors festzustellen.

Jedoch kann man bei biologischen Systemen manchmal den Zusammenhang von verschiedenen Sensorsignalen aus der räumlichen Anordnung der Nervenbahnen herleiten: Nervenverbindungen zwischen den Augen und dem Gehirn von höheren Tieren liegen so, daß benachbarte Punkte auf dem rezeptiven Feld des Auges mit benachbarten Punkten im zugehörigen Gehirnareal verbunden sind, und die Informationen dementsprechend korreliert und verarbeitet werden können. Um diese Information nachzubilden, wäre es z.B. möglich, eine Funktion zur Verfügung zu stellen, die einen Wert liefert, der angibt, wie stark zwei Signalleitungen miteinander korreliert sind. Korrelation wird dabei als eine Abstraktion vom Begriff der räumlichen Nähe benutzt, die ja nicht unbedingt gegeben sein muß. Diese Art der Informationsgewinnung wäre aber sehr mühsam und aufwendig und entspräche auch nicht den Gegebenheiten einer vom Charakter her sehr geometrischen Simulation: Bei einem organischen Auge können sich die einzelnen lichtempfindlichen Elemente sehr frei über eine Fläche verteilen, wodurch eine Positionsangabe relativ schwierig ist. Bei einer Kamera dagegen unterteilen sich die Felder klar in Zeilen und Spalten und lassen sich relativ einfach über Indizes ansprechen. Dieser Sensor hat also eine 'Breite' und eine 'Höhe'. Andere Sensoren können eine Ausdehnung nur in einer Richtung haben, wie z.B. beim Seitenlinienorgan des Fisches: Es besteht aus einer Reihe von Drucksensoren, die an beiden Seiten des Fischrumpfes entlanglaufen.

Als Verallgemeinerung bietet sich für die Simulation die Aussage an, daß ein Sensor beliebig viele Dimensionen haben kann – obwohl für menschliche Begriffe eine Ausdehnung in mehr als zwei Dimensionen ungewöhnlich, in mehr als drei Dimensionen kaum vorstellbar erscheint. Schließlich bietet die Simulation selbst nur drei Dimensionen in Breite, Höhe und Länge an. In jeder Dimensionen hat ein Sensor eine Ausdehnung, die die Anzahl der Sensoren in dieser Richtung angibt.

3 Szenenbeschreibung

3.1 Eigenschaften der Objekte

Die Modellierung eines Versuchsaufbaus in der Simulation, hier in Anlehnung an den Begriff aus der Bildverarbeitung auch 'Szene' genannt, besteht aus der Definition von mehreren Objekten, die in der Szene positioniert werden. Diese Objekte können sich wiederum aus Unterobjekten zusammensetzen. Zu den wichtigsten Eigenschaften eines Objektes gehören:

- Jedes Objekt hat eine Lage und Position in der Szene.
- Jedes Objekt kann körperlich sein. Es hat dann eine Oberfläche, die Licht reflektiert und eventuell Schatten wirft.
- Jedes Objekt hat einen Namen, über den es identifiziert und angesprochen wird.
- Ein Objekt kann bewegt werden. Unterobjekte bewegen sich synchron mit.
- Ein Objekt kann dynamische Elemente enthalten (hier Aktoren genannt), die vom Benutzer angesteuert werden können.
- Ebenso kann ein Objekt Sensoren enthalten, die Aufschluß über die ausgeführte Bewegung geben oder auf äußere Reize reagieren. Diese können abgefragt werden.
- Zusätzlich sollte sich jedes Objekt 'angemessen' auf einem virtuellen Zeichengerät darstellen können.

3.2 Die Definitionssprache

Szenen für die Benutzung durch *SimRobot* werden in separaten Definitionsdateien (üblicherweise **.scn*) in einer speziellen Beschreibungssprache beschrieben. Abb. 1 zeigt ein Beispiel. Diese Beschreibungssprache ist von der Struktur her recht einfach und hauptsächlich auf Erweiterbarkeit ausgelegt. Auf einen interaktiven Editor zur Erstellung der Szenen wurde bisher wegen des relativ großen Entwurfs- und Programmieraufwandes verzichtet.

Die Sprache beschränkt sich darauf, Objekte anhand ihrer Attribute und ihrer Unterobjekte zu beschreiben. Eine Definitionsdatei besteht lediglich aus der Definition des Objekts `WORLD` und einem Punkt. Eine Objektdefinition besteht aus:

- einem Schlüsselwort, das den Objekttyp bestimmt,
- einer Zeichenkette, die den Namen des Objekts enthält,
- wahlweise einer Liste von Attributen, die in runde Klammern eingeschlossen ist,
- wahlweise einer Liste von Unterobjekten, die in eckige Klammern eingeschlossen ist. Dies ist nur möglich, wenn das Objekt weitere Objekte enthalten kann.

Das Objekt `WORLD` nimmt dabei eine Sonderrolle ein, da es nur einmal definiert werden kann und Definitionen enthält, die sich auf andere Objekte auswirken.

3.3 Liste der Attribute

Im Nachfolgenden werden alle Objekte, Attribute und Ports aufgeführt, die für eine Szene und die Objekte definiert sind. Den Anfang machen die allgemein verwendbaren Attribute.

```
WORLD "test" (

  SURFACE "Standard" (1.0, 0.0, 1.0, 0.5, 0.5);
)
[
  POLYEDER "border" (
    BASEPOINTS (
        (-400, -400, 0),  (-400, 400, 0),
        (400, 400, 0),    (400, -400, 0)
    );
    SWEEP (0, 0, 100);
    SURFACE "Standard";
  );

  POLYEDER "column" (
    BASEPOINTS (
        (50, 0, 0), (50, 0, 260)
    );
    ROTATE 10;
    SURFACE "Standard";
    MOVETO (-140, 220, 0);
  );
  TURNABLE "gelenk1" (
    Z_AXIS;
    RANGE (40,180);
    START 90;
    FACTOR 20;
    MOVETO (0, 20, 1.5);
  )
  [
    POLYEDER "glkk" (
      BASEPOINTS (
          (250, 10, 0),   (250, -10, 0),
          (-30, -30, 0),  (-30, 30, 0)
      );
      SWEEP (0, 0, 60);
      SURFACE "Standard";
    )
  ]
].
```

Abbildung 1: Eine Definitionsdatei mit einer einfachen Szenendefinition

3.3.1 Allgemein verwendbare Attribute

Diese Attribute können auf jedes Objekt angewendet werden.

Attribute:

TURNX *Real*

Dreht das Objekt um die x-Achse (in Grad).

TURNY *Real*

Dreht das Objekt um die y-Achse (in Grad).

TURNZ *Real*

Dreht das Objekt um die z-Achse (in Grad).

SCALE *Real*

Skaliert ein Objekt und dessen Unterobjekte um den angegebenen Faktor. Ist der Faktor kleiner als 1, wird das Objekt verkleinert, ist er größer als 1, wird es vergrößert. Verwendungsbereich: Innerhalb aller Objekte. Es wird dabei je nach Objekt die Darstellung, die physikalische Größe und ggf. die Translationsgeschwingikeit angepaßt.

MOVETO *(Real,Real,Real)*

Bewegt das Objekt an die angegebenen Koordinaten.

REMARK *String*

Legt einen beliebigen String als Bemerkung fest. Dieser kann von dem Steuerprogramm mit `ObjectRemark` gelesen werden und so zur Konfiguration desselben durch die Szenenbeschreibung dienen.

COLOR *String bzw. (Real,Real,Real)*

Definiert die Farbe, in dem das Objekt angezeigt werden soll.

DETAILLEVEL bzw. DL *Integer*

Der *Detaillevel* ist ein Auswahlkriterium, um die Darstellung bestimmter Objekte der Szene einzuschränken und somit die Übersichtlichkeit zu erhöhen: Bei der Ausgaberoutine kann ein Zahlenwert angegeben werden, der mit dem *Detaillevel* eines Objektes vor dessen Ausgabe verglichen wird. Ist der *Detaillevel* höher, wird das Objekt nicht ausgegeben. Objekte mit höherem *Detaillevel* werden also in der Darstellung einer Szene eher unterdrückt.

3.3.2 Liste aller definierten Objektklassen

Hier werden die Objektklassen und die Attribute, die von den Objekten 'verstanden' werden, sowie die für das jeweilige Objekt definierten Ports aufgeführt. Ports sind mit *(A)* für Aktorport oder *(S)* für Sensorport markiert. Den Anfang der Liste machen die Attribute, die für jede Objektklasse gültig sind.

WORLD

Attribute:

AMBIENT *([Real], ...)*

Legt die ambiente 'Lichtstärke' für die verschiedenen Sensorklassen fest. Ambientes Licht ist das Licht, das gleichmäßig aus allen Richtungen kommt.

BACKGROUND *([Real], ...)*

Legt die 'Hintergrund-Lichtstärke' fest. Sozusagen die Farbe des fernen Horizonts.

SHADOWS

Berechnet den Schattenwurf von Körpern bezüglich der Emitter.

QUICKSHADING

Bei Berechnung von Kamerabildern mit dem Tiefenpufferverfahren werden keine Helligkeitsverläufe beachtet, sondern ein Mittelwert für die Helligkeit gebildet (ist schneller!).

STANDARDCOLOR *String bzw. (Real,Real,Real)*

Legt die Farbe fest, in der Objekte in der Szenendarstellung gezeigt werden sollen, die durch keines der anderen folgenden Attribute abgedeckt sind. Die Farbe kann auch noch individuell für einzelne Objekte gesetzt werden. Die Festlegung der Farbe erfolgt durch einen Namen (welcher von dem darstellenden Gerät unterstützt werden muß) oder durch einen RGB-Wert (Zahlentripel mit Werten zwischen 0 und 1).

BODYCOLOR *dito*

Legt die Standardfarbe der Körper in der Szenendarstellung fest.

EMITTERCOLOR *dito*

Farbe für Emitter.

SENSORCOLOR *dito*

Farbe für Sensoren.

WHISKERCOLOR *dito*

Farbe für Tasthaare.

TEXTURE *String LOAD String*

Liest eine Textur ein. Der erste String gibt den Namen an, unter dem die Textur ab sofort angesprochen werden kann, der zweite Name gibt den Dateinamen der Textur an. Entweder liegt die Textur schon umgewandelt vor, oder sie wird während des Einlesens konvertiert. Dann muß jedoch das Programm *TextureConvert* im aktuellen Pfad zu finden sein.

SURFACE *String ([Real], ...)*

Definiert eine Oberfläche. Eine Oberflächendefinition besteht aus dem Namen der Oberfläche und den Reflexionsgraden in den einzelnen Sensorklassen. Deren Werte liegen (normalerweise) zwischen 0 und 1. In eckigen Klammern können weitere Parameter zur genaueren Definition der Oberfläche folgen:

Attribute:

TEXTURE *String ([(Integer, Real, Real)], ...)*
Definiert eine Textur für die Oberfläche. Für jede Sensorklasse kann ein Tripel angegeben werden. Der erste Wert ist eine ganze Zahl zwischen 0 und 3 und gibt an, welche Grundfarbe des eingelesenen Bildes in der aktuellen Sensorklasse verwendet werden soll (0 = Helligkeit, 1 = Rot, 2 = Grün, 3 = Blau). Der zweite Wert gibt den dunkelsten Wert und der dritte Wert den hellsten Wert für die einzublendende Textur an. Wird das Zahlentripel an einer Stelle ausgelassen, so wird in dieser Sensorklasse der vorher angegebene Helligkeitswert und nicht die Textur verwendet.

SIZE *(Real, Real)*
Gibt die absolute Größe an, in der die Textur erscheinen soll.

RATIO *(Real, Real)*
Gibt das Seitenverhältnis an, in dem die Textur erscheinen soll. Die Textur wird für jede Fläche so gestreckt, daß die Fläche auf jeden Fall ganz bedeckt wird.

WRAPAROUND
Gibt an, daß die Textur zu allen Seiten hin wiederholt wird, wenn sie zu klein für die Fläche ist.

