Graphische Darstellung und Manipulation von Wissensbasen

Diplomarbeit an der Universität Ulm
Fakultät für Informatik
Abteilung: Künstliche Intelligenz

vorgelegt von:

Dieter Ludwig Finkenzeller

1. Gutachter: *Prof. Dr. F. W. von Henke*

2. Gutachter: *Prof. Dr. S. Biundo-Stephan*

1999

Inhaltsverzeichnis

1 Einleitung.....1
1.1 Konzeption.....2
1.2 Vergleich mit anderen Ansätzen.....6
1.2.1 GKB-Editor.....6
1.2.2 LOOM Ontosaurus.....7
1.3 Überblick.....8
2 Graphikwerkzeuge.....9
2.1 VRML 2.0.....9
2.2 Open Inventor.....15
2.3 OpenGL.....17
2.3.1 Profil eines möglichen OpenGL Programms.....19
2.3.2 Arbeitsweise von OpenGL.....20
2.4 Anforderung an das Graphikwerkzeug.....21
2.5 Die drei Graphikwerkzeuge im Vergleich.....21
2.6 Kollisionserkennung.....23
2.6.1 V-Collide.....23
2.6.2 Zusammenspiel von OpenGL und V-Collide.....25
3 Wissensrepräsentation.....27
3.1 Allgemeines.....27
3.2 Terminologische Logiken und LOOM.....29
3.2.1 Begriffsdefinitionen.....29
3.2.2 Syntax und Semantik.....30
3.2.2.1 Operatoren zur Konzeptgenerierung.....31
3.2.2.2 Operatoren zur Rollengenerierung.....32
3.2.2.3 Notation terminologischer Axiome.....33
3.2.2.4 Notation der Objekte.....34
3.2.3 LOOM.....35
3.3 Modell der realen Welt.....35
3.4 Objektrepräsentation.....36
4 Kommunikation.....43

4.1 Kommunikation über TCP/IP....43
4.2 Protokoll....44
4.3 Implementierung....50
4.3.1 LISP-Netz-Modul....50
4.3.2 C/C++-Netz-Modul....51
5 Erstellung des Gesamtsystems....53
5.1 Zielsysteme....53
5.1.1 Entwicklungsumgebung und Plattform....53
5.1.2 Wissensrepräsentationskomponente....54
5.1.3 Graphikkomponente....54
5.2 Modulbeschreibung....55
5.2.1 WR-Komponente....55
5.2.2 Graphikkomponente....56
5.2.2.1 Das Modul ai_object3D....57
5.2.2.1.1 Klasse object3D....57
5.2.2.1.2 Klasse sceneObject....57
5.2.2.1.3 Klasse controlObject....60
5.2.2.2 Das Modul ai_scene....61
5.2.2.3 Das Modul ai_face....63
5.2.2.4 Das Modul ai_control....64
5.2.2.5 Das Hauptprogramm ai....65
6 Bewertung....67
6.1 Messungen....67
6.1.1 Bearbeitungszeiten der Wissensbasis....68
6.1.2 Bearbeitungszeiten der Bildschirmdarstellung....69
6.1.3 Vergleich der Bearbeitungszeiten....72
6.2 Perspektiven....72
6.2.1 Erweiterung des Modells....73
6.2.2 Erweiterung der Graphikkomponente....74
6.3 Einsatzgebiete....75
7 Zusammenfassung....77
8 Anhang....79

8.1 VRML 2.0 Beispiel..79
8.2 Open Inventor: Beispiel zu C++...82
8.3 Open Inventor: Beispiel zum Dateiformat...83
8.4 LOOM: Modell der Büroumgebung - TBox...84
8.5 Netzwerkkomponente: LISP-Netz-Modul..86
8.6 Netzwerkkomponente: C/C++-Netz-Modul..87
9 Literaturverzeichnis..89

1 Einleitung

Die Wissensrepräsentation ist ein wichtiges Teilgebiet der Künstlichen Intelligenz. Hier wird unter anderem versucht, einen Teil der Realität zu modellieren. Dabei wird Wissen über die Realität in symbolischer Form repräsentiert und Schlußfolgerungen aus diesem Wissen gezogen. Da die Abteilung Künstliche Intelligenz mit Teilprojekten am Sonderforschungsbereich 527 „Integration symbolischer und subsymbolischer Informationsverarbeitung in adaptiven sensomotorischen Systemen“[1] beteiligt ist, steht in dieser Arbeit das Gebiet der Service-Robotik in heterogenen Umgebungen im Vordergrund. In diesem Zusammenhang liegt ein Schwerpunkt der Abteilung auf der Modellierung und Repräsentation von Wissen über solche Umgebungen (Szenarios). Welches Wissen muß nun in diesem Umfeld modelliert werden? Hier ist ein Modell zu erstellen, welches das Wissen über die Umgebung repräsentiert. Dabei ist das Modell in zwei Bereiche einzuteilen (vgl. Teilprojekt D1, SFB 527[2]):

- Im *Konzeptmodell* ist eine hierarchische Beschreibung der Eigenschaften der Objekte der Umgebung festzulegen.
- Konkrete Objekte bzw. Instanzen der Objekte des Konzeptmodells spiegeln eine konkrete Umgebung wider. Sie werden im *Weltmodell* zusammengefaßt.

Für die Erforschung, der in den einzelnen Teilprojekten untersuchten Interaktionen zwischen Methoden der symbolischen und subsymbolischen Informationsverarbeitung, wird ein autonomes Fahrzeug, der SFB-Demonstrator, eingesetzt. Unter anderem soll dieser schnell auf unerwartete Veränderungen der Umgebung reagieren. Unterschiedliche Szenarien, die als symbolisches Weltmodell vorliegen, müssen hierzu entworfen werden. Die Modellierung dieser Szenarien ist mit herkömmlichen Schnittstellen zu Wissensbasen recht mühsam und aufwendig. Desweiteren wird es schwierig, bei größeren Szenarios, den Überblick zu behalten, wodurch die Anfälligkeit für Fehler in der Modellierung steigt. Eine graphische Benutzerschnittstelle, die eine dreidimensionale Visualisierung des Szenarios bietet, kann hier Abhilfe schaffen. Mit ihr lassen sich Szenarios, wie mit einem Baukasten, aber in graphischer Form, modellieren. Die Manipulation der Objekte des Szenarios erfolgt bei einer völlig frei wählbaren dreidimensionalen Sicht auf das Szenario.

Ziel dieser Arbeit ist die Entwicklung einer graphischen Schnittstelle zur Manipulation dreidimensionaler Objekte aus einer Wissensbasis. Die graphische Schnittstelle spiegelt ein symbolisches Weltmodell der Wissensbasis, in Form eines dreidimensionalen Szenarios, wider. Angelehnt an die SFB-Thematik besteht diese Szene hauptsächlich aus einer Büroumgebung. Über die Oberfläche der graphischen Schnittstelle soll eine Bürowelt, die zunächst auf einen Raum beschränkt ist, aus einzelnen Objekten aufgebaut werden können. Die Entwicklung eines neuen Formalismus zur Repräsentation von Wissen ist

1 Informationen zum SFB 527 sind im Internet unter http://www.uni-ulm.de/SMART/index.e.html zu finden.

2 Eine Beschreibung zum Teilprojekt D1 kann im Internet unter http://www.informatik.uni-ulm.de/ki/sfb-d1.html eingesehen werden.

nicht Teil der Arbeit. Hier wird auf ein bestehendes System, LOOM[3], zurückgegriffen. Es ist Aufgabe dieser Arbeit ein geeignetes Protokoll für die Kommunikation zwischen der graphischen Oberfläche und der Wissensbasis ist zu definieren. Für die graphische Darstellung soll aus einer Menge existierender Werkzeuge ein geeignetes ausgewählt und angepaßt werden.

Die vorrangige Aufgabe liegt damit auf der Entwicklung eines geeigneten Kommunikationsprotokolls, einer graphischen Oberfläche und der Implementierung eines exemplarischen Gesamtsystems. Weiterhin soll eine kleine Wissensbasis mit dem Gesamtsystem erstellt werden, die sich an der Welt eines Service-Roboters orientiert.

1.1 Konzeption

Es soll Wissen über eine Büroumgebung dreidimensional dargestellt werden. Ebenso soll das Wissen von einem Benutzer über eine graphische Oberfläche bearbeitet werden können. Die darunterliegende Wissensbasis bleibt dem Benutzer verborgen. Außerdem hat der Benutzer keine Kenntnis darüber, ob sich die Wissensbasis auf seinem Arbeitsrechner oder auf einem anderen Rechner befindet (Netzwerktransparenz). Das Gesamtsystem besteht demnach aus folgenden drei Komponenten: die graphische Oberfläche (Graphikkomponente), die Wissensbasis (Wissensrepräsentationskomponente) und die Netzwerkkomponente (Sicherstellung der Netzwerktransparenz). Die nächsten Punkte umreißen die Aufgaben der einzelnen Komponenten.

- Die Wissensrepräsentationskomponente[4] enthält das Konzeptmodell der Büroumgebung. Hier werden die Eigenschaften der Objekte in Objektklassen eingeteilt und die Beziehungen, in denen die Objekte miteinander stehen können, beschrieben. Die Informationen zur geometrischen Form der Objekte werden nicht modelliert. Sie werden in der Graphikkomponente gehalten. Desweiteren wird Wissen über konkrete Objekte, also Instanzen der Objektklassen, verwaltet. Die Wissensrepräsentationskomponente erhält, über die Netzwerkkomponente, von der Graphikkomponente Anweisungen zur Änderung des gespeicherten Wissens und Anfragen bzgl. des Wissens. Die Graphikkomponente wird darüber in Kenntnis gesetzt, ob Änderungen angenommen oder zurückgewiesen wurden. Anfragen werden bearbeitet und das Ergebnis wird an die Graphikkomponente weitergegeben (siehe Kapitel 3).
- Die Graphikkomponente visualisiert die Information der Büroumgebung, die in der Wissensbasis gehalten wird. Sie beinhaltet die geometrischen Formen der einzelnen Objektklassen der Wissensbasis. Sie verarbeitet Benutzereingaben, schickt Änderungen und Anfragen über die Netzwerkkomponente an die WR-Komponente. Anschließend wartet sie auf eine Bestätigung bzw. Ablehnung einer Änderung bzw. auf das Ergebnis einer Anfrage (siehe Abschnitte 2.3, 5.1.3 und 5.2.2).
- Die Netzwerkkomponente verbindet die beiden anderen Komponenten. Hier ist ein High-Level-Protokoll festzulegen, über das die beiden Komponenten kommunizie-

3 LOOM ist ein Projekt der Artificial Intelligence research group des Information Sciences Institute der University of Southern California.
http://www.isi.edu/isd/LOOM/LOOM-HOME.html

4 Die Wissensrepräsentationskomponente wird im weiteren Verlauf mit WR-Komponente abgekürzt.

ren. Dabei erhält jede der beiden anderen Komponenten jeweils ein Netz-Modul, über das die Kommunikation abgewickelt wird. Eine detaillierte Beschreibung der Netzwerkkomponente wird in Kapitel 4 gegeben.

Die einzelnen Komponenten ergeben zusammen das in Abbildung 1 dargestellte Gesamtsystem. Die Abbildung zeigt den Zusammenhang der einzelnen Komponenten bzgl. des Informationsflusses der Daten von der WR-Komponente zur Graphikkomponente und zurück.

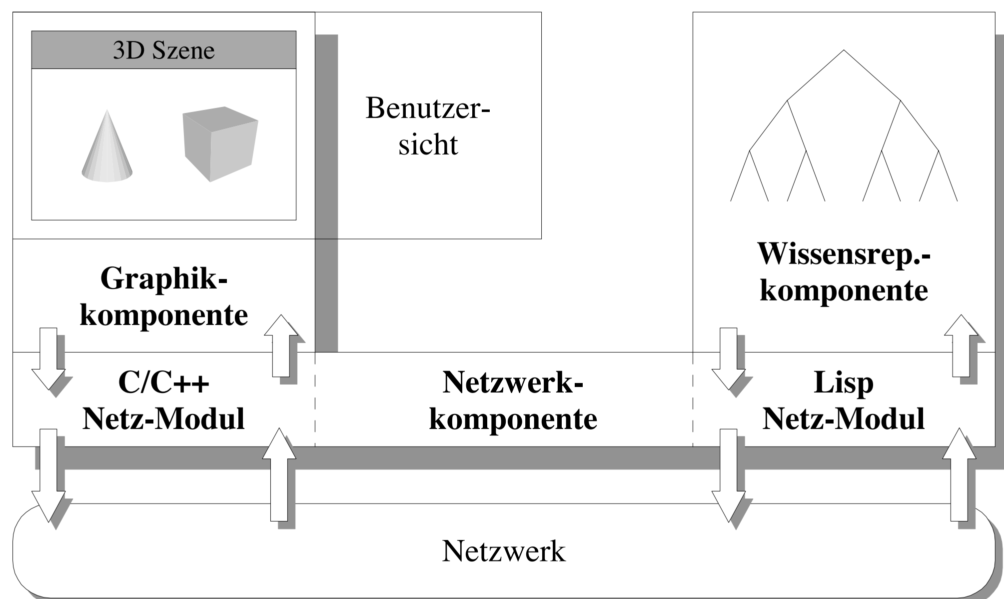

Abbildung 1: Gesamtsystem

Dem Benutzer zeigt sich das Gesamtsystem nur über eine graphische Oberfläche. Sie ist in zwei Fenster aufgeteilt. Eines der Fenster enthält das 3D-Modell der Büroumgebung (die Szene) und eine Kontrolleiste, die Elemente zur Bearbeitung der dreidimensionalen Szene bereitstellt. Das andere Fenster birgt Elemente zur Programmsteuerung und Felder, die die Attribute eines einzelnen 3D-Objekts zeigen. Nur über diese Oberfläche kann der Benutzer auf die Szene Einfluß nehmen. Die Objekte können gedreht, positioniert, gestapelt und gelöscht werden. Ebenso kann die gesamte Szene gedreht und verschoben werden. Der Benutzer kann neue Objekte bzw. Instanzen bereits definierter Objektklassen (siehe Kapitel 3) erstellen, aber keine neuen Objektklassen generieren. Neue Objektklassen können nur in der Wissensbasis selber definiert werden. Entsprechend muß in der Graphikkomponente für diese Klasse eine neue geometrische Form bereitgestellt werden.

In Abbildung 2 ist das erste Fenster der Oberfläche gezeigt, das eine Beispielszene und die Kontrolleiste beinhaltet. Mit den Elementen der Kontrolleiste lassen sich einzelne Objekte, wie auch die gesamte Szene, drehen und verschieben.

Abbildung 3 zeigt das zweite Fenster der Oberfläche, das die Attribute eines 3D-Objekts und die Programmsteuerung enthält, die 2D Benutzerschnittstelle. Mit den Elementen

auf der linken Seite kann ein Objekte angelegt, gelöscht und geändert werden. Desweiteren kann die aktuelle Szene in der Wissensbasis gespeichert und das Programm beendet werden. Die Felder auf der rechten Seite beinhalten Attribute des Objekts, wie Typ, Name, Name in der Wissensbasis, Farbe, Position und Ausrichtung. Die genaue Bedeutung, der einzelnen Elemente beider Fenster, ist in der *Benutzerdokumentation* [Fin 99] aufgeführt.