Ein fiktives Beispiel für eine Holzoberfläche:

```
TEXTURE "maserung" LOAD "holz.gif";
SURFACE "s-holz" (0.6, 0.4, 0.0, 0.6, 0.7) [
  TEXTURE "maserung" ((1, 0.0, 1.0),
                      (2, 0.0, 1.0),
                      (3, 0.0, 1.0),,);
  SIZE (10, 10);
  WRAPAROUND;
]
```

MACRO *Objektdefinition*

Dieses Attribut dient dem Zweck, ein einmal definiertes Objekt mehrmals in der Szene verwenden zu können. Das Makro kann anstelle einer Objektdefinition der Form

Typname "objektname" ...

verwendet werden und hat dann entweder die Form

`MACRO` *"makroname"* ...

oder

`MACRO` *"makroname"* `AS` *"objektname"* ...,

wobei im ersten Fall der Makroname auch den Objektnamen bildet.

PATH *String = String*

Setzt eine Abkürzung des Pfades zu einem Port oder Unterobjekt, vergleichbar den symbolischen Links von *Unix*.

Beispiel:

```
WORLD "Beispiel"
(
  AMBIENT (0.1); // Ambiente Helligkeit
  BACKGROUND (0.5); // Hintergrundhelligkeit
)
[
  POLYEDER "Flaeche"
  (
    BASEPOINTS ((0,0,0),(0,100,0),(100,100,0),(100,0,0));
    FACES ((0,1,2,3))
  )
].
```

GROUP

Eine `GROUP` ist ein Objekt, das sich aus beliebig vielen Unterobjekten zusammensetzt. Die Unterobjekte sind in einer einfach ring-verketteten Liste organisiert. Zusätzlich enthält ein `GROUP`-Objekt noch eine Liste mit 'symbolischen Links', die vom Benutzer definiert werden können, und zur Übersetzung von Pfadangaben herangezogen werden. Die meisten Methodenaufrufe werden von dieser Klasse rekursiv an ihre Unterobjekte weiterverteilt, da sie eigentlich nur die Eigenschaften ihrer Unterobjekte realisiert.

Attribute:

PATH *String = String*

Setzt eine Abkürzung des Pfades zu einem Port oder Unterobjekt, vergleichbar den symbolischen Links von *Unix*.

Beispiel:

```
MACRO
  GROUP "Makro1"
  [
    POLYEDER "Polyeder1" (...);
    POLYEDER "Polyeder2" (...);
  ];
```

TURNABLE/EXTENDIBLE

Dreh-/Teleskopgelenk, das sich um/entlang eine(r) von drei Achsen bewegen kann.

Attribute:

X_AXIS/Y_AXIS/Z_AXIS

Legt fest, entlang welcher oder um welche Achse die Bewegung erfolgt.

FACTOR *Real*

Empfindlichkeit der Steuerung eines Gelenks. Gibt den Wert an, mit dem ein Steuerbefehl multipliziert wird.

START *Real*

Startwert. Bei Drehgelenken Grad, bei Teleskopgelenken die Entfernung vom Nullpunkt.

RANGE *(Real,Real)*

Grenzen der Bewegung. Beim Drehgelenk Gradzahlen, beim Teleskopgelenk Entfernungen vom Nullpunkt. Der kleinere Wert muß zuerst angegeben werden.

PATH *String = String*

Setzt eine Abkürzung des Pfades zu einem Port oder Unterobjekt, vergleichbar den symbolischen Links von *Unix*.

Ports:

MOVE (A)

Drehung/Translation um angegebenen Anteil.

EXECUTED (S)

Wieviel des bei MOVE angegebenen Kommandos wurde ausgeführt.

WHERE (S)

Wie ist die Lage des Gelenks (relativ zu den angegebenen Grenzen)? Hat das Gelenk keine Grenzen, ist der Wert 0.

Beispiel:

```
TURNABLE "Gelenk1"
(
  X_AXIS; // Dreht sich um die x-Achse
  FACTOR 10; // Bei MOVE 1 dreht es sich um 10 Grad
  RANGE (0,180); // Bewegung von 0 bis 180 Grad
  START 90; // Beginne mit 90 Grad
  MOVETO (100,100,100); // Anfangsposition
);
```

VEHICLE

Objekt, das sich im Gegensatz zum `MOVEABLE` ohne Schranken frei im Raum bewegen kann. Ein `Vehicle` kann sich in sechs Freiheitsgraden bewegen, d.h. translatorisch entlang und rotatorisch um seine x-, y- und z-Achse. Wie das `Moveable` ist es körperlos und unsichtbar und benutzt das gleiche Bewegungsmodell, so daß die gleichen Aussagen auch hier gelten. Es besitzt sechs Ports zur Ansteuerung, aber nur einen Port zur Feststellung, ob die Bewegung ausgeführt wurde. Eine Lagefeststellung über Ports ist nicht vorgesehen.

Attribute:

RESTR_TRANS_X / RESTR_TRANS_Y / RESTR_TRANS_Z *(Real,Real,Real)*

Wie stark bewegt sich das Fahrzeug in Richtung seiner jeweiligen-Achse. Angegeben werden die Werte für Aktorkommandos -1, 0 und +1. Zwischenwerte werden linear interpoliert.

RESTR_ROT_X / RESTR_ROT_Y / RESTR_ROT_Z *(Real,Real,Real)*

Wie stark dreht sich das Fahrzeug um seine jeweiligen-Achse. Angegeben werden die Werte in Grad für Aktorkommandos -1, 0 und +1. Zwischenwerte werden linear interpoliert.

PATH *String = String*

Setzt eine Abkürzung des Pfades zu einem Port oder Unterobjekt, vergleichbar den symbolischen Links von *Unix*.

Ports:

TRANS_X / TRANS_Y / TRANS_Z (A)

Translatorische Bewegung in Richtung der jeweiligen-Achse ausführen.

ROT_X / ROT_Y / ROT_Z (A)

Rotatorische Bewegung in Richtung der jeweiligen-Achse ausführen.

EXECUTED (S)

Gibt an, ob die angeforderte Bewegung ausgeführt wurde. 1 steht für die erfolgreiche Ausführung.

Beispiel:

```
Vehicle "Auto"
(
  RESTR_TRANS_X (-10,0,150); // Bei MOVE 1 Temp 150, bei MOVE -1 Tempo -10
  RESTR_ROT_Z (-10,0,10); // Bei MOVE 1 10  Grad nach rechts
  MOVETO (100,100,100); // Anfangsposition
  TURNZ 45; // Anfangsrichtung
);
```

RADIALEMITTER

Ein Emitter, der in alle Richtungen gleichmäßig strahlt und dessen 'Leuchtstärke' mit dem Quadrat der Entfernung abnimmt. Schattenwurf durch Objekte wird bei Bedarf berücksichtigt.

Attribute:

RADIATION *([Real],...)*

Relative Lichtstärke in den einzelnen Sensorklassen.

FACTOR *Real*

Der Faktor, um den die relative Lichtstärke des Emitters verstärkt wird.

START *Real*

Die relative Anfangs-Lichtstärke, die durch Aktorkommandos zwischen 0 und 1 verändert werden kann.

Ports:

EMIT (A)

Verändern der relativen Gesamtlichtstärke. Nur positive Werte möglich.

Beispiel:

```
RADIALEMITTER "Lampe"
(
  FACTOR 10; // START x RADIATION wird um Faktor 10 verstaerkt
  RADIATION (0,1,0); // Strahlt nur in zweiter Klasse
  START 1; // Volle Kraft
);
```

SPOTEMITTER

Ein Emitter mit einer Vorzugsrichtung. Bei diesem Emitter kann man einen maximalen Strahlungskegelwinkel und einen Bündelungsfaktor angeben. Auch bei diesem Emitter nimmt die Strahlungsstärke mit dem Quadrat der Entfernung ab. Attribute und Ports wie `RADIALEMITTER`. Zusätzlich:

Attribute:

ANGLE *Real*

Halber Öffnungswinkel des Strahlungskegels. Die Strahlungsstärke nimmt zum Rand hin ab.

EXPONENT *Real*

Bestimmt die Abnahme der Lichtstärke zum Rand hin. Höherer Exponent bedeutet stärkere Konzentration zur Mitte hin.

Beispiel:

```
SPOTEMITTER "Strahler"
(
  RADIATION (0,1); // Strahlt nur in zweiter Klasse
  ANGLE 45; // 90 Grad Oeffnungswinkel
);
```

POLYEDER

Ein `POLYEDER` ist ein Körper, der sich aus polygonalen Flächen zusammensetzt. Diese Flächen müssen nicht geschlossen sein, da diese Klasse definitionsgemäß 'hohle' Körper darstellt: Wenn sich ein Polyeder vollständig in einem anderen Polyeder befindet, kann keine Überlappung festgestellt werden, da nur der Schnitt der Außenflächen überprüft wird. Intern werden Polyeder durch Eckpunkte und Kantenzüge repräsentiert, die die Eckpunkte verbinden und damit die Flächen bilden. Die Methode `MakeStep` speichert die alten Eckpunkte übrigens in einem eigenen Array, damit sie bei Nichtgelingen einer Bewegung mit `StepBack` ohne zusätzlichen Rechenaufwand wieder hergestellt werden können. Es gibt drei Methoden, um mit dieser Klasse einen Polyeder zu definieren:

- Man kann alle Eckpunkte und Kantenzüge einzeln angeben.
- Man gibt eine Reihe von Eckpunkten an, die als Kantenzug einer einzelnen Fläche interpretiert werden; diese Fläche wird um einen Vektor im Raum verschoben und bildet so ein Prisma, einen sogenannten 'Sweep'-Körper.
- Man gibt eine Reihe von Eckpunkten an, die als Kantenzug interpretiert werden, der mit wählbarer Auflösung um die z-Achse rotiert wird. Man erhält einen angenäherten Rotationskörper, der bei Bedarf oben und unten geschlossen oder zum Torus vervollständigt wird.

Attribute:

SURFACE *String*

Bestimmt die Oberflächeneigenschaft. Der Oberflächenname muß vorher vereinbart werden.

TRANSPARENT

Der Körper wird in einer Szenendarstellung nur mit den Umrißlinien dargestellt (und verdeckt damit also keine anderen Objekte).

BASEPOINTS *((Real,Real,Real),...)*

Bestimmt die Eckpunkte des Polyeders als Liste von Vektoren.

FACES *((Integer,...[; String [, Real]]),...)*

Bestimmt die Flächen des Körpers als Liste von Eckpunktlisten (die Eckpunkte werden als Index in die Liste der Eckpunkte – BASEPOINTS – angegeben). Start der Indizes ist wie immer bei Null. Wird am Ende der Liste noch ein String nach einem Semikolon angegeben, wird die Oberflächendefinition dieses Namens für diese Fläche benutzt. Eine evtl. Textur wird an der ersten Kante in der Kantenliste ausgerichtet. Deshalb kann nach dem Namen noch ein Drehwinkel in Grad angegeben werden, um den diese Textur zu dieser ersten Kante gedreht sein soll.

SWEEP *(Real,Real,Real)*

Erzeugt einen Sweep-Körper, indem die Eckpunkte - die BASEPOINTS - als Ecken einer Fläche angesehen werden, und diese Fläche um den angegebenen Vektor verschoben wird.

ROTATE *Integer*

Erzeugt einen Rotationskörper, indem die Eckpunkte – die BASEPOINTS – als der Kantenzug einer Seite eines Rotationskörpers angesehen und diese Punkte um die z-Achse gedreht werden. Der Integer gibt die Zahl der Rotationsschritte für eine Umdrehung an.

FACES, SWEEP und ROTATE schließen sich gegenseitig aus und können auch nur einmal angegeben werden.