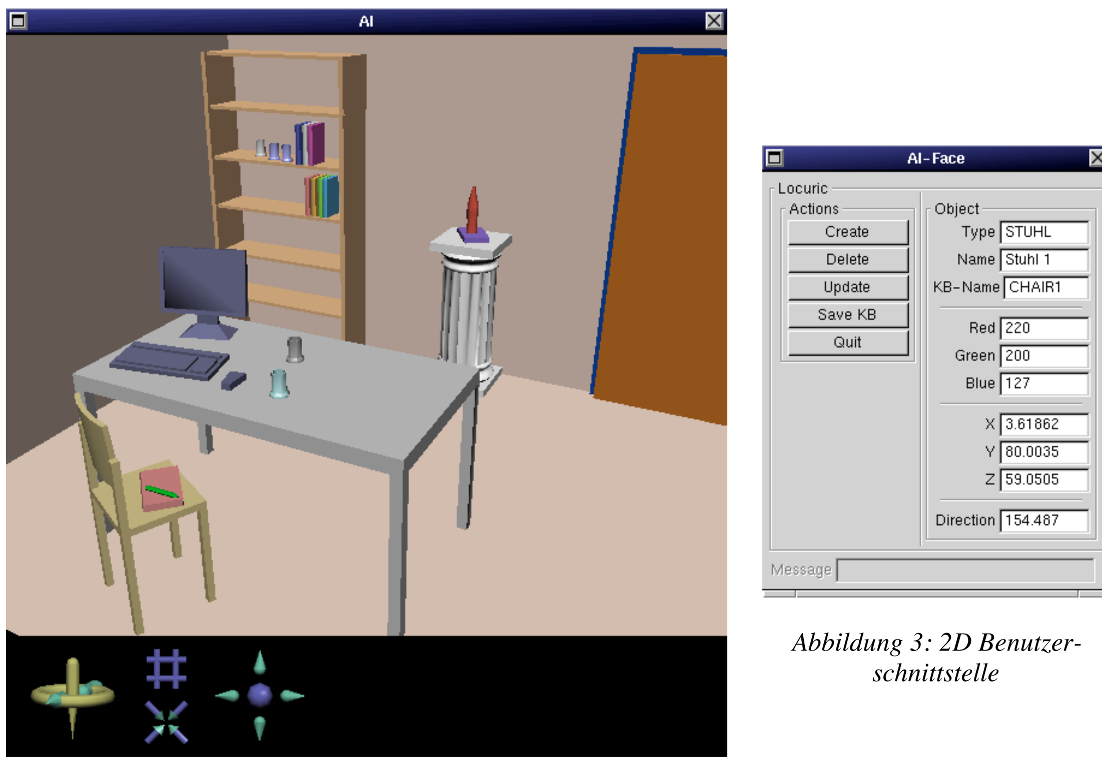

Abbildung 2: Szene und Kontrolleiste

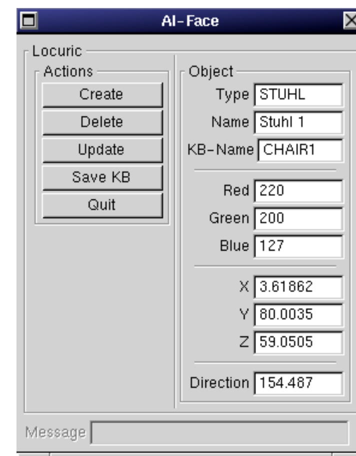

Abbildung 3: 2D Benutzerschnittstelle

Anhand einer Beispielsitzung soll kurz die Funktionalität des Gesamtsystems und das Zusammenspiel der drei Komponenten verdeutlicht werden. Dabei wird ein Stuhl, auf dem ein Buch und ein Stift liegen, auf einen Tisch gestellt. In Abbildung 2 ist der Grundzustand der Szene zu sehen. Wir werden uns nun alle weiteren Schritte, mit der jeweiligen Abbildung der Szene, ansehen, die zum Endzustand „der Stuhl steht auf dem Tisch“ führen.

Um den Stuhl bewegen zu können, muß er zuerst angewählt, d.h. mit der linken Maustaste angeklickt, werden. Der Stuhl zeichnet sich, wenn er angewählt ist, gegenüber den anderen Objekten durch kleine rote Quadrate an seiner Oberfläche aus (Abbildung 4). Dieser Zustand wird mit *aktiv* bezeichnet. Damit soll ausgedrückt werden, daß der Stuhl gehalten wird und man ihn nun mit der Maus bewegen kann.
Mit dem Anwählen des Stuhls ist ebenfalls eine Anfrage an die Wissensbasis verbunden, die ermittelt, welche Objekte direkt oder indirekt auf dem Stuhl liegen. In unserem Beispiel sind dies das Buch und der Stift.

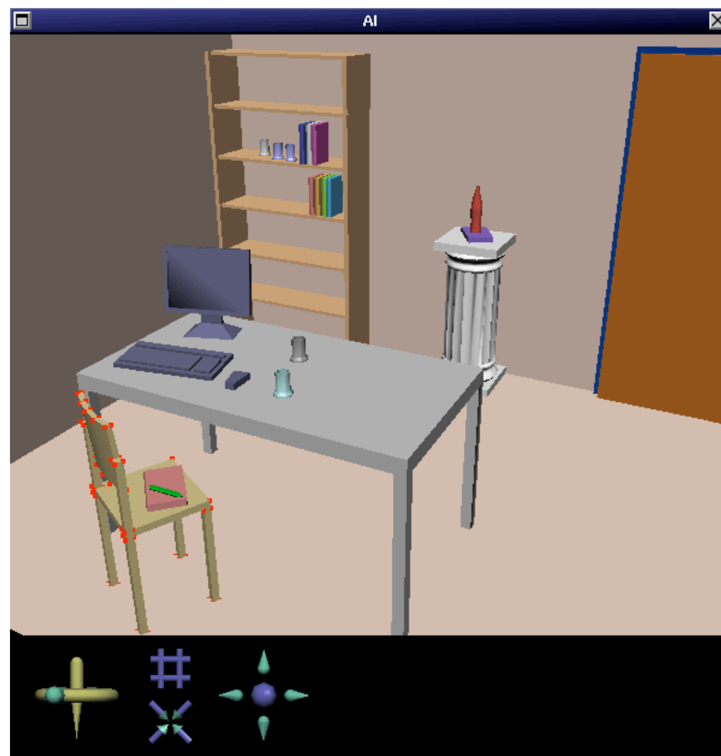

Abbildung 4: Stuhl, angewählt

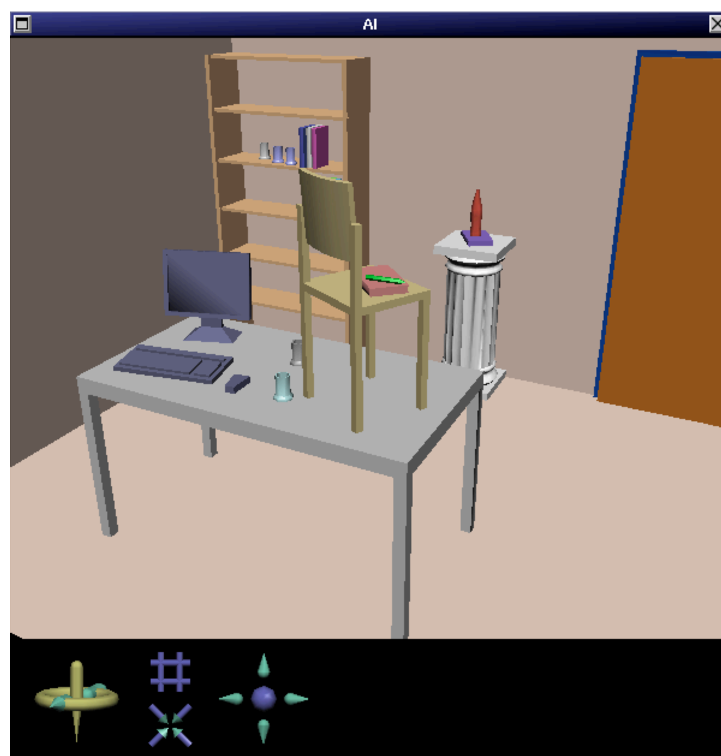

Abbildung 5: Endzustand, Stuhl steht auf dem Tisch

Als nächstes wird der Stuhl an seine neue Position auf den Tisch gebracht. Während der Stuhl bewegt wird, werden die beiden Objekte, die sich auf dem Stuhl befinden, synchron mitbewegt. Auf dem Tisch wird der Stuhl, durch erneutes Anklicken mit der Maus, losgelassen. Er ist somit nicht mehr aktiv und fällt nun, mit all den darauf befindlichen Objekten, auf den Tisch.

Die Wissensbasis erhält nun die Anweisung, den Stuhl auf den Tisch zu stellen. Da die Wissensbasis diese Änderung erlaubt, wird sie schließlich noch über die Position des Stuhles und der darauf befindlichen Objekte informiert. Die Kriterien, anhand derer die Wissensbasis Entscheidungen trifft, werden in Kapitel 3 dargelegt. Wäre es nicht zulässig, so würde die Wissensbasis den Versuch, den Stuhl auf den Tisch zu stellen, zurückweisen und der Stuhl und die darauf befindlichen Objekte nähmen wieder ihre alte Position ein, die sie vor der Aktion hatten. Die Konsistenz der Wissensbasis ist in jedem Fall gewährleistet. In Abbildung 5 ist der Endzustand „der Stuhl steht auf dem Tisch“ zu sehen.

1.2 Vergleich mit anderen Ansätzen

Einen völlig unterschiedlichen Ansatz verfolgen die beiden Arbeiten, GKB-Editor und LOOM Ontosaurus, die in diesem Abschnitt vorgestellt werden. Im Gegensatz zur vorliegenden Arbeit visualisieren sie nicht die „Semantik“ des modellierten Wissens, sondern die „Syntax“.

1.2.1 GKB-Editor

Einen anderen Aspekt der Visualisierung von Wissensbasen greift der GKB-Editor des SRI[5] auf. Hier wird die Struktur des modellierten Wissens von Wissensbasen graphisch repräsentiert. Das Modell wird als Graph dargestellt: Objekte sind die Knoten und Beziehungen zwischen den Objekten sind die Kanten des Graphen. Der Benutzer kann mit der Maus durch den Graph navigieren und ihn editieren. Dabei können Bereiche des Graphen herausgegriffen und vergrößert angezeigt werden. Die graphische Sicht auf das Modell abstrahiert von der darunterliegenden Wissensbasis. Dies bietet den Vorteil, daß über diese einheitliche Benutzeroberfläche mit verschiedenen Wissensbasen gearbeitet werden kann.

Der GKB-Editor besteht aus drei Modulen: der graphischen Benutzeroberfläche, einer Bibliothek mit generischen Wissensbasis-Funktionen (*generic frame functions*) und Bibliotheken, die die Abbildung der generischen Wissensbasis-Funktionen auf Funktionen der entsprechenden Wissensbasis vornehmen.

Der Ablauf, wie der GKB-Editor Benutzereingaben verarbeitet, sieht wie folgt aus: Benutzerinitiierte Anfragen, Änderungen usw. werden über „*generic frame functions*“ realisiert. Sie werden dann auf Funktionen und/oder Methoden der darunterliegenden Wissensbasis abgebildet. Die Kommunikation zwischen den Modulen erfolgt über das „*generic frame protocol*“.

5 SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025 USA
http://www.ai.sri.com/~gkb

Um Wissensbasen an den GKB-Editor anzuschließen, muß für jede Wissensbasis ein Modul vorgeschalten werden, das dieses „*generic frame protocol*“ unterstützt und eine Abbildung auf die Wissensbasis vornimmt. Mit dem „*generic frame protocol*“ soll die Benutzung von Wissensbasen vereinheitlicht werden.

Abbildung 6 zeigt den *Class Hierarchy Viewer*, mit dem man die Klassenhierarchie und Instanzen der Klassen betrachtet.

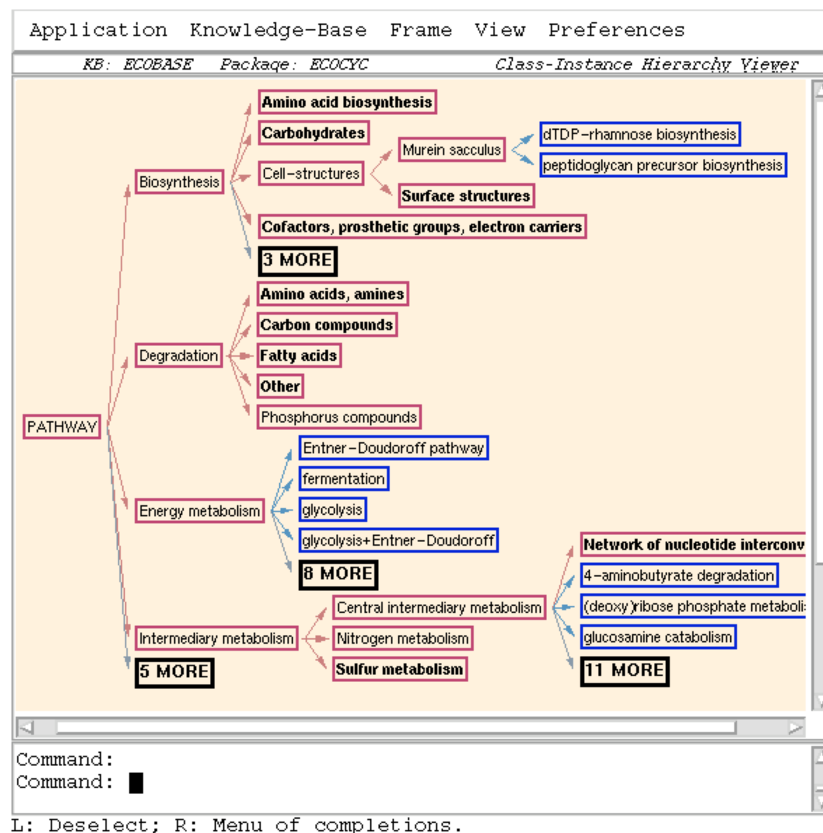

Abbildung 6: Class Hierarchy Viewer

1.2.2 LOOM Ontosaurus

Ontosaurus[6] ist ein Web-Browser für LOOM Wissensbasen. Ontosaurus visualisiert, wie der GKB-Editor, die Struktur des modellierten Wissens. Im Gegensatz zum GKB-Editor wird die Struktur nicht als Graph dargestellt, sondern hier macht man sich die Hyperlinks zu nutze, die wir von Internetbrowsern, wie Netscape oder MS Internet Explorer, kennen. Über diese Links kann man durch die Wissensbasis navigieren.

Der Browser besteht aus drei Hauptfenstern. Ein *Toolbar*-Fenster, das Elemente zur Bearbeitung der Wissensbasis beinhaltet, wie Speichern, Laden, Suchen, Editieren usw. Zwei weitere Fenster, *Reference* und *Content*, beinhalten die Daten der Wissensbasis. Diese Daten enthalten Links, über die man durch die Wissensbasis navigiert. Im

6 Der LOOM Web Browser wurde am Information Sciences Institute der University of Southern California erstellt.
http://www.isi.edu/isd/ontosaurus.html

Bookmark-Fenster, ein Unterfenster des *Reference*-Fenster, kann der Benutzer Referenzen auf Ontosaurus-Web-Seiten ablegen. Mit einem weiteren Unterfenster, *Bookmarkcontrol*, stehen dem Benutzer Möglichkeiten zur Bearbeitung seiner Ontosaurus-Bookmarks zur Verfügung. In Abbildung 7 ist die Benutzeroberfläche des Ontosaurus zu sehen.

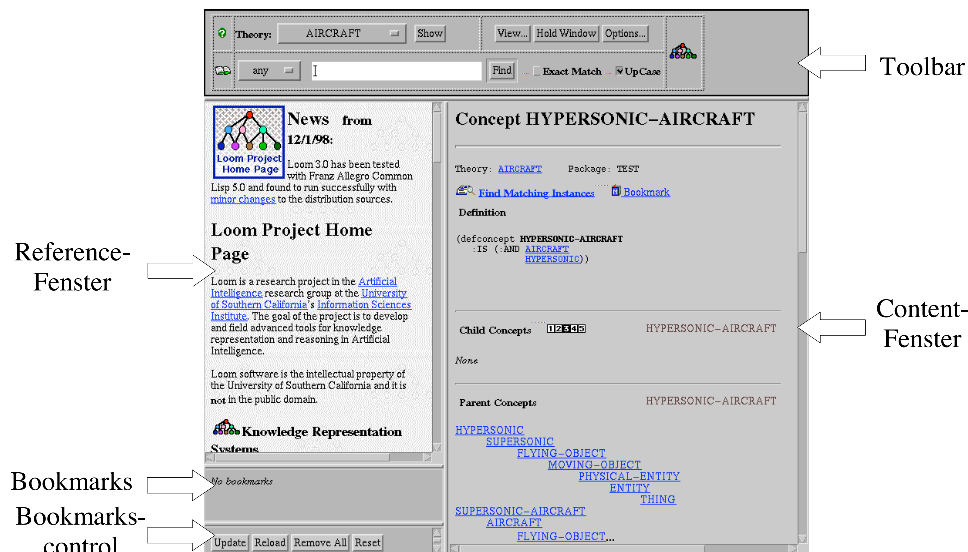

Abbildung 7: Ontosaurus

Der GKB-Editor gewährt stets einen Überblick über den gesamten bzw. einen Teil des Graphen des Modells, wohingegen Ontosaurus immer nur einen Knoten, mit Verzweigungen zu seinen Söhnen, darstellt.

1.3 Überblick

Zu Beginn eines jeden Kapitels wird der Inhalt seiner Anschnitte in einer kurzen Zusammenfassung wiedergegeben.

Im zweiten Kapitel werden drei Graphikwerkzeuge vorgestellt. Eines davon soll der Visualisierung der Büroumgebung dienen. Kapitel 3 widmet sich der Wissensrepräsentation. Hier werden die terminologischen Logiken und der Repräsentationsformalismus LOOM besprochen. Darauf basierend wird ein theoretisches Modell der Büroumgebung entwickelt. Kapitel 4 beschreibt die Netzwerkkomponente, über die die Wissensrepräsentationskomponente mit der Graphikkomponente verbunden wird. Ein entsprechendes Kommunikationsprotokoll wird aufgezeigt. Das Kapitel 5 behandelt den Zusammenschluß der Wissensrepräsentations- und Graphikkomponente zum Gesamtsystem. Kapitel 6 beinhaltet eine Bewertung des Gesamtsystems in Form von Messungen und Analysen. Desweiteren werden hier Perspektiven, Erweiterungsmöglichkeiten und mögliche Einsatzgebiete dargelegt. Eine abschließende Zusammenfassung bildet Kapitel 7. Hier werden die Kernaussagen der einzelnen Kapitel nochmals aufgegriffen. Kapitel 8 beinhaltet Beispielprogramme zu den Graphikwerkzeugen, terminologische Logiken, LOOM und zur Netzwerkkomponente.

2 Graphikwerkzeuge

Für das in dieser Arbeit zu erstellende Gesamtsystem ist ein adäquates Graphikwerkzeug auszuwählen, das den Anforderungen, auf die später eingegangen wird, gerecht wird. In diesem Kapitel werden nun drei Graphikwerkzeuge vorgestellt. Mit jedem dieser Werkzeuge können dreidimensionale Graphiken, teilweise mit der Möglichkeit der Benutzerinteraktion, erstellt werden.

Dabei sollen die drei Raumachsen des verwendeten Koordinatensystems, bzgl. des Betrachters, wie folgt definiert sein: Die positive x-Achse verläuft von links nach rechts, die positive y-Achse von unten nach oben und die positive z-Achse läuft auf den Betrachter zu, also von hinten nach vorne. Entsprechend wird auch die räumliche Anordnung der 3D-Objekte vorgenommen. Abbildung 8 zeigt die drei Raumachsen und einen Kegel. Seine Spitze ist „oben“ und er „steht“ auf der x ,z-Ebene.

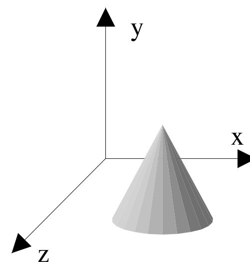

Abbildung 8: Lage der Raumachsen

Zu Beginn werden drei Werkzeuge, VRML 2.0, Open Inventor und OpenGL, nacheinander beschrieben[7]. Besonderheiten und Stärken des einzelnen Werkzeugs werden herausgearbeitet. In jedem dieser Abschnitte wird zu Beginn eine kurze Zusammenfassung des jeweiligen Werkzeugs gegeben. Im darauffolgenden Abschnitt werden die Anforderungen, die an das Graphiksystem gestellt werden, definiert. Eine tabellarische Gegenüberstellung der Werkzeuge wird dann, unter dem Aspekt der gestellten Anforderungen, ausgewertet und somit das am besten geeignete Werkzeug bestimmt.

2.1 VRML 2.0

VRML 2.0[8] ist ein auf ASCII basierendes Dateiformat, mit dem sich recht einfach dreidimensionale Szenen beschreiben lassen. Eine derartige Datei kann mit Hilfe eines externen Programms geladen und visualisiert werden. Verfügt dieses externen Programm, auch VRML-Browser genannt, über eine Anbindungen an Java und Javascript, so können in einer Szene Objekte dynamisch hinzugenommen und entfernt werden. In einem der folgenden Abschnitte ist diese Problematik Mittelpunkt der Diskussion.

7 Weitere Graphikwerkzeuge, wie MS Direct3D und Apple QuickTime3D standen nicht zur Auswahl, da das Gesamtsystem primär auf UNIX-Maschinen zu implementieren ist.

8 In dieser Arbeit wird nur VRML 2.0 betrachtet. VRML 1.0 bietet nur die Möglichkeit, statische Szenen zu erstellen und wurde deswegen nicht in Betracht gezogen.

Die geschichtliche Entwicklung von VRML 2.0, eine Beschreibung des Dateiformats und der VRML-Browser werden im folgenden Absatz gegeben.

VRML ist die Abkürzung für *Virtual Reality Modeling Language*. Die VRML 1.0 Spezifikation basiert auf dem Open Inventor Dateiformat und wurde von Silicon Graphics, Inc. entwickelt. Die zweite Spezifikation von VRML (VRML 2.0) wurde mit verbesserten Möglichkeiten der Benutzerinteraktion versehen. Außer einigen syntaktischen Gemeinsamkeiten sind VRML 1.0 und VRML 2.0 grundverschieden. Die Spezifikation wurde anfänglich vom Silicon Graphics VRML Team, Sony Research und einigen weiteren Gruppen entworfen. Weiter wurde dann VRML 2.0 von der *VRML email discussion group*, einigen Firmen und Einzelpersonen, dem VRML Konsortium, betreut. Daraus entstand VRML97, das schließlich VRML 2.0 ersetzte. VRML97 wurde als *International Standard ISO/IEC 14772* herausgegeben. Die Unterschiede zwischen VRML 2.0 und VRML97 sind von geringfügiger Natur, folglich werden beide meist als gleichwertig eingestuft.

Da die genaue Spezifikation des Dateiformates hier nicht von Interesse ist, wollen wir uns darauf beschränken, nur die Grundkenntnisse, um eine Szene beschreiben zu können, zu vermitteln. Die Beschreibung der Szene erfolgt in Form eines gerichteten azyklischen Graphen, dem VRML-Scene-Graph. Die Knoten des Graphen enthalten die Eigenschaften (Form, Farbe, Translation usw.) der 3D-Objekte bzw. weitere Knoten. Bei VRML werden die Eigenschaften der 3D-Objekte ebenfalls als Knoten bezeichnet. Desweiteren ist ein Knoten nicht von seinem Vorgängerknoten abhängig, d.h. er erbt nicht dessen Eigenschaften.

Aus Benutzer Sicht kann man eine Unterscheidung zwischen Knoten, die die Struktur des Graphen aufbauen, und Knoten, die die Eigenschaften eines 3D-Objekts beschreiben, treffen. In Abbildung 9 ist graphisch ein VRML-Scene-Graph, der einen blauen Würfel repräsentiert, dargestellt. Strukturbildende Knoten werden als Kreise gezeichnet und die weiteren Knoten, die die Eigenschaften eines Objektes beschreiben, sind durch Rechtecke gekennzeichnet. Knoten zur Strukturbildung sind in die *Transform*-Knoten und die *Shape*-Knoten. Alle anderen Knoten beschreiben die Eigenschaften des Objekts.

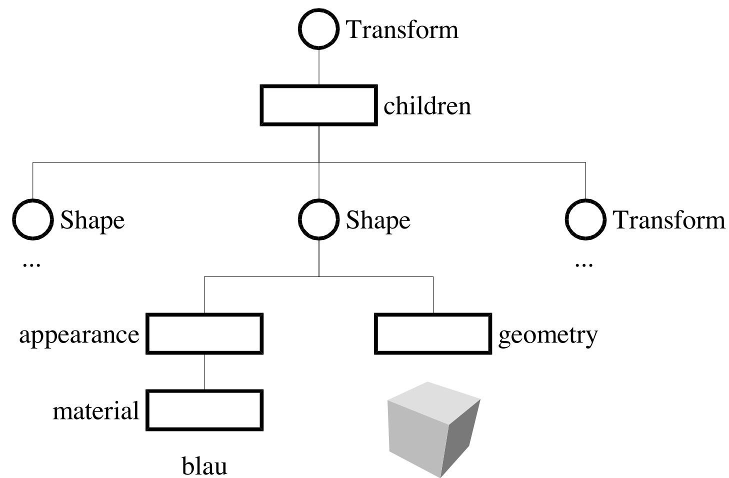

Abbildung 9: VRML-Scene-Graph für einen blauen Würfel, mit weiteren möglichen Knoten

VRML selber unterscheidet diese Knoten nicht, sondern jeder Knoten ist hier von einem bestimmten Typ, der wiederum nur bestimmte Typen anderer Knoten beinhalten darf. Synonym zur Bezeichnung Knoten wird in VRML der Begriff Feld verwendet. Auf die Beschreibung der Typen bzw. Felder wird im weiteren Verlauf genauer eingegangen. Zu diesem Zweck sehen wir uns die Textform des VRML-Scene-Graph aus Abbildung 9, die in Beispiel 1 angeben ist, näher an.

```
#VRML V2.0 utf8
Transform {
          children [
                    Shape { ...
                    }
                    Shape {
                              appearance Appearance {
                                        material Material {
                                                  diffuseColor 0 0 1
                                        }
                              }
                              geometry Box { }
                    }
                    Transform { ...
                    }
          ]
}
```

Beispiel 1: blauer Würfel

Die erste Zeile ist der Header der VRML 2.0 Datei, die jede VRML 2.0 Datei enthalten muß, [VRL 97]. *Transform* definiert einen neuen Wurzelknoten. In *children* sind seine Sohnknoten, auch Felder genannt, definiert. Hier können nun geometrische Objekte, mit

Shape eingeleitet, oder weitere Knoten, mittels *Transform*, angegeben werden. In unserem Beispiel wird hier mittels *Shape*, das auch ein Knoten ist, ein Würfel definiert. *Shape* enthält genau zwei Knoten, *appearance* und *geometry*. Die zu einem Knoten gehörigen Sohnknoten müssen nicht zwingend angegeben werden. Diese Knoten erhalten in diesem Fall ihren Standardwert. Würde man in unserem Beispiel den *appearance* Knoten weglassen, hätte das zur Folge, daß der Würfel in einem mittleren Grau erschiene. Mit *geometry Box {}* wird schließlich die Form des Würfels beschrieben. In den geschweiften Klammern bei *Box {}* kann noch die Ausdehnung des Würfels in x-, y- und z-Richtung (z.B. *size 2 2 2*) angegeben werden. Der Typ dieses Knotens ist SFVec3f. Was besagt, daß es sich hierbei um genau einen Vektor mit drei Gleitkommazahlen handelt. Weitere mögliche Datentypen sind SFFloat, genau eine Gleitkommazahl, MFInt32, beliebig viele ganze Zahlen, usw. Tabelle 1 gibt einen kleinen Einblick in die VRML 2.0 Spezifikation. In geschweiften Klammern werden die Typen der Felder und ihr Standardwert angegeben, z.B. *Sphere { SFFloat radius 1}*, steht für eine Kugel mit dem Durchmesser *1*. Sie hat als einziges Feld *radius* vom Typ *SFFloat* und dieser ist mit dem Wert *1* standardmäßig belegt. Eine detaillierte Auflistung aller Knoten und Typen ist in [VRL 97] aufgeführt.

Knoten	**Beschreibung**
Shape { SFNode appearance NULL SFNode geometry NULL}	hier wird ein Objekt mit definiertem Erscheinungsbild (appearance) und Form (geometry) angelegt;
Box {SFVec3f size 2 2 2}	legt einen Quader, mit Durchmesser x,y,z jeweils gleich 2, an;
Sphere { SFFloat radius 1 }	eine Kugel mit Radius 1 wird angelegt;

Tabelle 1: VRML-Knoten

Um obiges Beispiel 1 betrachten zu können benötigen wir noch ein spezielles Programm, das VRML 2.0 anzeigen kann. Für MS Windows sind die beiden gebräuchlichsten VRML-Browser[9] CosmoPlayer[10] und WorldView[11]. Sie sind aber nur als Plug-In[12] für Netscape bzw. für MS Internet Explorer realisiert. Beide stellen eine Sprachanbindung für Java und Javascript zur Verfügung. Der tiefere Sinn, der sich dahinter verbirgt, wird im nächsten Abschnitte diskutiert. Dadurch, daß beide Browser als Plug-In verfügbar sind, können VRML-Szene direkt in Web-Seiten integriert werden und Szenen aus dem Netz können sofort betrachtet werden. Mit beiden ist es dem Benutzer möglich, sich frei in der Szene zu bewegen. Die Szene kann gedreht, gezoomt und bewegt werden. Sie ist aber völlig statisch, in dem Sinne, daß der Benutzer einzelne Objekt zwar bewegen und drehen darf, aber keine Objekte hinzunehmen bzw. entfernen kann.

9 Die meisten VRML-Browser bilden ihre Graphikroutinen auf OpenGL ab.
10 CosmoPlayer, ehemals von Silicon Graphics, Inc., jetzt Platinum Technologie, Inc.: Im Internet: http://www.cosmosoftware.com
11 WorldView, ehemals von InterVista jetzt Platinum Technologie, Inc.: Im Internet: http://www.intervista.com
12 Ein Plug-In ist eine nachladbare Funktionsbibliothek, die die Funktionalität eines Browsers zur Darstellung eines nicht direkt unterstützten Datenformates erweitert.

Abbildung 10 und 11 zeigen die zwei Varianten der Konsole[13] des CosmoPlayer[14] unter Netscape. Mit der ersten Form kann sich der Benutzer durch die Szene bewegen. Er verändert hier die Position und Richtung seines Blickpunktes bzw. Blickrichtung.