Beispiel:

```
POLYEDER "Zylinder"
(
  DL 0; // Immer zeichnen
  MOVETO (-140,220,2); // Position
  SURFACE "Standard"; // Oberflaeche
  BASEPOINTS ((0,0,0),(100,0,0),(100,100,0),(0,100,0),
              (0,100,100),(100,100,100));
  FACES ((0,1,2,3),(2,3,4,5),(4,5,0,1));
        // Punkte, aus denen Flaechen werden
);
```

WHISKER

Ein Abstandssensor, der auf Körper reagiert. Modelliert ein 'Tasthaar', das um so stärker reagiert, je mehr es von einem Körper 'eingeklemmt' wird. (Realisiert durch eine Schnittberechnung zwischen einer Strecke und den Körpern der Szene.) Das Resultat liegt im Intervall $[0;1]$. Das Tasthaar zeigt in Richtung der z-Achse.

Attribute:

LENGTH *Real*

Gibt die Länge des Tasthaares an.

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
WHISKER "Schnurrhaar"
(
  LENGTH 10; // Laenge 10
  MOVETO (100,100,100); // Position
);
```

TACTILE

Ein Tasthaar, das nicht den Abstand, sondern die 'Farbe' der berührten Fläche in den angegebenen Sensorklassen mißt. Als Provisorium für jene Fälle gedacht, in denen Oberflächen in ihrer 'Qualität' ertastet werden sollen.

Attribute:

LENGTH *Real*

Gibt die Länge des Tasthaares an.

CLASS *Integer*

Gibt die Sensorklasse an, in der der Sensor Signale wahrnimmt. Wenn CLASS mehrmals angegeben wird, wird der Ausgabevektor entsprechend aufgestockt.

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
TACTILE "Schnurrhaar"
(
  LENGTH 10; // Laenge 10
  CLASS 1; // registriert Sensorklasse 1
  MOVETO (100,100,100); // Position
);
```

RECEPTOR

Ein Lichtstärkenmesser. Mißt die Gesamtlichtstärke in den vom Benutzer angegebenen Sensorklassen. Die effektiv beobachtete Lichtstärke nimmt in Übereinstimmung mit gängigen Lichtmodellen mit dem Cosinus des Einfallswinkels ab.

Attribute:

CLASS *Integer*

Wie bei TACTILE.

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
RECEPTOR "Phototransistor"
(
  CLASS 1; // registriert Sensorklasse 1
  MOVETO (100,100,100); // Position
);
```

WHISKERFIELD

Eine rechteckiges Feld aus Tasthaaren. Liefert sozusagen die Reliefdarstellung eines direkt benachbarten Körpers.

Attribute:

RESOLUTION *(Integer,Integer)*

Festlegen der Auflösung des Sensors in Breite und Höhe.

SIZE *(Real,Real)*

Gibt die Grundfläche an, auf der sich die Tasthaare verteilen. Die Tasthaare stehen senkrecht auf dieser Fläche.

LENGTH *Real*

Gibt die Länge der Tasthaare an.

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
WHISKERFIELD "Bart"
(
  RESOLUTION (10,10); // 10 x 10 Haare
  SIZE (1,1); // Grundflaeche 1 x 1
  MOVETO (100,100,100); // Position
);
```

CAMERA

Eine Raytracing– oder Z–Buffer-Kamera. Liefert ein Bild in den angegebenen Sensorklassen. Kamera zeigt in Richtung der z-Achse.

Attribute:

RESOLUTION *(Integer,Integer)*

Festlegen der Auflösung des Sensors in Breite und Höhe.

SIZE *(Real,Real)*

Gibt die Grundfläche der Blickpyramide an. Die 'Sehstrahlen' werden von der Spitze der Pyramide aus durch die Grundfläche in Richtung der z-Achse geschickt. Dadurch bestimmt sich der Sichtwinkel der Kamera. Die Höhe der Pyramide ist 1.

CLASS *Integer*

Gibt die Sensorklasse an, in der der Sensor Signale wahrnimmt. Wenn CLASS mehrmals angegeben wird, wird der Ausgabevektor entsprechend aufgestockt.

DEPTH

Legt fest, daß die Kamera ein Tiefenbild berechnet.

ZBUFFER

Das Kamerabild nicht mit einem Strahlverfolgungs sondern mit einem Tiefenpufferverfahren berechnen. Dies ist zwar schneller, braucht aber auch mehr Speicherplatz (wer ihn hat, sollte nicht zögern, es zu benutzen).

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
CAMERA "CCD"
(
  RESOLUTION (20,20); // Aufloesung 20 x 20 Punkte
  CLASS 1; // Kamera sieht Sensorklasse 1
  MOVETO (100,100,100); // Position
  TURNZ 45; // Drehung
  ZBUFFER;
);
```

FACETTE

Eine Raytracing-Kamera mit veränderter Geometrie. Sehstrahlen haben den gleichen Winkelabstand voneinander.

Attribute:

RESOLUTION *(Integer,Integer)*

Festlegen der Auflösung des Sensors in Breite und Höhe.

SIZE *(Real,Real)*

Gibt den Öffnungswinkel der Facette an. 'Die Sehstrahlen' gehen im gleichen Winkelabstand vom Sehpunkt aus.

CLASS *Integer*

Gibt die Sensorklasse an, in der der Sensor Signale wahrnimmt. Wenn CLASS mehrmals angegeben wird, wird der Ausgabevektor entsprechend aufgestockt.

Ports:

VALUE (S)

Wert des Sensors.

Beispiel:

```
FACETTE "Auge"
(
  RESOLUTION (60,20); // Aufloesung 60 x 20 Punkte
  CLASS 1; // Auge sieht Sensorklasse 1
  MOVETO (100,100,100); // Position
  TURNZ 45; // Drehung
);
```

4 Interaktive Steuerung

4.1 Starten der Simulation

Die stand-alone Version *SimRobot* kann von einer Kommandozeile her aufgerufen werden; als Argument wird dann der Name einer Definitionsdatei angegeben. Zusätzlich können auf der Kommandozeile noch Optionen angegeben werden. Zur Zeit sind definiert:

`-d` **(Debug)** Zeige die von der Standardeingabe erhaltenen Kommandos auf der Unix-Pipe *stderr* (also im allgemeinen im aktuellen Kommandofenster) an.

`-m` **(Monochrome)** Darstellung der Szene und der eigentlichen Sensor- und Aktorfenster wie auf einem monochromen Display.

Vollständige Befehlszeile:

```
SimRobot <definitionsdatei> [-d] [-m]
```

Nach dem Start des Programms und dem wenigstens teilweise erfolgreichen Einlesen der Definitionsdatei werden zwei Fenster geöffnet: Das Kontrollfenster, von dem aus alle anderen Fenster geöffnet werden, und ein Szenefenster, in dem die aktuelle Szene dargestellt wird. Auf zwei Arten kann nun mit der Simulation interagiert werden: Zum einen über Menüs (Sektion 4.2) und zum anderen über die Kommandozeile (Sektion 4.3).

4.2 Menüsteuerung

Alle Fenster bis auf das Kontrollfenster haben einen sogenannten 'Push-Pin', der dazu dient, das Fenster von der Oberfläche zu nehmen. Bei den Szenefenstern führt ein Wegnehmen dazu, daß das Fenster wirklich gelöscht wird. Vergrößerung, Blickwinkel und Darstellungsart gehen damit verloren. Aktorpanel und Sensorpanel werden dagegen nur unsichtbar gemacht und durch Anklicken der entsprechenden Knöpfe im Kontrollfenster wieder hervorgeholt. Aus diesem Grund gibt es diese beiden Fenster auch jeweils nur einmal. Die Bedeutung der einzelnen Bedienelemente ist in den Tabellen 1 bis 5 und den dazugehörigen Abbildungen (2 – 8) erläutert.

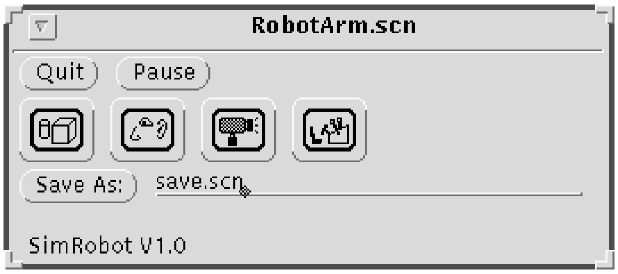

Abbildung 2: Das Kontrollfenster

Element	Bedeutung
Quit	Simulation abbrechen. Es wird keine Abfrage vorgenommen, also Vorsicht.
Pause/Continue	Unterbrechen/Weiterführen der Kommandoauswertung. Damit kann das Steuerprogramm für eine beliebige Zeit angehalten werden.
Icon left	Öffnen eines Szenefensters. Es muß ein beliebiges Objekt der Szene ausgewählt werden, das dann mit seinen Unterobjekten dargestellt wird. Soll die gesamte Szene dargestellt werden, muß das erste Objekt in der Liste (World), angewählt werden.
Icon mid-left	Öffnen des Sensorpanels für die Sensorüberwachung.
Icon mid-right	Öffnen eines Bildfensters für ein Kamerabild oder das Bild eines sonstigen zweidimensionalen Sensors.
Icon right	Öffnen des Aktorpanels für die Aktorsteuerung.
SaveAs	Speichern des aktuellen Zustands unter dem angegebenen Namen.

Tabelle 1: Die Steuerelemente im Kontrollfenster

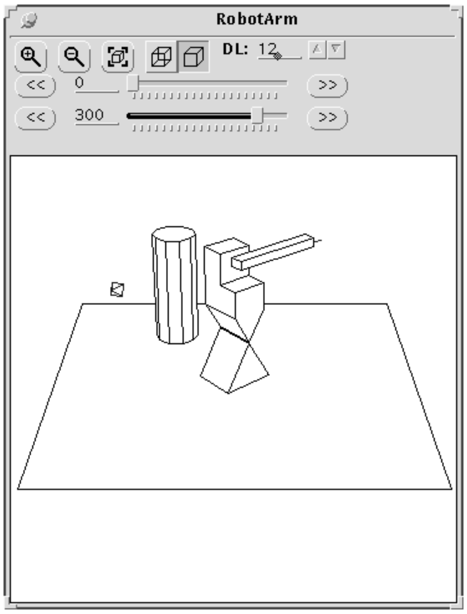

Abbildung 3: Ein Szenenfenster

Element	Bedeutung
Icon left	Vergrößert die Szene um den Faktor 1,2. Die Szene schrumpft und wächst außerdem mit dem Fenster.
Icon mid-left	Verkleinert die Szene um den Faktor 1,2.
Icon middle	Passe den Vergrößerungsfaktor dem dargestellten Objekt an. Die Vergrößerung wird so gewählt, daß das Objekt gerade gut in das Darstellungsfenster paßt. Die Vergrößerung ist dabei auch vom Blickwinkel abhängig, da die Ausdehnung der Szene nur nach dem erzeugten Bild bestimmt wird.
Icon mid-right / right	Wählt die Art der Darstellung.
DL	Detaillevel. Gibt die Detailtiefe an, bis zu der Objekte dargestellt werden. Der Detaillevel eines Objektes orientiert sich an seinem Typ, kann aber in der Definitionsdatei für jedes Objekt verändert werden.
Schiebebalken	Einstellung des Blickwinkels auf die Szene. Oberer Balken: Drehung um z-Achse. Unterer Balken: Drehung um y-Achse. Die äußeren Knöpfe führen eine Drehung um 10 Grad aus. Die Drehung um die z-Achse wird zuerst ausgeführt.

Tabelle 2: In einem Szenefenster werden die gesamte Szene oder Teilobjekte der Szene dargestellt. Hier eine Auflistung der Elemente.

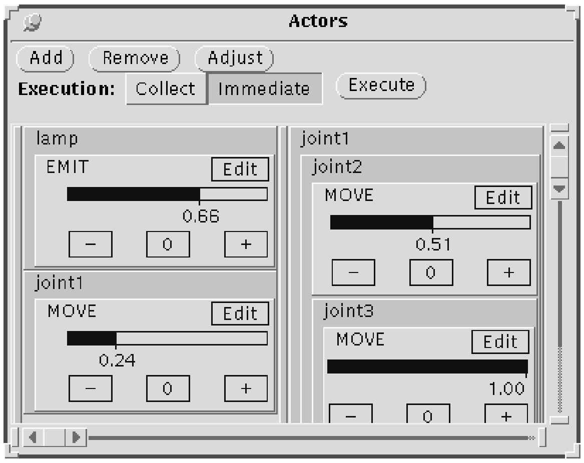

Abbildung 4: Das Aktorpanel

Element	Bedeutung
Add	Öffnen einer Liste mit allen Objekten und Aktorports der Szene. Klicken auf einen Aktorport in der Liste öffnet die entsprechende Steuereinheit, Klicken auf ein Objekt bedeutet, daß eine Steuereinheit für jeden Aktorport des Objekts und die seiner Unterobjekte geöffnet wird. Die Hierarchie der Objekte und Ports wird in der Liste durch Einrückung symbolisiert.
Remove	Entfernen der markierten Steuereinheiten.
Adjust	Anpassen der Spaltenaufteilung der Steuereinheiten an die Höhe des Anzeigefeldes.
Execution: Collect/ Immediate	Wählen, ob ein Aktorkommando sofort beim ersten Klick ausgeführt werden soll. Wenn 'Collect' gewählt wurde, werden alle Kommandos gesammelt, und erst ausgeführt, wenn 'Execute' angewählt wird.
Execute	Ausführen der gesammelten Aktorkommandos.
Anzeigefeld	Hier werden die Steuereinheiten angezeigt. Klicken auf einen 'Edit'-Button bedeutet, daß die Werte der Steuereinheit verändert werden sollen. Dazu wird ein neues Fenster geöffnet, das an anderer Stelle beschrieben ist. Klicken auf den Schieber einer Steuereinheit stellt einen Wert zwischen 0 und 1 ein. Klicken auf '+', '−' oder '0' stellt ein Aktorkommando in Höhe des positiven oder negativen gewählten Wertes oder mit Wert 0 ein. Wird hier statt der linken die rechte Maustaste betätigt, wird die Default-Wahl eingestellt, die bei wiederholter Anwendung der 'Execute'-Taste automatisch verwendet wird. Ein zweiter Klick hebt die Selektion wieder auf. Mausklicks an anderen Stellen werden dazu verwendet, Steuereinheiten zu selektieren.

Tabelle 3: In einem Aktorpanel wird für jeden gewählten Aktor ein kleines Steuerelement angezeigt, mit dem man den jeweiligen Aktor durch Mausklick ansteuern kann.

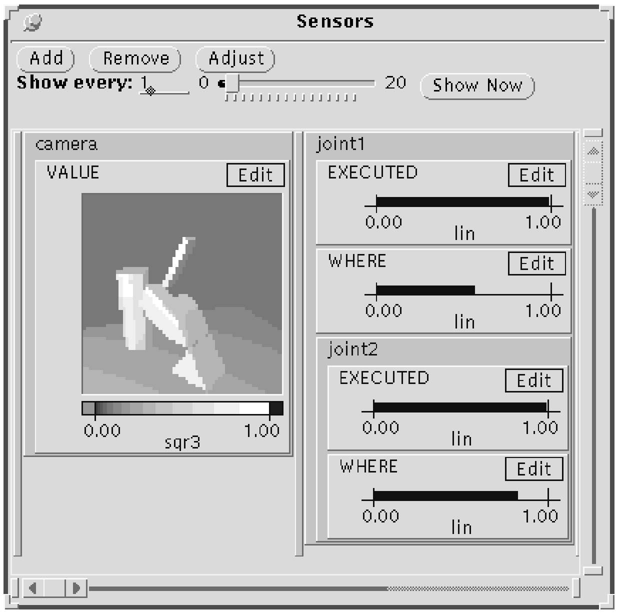

Abbildung 5: Das Sensorpanel

Element	Bedeutung
Add	Wie bei Aktorpanel, jedoch für Sensorports.
Remove	dito
Adjust	dito
Show every:	Gibt an, wie oft die Sensoren angezeigt werden sollen. Bei einer 3 werden die Sensoren nur bei jedem dritten Zeitschritt angezeigt. Da die Simulation bei der stand-alone Version die Sensoren nur berechnet, wenn sie auch abgefragt werden, kann dies eine Beschleunigung bewirken (allerdings nur, wenn das Steuerprogramm diese Sensoren nicht regelmäßig abfragt). Bei einer Null als Wert werden die Sensoren ueberhaupt nicht mehr abgefragt. Diese Einstellung wirkt jedoch nicht, wenn XWindows ein Neuzeichnen des Fensters initiiert (z.B. wenn Adjust gedrückt wurde, oder eine Verdeckung des Fensters beseitigt wurde und XWindows den alten Bildinhalt nicht intern gespeichert hat).
Show Now	Zeige und berechne die Sensoren jetzt neu. Setzt den Zeitschrittzähler wieder auf den Anfang.
Anzeigefeld	Hier werden die Kontrolleinheiten angezeigt. Nur der 'Edit'-Button kann für eine Einheit angeklickt werden, und öffnet ein Fenster, in dem die Darstellung eines Sensorwerts beeinflußt werden kann, indem die Skala verändert wird. Ansonsten wird der Klick verwendet, um die Einheiten zu selektieren.

Tabelle 4: In einem Sensorpanel werden für jeden gewählten Sensorport kleine Kontrolleinheiten angezeigt, auf denen der aktuelle Wert des Sensors dargestellt wird.

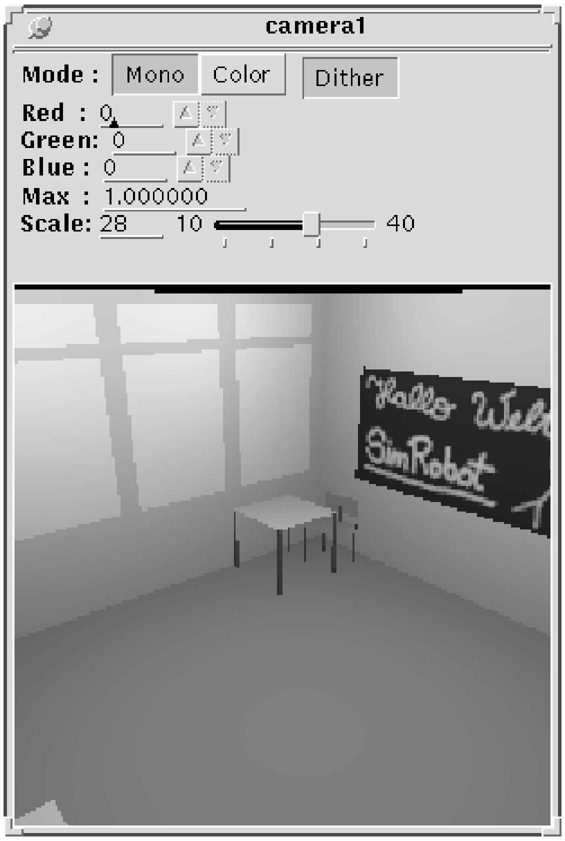

Abbildung 6: Ein Kamerafenster

Element	Bedeutung
Mode	Umschalten zwischen Grautonbild und Mehrfarbbild. Im Modus 'Mono' kann nur eine Bildebene des Sensors dargestellt werden, im Modus 'Color' können bis zu drei verschiedene Bildebenen dargestellt werden, die auf die Grundfarben Rot, Grün und Blau abgebildet werden.
Dither	Wenn 'Dither' aktiviert ist, wird das entstehende Bild mit dem Algorithmus von Floyd-Steinberg gedithert. Dadurch steigt vor allem im Farbmodus die Bildqualität, aber der Bildaufbau dauert etwas länger.
Red/Green/Blue	Einstellen der dargestellten Bildebene für die einzelnen Grundfarben.
Max	Maximum der dargestellten Sensorintensität einstellen. Helligkeitswerte über diesem Maximum werden 'gekappt'.
Scale	Kontrastverstärkung.

Tabelle 5: In einem Kamerafenster wird das Bild einer Kamera (oder eines anderen bilderzeugenden Sensors aufbereitet und angezeigt.

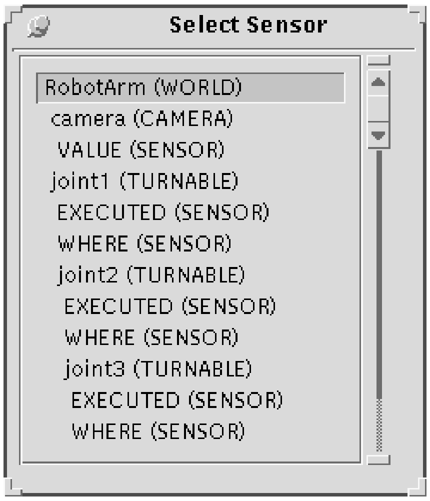

Abbildung 7: Ein Selektionsfenster. Direkte Unterobjekte werden jeweils um eine Stelle eingerückt.

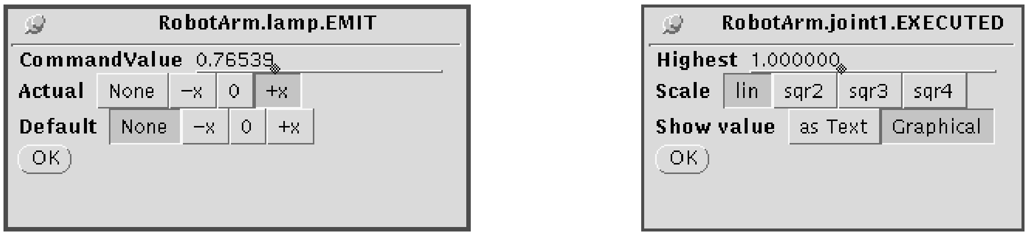

Abbildung 8: Das Editfenster dient dazu, die Darstellung in einem Sensor- oder die Werte eines Aktorpanels zu verändern.

4.3 Kommandozeilensteuerung

Wie erwähnt, erfolgt die Ansteuerung der stand-alone Version über Pipes. In bester *Unix*-Tradition werden dazu zwei besondere Pipes verwendet, nämlich die Standardeingabe und -ausgabe. Aus Gründen der Wartbarkeit sind die Kommandos und ihre Resultate benutzerlesbar kodiert, so daß man zum Testen die Kommandos an die Simulation auch über die Kommandozeile geben kann, und dementsprechend auch die Resultate am Bildschirm erhält. Zudem werden auf diese Weise Unverträglichkeiten der internen Zahlendarstellung vermieden, die bei der Kommunikation zwischen verschiedenen Maschinen auftreten könnten.

Die Simulation sammelt solange Zeichen von der Standardeingabe, bis ein Zeilenende-Zeichen die Vollständigkeit einer Kommandozeile signalisiert. Erst dann wird der String ausgewertet, das gefundene Kommando ausgeführt, und das Ergebnis der Operation auf die Standardausgabe geschrieben. Befindet sich die Simulation im 'Pause'-Modus, wird die Auswertung aufgeschoben, bis dieser Modus wieder beendet wird. Dadurch kann ein Benutzer in Ruhe eine Szene auswerten und Veränderungen vornehmen, ohne daß das Steuerprogramm 'dazwischenfunken' kann.

Das Format eines Kommandos ist leicht zu beschreiben. Am Anfang des Kommandos steht in runden Klammern die Nummer der Anfrage. Mit dieser Nummer wird auch die Rückantwort versehen. Fehlt diese Angabe, wird bei der Rückantwort -1 als Wert angegeben. Diese Angabe dient dazu, die Zuordnung von Kommando und Rückmeldung eindeutig zu machen.

Danach folgt das Kommandowort. Dieses ist in fast allen Fällen der Funktionsname, so wie er auch in der Modulbeschreibung der Simulationsschnittstelle in der Datei `"sim3D.h"` zu finden ist. Allein die Funktion `GetMacro` gibt es in zwei Formen, so daß deshalb die Kommandowörter `GetMacroStr` und `GetMacroInt` für die beiden Formen stehen. Mit der ersten wird ein Macro anhand seines Namens bestimmt, mit der zweiten wird ein Macro anhand seiner Nummer ausgewählt.

Anschließend folgen eventuelle Parameter, die durch Leerzeichen voneinander getrennt sind. Diese Parameter entsprechen denen aus `"sim3D.h"`. Ganze Zahlen und Fließkommazahlen werden so angegeben, wie sie das gewöhnliche `scanf` unter C erkennt. Pointer werden durch ihren Zahlenwert repräsentiert. [2] Bei Zeichenketten müssen Leerstellen und nichtdruckbare Zeichen durch Sequenzen anderer Zeichen ersetzt werden, um das Format nicht zu verletzen. Leerzeichen werden durch `\_` ersetzt, Kontrollzeichen durch das `\` und einer dreistelligen Dezimal-Nummer, die den ASCII-Code des Zeichens angibt. Zusammengesetzte Werte, die einem `struct` unter C oder C++ entsprechen, werden durch geschweifte Klammern zusammengefaßt und mit Komma getrennt. Enthalten diese Werte wiederum zusammengesetzte Werte, wiederholt sich die Klammerung.

[2] Ausnahme: Bei der Funktion `GetSensorValue` wird natürlich nicht der Pointer auf das `Real`-Array übergeben, da dieser ja nur den Platz für den Rückgabewert repräsentiert. Bei der Rückantwort werden stattdessen zuerst die Anzahl der zu übertragenden Werte und dann die Werte selbst – durch Leerzeichen getrennt – übertragen.