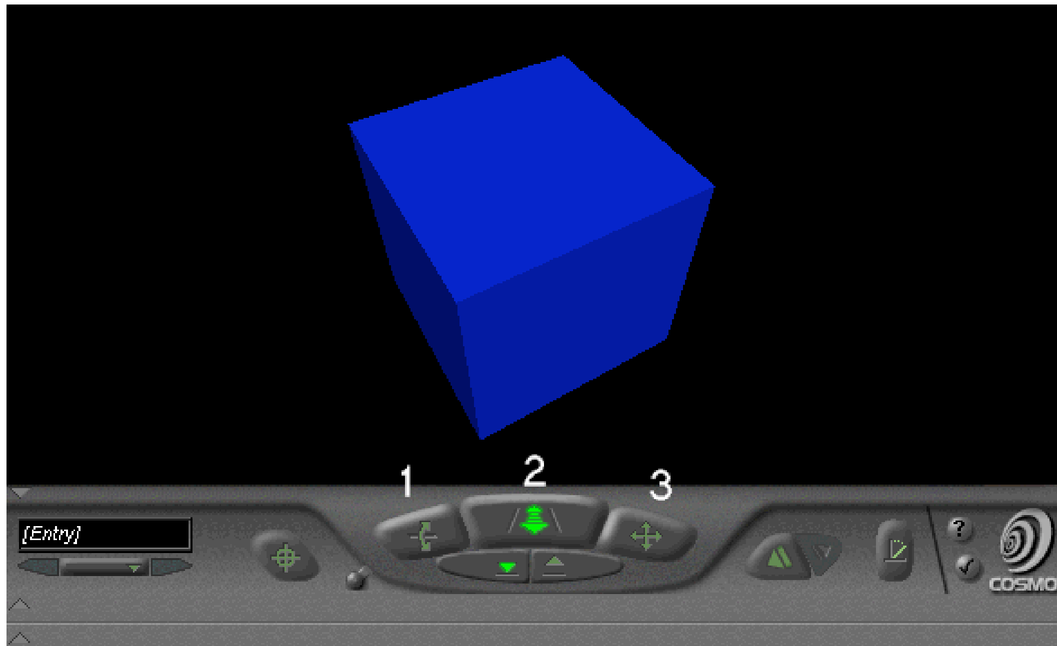

Abbildung 10: 1. Form des Bedienfeldes

Mit Button 1 dreht sich der Betrachter in der Szene in eine gewünschte Richtung. Button 2 bewegt den Betrachter zur Szene hin bzw. von ihr weg. Der 3. Button verändert die Position des Betrachters in y- bzw. x-Richtung.

Die zweite Form erlaubt, aus Benutzersicht, bei festem Blickpunkt, die Szene zu verschieben und zu drehen.

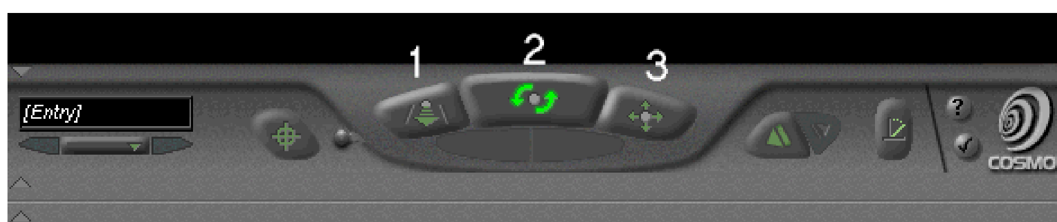

Abbildung 11: 2. Form des Bedienfeldes

Button 1 bewegt die Szene zum Betrachter hin bzw. von ihm weg. Mit Button 2 kann die Szene beliebig gedreht werden. Button 3 schließlich verschiebt in y- bzw. x-Richtung.

Andere VRML-Browser, auch für UNIX-Systeme, bieten bisher keine Sprachanbindung. Weiterhin unterstützen diese VRML-Browser meist nur einen Teil der

13 Die Konsole befindet sich am Unteren Fensterrand, wobei hier nur die drei großen mittleren Buttons von Interesse sind.
14 Abbildung 10 zeigt den oben definierten blauen Würfel, betrachtet von der linken, oberen Ecke.

VRML 2.0-Spezifikation. Hier sei VRwave[15] genannt der nahezu VRML 2.0 vollständig abdeckt und für die meisten Plattformen verfügbar ist. Im weiteren Verlauf gilt das Interesse den 3D-Objekten und den Eigenschaften, die man ihnen zuweisen kann.

Objekte können so definiert werden, daß sie später vom Benutzer separat angewählt, bewegt und rotiert werden können. Ebenso wird die Möglichkeit, Animationen zu erstellen, angeboten. Ein kleines Demonstrationsprogramm zu VRML 2.0 ist im Anhang aufgeführt (*8.1 VRML 2.0 Beispiel*). Es definiert vier blaue Symbole und einen roten Würfel. Das Symbol rechts oben dreht sich um seine linke untere Ecke (Animation). Wenn man den Würfel mit der linken Maustaste anklickt und die Maustaste gedrückt hält, kann man den Würfel drehen. Gleichzeitig drehen sich auch die vier Symbole gemeinsam mit.

Eine in VRML 2.0 beschriebene Szene legt nur die im VRML-Scene-Graph definierten Objekte an, d.h. ein dynamisches Erzeugen und Entfernen der Objekte wird vom VRML-Scene-Graph nicht unterstützt. Wird eine entsprechende Dynamik gewünscht, so muß dies über eine Sprachanbindung an Java oder Javascript geschehen. Nur dadurch erhält hat man die Möglichkeit, dynamisch 3D-Objekte anzulegen bzw. zu entfernen. Erst in diesem Fall kann ein Benutzer in die Objektwelt der Szene aktiv eingreifen. Realisiert wird dies durch einen speziellen Knoten, dargestellt in Tabelle 2.

```
Script { exposeField   MFString url []
         field         SFBool directOutput FALSE
         field         SFBool mustEvaluate FALSE

         eventIn       evenType eventName
         field         fieldType fieldName initialValue
         eventOut      evenType eventName}
```

Tabelle 2: Skriptknoten

Im Feld url[16] wird eine Datei spezifiziert, die sich auf einem speziellen Server befindet. Ebenso wird damit das Protokoll spezifiziert, über das auf diese Datei zugegriffen wird. Hinter der Datei verbirgt sich das Skript, das ausgeführt werden soll. Das Skript seinerseits muß VRML-Syntax erzeugen, so daß Objekte dynamisch erzeugt werden können. Über die weiteren Felder können dem Skript noch Parameter übergeben und Rückgabewerte entgegengenommen werden. Da die, zur vorliegenden Arbeit parallel durchgeführte Arbeit [Fie 99] schon früh zeigte, daß die dynamisch Objektgenerierung extrem langsam und somit für diese Arbeit nicht ausreichend schnell ist, wurden weitere Untersuchungen von VRML 2.0 eingestellt.

15 Vrwave © 1996-1997 IICM; ist ein in Java geschriebener VRML 2.0-Browser. Im Internet zu finden unter: http://www.iicm.edu/vrwave

16 URL: IETF RFC 1738 Uniform Resource Locator, Internet standards track protocol http://ds.internic.net/rfc/rfc1738.txt

2.2 Open Inventor

Mit Open Inventor[17], das auf OpenGL basiert, lassen sich Programme mit 3D Benutzerschnittstelle realisieren. Das Augenmerk liegt hier auf der Interaktion mit dem Benutzer. In Befehlsbibliotheken werden die Kommandos bereitgestellt. Open Inventor ist objektorientiert, bietet aber auch eine C-Schnittstelle. Szenenbeschreibungen können auch in einem speziellen Dateiformat abgelegt werden.

Objekte einer Szene werden bei Open Inventor, ähnlich zu VRML 2.0, in einem Szenengraphen angeordnet. Dabei können mehrere Szenengraphen parallel existieren, die in einer Szenendatenbank verwaltet werden. Wie bei VRML 2.0 wird für ein 3D-Objekt ein Knoten angelegt, welcher zugehörige Attribute wie Farbe, Form, Textur, Position im Raum usw. enthält. In Abbildung 12 ist ein Szenengraph dargestellt[18]. Hier wird das Kalottenmodell eines Wassermoleküls beschrieben, wie man es aus dem Chemieunterricht kennt. Der erste Knoten, die Wurzel, definiert ein neues Objekt, namens *Wassermolekül*. Sollen weitere Wassermoleküle angelegt werden, so geschieht dies über dieses neu definierte Objekt. Wie sich im weiteren Verlauf zeigen wird, spielt die Reihenfolge der Knoten, von oben nach unten, von links nach rechts, eine entscheidende Rolle für eine spezielle Art der Vererbung, auf die wir später zu sprechen kommen. Der erste Knoten in der zweiten Ebene legt ein neues Objekt *Sauerstoff* an. Es hat eine kugelförmige Form und ist von roter Farbe. Das Objekt liegt im Ursprung des globalen Koordinatensystem und besitzt die Standardausdehnung einer Kugel, d.h. der Radius soll z.B. gleich 1.0 sein. Der zweite Knoten beschreibt ein weiteres Objekt, *Wasserstoff_1*. Die Form entspricht wieder einer Kugel, die Farbe ist diesmal weiß. In *Transform.* wird die Position (z.B. x=0.8, y=0.8, z=0.0) und Skalierungsfaktor (z.B. für x, y, z jeweils 0.75) dieser Kugel angegeben. Der letzte Knoten definiert ein Objekt *Wasserstoff_2*. Wieder ist die Form eine Kugel, die Position aus *Transform.* soll jetzt z.B. x=-0.8, y=0.8, z=0.0 sein. Der Skalierungsfaktor und die Farbe werden vom vorherigen Objekt (*Wasserstoff_1*) geerbt. Die Form der Vererbung ist in dem Sinn ungewöhnlich, daß ein Knoten nicht unbedingt von einem hierarchisch höheren Knoten erbt, sondern von seinem unmittelbaren Vorgängerknoten.

In textueller Form, sei es als C++ Programm oder Inventor Dateiformat, wird der Szenengraph, Ebene für Ebene, Knoten für Knoten, abgebildet. Da jeder Knoten Daten von seinen Vorgängern erbt, ist die Reihenfolge, in der die Knoten definiert werden, zu beachten.

17 Open Inventor ist eine Entwicklung von Silicon Graphics, Inc.
18 Der Szenengraph ist eine modifizierte Version dessen aus [Wer 94], Seite 47.

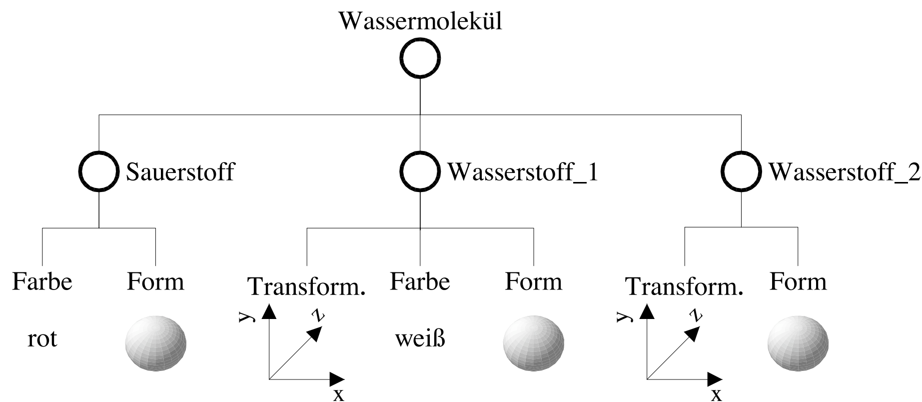

Abbildung 12: Open Inventor Szenengraph für ein Wassermolekül

Mit dem Szenengraph kann man sich so einen ganzen Chemiebaukasten mit den verschiedensten Molekülen definieren. Szenen werden nach dem Baukastenprinzip erstellt, d.h. vorgefertigte Objekte können nach Belieben zusammengesetzt werden.

Ebenso stellt Open Inventor dem Benutzer bereits definierte Elemente zur Benutzerinteraktion zur Verfügung. So können zu Objekten sog. Manipulatoren definiert werden, mit denen das Objekt gedreht, positioniert oder in seiner Größe verändert werden kann. Wählt man nun so ein Objekt mit der Maus an, wird der entsprechende Manipulator sichtbar und man kann das Objekt, wieder mit der Maus, bearbeiten. In *8.2 Open Inventor: Beispiel zu C++*, im Anhang, ist ein derartiges Objekt definiert. Hier wird ein roter Kegel angelegt, der mit der Maus gedreht werden kann. Um nun den Kegel drehen zu können, wird ein Manipulator definiert. Er zeigt sich in der Szene als drei Ringe um den Kegel. Wird ein einzelner Ring mit der linken Maustaste angeklickt und gehalten, so wird das Objekt mit Manipulator um die Achse, die durch den Mittelpunkt des Rings verläuft, senkrecht zum Ring, gedreht. Beim Anklicken des Schnittpunkts zweier Ringe, wird das Objekt und der Manipulator beliebig gedreht. Zu sehen ist die Szene in Abbildung 13.

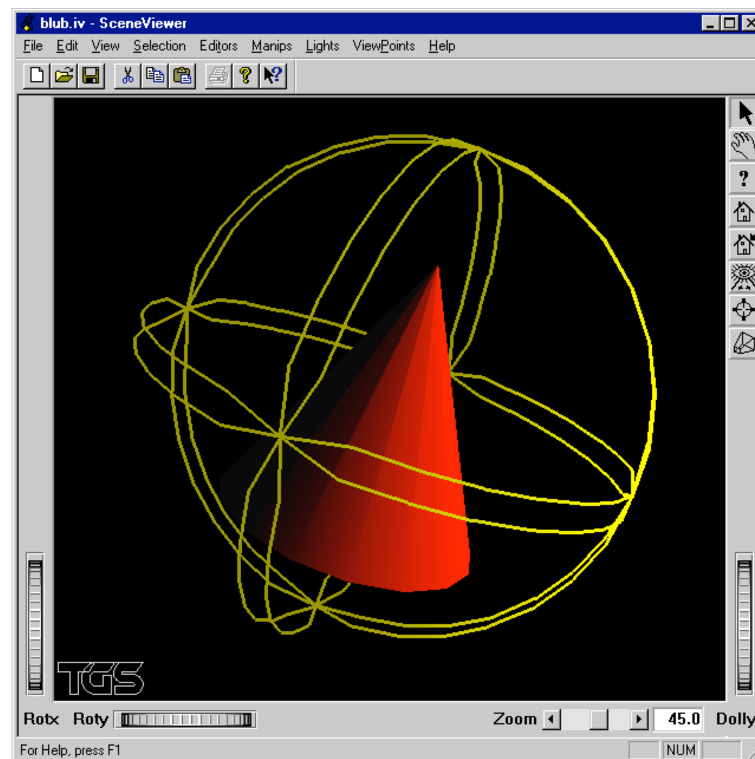

Abbildung 13: Manipulator

Wie weiter oben erwähnt, bietet Open Inventor ein Dateiformat, in welchem Szenenbeschreibungen vorgenommen werden können. In *8.3 Open Inventor: Beispiel zum Dateiformat*, im Anhang, ist ein Auszug aus einer Datei angegeben. Definiert wird eine Windmühle, deren Flügel sich drehen[19].

Die eigentliche Aufgabe von Open Inventor gilt der Verwaltung der Szenengraphen. Die Berechnung der Szene wird von OpenGL übernommen. Das Zeichnen der Szene ist aber optional.

2.3 OpenGL

OpenGL stellt eine Schnittstelle zum Erstellen und Bearbeiten dreidimensionaler Graphik zur Verfügung. Alle entsprechenden Kommandos sind in einer Befehlsbibliothek zusammengefaßt. Die Anbindung erfolgt i.A. über die Programmiersprachen C/C++, wobei OpenGL selber nur eine C-Schnittstelle bietet. Benutzereingaben werden von OpenGL selber nicht verarbeitet. OpenGL berechnet die Szene zwar, sie liegt dann als Bitmapgraphik im Speicher vor, bietet aber keine Funktionen, diese auf dem Bildschirm auszugeben. Deshalb müssen Ausgabe der Graphik und Benutzerinteraktion über externe Softwarekomponenten, auf die später eingegangen wird, bereit gestellt werden.

OpenGL ist ein weitverbreiteter Graphikstandard, entwickelt von Silicon Graphics, Inc. Folgende Leitlinien standen bei der Entwicklung im Vordergrund. Das Graphiksystem soll plattformunabhängig sein, somit ist die Implementierung der Applikation nicht von

19 Das Beispiel ist aus [Wer 94] Seite 7 entnommen.

der vorhandenen Graphikhardware abhängig. Die Abbildung auf die Graphikhardware ist allein Aufgabe der jeweiligen OpenGL Implementierung. Fensterverwaltung und die Verarbeitung von Benutzereingaben werden nicht bereit gestellt, somit kann sich die Implementierung von OpenGL „allein" mit der Berechnung dreidimensionaler Szenen beschäftigen. Weiterhin werden nur geometrische Grundobjekte[20] (Punkte, Linien, Polygone) und 3D Standardoperationen (perspektivische, parallele Projektionen; Lichtquellen; Texturen, Antialiasing (Zeichnen der Kanten ohne „Treppeneffekt") und atmosphärische Effekte) angeboten. Komfortablere Applikationen zum Erstellen von 3D Szenen können darauf aufsetzen.

Dem Programmierer zeigt sich demnach OpenGL bezüglich der angebotenen geometrischen Grundobjekte recht spartanisch. Komplexe Objekte können nur durch Zusammenfügen mehrerer Grundobjekte entstehen. Attribute, wie Form, Farbe, Textur, Position usw. können, im Gegensatz zu VRML und Open Inventor, einem Objekt nicht direkt zugeordnet werden. Im Programmcode spiegelt sich ein Objekt mit seinen Attributen nur als eine Folge von Funktionsaufrufen wider. Eine Graphstruktur, wie sie bei den beiden vorherigen Graphikwerkzeugen zu sehen ist, ist somit nicht gegeben.[21] Eine zusätzliche Bibliothek (GLU: OpenGL Utility Library), die bei jeder OpenGL-Implementierung enthalten ist, bietet hier eine kleine Erleichterung. Einige geometrische Grundobjekte (Zylinder, Kugel, Kegel und einige andere) sind hier bereits vorgefertigt. Ebenso enthält GLU Funktionen die einige „Kameraeinstellungen" (Art der Projektion, Blickpunkt des Betrachters) vereinfachen.

OpenGL ist rein auf die Berechnung dreidimensionaler Graphik ausgelegt. Deshalb finden sich, wie oben erwähnt, in OpenGL weder Funktionen zur Fenstersteuerung noch zur Benutzerinteraktion. Eine weitere Bibliothek (GLUT[22]: OpenGL Utility Toolkit, [Kil 96]) vereint diese beiden Aspekte. Sie ist unabhängig von OpenGL und für unterschiedliche Plattformen erhältlich. Eine weitere Bibliothek, basierend auf GLUT ist GLUI[23], [Rad 98]. Mit ihr können übliche (2D) Benutzerschnittstellen generiert werden. Weitere Bibliotheken, die sich ausschließlich mit der Fenstersteuerung beschäftigen sind *glX* ([Kil 94]) für UNIX/XWindows, *wgl* für MS Windows, *pgl* für IBM OS/2 und *agl* für Apple Macintosh.

In den beiden folgenden Abschnitten wird OpenGL näher beleuchtet. Zuerst wird das Profil eines möglichen OpenGL Programms beschrieben und anschließend wird die Arbeitsweise, wie OpenGL bei der Berechnung der Szene vorgeht, betrachtet.

20 Diese geometrischen Grundobjekte lassen sich meist gut auf die Graphikhardware abbilden.

21 Man spricht bei OpenGL auch von einer sog. Zustandsmaschine, d.h. Eigenschaft, wie Farbe, Rotation, Skalierung usw., die einmal gesetzt wurden gelten solange bis sie überschrieben werden. Will man z.B. drei rote Kugel haben, so muß die Farbe für die Oberfläche der Kugeln nur einmal bei der ersten Kugel gesetzt werden alle weiteren Kugeln erben dann die Farbe.

22 OpenGL Utility Toolkit (GLUT) ist eine Bibliothek für eine system-unabhängige Fenstersteuerung und Benutzerinteraktion. Sie wird von Mark Kilgard entwickelt und ist im Internet unter http://reality.sgi.com/mjk erhältlich.

23 OpenGL User Interface Library (GLUI) ist eine system-unabhängige Bibliothek für die Erstellung graphischer Benutzerschnittstellen (Buttons, Textfelder usw.) von Paul Rademacher.

2.3.1 Profil eines möglichen OpenGL Programms

Richtig wäre es von einem Programm zu sprechen, das OpenGL verwendet. Vorgestellt wird nämlich ein Programm, das eine mit OpenGL erstellte Szene auf dem Bildschirm darstellt und auf Benutzereingaben reagiert. Wie bereits oben erwähnt unterstützt OpenGL weder Bildschirmausgaben noch Benutzerinteraktionen. Hier finden die oben angegebenen Bibliotheken GLUT, glX, wgl usw. Verwendung. Bei der Beschreibung des Programms soll aber nicht auf die Ebene des Programmcodes gegangen werden, sondern es wird die Funktion einzelner Abschnitte beschrieben, in welche sich das Programm aufgliedern läßt. Nachfolgende Liste enthält jeweils fettgedruckt diese Abschnitte mit Beschreibung. Detaillierte Informationen zu OpenGL sind in [WND 97] nachzulesen.

1. **Erzeugen und Initialisieren eines Fensters auf dem Bildschirm;**
   Zu Beginn des Programms wird ein Fenster geöffnet und initialisiert, in welches die graphische Ausgabe erfolgt. Wie bereits erwähnt, stellt OpenGL derartige Routinen nicht bereit. Eine andere Bibliothek muß also diese Funktionalität zur Verfügung stellen. Genannt seien hier *glX* für Unix/XWindows, *wgl* für MS Windows, *pgl* für OS/2 und *agl* für Apple Macintosh. In unserem Fall wird GLUT zur Fenstersteuerung verwendet. GLUT ist sowohl für Unix/XWindows als auch für MS Windows verfügbar. Die Plattformunabhängigkeit für den OpenGL-Teil des Programms ist stets gewährleistet. Mit GLUT ist sogar das gesamte Programm auf den beiden System Unix/XWindows und MS Windows lauffähig.

2. **Fensterinhalt löschen, bzw. mit definierter Farbe füllen;**
   Das Fenster wird zum Zeichnen vorbereitet - die Szene erhält die angegebene Farbe als eine Hintergrund.

3. **Allgemeine Initialisierung vornehmen;**
   Hier werden Einstellungen vorgenommen, die die Darstellung der Szene beeinflussen. Folgende Liste enthält eine Zusammenfassung möglicher Parameter.
   - Transparenz (α-Blending): hier kann man festlegen ob Objekte transparent oder stets gefüllt gezeichnet werden;
   - Z-Buffer: sollen die Objekte ihrer Anlegereihenfolge nach oder ihrer räumlichen Tiefe nach, verdeckend gezeichnet werden;
   - Lichtquelle: Festlegen, ob und welche Lichtquellen zugelassen sind
   - Face-Culling: Festlegen, welche Seiten eines Polygons gezeichnet werden sollen, beide oder nur Vorder- bzw. Rückseite;
   - Doublebuffer: soll für das Zeichnen der Szene die Technik des Doublebuffering Verwendung finden?, d.h. die Graphik der Szene wird zweimal im Speicher gehalten, während die eine Graphik gezeigt wird, wird die neue Graphik berechnet, dadurch wird stets die vollständig gerechnete Szene angezeigt; anderfalls, beim Singlebuffering, sieht man den Aufbau der Szene;

4. **Kameraeinstellungen vornehmen;**
   Die Position des Betrachters, seine Blickrichtung und seine Entfernung zur Szene, werden hier festgelegt. Analog zu einer echten Kamera ist alles, vom Zoom bis zum Tele, wie auch Weitwinkel, möglich (siehe [WND 97], S. 91-137).

5. **Lichtquellen anlegen;**
   Soll die Szene beleuchtet werden, so können bis zu acht unabhängige Lichtquellen definiert werden. Die Art der Lichtquelle kann punktförmig oder ein Scheinwerferlicht (kegelförmig) sein. Die Farbe des Lichtes kann frei definiert werden. Die Intensitätsabnahme des Lichtes läßt sich mit zunehmender Entfernung als konstant, linear oder quadratisch definieren. OpenGL bietet noch weitere Effekte, wie Nebel usw., die in [WND 97], S. 169-211, detailliert beschrieben werden.

6. **Farbe bzw. Textur festlegen;**

7. **Objekt definieren;**
   Zusammensetzen eines 3D-Objekts aus OpenGL Grundobjekten. Dieses Objekt erhält die zuvor definierte Farbe, Textur bzw. Oberflächeneigenschaften.
   Punkt 6. und 7. werden so oft wiederholt, bis alle Objekte, die die Szene definieren, angelegt wurden. Haben mehrere Objekte die selbe Oberfläche bzw. Materialeigenschaft, so muß Punkt 6. nur einmal zu Beginn angegeben werden.

8. **Szene zeichnen;**
   Diese Anweisung veranlaßt OpenGL, die Berechnung der Szene zu starten. Nach Beendigung liegt die Szene als Bitmapgraphik im Speicher vor und GLUT übernimmt die Darstellung der Graphik auf dem Bildschirm. Hierfür generiert GLUT ein Fenster, in das die Graphik gezeichnet wird.

9. **Warten auf Benutzerereignisse und ggf. Bild neu zeichnen;**
   OpenGL stellt hierfür keine Befehle zur Verfügung. Die unter Punkt 1. erwähnten Bibliotheken kommen auch hier zum Einsatz. Wiederum ist GLUT die Bibliothek der Wahl (siehe [Kil 96], S. 16-27).

2.3.2 Arbeitsweise von OpenGL

In Punkt 8. spielt sich der eigentliche Kern von OpenGL ab. Hier erfolgt eine mathematische Beschreibung der Szene. Folgende Liste zeigt die Phasen, die OpenGL durchläuft, bis das endgültige Bild fertig ist.

1) Aus einfachen Grundobjekten (Punkte, Linien, Polygone) generiert OpenGL komplexe Objekte. Die Objekte liegen in einer mathematischen Form vor.

2) Jetzt werden die Objekte im dreidimensionalen Raum angeordnet, d.h. OpenGL führt die dem Objekt zugeordnete Transformation durch (drehen, bewegen). Der Standort des Betrachters und seine Blickrichtung werden angelegt. Damit wird der, perspektivische oder parallele, Sichtbereich auf die Szene definiert.

3) Nun wird die Farbgebung jedes Objektes berechnet. Die Farbe eines Objektes setzt sich aus seiner vorgegebenen Farbe bzw. Textur, den vorhandenen Lichtquellen und den atmosphärischen Effekten (Nebel, Tiefenunschärfe) zusammen.

4) Die mathematische Beschreibung der Szene wird jetzt in Pixel, entsprechend einer Farbtiefe, umgesetzt. Das fertige Bild liegt nun als Bitmap im Speicher vor.

Eine zusätzliche Bibliothek (GLUT, glX, agl, usw.) übernimmt nun die Darstellung der in Punkt 4) generierten Graphik.

2.4 Anforderung an das Graphikwerkzeug

In diesem Abschnitt werden die Aufgaben, die das Graphikwerkzeug bewältigen sollte, gestellt. Die wichtigsten Eigenschaften, die das Werkzeug bieten soll, sind in folgender Liste aufgeführt.

- Das Werkzeug sollte einfach in der Handhabung sein. Es soll gut erlernbar sein, d.h. Szenen können schnell und mit geringem Aufwand erstellt werden. Die Definition der Objekte ist einfach. Attribute wie Form, Farbe, Position, Drehung können direkt zum Objekt angegeben werden.
- Die Möglichkeit, Benutzereingaben zu bearbeiten, soll bereits das System bereit stellen. Ob ein Objekt mit der Maus zu manipulieren ist, soll über eine Eigenschaft des Objekts geregelt sein, die ein- bzw. ausgeschaltet werden kann.
- So sich Objekte in der Szene änderen, übernimmt das System das erneute Zeichnen der Szene automatisch, d.h. das System verfügt über eine Komponente für die Bildschirmausgabe.
- Das System soll Rückmeldung darüber geben, ob sich Objekte in der Szene durchdringen, d.h. eine Komponente für Kollisionserkennung sollte bereits integriert sein.
- Objekte müssen dynamisch anzulegen und zu entfernen sein.
- Der Benutzer muß mit dem System flüssig arbeiten können. Der Zeitaufwand Objekte anzulegen und zu löschen sollte möglichst gering sein. Die dadurch neu entstandene Szene sollte ebenfalls schnell berechnet und gezeichnet werden.
- Das System selber muß stabil laufen und robust gegenüber Benutzerfehlern sein.

2.5 Die drei Graphikwerkzeuge im Vergleich

Im vorhergehenden Abschnitt wurden die Anforderungen an das Graphikwerkzeug vorgestellt. Unter diesem Aspekt werden nun die drei Werkzeuge verglichen. Tabelle 3 enthält die Eigenschaften der jeweiligen Werkzeuge.

Eigenschaft	VRML	Open Inventor	OpenGL
Sprache	Dateiformat, VRML-Scene-Graph; Anbindung an Java/Javascript möglich	C++, auch C	C/C++
vordefinierte Objekte	ja	ja	nein, nur mit GLU und GLUT

Eigenschaft	VRML	Open Inventor	OpenGL
Benutzerinteraktion	ja	ja	nein
eigenständige Darstellung der Szene	nur mit VRML-Browser	ja	nein; nur mit GLUT, glX, wgl, pgl und agl
dynamische Objektgenerierung	nein, nur über Anbindung Java/Javascript	ja	ja
benutzerfreundlich	ja	ja, über das Dateiformat, sonst nicht	nein
Objektrepräsentation bzw. -verwaltung	VRML-Scene-Graph	Szenengraph und Szenendatenbank	keine
Vererbung der Eigenschaft und Attribute vom Vorgänger	nein	ja	ja
Hardwareunterstützung	ja, bilden Graphikaufrufe meist auf OpenGL ab	ja, bilden Graphikaufrufe auf OpenGL ab	ja, direkt
Geschwindigkeit, Objekte dynamisch anzulegen und zu entfernen	langsam	schnell	schnell
Kollisionserkennung	nein	nein	nein
frei erhältlich	ja	nein	ja
Plattformverfügbarkeit	externe VRML-Browser für alle Plattformen, mit Java/Javascript Anbindung nur für MS Windows und Silicon Graphics Workstations	alle Plattformen	alle Plattformen

Tabelle 3: Gegenüberstellung der drei Graphikwerkzeuge

Open Inventor schneidet hier am besten ab, wurde aber aus folgenden Gründen trotzdem nicht verwendet: Open Inventor ist nicht frei erhältlich und muß lizensiert werden.

Darüber hinaus sind Runtimelizenzen zu entrichten. Weiterhin war zu Beginn der Diplomarbeit weder Literatur bzgl. Open Inventor noch das Produkt selber verfügbar. Eine vernünftige Einarbeitung war deshalb nicht möglich und das Produkt wäre auch zu kostenintensiv gewesen.

VRML schied aus, da das dynamische Löschen der Objekte zu langsam ist. Detaillierte Angaben sind in der Diplomarbeit von S. Fiedler [Fie 99] nachzulesen. Außerdem laufen wenigstens die hier betrachteten VRML-Browser nur eine gewisse Zeit stabil. VRwave stürzt bei größeren VRML-Welten einfach ab. CosmoPlayer und InterVista, jeweils als Netscape-Plugin, stürzen meist nach dem Anzeigen mehrerer VRML-Welten samt Netscape ab.