```
(0) GetActorPort gelenk1.MOVE
  (0) OK : {67546, 0}
(1) GetSensorPort gelenk1.EXECUTED
  (1) OK : {67546, 0}
(2) GetSensorPort gelenk1.GNFTZ
  (2) ERROR 5 : {0, 0}
(3) SetActorValue {67546, 0} -0.45
  (3) OK :
(4) DoTimeStep
  (4) OK :
(5) GetSensorValue {67546, 0}
  (5) OK : -0.45
```

Abbildung 9: Beispiel für eine Kommunikation mit der Simulation. Die eingerückten Zeilen stellen die Rückantworten dar. Dabei gibt in Klammern die erste Zahl die Objektkennung und die zweite Zahl die Portnummer an.

Als Rückantwort wird auf die Standardausgabe ein String mit folgendem Format geschrieben.

- An erster Stelle in Klammern die Kommandonummer, auf die die Antwort erfolgt.
- Dann folgt entweder

  FAULT , wenn das Kommando nicht ausgewertet werden konnte.

  ERROR *nr* : für eine Ausführung des Kommando, die einen Fehlerstatus lieferte. *nr* ist die Nummer des Fehlers, wie sie in `"sim3DErrors.h"` definiert ist.

  OK : für eine fehlerfreie Ausführung
- Nach ERROR oder OK folgen dann noch die Rückgabewerte, die im gleichen Format angegeben werden, wie die Argumente.

5 Programmsteuerung

5.1 Die Schnittstelle zur Simulation

Neben der im vorigen Kapitel vorgestellten interaktiven Steuerung von *SimRobot* soll nun auf die weitaus wichtigere Steuerung über eigene Programme (Controller) eingegangen werden.

Wie schon oben erwähnt, gibt es zwei Versionen des Programms, die benutzt werden können: Eine Version startet die Simulation als eigenen Prozess und spricht sie über Pipes an. Diese ist nur unter UNIX und XWindows lauffähig. Eine zweite Version ist ein Subset des Programms, das ohne eigene Oberfläche als Teil des Steuerprozesses läuft. Diese Version ist prinzipiell unter den meisten Betriebssystemen lauffähig. Beide Versionen lassen sich von der Ansteuerung her fast identisch benutzen. Die stand-alone Version wird mit

```
#include "sim3DInc.h"
```

in das Steuerprogramm eingebunden, die Version ohne Oberfläche mit

```
#include "sim3D.h".
```

Beide Module definieren eine Klasse `SIMULATION`, die jeweils eine Szene repräsentiert. In der ersten Version wird dabei durch den Konstruktor ein neuer Prozeß mit dem eigentlichen Programm gestartet. Die Methoden der Klasse werden überwiegend durch Kommunikation mit dem Programm abgearbeitet. Der Destruktor der Klasse fordert den Prozeß auf, sich zu beenden. Die zweite Version greift dagegen direkt auf die Datenstrukturen zu, die mit dem Konstruktor aufgebaut worden sind. Um einen Angleich der Funktionsweisen der Versionen zu erreichen, wird auch in der stand-alone Version bei jeder Anfrage erst auf die Reaktion des Simulationsprozesses gewartet, bevor die Kontrolle an das Steuerprogramm zurückgegeben wird.

Die Klasse `SIMULATION` definiert nun die verschiedenen Methoden, welche dem Programmierer zur Kommunikation über die Schnittstelle zur Verfügung stehen. Alle Methoden sind unten aufgelistet und kurz erläutert. Ferner läßt sich der Code in einem der beiden oben genannten HeaderDateien sowie der zugehörigen `.cc` Datei nachlesen.

5.2 Starten der Simulation

Laden einer Szenendefinition (und evtl. Starten eines neuen Prozesses):

```
SIMULATION* sim;
sim = new SIMULATION ("dateiname"); // SIMULATION ist eine Klasse
```

Wurde ein neuer Prozess gestartet, dann sollten unter UNIX zwei Signale abgefangen werden, die anzeigen, daß ein Prozeß abgebrochen wurde. Zu diesem Zweck wird z.B. eine Funktion `SimQuit` definiert, die dann vom Steuerprogramm aufgerufen werden soll, und vom aktuellen Zustand rettet, was zu retten ist.

```
void SimQuit ();          // Behandlungsroutine fuer Abbruch der Simul.
signal (SIGCHLD, SimQuit); // Hier die eine eigene Routine angeben
signal (SIGPIPE, SimQuit); // dito
```

Der Simulationsprozeß wird vom Steuerprozeß kontrolliert abgebrochen, indem der Destruktor für das Objekt aufgerufen wird. Also:

```
delete sim;
```

5.3 Ansteuern der Objekte, Lesen der Sensoren

- Bestimmen von Ports für Sensoren und Aktoren

```
ACTORPORT ap;
SENSORPORT sp1;
```

- Namensauflösung

  In *SimRobot* werden alle Sensoren und Aktoren (repräsentiert durch sogenannte 'Ports'), über einen symbolischen Namen angesprochen. Dieser Name setzt sich aus den Namen aller Objekte und Unterobjekte, zu denen der Sensor gehört, sowie dem Namen des Ports zusammen. Diese Methode der Namensgebung ist direkt von der Bildung von Pfadnamen z.B. unter *Unix*-Dateisystem inspiriert, jedoch wurde als Trennzeichen nicht der Schrägstrich, sondern der Punkt gewählt.

```
ap = sim->GetActorPort ("objekt1.unterobjekt3.PORTNAME");
sp = sim->GetSensorPort ("objekt2.unterobjekt2.PORTNAME");
```

  Diese Methode schützt jedoch noch nicht ausreichend gegen Veränderungen der Szene. Bekommt beispielsweise eine Kamera an einem Roboterarm ein eigenes Gelenk, um es zusätzlich ausrichten zu können, so ist dieses Gelenk ein Unterobjekt, das im obigen Pfadnamen des Sensors auftauchen müßte. Zwar hat sich durch diese Veränderung die Bedeutung der Kamera grundsätzlich verändert (Denn nun muß das Kamerabild zusätzlich in Bezug zur Bewegung des neuen Gelenks gesetzt werden), aber trotzdem wäre es schön, wenn man sie weiterhin unter dem gleichen Namen verwenden könnte. Dann müßte die Controller Datei nicht bei jeder derartigen Szenenveränderung angepaßt werden. Aus diesem Grunde wurde die Möglichkeit eingeführt, für ein Objekt in der Definitionsdatei Pfade einzuführen (siehe Attribut `PATH`). Mit ihnen können Unterobjekte oder Ports der Objekte angesprochen werden, ein Verweis auf einen parallel liegendes Objekt ist jedoch nicht möglich. Dies wird dadurch erreicht, daß Objekt-/Portnamen auf dem Weg durch die Objektstruktur bei jedem erreichten Objekt um den Objektnamen gekürzt werden. Dann wird das unmittelbar nächste Namenssegment mit den definierten Pfaden verglichen und bei Übereinstimmung durch den Link ersetzt. Erst dann wird das nunmehr erste Namenssegment mit den Namen der Unterobjekte und Ports verglichen.

  Da dieser Vorgang aufwendig ist, sollten die Portnamen nur einmal bei der Initialisierung des Steuerprogramms ausgewertet werden. Dabei erhält man einen abstrakten Port, unter dem der Aktor bzw. Sensor ab sofort angesprochen wird. Dieser Port enthält intern die Adresse des Objekts sowie eine Nummer, unter der dieser Port dem Objekt bekannt ist. Deshalb kann ein Port nicht nach einem erneuten Laden einer Definitionsdatei weiter verwendet werden, da selbst bei absolut gleicher Definitionsdatei die Objekte im Hauptspeicher des Rechners neue Adressen erhalten haben können.

- Setzen eines Aktorwertes

  Absetzen eines Kommandos an die Simulation (Bewegungskommandos liegen zwischen -1 und 1):

```
sim->SetActorValue (ap, 0.5);
```

- Lesen eines Sensorwertes

  Der Wert eines Sensors kann ein Array von reellen Zahlen sein, das sich auch noch über mehrere Dimensionen erstreckt. Es gibt zwei Funktion um zum einen die Zahl der Dimensionen (liefert z.B. 0 für Einzelsensoren wie dem Tasthaar, und 2 für Kameras mit einer wahrgenommenen Sensorklasse und 3 bei mehrfarbigen Bildern) und zum anderen die Ausdehnung des Array in einer Dimension zu bestimmen. Der Speicher für dieses Array ist vom Benutzer bereitzustellen. Die erforderliche Größe berechnet man, indem man die einzelnen Ausdehnungen miteinander multipliziert. Hier ein Anwendungsbeispiel für den Sensorport `sp`:

```
INTEGER dim = sim->GetSensorDim (sp);
INTEGER i, size = 1;
for (i=0;i<dim;i++)
  size = size * sim->GetSensorDimSize (sp, i);

SHORTREAL* buffer = new SHORTREAL [size];
sim->GetSensorValue (sp, buffer);

// ... Auswertung des Sensors ...

delete [] buffer;
```

  Man kann das Belegen und Freigeben des Speichers auch von der Schnittstelle erledigen lassen:

```
SHORTREAL* buffer = sim->NewSensorBuffer (sp);
sim->GetSensorValue (sp, buffer);
// ... Auswertung ...
sim->DeleteSensorBuffer (sp, buffer);
```

  Diese Prozedur kann vereinfacht werden, wenn definitiv nur ein Wert zurückerwartet wird. Wenn dem nicht so ist, kann es aufgrund der Eigenheiten von C und C++ zu einem Systemabsturz oder undefinierten Reaktionen des Programms kommen, also Vorsicht. Ansonsten läßt sich bei entsprechender Sorgfalt ein Sensor, der die Dimension Null hat, eben auch so lesen:

```
SHORTREAL result;
sim->GetSensorValue (sp, &result);
```

  Ein primitives Steuerungsbeispiel, das die prinzipielle Vorgehensweise veranschaulicht, ist in "`test.cc`" gegeben. Jedoch werden in dem Beispiel keine Fehler abgefangen. Routinen, um Fehler bei der Behandlung der Schnittstellenfunktionen zu erkennen, sind in "`sim3DErrors.h`" definiert. Einen elementaren Test auf Fehler bietet außerdem die Methode `WasOk` der Klasse `SIMULATION`. Liefert `WasOk` nicht `TRUE`, so sollte auf jeden Fall `sim3DTakeLastError` aufgerufen werden, um die Fehlernummer wieder auf 'Kein Fehler' zu setzen.

- Ausführung aller bisher abgegebenen Kommandos

```
sim->DoTimeStep ();
```

5.4 Abfragen der Szene

Den Handle eines Unterobjektes erhält man mit