OpenGL hat zwar nicht den Komfort der Objektverwaltung, wie bei VRML und Open Inventor, läuft dafür aber stabil und ist schnell beim Anlegen und Löschen von Objekten. Ebenso bietet OpenGL selber keine Möglichkeit Benutzereingaben zu verwalten, so wie es VRML und Open Inventor bereit stellen. Dies läßt sich aber über zusätzliche Bibliotheken (GLUT, GLUI) lösen. Dafür ist aber für die meisten Plattformen ein Freeware-OpenGL samt den optionalen Bibliotheken verfügbar, was letztlich dazu führte OpenGL als Graphikwerkzeug zu verwenden.

Leider stellt keines der drei Werkzeuge eine Komponente für Kollisionserkennung bereit. Eine zusätzliches Softwarepaket, das im nächsten Abschnitt beschrieben wird, mußte deshalb hinzu genommen werden.

2.6 Kollisionserkennung

In diesem Abschnitt wird ein Sofwarepaket - V-Collide - zur Kollisionserkennung vorgestellt. Der Abschnitt ist wie folgt aufgegliedert: zunächst wird die Idee von V-Collide und dem darunterliegenden RAPID kurz vorgestellt (detaillierte Information zu V-Collide in [HLC 98] und RAPID in [KGL 98]). Danach erfolgt eine Beschreibung wie mit V-Collide eine Kollisionserkennung für OpenGL realisiert wird.

2.6.1 V-Collide

Jedes 3D-Objekt wird in V-Collide durch eine Menge von Dreiecken, die die Form des Objekts bestimmen, und einer Transformationsmatrix, die die Position und die Drehung des Objekts beschreibt, repräsentiert. Die Erkennung von Kollisionen zwischen den Objekten erfolgt in V-Collide [HLC 98] hierarchisch, gestaffelt in mehrere Stufen. Nach dem *n-body sweep-and-prune* Algorithmus aus I-Collide [CLM 95] werden zuerst Paare von Objekten identifiziert, die möglicherweise an einer Kollision teilnehmen. Diese Paare von Objekten werden dann mit RAPID [KGL 98] auf tatsächliche Kollision überprüft.

Die oben angegebenen Schritte werden nun nacheinander betrachtet.

1) Der *n-body sweep-and-prune* Algorithmus
   Die erste Stufe von V-Collide berechnet für jedes Objekt der Szene einen umschließenden, an den Achsen ausgerichteten Quader (AABB: axis-aligned bounding box).

Die Endpunkte dieser Quader werden in drei Listen, für jede Achse eine, aufsteigend sortiert, gehalten. Wenn sich Objekte bewegen, müssen die Listen neu sortiert werden. Da sich aber bei einer Animation die Objekt von Einzelbild zu Einzelbild i.A. nur wenig bewegen, wird für das Sortieren *Direktes Einfügen* verwendet. Nur die Paare von Objekten, deren AABBs sich in allen drei Dimensionen überlappen, werden an die nächste Stufe weitergereicht. In Abbildung 14 wird obiger Algorithmus für den zweidimensionalen Raum veranschaulicht. Es sind vier Objekte zu sehen, wobei Objekt 2 und 3 sich überschneiden. Nur die Objekte überlappen sich, deren jeweiligen Abschnitte sich in x- und y-Richtung überschneiden.

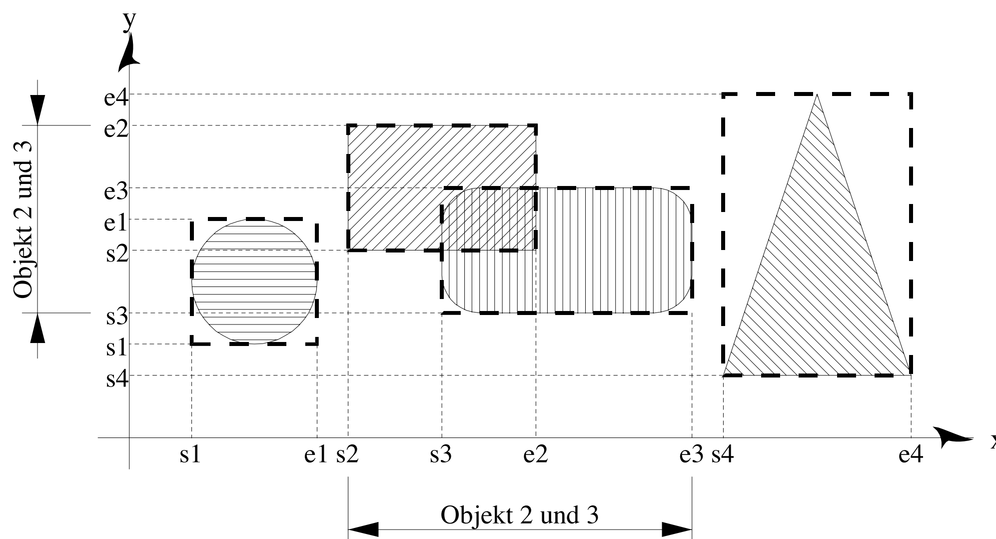

Abbildung 14: sweep and prune für 2D Objekte

2) RAPID: Berechnung optimal umschließender Quader
In dieser Stufe wird für jedes Objekt eines solchen Paares ein Baum optimal umschließender Quader mit zugehörigem Richtungsvektor berechnet. Ein derartiger Quader wird als OBB (oriented bounding box) bezeichnet. Die Wurzel des Baumes ist ein Quader, der das gesamte Objekt enthält. Die Sohnknoten sind weitere OBBs, die Objektteile (i.A. Dreiecke) des gesamten Objekts enthalten. Sie sind die Blätter des Baums.

3) RAPID: Finden der überschneidenden OBBs eines Objektpaars
In diesem Schritt werden auf Blattebene die OBBs des Baumes des ersten Objekts mit den OBBs des Baumes des zweiten Objekts auf Kollision untersucht. Die Paare von überlappenden OBBs werden an die nächste Stufe weitergereicht.

4) RAPID: exakte Kollisionserkennung
Im letzten Schritt werden die Dreiecke der überlappenden OBB-Paare genau auf Überschneidung untersucht.

5) Rückmeldung welche Objektpaare an einer Kollision beteiligt waren.

In Abbildung 15 wird der Ablauf der Kollisionserkennung graphisch dargestellt.

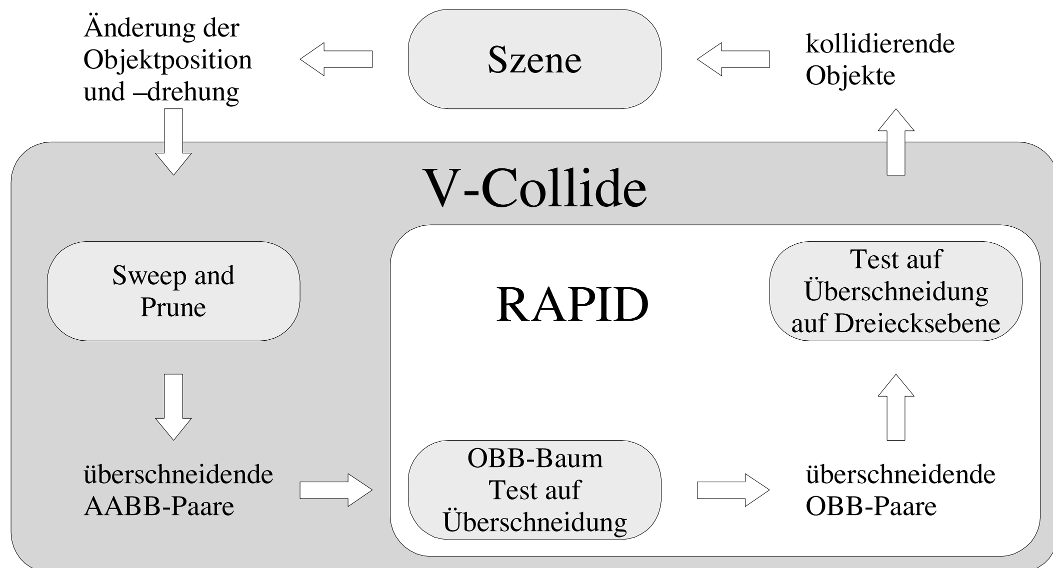

Abbildung 15: Architektur von V-Collide

2.6.2 Zusammenspiel von OpenGL und V-Collide

V-Collide ist ein völlig eigenständiges Softwarepaket mit Schnittstellen zu C/C++. 3D-Objekte werden in einer sog. *Collisiondetectionengine* zusammengefaßt. 3D-Objekte werden in V-Collide nur durch Dreiecke beschrieben. Jedes Objekt hat seine zugehörige Transformationsmatrix. Sie enthält Drehung und Position des Objekts und wird bei jeder Bewegung des Objekts entsprechend angepaßt. Eine Skalierung der Objekte ist nicht zulässig bzw. wird noch nicht unterstützt. Eine Kollisionserkennung für OpenGL kann man nun wie folgt erreichen:

1) jedes Objekt liegt als Menge von Dreiecken vor;

2) jedem Objekt wird genau eine Transformationsmatrix zugeordnet;

3) Position und Richtung eines Objekts werden stets in oben genannter Matrix zusammengefaßt;

4) die Form, also die Dreiecke, und die Tansformationsmatrix eines jeden Objekts werden OpenGL, wie auch V-Collide bekannt gemacht;

Zuerst werden alle Objekte so angelegt, daß sie sich nicht berühren und überschneiden. Sollen dann Objekte bewegt werden, so wird diese Bewegung zuerst an V-Collide weitergeleitet. Hier wird auf Kollision geprüft und ggf. wird dann die Bewegung zurückgenommen bzw. an OpenGL weitergegeben. Die darzustellende Szene liegt also doppelt vor und bei jeder zulässigen Bewegung müssen OpenGL und V-Collide die Daten der geänderten Objekte erhalten.

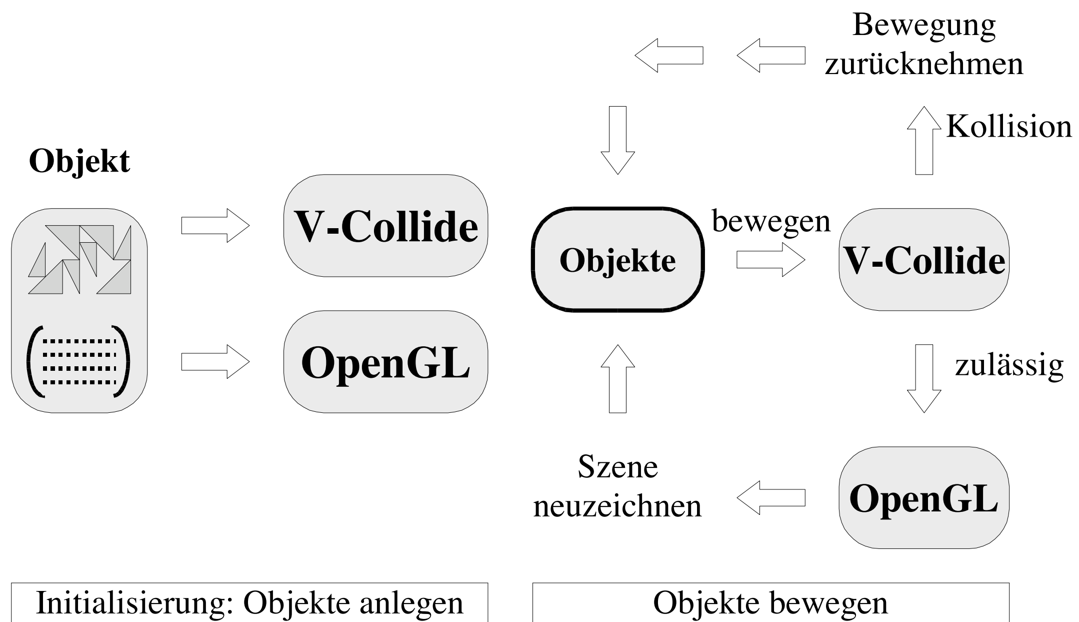

Abbildung 16: Kollisionserkennung für OpenGL mit V-Collide

3 Wissensrepräsentation

Im vorangegangenen Kapitel wurden markante Eigenschaften dreier Graphikwerkzeuge herausgearbeitet, bzgl. bestimmter Anforderungen bewertet und ein geeignetes ausgewählt.

Der nächste Schritt zum Gesamtsystem führt uns zur Wissensrepräsentation. Unsere Aufgabe liegt darin ein möglichst gutes Modell der Umgebung eines Service-Roboters, also eine Bürowelt, zu repräsentieren.

In diesem Kapitel erfolgt deshalb erst eine kurze Diskussion über die Anwendungsgebiete der Wissensrepräsentation und einigen verschiedenen Vorgehensweisen, Wissen zu repräsentieren. Es wird das Konzept der terminologischen Logiken, oder auch Begriffs- oder taxonomische Logiken genannt, herausgegriffen und im Detail beschrieben. Im darauffolgenden Abschnitt wird der Repräsentationsformalismus LOOM, eine Implementierung einer terminologischen Logik, vorgestellt. Sie ist Bestandteil der WR-Komponente, die in Abschnitt 5.1.1 vorgestellt wird. Im Anschluß daran wird ein, auf einen Raum beschränktes, Modell einer Bürowelt angegeben.

3.1 Allgemeines

In der Wissensrepräsentation wird Wissen über einen Ausschnitt der realen Welt repräsentiert. Wie schon von den Datenbanken her bekannt, kann auch hier das repräsentierten Wissen unter verschiedenen Gesichtspunkten ausgewertet werden. Die eigentlich interessante Eigenschaft der Wissensrepräsentation, die sie gegenüber herkömmlichen Datenbanken auszeichnet, ist die Möglichkeit, aus den vorhandenen Daten durch Inferenzmechanismen zuvor implizite Kenntnisse bzw. Informationen explizit zu machen.

Einige Anwendungsbeispiele der Wissensrepräsentation sind:

- *Diagnose*: dabei ist die Repräsentation von Wissen über ein Gerät, sowie über mögliche Fehlerursachen nötig;
- *Planung*: Wissen über mögliche Aktionen und mögliche Pläne des Gegners ist hier von Bedeutung;
- *Roboternavigation*: Wissen über die Umgebung und Hindernisse usw. ist relevant.

Der vorliegenden Arbeit liegt letzteres Anwendungsfeld zugrunde.

Wie lassen sich aber nun die Aufgaben der Wissensrepräsentation aufteilen? Eine Mögliche Aufgliederung wird nun vorgestellt. Dabei werden die Aufgaben in drei Interessensbereiche einteilt.

1. Das erste Gebiet, die *Modellierung*, beschäftigt sich mit der Repräsentation des Wissens eines Anwendungsgebietes. In unserem Fall handelt es sich dabei um Wissen über die Bürowelt.
2. Im nächsten Gebiet, den *Formalismen*, steht der Entwurf von Formalismen zur

Bearbeitung des Wissens im Vordergrund. Hier sind verschiedenste Ansätze vertreten: Vererbungsnetze, Constraints (Constraint-Netze, Constraint-Propagierung), Qualitative Modellierung (z.B. Allensches Intervallkalkül) und Nicht-monotone Logik (z.B. Defaultlogik)[24].

3. Das letzte Gebiet, das *System*, beschäftigt sich mit der Implementierung einer Applikation, die der Verwaltung und Verarbeitung des repräsentierten Wissens dient. Ein System, dem z.B. eine terminologische Logik zu Grunde liegt, sollte entsprechende Dienste anbietet, mit denen sich die modellierte Information bearbeiten, anordnen und abfragen läßt. Ebenso sollte das System in der Lage sein durch Mechanismen Schlußfolgerungen, d.h. implizites Wissen, aus dem expliziten Wissen abzuleiten. Z.B. für terminologische Logiken soll das System folgenden Katalog von Fragen[25] (unter Zuhilfenahme entsprechender (Inferenz-) Mechanismen) beantworten können:

   - Ist eine eingeführtes Konzept sinnvoll definiert? (*satisfiability*)
   - Ist eine Konzept allgemeiner als ein anderes? (*subsumption*)
   - Wo genau befindet sich ein Konzept in der Konzepthierarchie? (*classification*)
   - Ist das modellierte Wissen konsistent? (*consistency*)
   - Welche Fakten lassen sich aus dem modellieren Wissen folgern? (*instantiation*)
   - Welche Konzepte instanzieren ein bestimmtes Objekt? (*realization*)
   - Welche Instanzen gibt es zu einem gegebenen Konzept? (*retrieval*)

   Eine Beurteilung der Algorithmen, die diese Eigenschaften implementieren, erfolgt anhand folgender Kriterien:

   1. Korrektheit
      Antworten, Ergebnisse müssen im Sinne der Semantik wahr sein.
   2. Vollständigkeit
      Alle Ergebnisse müssen gefunden werden bzw. existiert ein Ergebnis, so wird es ausgegeben.
   3. Entscheidbarkeit, Komplexität
      Sind die gestellten Anfragen entscheidbar und wie schnell liegen die Ergebnisse der Anfragen vor?

Im nächsten Abschnitt wird der Formalismus einer terminologischen Logik eingeführt. Im Anschluß daran wird das System LOOM, das auf diesem Formalismus beruht, vorgestellt.

24 In dieser Arbeit wird auf die Beschreibung der Ansätze, mit Ausnahme der terminologischen Logiken, verzichtet. In der entsprechenden Literatur werden die anderen Ansätze vorgestellt, [OLN 93].
25 Der entsprechende englische Fachbegriff ist jeweils, in Klammern, nach der Frage genannt.

3.2 Terminologische Logiken und LOOM

Ein geeigneter und bereits gut untersuchter Formalismus sind die terminologischen Logiken. Sie wurden schon mehrfach in Wissensrepräsentationssystemen implementiert, so z.B. in CLASSIC [BPS 94], KL-ONE [BS 85], LOOM [Mac 91] usw. In diesem Abschnitt werden Begriffsdefinitionen, Syntax und Semantik einer terminologischen Logik ([BBH 90], [BBH 92], [Neb 90], [BL 94]) beschrieben.

3.2.1 Begriffsdefinitionen

Terminologische Wissensrepräsentation orientiert sich an den Objekten der realen Welt. Objekte werden durch ihre Eigenschaften und durch ihre Beziehungen untereinander beschrieben. In terminologischen Logiken werden die Eigenschaften „Konzepte" und die Beziehungen „Rollen" genannt. Konzeptnamen (einstellige Prädikate) und Beziehungen[26] (zweistellige Prädikate) können unter Verwendung von Operatoren zu komplexeren Konzepten zusammengesetzt werden. Mögliche Konzeptnamen wären z.B. *reich*, *Frau* und *Mann* und eine mögliche Rolle wäre *liebt*. Elemente, die eine Rolle erfüllen werden Rollenfüller bzw. Füller genannt.

Das abstrakte Modell, eines Teilbereichs der Welt, wird durch die Menge der Konzeptnamen und Rollen[27], die terminologische Axiome genannt werden, in der sog. TBox (siehe Abschnitt *3.2.2.3 Notation terminologischer Axiome*) beschrieben.

Konkrete Objekte der realen Welt, die sich in ihren Eigenschaften einem, oder mehreren Konzepten zuordnen lassen, werden in der sog. ABox (siehe Abschnitt *3.2.2.4 Notation der Objekte*) vereint, d.h. in der ABox werden also Aussagen über die Objekte des modellierten Teilbereichs der Welt getroffen. Sie sind Instanzen der Konzeptnamen der TBox.

Bei terminologischen Logiken gilt die *Unique Name Assumption* (UNA), die besagt, daß ein Name genau ein Objekt referenziert. Meist wird hier von einer *Open World Assumption* ausgegangen, was zur Folge hat, daß nicht genannte Fakten nicht als implizit falsch angenommen werden. Für unser Modell der Bürowelt werden wir aber vom Gegenteil, der *Closed World Assumption,* Gebrauch machen. Zyklische Definitionen komplexer Konzepte sind generell nicht verboten. In einigen Fällen sind Inferenzen dann nicht mehr möglich. Für „sinnvolle" zyklische Definitionen existieren aber eine Semantik und Algorithmen, die Inferenzen zulassen [Neb 91].

In den folgenden Abschnitten wollen wir uns eine Terminologische Logik genauer ansehen. Hier werden Symbole, Syntax und Semantik eingeführt, damit sich Zusammenhänge der realen Welt, formal einheitlich, modellieren lassen.

Um die eingeführten Symbole, Syntax und Semantik mit Leben zu füllen, wird uns ein Beispiel, das sich mit Wissen über familiäre Beziehungen beschäftigt, begleiten. Nachfolgend ist eine umgangssprachliche Beschreibung über familiäre Beziehungen gegeben. Wir haben damit Wissen über die Eigenschaften der Familienmitglieder und über deren

26 Im weiteren Verlauf wird eine Beziehung zwischen Konzepten auch als Rolle bezeichnet.
27 Konzepte und Rollen werden meist als *Begriffe* bezeichnet.

Beziehung untereinander.

Es gibt *Personen*.
Es gibt ein *Geschlecht*.
männlich ist ein *Geschlecht*.
weiblich ist ein *Geschlecht*.
Eine *Frau* ist eine *Person* deren *Geschlecht weiblich* ist.
Ein *Mann* ist eine *Person* deren *Geschlecht männlich* ist.
Eltern sind *Person* die *Kinder* haben, die wiederum *Personen* sind.
Mütter sind *Eltern* deren *Geschlecht weiblich* ist.
Väter sind *Eltern* deren *Geschlecht männlich* ist.

Nun soll es noch einige konkrete Personen geben, die sich wie folgt beschreiben lassen:

John ist ein *Vater*.
Tom ist ein *Kind* von *John*.
Mary ist eine Frau.

3.2.2 Syntax und Semantik

In diesem Abschnitt wird eine Syntax mit entsprechender Semantik gegeben, mit der sich Wissen über die reale Welt (zum Teil) beschreiben läßt.

Im Weiteren sollen folgende Abkürzungen bzw. Symbole verwendet werden:

- *CN* bezeichne die Menge der Konzeptnamen (z.B. *Person, Frau, Mann, Eltern, Mutter, Vater, Mutter_vieler_Kinder*).
- *RN* sei die Menge der Rollennamen (z.B. *hat-Kind*).
- *AN* ist die Menge der Attributnamen. (z.B. *hat-Geschlecht*).
- Die Menge der Individuennamen sei mit *IN* gegeben (z.B. *männlich, weiblich*).
- Die Menge der Objektnamen sei mit *ON* bezeichnet (z.B. *John, Mary, Tom*).

Um mit diesen atomaren Symbolen nun komplexe Ausdrücke aufbauen zu können, bedarf es der Einführung entsprechender Operatoren, die in den nächsten Abschnitten *3.2.2.1 Operatoren zur Konzeptgenerierung* und *3.2.2.2 Operatoren zur Rollengenerierung* beschrieben werden. Auf eine vollständige Aufführung aller Operatoren wird jedoch verzichtet und der geneigte Leser sei auf die einschlägige Literatur verwiesen.

Die Semantik wird mit Hilfe einer Interpretation $\mathfrak{I} = (\Delta^{\mathfrak{I}}, \cdot^{\mathfrak{I}})$ definiert. Diese besteht aus einem Interpretationsbereich bzw. Universum $\Delta^{\mathfrak{I}}$ und der Interpretationsfunktion $\cdot^{\mathfrak{I}}$. Damit lassen sich Konzeptnamen auf eine Teilmenge des Universums abbilden:

$$CN^{\mathfrak{I}} \subseteq \Delta^{\mathfrak{I}}$$

Rollennamen werden auf folgende Weise abgebildet:

$$RN^{\mathfrak{I}} \subseteq \Delta^{\mathfrak{I}} \times \Delta^{\mathfrak{I}}$$

Dies ist eine Menge von Paaren von Elementen aus $\Delta^{\mathcal{I}}$. Die Menge der Tupel, die eine Rolle erfüllen, wird mengentheoretisch mit $\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d)\}$ beschrieben.

Attributnamen werden in nachstehender Form interpretiert:

$$AN^{\mathcal{I}} : dom\ AN^{\mathcal{I}} \rightarrow \Delta^{\mathcal{I}}, \text{ mit } dom\ AN^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$$

Schließlich wird noch jedem Individuennamen und Objektnamen ein Element

$$I^{\mathcal{I}} \in \Delta^{\mathcal{I}}$$

zugewiesen.

3.2.2.1 Operatoren zur Konzeptgenerierung

Tabelle 4 enthält einen Auszug zulässiger Operatoren zum Aufbau komplexer Konzepte. Es werden jedoch nur die Operatoren wiedergegeben, die für das in Abschnitt *3.3 Modell der realen Welt* beschriebene Modell notwendig sind. Die verwendete konkrete Syntax entspricht der Notation in LOOM, [Bri 95]. Abschnitt *3.2.3 LOOM* gibt eine kurze Einführung in LOOM. Im weiteren Verlauf werden immer wieder einige Beispiele in abstrakter Syntax sowie in konkreter Syntax angegeben. Die Beschreibung der Semantik ist mengentheoretisch. Auf eine prädikatenlogische Sicht der Semantik wird hier verzichtet, sie kann in der entsprechenden Literatur nachgelesen werden.

Konkrete Syntax	Abstrakte Syntax	Semantik und Beschreibung
-keine-	$\top$	$\Delta^{\mathcal{I}}$ gesamtes Universum
-keine-	$\bot$	$\emptyset$ leeres Universum
(:and C_1 ... C_n)	$C_1 \sqcap \ldots \sqcap C_n$	$C_1^{\mathcal{I}} \cap \ldots \cap C_n^{\mathcal{I}}$ Konjunktion von Konzepten
(:or C_1 ... C_n)	$C_1 \sqcup \ldots \sqcup C_n$	$C_1^{\mathcal{I}} \cup \ldots \cup C_n^{\mathcal{I}}$ Disjunktion von Konzepten
(:at-least n R)	$\geq nR$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \rvert \geq n\}$ Rolle muß min. n Füller haben
(:at-most n R)	$\leq nR$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \rvert \leq n\}$ Rolle hat max. n Füller
(:exactly n R)	nR	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \rvert = n\}$ Rolle hat genau n Füller
(:exactly n R C)	$nR : C$	$\{d \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(d) \rvert = n \wedge R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$ Rolle hat genau n Füller und alle müssen das Konzept C erfüllen

Konkrete Syntax	Abstrakte Syntax	Semantik und Beschreibung
(:all R C)	$\forall R : C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$ alle Rollenfüller müssen das Konzept C erfüllen
(:some R C)	$\exists R : C$	$\{d \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$ wenigstens ein Füller der Rolle R genügt dem Konzept K
(:same-as R_1 R_2)	$R_1 = R_2$	$\{d \in \Delta^{\mathcal{I}} \mid R_1^{\mathcal{I}}(d) = R_2^{\mathcal{I}}(d)\}$ die Füller der beiden Rollen R_1 und R_2 sind identisch
(:subset R_1 R_2)	$R_1 \subseteq R_2$	$\{d \in \Delta^{\mathcal{I}} \mid R_1^{\mathcal{I}}(d) \subseteq R_2^{\mathcal{I}}(d)\}$ die Füller der Rolle R_1 bilden eine Teilmenge der Füller der Rolle R_2

Tabelle 4: Operatoren zur Konzeptgenerierung

Es sei vermerkt, daß folgende zwei Operatoren aus dieser Tabelle (:same-as R_1 R_2) und (:subset R_1 R_2) eine Sonderstellung einnehmen. Mit ihnen sind, im Unterschied zu allen anderen Operatoren, keine Konzepte verbunden. Sie werden deshalb auch mit der Bezeichnung, *Role-Value-Map*-Konstruktoren, versehen.

3.2.2.2 Operatoren zur Rollengenerierung

Analog zu den Operatoren zur Konzeptgenerierung wird nun eine Liste von Operatoren zur Erstellung von Rollen vorgestellt.

<table>
<tr><th>Konkrete Syntax</th><th>Abstrakte Syntax</th><th colspan="2">Semantik und Beschreibung</th></tr>
<tr><td>(:and R_1 ... R_n)</td><td>$R_1 \sqcap ... \sqcap R_n$</td><td colspan="2">$R_1^{\mathcal{I}} \cap ... \cap R_n^{\mathcal{I}}$
Rollen-Konjunktion</td></tr>
<tr><td>(:inverse R)</td><td>R^{-1}</td><td colspan="2">$\{(d, d') \mid (d', d) \in R^{\mathcal{I}}\}$
bei inversen Rollen werden die Elemente des Füllers getauscht</td></tr>
<tr><td>(:domain C_1)</td><td rowspan="2">$C_1 \times C_2$</td><td rowspan="2">$C_1^{\mathcal{I}} \times C_2^{\mathcal{I}}$</td><td rowspan="2">das erste Elemente des Füllers muß eine Instanz des Konzeptes C_1 sein und das zweite eine Instanz des Konzepts C_2</td></tr>
<tr><td>(:range C_2)</td></tr>
</table>

Konkrete Syntax	Abstrakte Syntax	Semantik und Beschreibung
(:compose R_1 ... R_n)	$R_1 \circ ... \circ R_n$	$R_1^{\mathfrak{I}} \circ ... \circ R_n^{\mathfrak{I}}$ Rollenkompositon: jeweils zwei aufeinanderfolgende Rollen werden über ein gemeinsames Element verbunden:(a,b|b,c|c,d|d ...)

Tabelle 5: Operatoren zur Rollenerstellung

In den nächsten beiden Abschnitten sehen wir an Beispielen, wie die Operatoren angewendet werden. Dabei werden einige umgangssprachlich formulierten Fakten aus unserem Beispiel aus Abschnitt 3.2.1 formal aufgeschrieben.

3.2.2.3 Notation terminologischer Axiome

Mit Hilfe der terminologischen Axiome, beschreiben wir die Eigenschaften der Objekte unseres Weltausschnitts aus dem Beispiel in Abschnitt 3.2.1. Diese Informationen werden in der sog. TBox zusammengefaßt. Wir können zwar komplexe Konzepte und Rollen erstellen, jedoch haben wir noch keine Möglichkeit, diesen Namen zuzuordnen. Deshalb wird in Tabelle 6 eine Konvention vorgestellt, mit der dies möglich ist. Dabei werden Konzeptdefinitionen bzw. Rollendefinitionen stets mit **(defconcept ...)** bzw. **(defrelation ...)** eingeleitet.

Konkrete Syntax	Abstrakte Syntax	Semantik
(defconcept *CN* :is *C*)	$CN \doteq C$	$CN^{\mathfrak{I}} = C^{\mathfrak{I}}$
(defrelation *RN* :is *R*)	$RN \doteq R$	$RN^{\mathfrak{I}} = R^{\mathfrak{I}}$
(defconcept *CN* :is-primitive *C*)	$CN \sqsubseteq C$	$CN^{\mathfrak{I}} \subseteq C^{\mathfrak{I}}$
(defrelation *RN* :is-primitive *R*)	$RN \sqsubseteq R$	$RN^{\mathfrak{I}} \subseteq R^{\mathfrak{I}}$

Tabelle 6: Definition terminologischer Axiome

Wie diese Tabelle zeigt, wird zwischen primitiven (bezeichnet mit *:is-primitive*) und definierten (bezeichnet mit *:is*) Begriffen unterschieden. Sowohl primitive als auch definierte Begriffe entstehen durch Kombination anderer Begriffe (im weiteren als Bedingungen bezeichnet) unter Zuhilfenahme der Konstrukte des Formalismus.