```
SIM3DOBJECT GetObject (STRING);
```

wobei diese Methode bei Angabe des leeren Strings " " den Handle des *Wurzelobjektes* der jeweiligen Simulation liefert. Die Anzahl der direkten Unterobjekte eines Objektes erfragt man mit

```
INTEGER SubObjectCount (SIM3DOBJECT obj);
```

und den Handle des n-ten Unterobjektes erhält man dann durch

```
SIM3DOBJECT SubObject (SIM3DOBJECT obj, INTEGER n);
```

Von jedem Objekt kann man den Objektnamen, den in der Definitionsdatei mit dem Attribut `REMARK` angegebenen Bemerkungsstring und den Namen der Klasse mit den folgenden Methoden abrufen:

```
STRING ObjectName   (SIM3DOBJECT obj);
STRING ObjectRemark (SIM3DOBJECT obj);
STRING ObjectClass  (SIM3DOBJECT obj);
```

Desgleichen lassen sich die Anzahl der direkten Sensor- und Aktorports des angegebenen Objekts und deren Namen abrufen:

```
INTEGER SensorPortCount (SIM3DOBJECT obj);
INTEGER ActorPortCount  (SIM3DOBJECT obj);
STRING  SensorPortName  (SIM3DOBJECT obj, INTEGER n);
STRING  ActorPortName   (SIM3DOBJECT obj, INTEGER n);
```

In der direkt eingebundenen Version (`"sim3D.h"`) kann man sich ein Objekt auf einen `GRAPHICPORT` ausgeben lassen. Zusätzlich gibt man dabei die Detailtiefe, bis zu der Objekte ausgegeben werden sollen, und eine Standardlänge für die Ausgabe von ausdehnungslosen Objekte (z.B. Emitter) an. In `"HiddenGraph.h"` ist ein `GRAPHICPORT` definiert, der verdeckte Linien berücksichtigt. Eine eigene Anwendung sollte in einer abgeleiteten Klasse wenigstens die Methoden SetPixel, SetLine und evtl. RegisterColors definieren. Die Ausgabe-Methode für ein Objekt lautet:

```
void DrawObject (SIM3DOBJECT obj, GRAPHICPORT* gp,
                 INTEGER detail, REAL scale);
```

5.5 Manipulieren der Szene

Die Schnittstelle bietet auch die Möglichkeit, die Szene zur Laufzeit in einem gewissen Rahmen zu verändern. Man hat die Möglichkeit, vorhandene Objekte in die Szene neu einzufügen, zu positionieren und zu löschen. Objekte können auf der Ebene der direkten Unterobjekte der Simulationswurzel mit

```
SIM3DOBJECT AddObjectAs (SIM3DOBJECT obj, STRING asname);
```

eingefügt werden, dabei wird ein neues Objekt mit dem angegebenen Namen erzeugt. Ein Objekt kann über

```
void RemoveObject (SIM3DOBJECT obj);
```

aus der Szene gelöscht werden.

Lesen und Setzen der Position von Objekten erfolgt durch

```
MOVEMATRIX  GetLocation (SIM3DOBJECT obj);
void        SetLocation (SIM3DOBJECT obj, MOVEMATRIX m);
```

Die Struktur `MOVEMATRIX` setzt sich aus einem Translationsvektor und einer Rotationsmatrix zusammen. Die genaue Definition ist in "`Vector.h`" nachzulesen. Auch alle notwendigen Operationen sind dort zu finden.

Man hat auch die Möglichkeit, sich aus den Makros zu bedienen, die in der Definitionsdatei erstellt wurden. Den Handle zu einem Makro kann man sich entweder über die Nummer oder über den Namen des Makros beschaffen. Im ersteren Fall benötigt man die Methoden

```
INTEGER MacroCount ();
SIM3DOBJECT GetMacro (INTEGER n);
```

Im zweiten Fall nimmt man

```
SIM3DOBJECT GetMacro (STRING);
```

6 Erweiterbarkeit

Die Frage "Wie schreibe ich meine eigene Klasse?läßt sich nicht generell beantworten, da je nach Funktion der Klasse unterschiedlich viel Aufwand getrieben werden muß. Auf jeden Fall muß der bestehende Quellcode an mindestens einer Stelle verändert und ein neues Modul geschrieben werden, um eine neue Klasse benutzen zu können. Diese Klasse wird von einer der beschriebenen Klassen abgeleitet. Welche der Methoden dabei umdefiniert werden müssen, wird im nachfolgenden Text beschrieben. Die Methoden treten zum überwiegenden Teil in "`sim3DBasics.h`ßum ersten Mal auf und sind dort mehr oder weniger kommentiert. Es wird für die nachfolgenden Erläuterungen angenommen, daß der neue Klassenname `SIM3DNeuerTyp` sei.

Vorgehensweise

Es muß eine eindeutige Integer-Konstante mit Namen `S3DNeuerTyp` vereinbart werden, die zur Identifikation des Typs dient. Zahlenwerte ab 1000 stehen für den Benutzer zur Verfügung.

In der Definition der neuen Klasse müssen auf jeden Fall folgende Methoden definiert werden:

- `SIM3DNeuerTyp (STRING name = "");`

  Der Konstruktor der neuen Klasse sollte den Konstruktor der Vorgängerklasse mit dem übergebenen Namen `name` aufrufen, und auf diese Weise die Datenstrukturen initialisieren und dabei den Namen des Objekts setzen.

- `SIM3DNeuerTyp (const SIM3DNeuerTyp& obj);`

  Der Kopierkonstruktor sollte den Kopierkonstruktor des Vorgängers aufrufen, und alle selbstdefinierten Felder kopieren (soweit das sinnvoll ist).

- `~SIM3DNeuerTyp ();`

  Der Destruktor sollte alle zusätzlich allozierten Datenstrukturen wieder freigeben.

- ```
  virtual STRING GetClass ()
            { return "NEUERTYP"; };
  ```

  Die Methode `GetClass` liefert den Namen der Klasse ohne '`SIM3D`' in Grossbuchstaben. Der gleiche Name sollte in der Funktion `ReadObject` in Modul `"sim3DReader.cc"` eingetragen werden. Wie das zu geschehen hat, ist (hoffentlich) beim Betrachten der Funktion ersichtlich.

- ```
  virtual BOOLEAN  IsA (INTEGER typ)
    { return (typ == S3DNeuerTyp) || SIM3DVorgaenger::IsA (typ); }
  ```

  Diese Methode dient dazu, den Typ der Klasse auf einfache Weise bestimmen zu können.

- ```
  virtual SIM3DElement* Copy()
    { return new SIM3DNeuerTyp (*this); }
  ```

  Erzeuge eine vollständige Kopie des Objekts.

Methoden, die nur im Bedarfsfall neu definiert werden:
  ```

- `virtual APORTDEF* GetAPortDef ();`

  Wenn das Objekt ansteuerbar ist, so müssen Aktorports definiert werden. `GetAPortDef` liefert den Zeiger auf eine Struktur, die die erforderlichen Daten (Name des Ports und eine Funktion, die die Ansteuerung erledigt) enthält. Diese Struktur ist in `"sim3DPortDef.h"` definiert, ein Anwendungsbeispiel ist z.B. in `"sim3DVehicle"` zu finden. Soll kein Port definiert sein, obwohl die Vorgängerklasse Ports definiert, so muß `GetAPortDef` *NULL* liefern.

- `virtual SPORTDEF* GetSPortDef ();`

  Wenn von dem Objekt irgendwelche Sensorwerte gelesen werden sollen, so müssen Sensorports definiert werden. Dies funktioniert wie bei den Aktorports, nur daß bei mehrdimensionalen Sensoren noch die Dimension und eine Funktion zum Feststellen der Größe des Sensors angegeben werden muß. Beispiele finden sich in `"sim3DSensors"`.

- `virtual void ReadAttr (const STRING& attr, SCANNER& scan);`

  Wenn die Klasse zusätzliche Attribute definiert, so müssen diese aus der Definitionsdatei gelesen werden. `ReadAttr` muß in diesem Fall den Namen eines in `attr` übergebenen Attributs erkennen und aus dem Objekt `scan` mit den in `"sim3DFile.h"` definierten Methoden und Funktionen lesen. Attribute, die nicht erkannt werden, werden an die gleiche Methode der Vorgängerklasse übergeben. Für Beispiele, siehe `"sim3DBasics.h"`.

- `virtual void WriteAttrs (BACKUP& backup);`

  Die gleichen Attribute, die gelesen werden, müssen auch wieder abgespeichert werden können. Um die entstehende Datei kurz zu halten, sollten Tests durchgeführt werden, die ein Attribut nicht abspeichern, wenn es den Default-Wert hat. Es werden Methoden und Funktionen aus `"sim3DFile.h"` verwendet. Es sollte auf jeden Fall das `WriteAttrs` der Vorgängerklasse aufgerufen werden, um die dort definierten Attribute zu speichern. Siehe `"sim3DBasics.h"` für Beispiele.

- `virtual void RegisterAdd ();`

  Diese Methode wird aufgerufen, wenn ein Objekt in eine Szene eingefügt wird. Wenn dabei irgendwelche Maßnahmen ergriffen werden müssen, weil Abhängigkeiten von der Wurzel bestehen, so ist hier der richtige Platz dafür. Die gleichnahmige Methode der Vorgängerklasse sollte bei einer Neudefinition aufgerufen werden.

- `virtual void RegisterRemove ();`

  Gegenstück zu RegisterAdd, das aufgerufen wird, wenn ein Objekt aus einem anderen Objekt (und damit aus der Szene) herausgenommen wird.

- `virtual MOVEMATRIX  GetLocalTransform ();`

  Diese Methode liefert die Lage und Position des Objekts relativ zu dem Bezugssystem des Oberobjektes. Eine `MOVEMATRIX` ist eine 3×4-Matrix, die eine Transformation darstellt, die mit anderen verknüpft werden kann, um so im Endeffekt die absolute Position in der Szene zu erhalten. Bei Gelenken ist die Stellung des Gelenks schon im Resultat der Funktion berücksichtigt.

- `virtual void MakeTimeStep (MOVEMATRIX M);`

  Durchführen eines Zeitschrittes. Diese Methode wird rekursiv für alle Unterobjekte eines Objekts aufgerufen und führt deren Aktorkommandos aus. Es wird das Bezugssystem für das Objekt übergeben, das aber noch mit dem lokalen Bezugssystem (`GetLocalTransform`) verknüpft werden muß, bevor das `MakeTimeStep` der Unterobjekte aufgerufen wird. Es sollte in einer Redefinition die gleichnahmige Methode der Vorgängerklasse aufgerufen werden.

- `virtual void MakeStep (MOVEMATRIX M);`

  Durchführen einer Bewegung (z.B. eines Gelenkes). Ruft `MakeStep` für alle Unterobjekte auf. Parameter `M` hat die gleiche Bedeutung wie bei `MakeTimeStep`. Alle Objekte, die auf diese Methode reagieren, sollten ihre Position nach diesem Aufruf anpassen, dabei aber die alten Wert speichern, so daß sie mit StepBack wieder hergestellt werden können. Normalerweise müssen nur Körper auf diese Methode reagieren. Aufgerufen wird `MakeStep` von Objekten, die das Bezugssystem für ihre Unterobjekte intern ändern, in der Methode `MakeTimeStep`. Siehe also `"sim3DBody.h"` und z.B. `"sim3DMoveable.h"`.

- `virtual void StepBack ();`

  Gegenstück zu `MakeStep`; in diesem Fall wird die alte Position mit `StepBack` wieder hergestellt. Aufrufe von `MakeStep` und `StepBack` können nicht geschachtelt werden.

- `virtual BOOLEAN DoesIntersect (SIM3DElement* obj);`

  Diese Methode testet, ob sich das Objekt mit dem angegebenen Objekt überschneidet. Überschneidungen finden im allgemeinen nur zwischen Körpern statt. Diese Methode wird verwendet, um die Zulässigkeit von Bewegungen zu überprüfen. Wenn eine Überschneidung vorliegt, sollte die vorgenommene Veränderung zurückgenommen werden. Siehe auch `"sim3DBody.h"`.

- `virtual void MakeNewBoundingBox ();`

  Berechne das umgebende Rechteck des Objekts neu. Nur für Körper bzw. sichtbare Objekte interessant.

- `virtual COLOR GetColor ();`

  Nur interessant, wenn für eine Klasse eine besondere Standardfarbe definiert werden soll, die verwendet wird, wenn in der Definitionsdatei keine Objektfarbe definiert wurde. In dieser Farbe wird das Objekt in der Szenendarstellung gezeichnet. Beispiel in `"sim3DSensors.h"`.

- ```
  virtual void Draw (GRAPHICPORT* gp, MOVEMATRIX m,
                     INTEGER detail, REAL scale);
  ```

  `Draw` dient dazu, das angesprochene Objekt auf einem virtuellen Zeichengerät `gp` auszugeben. `m` ist die Transformation, die vorher auf das Objekt anzuwenden ist, `detail` gibt ein Schranke an, bis zu der Details eines Objekts gezeichnet werden sollen (zu diesem Zweck läßt sich für jedes Objekt ein *Detaillevel* vereinbaren, ein Zahlenwert, der mit `detail` verglichen wird und damit angibt, ob ein Objekt noch gezeichnet werden soll oder nicht. Diese Maßnahme wird notwendig, wenn z.B. sehr viele Sensoren definiert sind und das Bild dadurch sehr unübersichtlich wird). `scale`
  ```

stellt ein Bemaßungskriterium für Objekte zur Verfügung, die keine eigentliche Ausdehnung haben, z.B. Emitter, die eigentlich punktförmig sind, und sollte die in der Szenendarstellung mindestens gewünschte Größe angeben. Erklärungen finden sich in `"BasicGraphics.h"` und `"sim3DBasics.h"`, Beispiele in `"sim3DSensors.h"` und `"sim3DBody.h"`.

- ```
  virtual BOOLEAN CalcRayIntersection (RAY ray, BOOLEAN infinite,
                          REAL& where, SIM3DElement*& obj, INTEGER& facenr);
  ```

  Diese Methode berechnet den Schnitt eines Strahls mit den Körpern eines Objekts und seiner Unterobjekte. Als Parameter wird ein Strahl (`ray`) übergeben und die Angabe, ob der Strahl unendlich lang sein soll (`infinite`). Wenn ein Schnittpunkt vorliegt, liefert die Funktion `TRUE` zurück. Als weitere Werte erhält man dann, auf welchem Punkt des Strahls der erste Schnitt vorliegt (`where`), das betreffende Objekt (`obj`) und welche Fläche des Objekts (`facenr`) getroffen wurde. Diese Methode wird zur Sensorberechnung von Kamera, Tasthaar und Taktil benutzt. Siehe auch `"sim3DBody.h"`.

- `SENSCLASSVECTOR GetColorOfFaceAt (INTEGER n, VECTOR p)`

  Liefert die Farbe einer Fläche eines Objekts an einer bestimmten Stelle. Nur für sichtbare Körper interessant.

- `SHORTREAL GetColorOfFaceAt (INTEGER cl, INTEGER n, VECTOR p);`

  Liefert die Helligkeit einer Fläche eines Objekts an einer bestimmten Stelle und in der angegebenen Sensorklasse. Nur für sichtbare Körper interessant.

- `VECTOR GetNormalOfFaceAt (INTEGER n, VECTOR p);`

  Liefert den Normalenvektor einer Fläche eines Objekts an einer bestimmten Stelle. Nur für sichtbare Körper interessant.

- `void DrawBodyOnZBuffer (ZBUFFER&);`

  Der Körper zeichnet sich selbst in ein ZBUFFER-Objekt. Dazu benutzt es die Methoden der Klasse ZBUFFER. Nur für sichtbare Körper interessant.
  ```

7 Ausblick

Wie wohl jedes Programm weist auch dieses noch Aspekte auf, die verbesserungswürdig sind, und noch nicht in Angriff genommen wurden. Einige sollen hier genannt werden und zur Diskussion anregen.

7.1 Allgemeine Erweiterungsmöglichkeiten

- Bei der Komposition einer Szene wird nicht getestet, ob ein neues Objekt räumlich überhaupt an die angegebene Stelle paßt. Dies kann zu interessanten Verknüpfungen benutzt werden, wenn Objekte sich nämlich durchdringen *sollen*. Bei Objekten, die sich frei bewegen sollen, ist dieses aber unerwünscht, denn es wird verhindern, daß sich das Objekt bewegen kann, da es in einem (kleinen) Bewegungsschritt den Überschneidungsbereich wohl kaum verlassen kann.

- Das Verfahren, Simulations-Objekte und ihre Ports über die Adressen der Objekte anzusprechen, ist ungünstig, wenn z.B. eine Szene noch einmal neu geladen werden soll. Dabei werden sich nämlich mit größter Wahrscheinlichkeit die Adressen der Objekte verändern und müssen also neu bestimmt werden. Der Benutzer muß diesen Schritt explizit selbst durchführen.

- Aus diesem Grund ist es nicht so einfach möglich, die Kommunikation zwischen Steuerungsprogramm und stand-alone Version von *SimRobot* abzuspeichern und für ein späteres Replay einer Simulation zu benutzen. Hier fehlt ein geeigneter Indirektionsschritt, der die Adressierung der Objekte von den physikalischen Adressen unabhängig macht und reproduzierbar aber trotzdem schnell ist. Aus diesem Grund gibt es in der Schnittstelle keine Funktion für ein 'Reload', mit dem man den Grundzustand wiederherstellen könnte.

- Die interaktive Steuerung sollte (genau wie die Schnittstelle zur Controller-Steuerung) die Möglichkeiten bieten, Objekte in die Szene einzufügen, zu löschen und zu positionieren.

- Die graphische Oberfläche bedarf sicherlich noch einer weiteren Überarbeitung.

- Geschwindigkeit ist und bleibt ein Problem der Simulation. Aus Gründen der Erweiterbarkeit können viele Optimierungen nicht durchgeführt werden.

7.2 Die Simulation von Kräften

Es sollten Kräfte simuliert werden. Zu den simulierten Effekten gehören insbesondere:

- Objekte können weggeschoben werden

- Objekte können kippen und fallen

- Objekte können genommen werden

Kraft ist definiert als Beschleunigung mal Masse. Das heißt, Kraft steht in erster Linie für die Beschleunigung, die auf einen Körper wirkt. Doch die resultierende Bewegung ist das Resultat unterschiedlichster Einflüsse. Verantwortlich für die Gesamtbewegung aller Objekte sind:

- Unregelmäßige Krafteinwirkung
- Reibung
- Hindernisse
- allgemeine mechanische Regeln

Dadurch treten eine Vielzahl von Effekten auf, die uns aus dem tägliche Leben bekannt sind. Dafür ein Beispiel: Ein Gegenstand, der über eine Tischkante geschoben wird, fällt nicht erst in dem Augenblick, in dem er die Kante vollständig überschritten hat. Sobald sein Schwerpunkt über die Kante bewegt wurde, beginnt, wenn keine anderen Kräfte wirken, eine Drehbewegung, die dadurch ausgelöst wird, daß eine Hälfte des Körpers der Gravitation ungeschützt ausgesetzt ist, während bei der anderen Hälfte der Tisch der Gravitation entgegenwirkt. Diese Drehung bewegt den Körper nach unten, eventuell aber auch zur Seite weg, weil ansonsten der Weg nach unten behindert wäre. Trifft der Gegenstand am Boden auf, wird er aufgrund elastischer Eigenschaften nicht sofort flach am Boden liegen bleiben, sondern mehr oder weniger stark zurückspringen.

Aus der klassischen Schulphysik wissen wir, daß die Reibungskräfte, die zwischen zwei Körpern wirken, sich aus dem Auflagedruck, dem Reibungskoeffizienten und den Bewegungsvektoren der beiden Körper ergeben. Wohlweislich vermeidet die Schulphysik aber Beispiele, die zu weit über die schiefe Ebene hinausgehen oder mehr als zwei Körper beinhalten. Wenn nämlich z.B. der Bewegungsvektor (bei einer Drehung) nicht für die ganze Auflagefläche gleich ist, oder mehr als zwei Körper sich gegenseitig beeinflussen, entstehen Zusammenhänge, die sich nicht mehr durch einfache Gleichungen ausdrücken lassen.

Weil die Berechung von Kräften zwischen Körpern zu aufwendig ist, verzichtet die Simulation zur Zeit vollständig darauf. Das hat auch Einfluß auf die zur Verfügung gestellten Elemente:

- Fahrzeuge können sich entweder nur in einer Ebene bewegen, die unabhängig von einem Untergrund ist, oder frei im Raum.
- Laufmaschinen werden (anfänglich) nicht zur Verfügung gestellt.
- Roboterarme können nicht greifen oder verschieben.
- Drucksensoren sind Kraftsensoren und werden nicht bereitgestellt.

Literatur

[1] U. Siems: *„SimRobot–Ein Programm zur Simulation sensorbestückter Agenten in einer dreidimensionalen Umwelt"*, Diplomarbeit, FB 3 Informatik, Universität Bremen, 1994.

[2] M. Heger: *„Hierarchisches konnektionistisches Konzept zur Reflexkontrolle einer Laufmaschine"*, Diplomarbeit, Universität Karlsruhe, 1992.

[3] P. van der Smagt: *„Simderella (likes to sleep)",* Department of Computer Systems and Mathematics, University of Amsterdam, 1993.

[4] M. Beier, S. Hapke, C. Hoppe, N. Körtge, A. Röpnack, U. Siems: *„FORANN-Projektbericht Nr.6: SimSalaBIM",* in: B. Krieg-Brückner, J. von Holten, R. Seifert: *„Berichte des Projekts FORANN",* FB 3 Informatik, Universität Bremen, 1992.

[5] J. D. Foley, A. van Damme, S. Feiner, J. Hughes *„Computer Graphics: Principles and Practice", Second Edition,* Addison-Wesley, Reading (Mass.), 1990.

A Simulationsobjekte

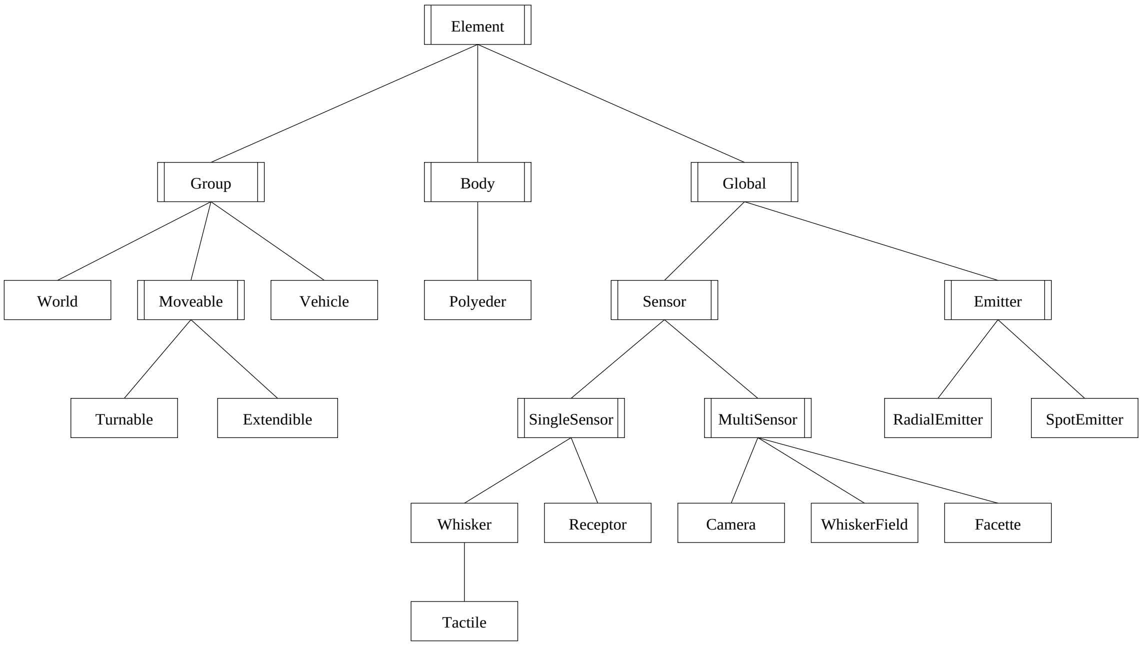

Abbildung 10: Vererbungshierarchie der Simulationsobjekte unter *SimRobot*; die abstrakten Klassen sind mit doppelt senkrechten Strichen versehen.

Hier die Auflistung aller definierten abstrakten Klassen (d.h. Klassen, die keine Objekte erzeugen und deshalb auch nicht in Kapitel 3 beschrieben wurden) von Simulationsobjekten mit Erklärung ihrer Bedeutung. Die Klassennamen sind im C++-Quelltext mit einem vorangestellten '`sim3D`' versehen, um Benennungsprobleme zu umgehen und die Zugehörigkeit zu verdeutlichen. Dieser Prefix wurde hier weggelassen, weil die Klassen in der Definitionssprache über die kurzen Klassennamen angesprochen werden.

Element `SIM3DElement` ist die Grundklasse für die gesamte Vererbungshierarchie. Es stellt alle in Kapitel 3.1 beschriebenen Eigenschaften bereit und definiert nötige und nützliche Hilfsfunktionen.

Moveable `SIM3DMoveable` ist die Grundlage für alle Gelenke, die einen Freiheitsgrad haben und sich innerhalb dieses Freiheitsgrades prinzipiell auch nur in einem festen Bereich bewegen können. Diese Klasse definiert die Ports für Steuerung, Bewegungs- und Lagefeststellung, und stellt den allgemeinen Mechanismus zur Ansteuerung bereit. Das Gelenk selbst ist völlig körperlos und unsichtbar. Die Auswirkung einer Bewegung besteht darin, daß das Bezugssystem für die Unterobjekte verändert wird. Das aktuelle Bewegungsmodell sieht vor, daß eine Bewegung nur dann ausgeführt wird, wenn dies ohne Überschneidung der Körper des Gelenks mit denen der restlichen Szene möglich. Bewegung erfolgt immer nur relativ zur vorherigen Position.

Body Grundklasse für alle Körper. Körper zeichnen sich dadurch aus, daß sie sich (im ungünstigsten Falle) mit anderen Körpern schneiden können und eine diskrete Oberfläche mit gewissen Oberflächeneigenschaften haben. Diese Oberfläche wird von `SIM3DBody` verwaltet.

Global Einzelobjekt, das seine absolute Position in der Szene zwischenspeichert. Hat ansonsten keine Bedeutung.

Sensor Diese Klasse faßt alle externen Sensoren zusammen. Die einzige Eigenschaft, die in dieser Klasse vermittelt wird, ist von interner Bedeutung: Sensoren, die dieser Klasse angehören, werden nur ausgewertet, wenn der entsprechende Port gelesen wird (Es sei denn, es wurde ein globales Flag gesetzt, das die Berechnung bei jeder Veränderung erzwingt).

SingleSensor Diese Klasse definiert den Port für Sensoren, die nur einen einzelnen Wert liefern.

MultiSensor Diese Klasse definiert den Port für zweidimensionale Sensoren, sowie die Auflösung und die Größe eines solchen Sensors.

Emitter Eine abstrakte Lampe (oder eine Schallquelle oder was auch immer). Ein Emitter hat eine Strahlungscharakeristik mit relativen Strahlungsstärken in den einzelnen Sensorklassen, und liefert für einen beliebigen Punkt im Raum eine Gesamt-Strahlungsstärke und eine Strahlungsrichtung (Vektor). Zusätzlich läßt sich ein Emitter über einen Port in seiner Strahlungsstärke steuern.

World Jede Szene hat natürlich eine 'Wurzel', die alle Objekte aufnimmt. Diese Klasse nimmt außerdem alle Informationen auf, die global in der gesamten Szene gebraucht werden. Dies sind unter anderem

- Die Liste aller in der Szene definierten Emitter
- Die ambiente Strahlungsstärke (Stärke des ungerichteten Lichts)
- Die Helligkeit und Farbe des Hintergrundes
- Eine Liste aller geladener oder definierten Texturen
- Eine Liste aller in der Definitionsdatei beschriebenen Oberflächen
- Eine Liste aller in der Definitionsdatei beschriebenen Makro-Objekte
- Sollen Schatten berechnet werden?
- Sollen Sensoren sofort nach jedem Schritt berechnet werden?

Außerdem stellt `SIM3DWorld` zusätzliche Funktionen zur Sensorauswertung bereit.

B Liste der Module

BasicGraphics – Virtuelles, dreidimensionales Ausgabegerät für Szenen, Verwaltung der Farben für die Ausgabe

BigSensor2D – Darstellung und Verwaltung von Kamera-Fenstern. Für die stand-alone Version.

Coding – Kodierung und Dekodierung beliebiger Strings zur Übertragung über Pipes. Für die stand-alone Version.

CommandReader – Lesen und Auswerten eines Kommandostrings. Für die stand-alone Version.

Common – Definition von verläßlichen Datentypen (INTEGER, REAL, ...).

ControlPanel – Darstellung und Verwaltung der Fenster zur Darstellung von Sensorwerten und Aktoreinheiten. Für die stand-alone Version.

DisplayPanel – Darstellung und Verwaltung von Szene-Fenstern. Für die stand-alone Version.

Geometry – Verwaltung von Sehstrahlen, Intervallen auf dem Zahlenstrahl und umgebenden Quadern.

HiddenGraph – Graphikport mit Hidden-Line Darstellung. Besonders für die stand-alone Version.

SensorClasses – Verwaltung beliebig langer Zahlenvektoren mit Defaultwert für nicht definierte Elemente.

SimRobot – Hauptmodul der stand-alone Version.

Vector – Vektor-, Matrix- und sonstige geometrische Operationen im dreidimensionalen Raum.

XInits – Datenstrukturen und Initialisierung für XWindows- und XView-Gebrauch.

XViewGraphics – Hidden-Line Graphikport für die Verwendung mit dem graphischen Toolkit XView unter XWindows. Besonders für die stand-alone Version.

ZBuffer – Modul für den z-Buffer-Algorithmus.

mystring – Eigene Stringverwaltung.

sim3D – Schnittstelle der Simulation für direkte Einbindung des Simulationskerns in eigene Programme.

sim3DBasics – Die Grundklassen `SIM3DElement`, `SIM3DGroup` und `SIM3DGlobal`.

sim3DBody – Die Körperklassen `SIM3DBody` und `SIM3DPolyeder`.

sim3DDynamics – Dynamische Arrays für Strings, Reals und Integers.

sim3DEmitter – Die Emitterklassen `SIM3DEmitter`, `SIM3DRadialEmitter` und `SIM3DSpotEmitter`.

sim3DErrors – Fehlerbehandlungsroutinen und Fehlerkonstanten.

sim3DFile – Die Klassen `SCANNER` und `BACKUP` zum Lesen und Schreiben von Definitionsdateien.

sim3DInc – Schnittstelle der Simulation für eigenständiges *SimRobot*-Programm

sim3DMoveable – Die Gelenkklassen `SIM3DMoveable`, `SIM3DTurnable` und `SIM3DExtendible`.

sim3DPorts – Datenstrukturen zur Portdefinition in allen Simulations-Klassen.

sim3DReader – Lesen eines Objektes von einem `SCANNER`-Objekt. Neue Objekttypen müssen hier eingetragen werden.

sim3DWorld – Die Wurzelklasse `SIM3DWorld`. Diese Klasse enthält alle globalen Daten und ist zuständig für alle allgemeinen Anfragen an eine Szene.

sim3DSensors – Alle externen Sensoren:

`SIM3DSensor`, `SIM3DSingleSensor`, `SIM3DMultiSensor`, `SIM3DWhisker`, `SIM3DTactile`, `SIM3DReceptor`, `SIM3DCamera`, `SIM3DWhiskerField`.

sim3DVehicle – Eine Fahrzeugklasse `SIM3DVehicle`.

C Syntax der Definitionssprache

Die Syntax der Definitionssprache läßt sich nur teilweise angeben, weil für die Interpretation der Definitionsdatei die Simulationsobjekte teilweise selbst verantwortlich sind (Attribute). Dieser Fall ist in der nachfolgenden Darstellung der Syntax in Backus-Naur-Form durch ... angedeutet. Welche Attribute von welchen Objekten verstanden werden, ist in der Auflistung der Attribute in Kapitel 3.2 beschrieben.

Für Kommentare gelten die Konventionen von C++. Das heißt, Text, der zwischen `/*` und `*/` steht, oder Text, der in einer Zeile hinter `//` steht, wird ignoriert. Kommentare können geschachtelt werden.

Bezeichner (*ident*) werden intern automatisch in Großbuchstaben umgesetzt.

Backus-Naur-Form

```
all        ::= "WORLD" [ objectdef ] .
objectdef ::= string
               [ "(" attribute { ";" attribute } ")" ]
               [ "[" object { ";" object } "]" ]
object     ::= classname objectdef
             | "MACRO" string [ "AS" string ]
classname  ::= "GROUP"                    // Objektklassen
             | "TURNABLE"
             | "EXTENDIBLE"
             | "VEHICLE"
             | "POLYEDER"
             | "SPOTEMITTER"
             | "RADIALEMITTER"
             | "CAMERA"
             | "FACETTE"
             | "WHISKER"
             | "TACTILE"
             | "RECEPTOR"
             | "WHISKERFIELD"
             | otherident
otherident ::= ident              // Zusaetzliche Klassennamen koennten
                                  // definiert werden!
attribute  ::= ident [ parameter ]
parameter  ::= integer
             | real
             | string
             | ident
             | vector
             | sensvector
             | ...
integer    ::= digit { digit }
digit      ::= "0" | "1" | "2" | ... | "9"
real       ::= [ "+" | "-" ] integer [ "." integer ]
                  [ ("E" | "e") [ "+" | "-" ] integer ]
string     ::= """ { printablechar | escapedchar } """
printablechar ::=      // Alle Zeichen bis auf ", \ und Kontrollcodes
escapedchar ::= "\" somechar
somechar   ::=         // beliebiges Zeichen
```

```
ident      ::= letter { letter | digit | "_" }
letter     ::= "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
vector     ::= "(" real "," real "," real ")"
sensvector ::= "(" [ real ] { "," [ real ] } ")"

properties ::= sensvector
```

D Grafikformat der Texturen

SimRobot nutzt das Programmpaket *xli* zur Konvertierung verschiedenster Grafikformate in ein einheitliches Format. Die Konvertierung kann vorab geschehen, oder zur Laufzeit (*SimRobot* erkennt sein internes Format an einer 'magic number' und startet den Konverterer als Prozeß, wenn ein anderes Format vorliegt). Jedoch werden nicht alle Formate mit diesem Programm abgedeckt. Deshalb an dieser Stelle jetzt eine knappe Beschreibung des intern verwendeten Formats in Tabelle 6 für diejenigen, die sich ihren Konverter selbst schreiben wollen.

Tabelle 6: Header des *SimRobot*-Texturformates

	32-bit	
0	4F5A5101h	Magic Number
1	Byte pro Pixel	Bedeutung: 0: Ein Bit pro Pixel – 1,2: Ein oder zwei Byte pro Pixel (Index in Farbtabelle) – 3: Drei Byte pro Pixel (direkte Farbinformation)
2	Breite in Pixeln	
3	Höhe in Pixeln	
4	Größe der Farbtabelle	Anzahl der Einträge (n)
5	Größe des Bildes	in Bytes
$6 \ldots 5+n$	Farbe: 00RRGGBBh	Farbtabelle mit n Einträgen