Definierte Begriffe werden durch Kombination anderer Begriffe festgelegt. Diese Kombination ist für Objekte sowohl eine notwendige als auch eine hinreichende Bedingung, d.h. Objekte, die zu dieser Klasse gehören, erfüllen notwendigerweise diese Bedingung. Für alle anderen Objekte, die diese Bedingung erfüllen, ist das hinreichend, um als Instanzen dieser Klasse erkannt zu werden. Es besteht eine Äquivalenz.

Die Definition primitiver Begriffe hingegen stellt nur eine notwendige Bedingung dar. In diesem Fall kann von einer Implikation gesprochen werden: Objekte, die zu dieser Klasse gehören, erfüllen die Bedingung. Aber Objekte, die diese Bedingung erfüllen, sind nicht automatisch Instanzen dieser Klasse. So kann man z.B. *Rohbau* mit folgender Bedingung definieren: *vier nicht verputzte Wände* mit *Dach* darauf. Die meisten

Blockhäuser erfüllen genau diese Bedingung, sie sind aber nicht automatisch ein *Rohbau*.

In Tabelle 7 ist ein Auszug der Terminologien zu sehen, die sich aus unserem Beispiel ergeben. Da LOOM nicht zwischen Attributen und Rollen unterscheidet, ist hier *hat-Geschlecht* eine Rolle. *Männl.* und *Weibl.* werden zu Konzepten.

TBox	
Abstrakte Syntax	**Konkrete Syntax**
$Person \sqsubseteq \top$	(defconcept *Person*)
$Geschlecht \sqsubseteq \top$	(defconcept *Geschlecht*)
Männl. $\sqsubseteq$ *Geschlecht*	(defconcept *Männl.* :is-primitive *Geschlecht*)
Weibl. $\sqsubseteq$ *Geschlecht*	(defconcept *Weibl.* :is-primitive *Geschlecht*)
hat-Geschlecht $\sqsubseteq$ *Person* $\times$ *Geschlecht*	(defrelation *hat-Geschlecht* :domain *Person* :range *Geschlecht*)
Mann $\sqsubseteq$ *Person* $\sqcap$ *1 hat-Geschlecht*:*Männl.*	(defconcept *Mann* :is-primitive (:and *Person* (:the *Geschlecht Männl.*)))
Frau $\sqsubseteq$ *Person* $\sqcap$ *1 hat-Geschlecht*:*Weibl.*	(defconcept *Frau* :is-primitive (:and *Person* (:the *Geschlecht Weibl.*)))
hat-Kind $\sqsubseteq$ *Person* $\times$ *Person*	(defrelation *hat-Kind* :domain *Person* :range *Person*)
Eltern $\doteq$ *Person* $\sqcap$ $\exists$*hat-Kind*:*Person* alternativ: *Eltern* $\doteq$ *Person* $\sqcap$ $\exists$*hat-Kind*	(defconcept *Eltern* :is (:and *Person* (:some *hat-Kind Person*)) alternativ: (defconcept *Eltern* :is (:and *Person* (:at-least 1 *hat-Kind*))
Vater $\sqsubseteq$ *Mann* $\sqcap$ *Eltern*	(defconcept *Vater* :is-primitive (:and *Mann Eltern*))

Tabelle 7: Terminologien der TBox

3.2.2.4 Notation der Objekte

Nachdem die Eigenschaften, die Konzepte, der Objekte unseres Weltausschnitts defi-

niert wurden, können letztendlich Instanzen der Konzepte angelegt werden. Die Form ist in Tabelle 8 gezeigt:

Konkrete Syntax	Abstrakte Syntax	Semantik
(tell (create *ON C*))	$ON \in C$	$ON^{\mathcal{I}} \in C^{\mathcal{I}}$
(tell (*R ON ON'*))	$\langle ON, ON' \rangle \in R$	$\langle ON^{\mathcal{I}}, ON'^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$

Tabelle 8: Definition assertionaler Axiome

In unserer Beispielwelt seien folgende Objekte gegeben:

ABox	
Abstrakte Syntax	**Konkrete Syntax**
$John \in Vater$	(tell (create *John Vater*))
$Mary \in Frau$	(tell (create *Mary Frau*))
$\langle John, Tom \rangle \in hat\text{-}Kind$	(tell (*hat-Kind John Tom*))

Tabelle 9: Objekte der ABox

Nachdem die theoretischen Grundlagen gelegt wurden, wird nun LOOM, eine Implementierung der terminologischen Logiken, vorgestellt.

3.2.3 LOOM

Das System LOOM gehört zur Klasse der ausdrucksstarken und schnellen Systemen mit strukturellem und damit unvollständigem Schlußfolgerungsverfahren [HKN 94]. Zusätzlich zu den beschreibungslogischen Diensten bietet LOOM regelbasiertes Schlußfolgern und objektorientierte Methoden basierend auf den Konzeptbeschreibungen. Systeme, die ein erweitertes Angebot an Schlußfolgerungsverfahren zur Verfügung stellen, werden als *hybride Inferenzsysteme* bezeichnet. Im nächsten Abschnitt soll, an einem Modell einer Büroumgebung, LOOMs Fähigkeit Schlußfolgerungen zu ziehen gezeigt werden. Dabei wird ebenso vom objektorientierten Ansatz gebraucht gemacht.

3.3 Modell der realen Welt

In dieser Arbeit wird mit Einschränkungen ein Teil der realen Welt modelliert. Als Repräsentationsformalismus wird LOOM verwendet. Folglich werden die Formalisierungen in LOOM-Notation gegeben. Zusätzlich wird, so es sinnvoll erscheint, noch die abstrakte Notation aufgeführt.

Ziel dieses Teilbereichs der Arbeit ist, ein Modell für eine Büroumgebung zu erarbeiten. Dies wirft viele Fragen auf: welche Objekte werden in das Modell aufgenommen? Welche graphischen Details soll das Modell enthalten? Wie werden die Eigenschaften der Objekte (Tisch, Stuhl, Zimmer usw.) modelliert? Wie detailliert wird das Modell, bzgl. der Objekteigenschaften? Wie sehen Beziehungen unter Objekten aus? In wie weit sollen die Gesetze der Physik modelliert werden?

Es muß ein abstraktes Modell gefunden werden, das sich in der Wissensbasis einfach beschreiben läßt, aber noch genug Information enthält um eine eindeutige graphische Darstellung zuzulassen. Das fertige Modell enthält demnach so wenig geometrische Information wie möglich, aber so viel wie nötig.

3.4 Objektrepräsentation

In diesem Abschnitt werden Objekte, wie sie in einem normalen Büro vorkommen, beschrieben bzw. modelliert. Also Tische, Stühle, Rollcontainer, Schränke, Regale, Gläser, Flaschen, Monitor, Tastatur, Maus, Papier, Bücher, Stifte usw. Die Objekte werden allein durch ihre Eigenschaften beschrieben. Wir wollen uns dabei auf die Stapelbarkeit und Beweglichkeit[28] von Objekten beschränken. So hat z.B. ein Tisch die Eigenschaften, daß er beweglich ist und ganz bestimmte Objekte darauf liegen können. Ein Schrank hingegen ist unbeweglich. Er kann ebenso wie der Tisch bestimmte Objekte aufnehmen. Welche Objekte nun auf einem bestimmten anderen Objekt liegen dürfen, wird über sog. *Methoden* realisiert, auf die wir später noch zu sprechen kommen. Insgesamt werden drei Grundeigenschaften, mit welchen sich Objekte beschreiben lassen, definiert: Stapelbarkeit, Translation (das Objekt darf im Raum frei bewegt werden) und Rotation bzgl. der y-Achse (die Ausrichtung des Objekts ist frei wählbar).

Die Notwendigkeit, Beweglichkeit in Translation und Rotation aufzusplitten, ist im Modell der Bürotür begründet. Sie befindet sich an einer festen Position, kann aber geöffnet (Veränderung der Ausrichtung bzgl. der linken oder rechten Kante der Tür) werden, d.h. die Tür ist selber nur im Sinne der Rotation um die y-Achse beweglich. Im allgemeinen Sprachgebrauch ist eine Tür wie ein Tisch beweglich. Nur, daß sich für uns, offensichtlich klar, von Tür nach Tisch der Kontext bzgl. der Beweglichkeit ändert. Das ist genau das Wissen, das zusätzlich modelliert werden muß. Es mag zwar nicht das optimale Beispiel sein, verdeutlicht aber die Problematik, die bei der logischen Modellierung alltäglicher Dinge ständig auftritt. Ein Erweiterung des Modells bzgl. dieser Problematik der Rotation wird in Abschnitt 6.2.1 angesprochen.

Nun aber wieder zurück zu unserer Modellierung der Büroumgebung. Nachfolgend werden einige Konzepte und Rollen/Relationen, die die Eigenschaften der Bürowelt beschreiben, vorgestellt. Dabei wird stets eine Beschreibung dessen, was modelliert werden soll, vorgenommen. Danach erfolgt, so es sinnvoll erscheint, die abstrakte Syntax. Der entsprechende LOOM-Ausdruck wird stets gegeben.

Zuerst wird das Basis-Konzept definiert. Von ihm werden alle weiteren Konzepte, Objekte betreffend, ableitet. LOOM Syntax:

```
(defconcept all-objects)
```

Jetzt wird ein Konzept definiert, das ein nicht bewegliches Objekt *object* beschreibt, mit jeweils genau einem Namen *name* (kann irgend eine Zeichenfolge sein, in LOOM als *string* bezeichnet), einer Farbe *rgba* (rot, grün, blau und Durchsichtigkeit sind genau in

28 Beweglichkeit beinhaltet hier nicht nur die Möglichkeit der Translation, sondern auch die der Rotation bzgl. der y-Achse (siehe Abbildung 8: Lage der Raumachsen).

dieser Reihenfolge anzugeben und haben einen Wertebereich von 0 bis 255, z.B. „255 0 0 0“ für Vollrot), Position *position* (die Koordinaten des Objekts in der x, z-Ebene, z.B. „12.0 20.0“, siehe auch Abbildung 8: Lage der Raumachsen), Ausrichtung (Rot. um die y-Achse) *direction* (im Gradmaß anzugeben von 0 bis 360, z.B. „120.0“) und einem Skalierungsfaktor *scale*[29]. Detaillierte Informationen zu geometrischen Formen werden nicht in der Wissensbasis gehalten. Die abstrakte Syntax zu diesem Konzept lautet:

$object \sqsubseteq all\text{-}objects \sqcap (1\ rgba) \sqcap (1\ position) \sqcap (1\ direction) \sqcap (1\ scale)$

In LOOM Notation:

```
(defconcept object :is-primitive
  (and all-objects
    (exactly 1 name)
    (exactly 1 rgba)
    (exactly 1 position)
    (exactly 1 direction)
    (exactly 1 scale)
  )
)
```

Als nächstes werden, für eben eingeführtes Konzept, fünf Rollen/Relationen *name, rgba, position, direction* und *scale* angelegt, deren jeweils erstes Argument (*:domain*) ein Konzept vom Typ *object* sein muß. Das zweite Argument (*:range*) ist vom Typ *string*, wobei *string* ein in LOOM vorgefertigtes Konzept ist, das Zeichenketten beschreibt. In LOOM Notation:

```
(defrelation name :domain object :range string)
(defrelation rgba :domain object :range string)
(defrelation position :domain object :range string)
(defrelation direction :domain object :range string)
(defrelation scale :domain object :range string)
```

position, rgba, direction und *scale* sind vom Typ *string*, da die Wissensbasis diese Daten nur speichert und nicht damit rechnet.

Für die abstrakte Syntax sollen die ersten beiden Rollen genügen. Die restlichen werden analog definiert:

$name \sqsubseteq object \times string$
$rgba \sqsubseteq object \times string$

Nun werden drei Konzepte angelegt, die die Eigenschaft der Beweglichkeit widerspiegeln. Zuerst die abstrakte Notation:

$translate\text{-}object \sqsubseteq object$
Objekte, die nur in x-, y- und z-Richtung verschoben werden dürfen;

$rotate\text{-}object \sqsubseteq object$
Objekte, die nur um die y-Achse gedreht werden dürfen;

29 Die Möglichkeit der Skalierung der Objekte wurde im Gesamtsystem nicht implementiert, da die verwendete Kollisionserkennungssoftware bisher keine Skalierung der Objekte unterstützt.

move-object $\sqsubseteq$ *translate-object* $\sqcap$ *rotate-object*
Objekte, die die Eigenschaften von *translate-object* und *rotate-object* erben, also frei beweglich sind;

Die drei Konzepte in LOOM Notation sehen dann wie folgt aus:

```
(defconcept translate-obj :is-primitive object)
(defconcept rotate-obj :is-primitive object)
(defconcept move-obj :is-primitive (and translate-obj rotate-obj))
```

Nun werden drei Konzepte für frei bewegliche Objekte, *Stift, Glas* und *Flasche* angelegt. Die Definition in abstrakter Syntax:

Stift $\sqsubseteq$ *move-obj*
Glas $\sqsubseteq$ *move-obj*
Flasche $\sqsubseteq$ *move-obj*

und in LOOM Notation:

```
(defconcept Stift :is-primitive move-obj)
(defconcept Glas :is-primitive move-obj)
(defconcept Flasche :is-primitive move-obj)
```

Die nächsten drei Konzepte beschreiben die unbeweglichen Objekte, *Boden, Saeule, Tuerstock* und *Wand* In abstrakter Syntax:

Boden $\sqsubseteq$ *object*
Saeule $\sqsubseteq$ *object*
Tuerstock $\sqsubseteq$ *object*
Wand $\sqsubseteq$ *object*

in LOOM Notation:

```
(defconcept Boden :is-primitive object)
(defconcept Saeule :is-primitive object)
(defconcept Tuerstock :is-primitive object)
(defconcept Wand :is-primitive object)
```

In Abschnitt *8.4 LOOM: Modell der Büroumgebung - TBox* werden alle Konzepte und Relationen des Modells aufgelistet. Einen Ausschnitt des Klassenbaums ist in Abbildung 17 gegeben.

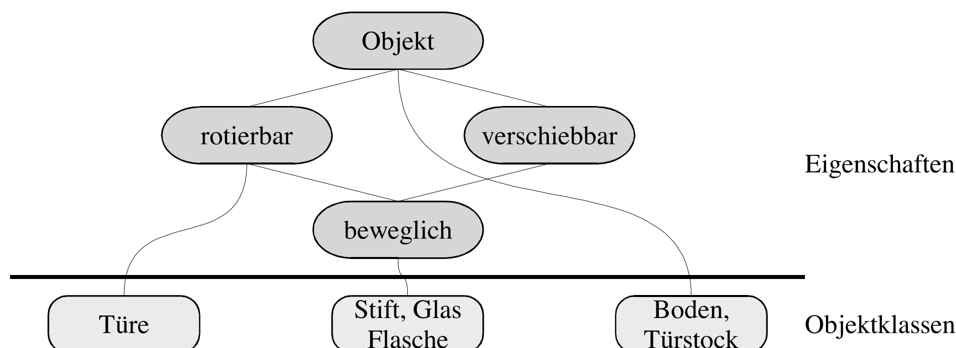

Abbildung 17: Klassenbaum (Ausschnitt)

Die Beziehung zwischen den Konzepten wird über folgende Relation

```
(defrelation A-on-B
 :domain object
 :range object
 :characteristics (:single-valued))
```

realisiert. Sie besagt, daß nur Objekte vom Typ *object* auf Objekten vom Typ *object* liegen dürfen. Desweiteren besagt *:single-valued*, daß ein einzelnes Objekt nie auf mehr als einem weiteren Objekt liegen darf, d.h. die Relation darf nur jeweils einen Füller haben. In der aktuellen Definition dürfen wir z.B. auch einen Tisch auf ein Glas stellen. Da das aber wenig Sinn macht, wird ein Tupel in die *A-on-B*-Relation nicht direkt, sondern über eine sog. *Methode* einer sog. *Aktion* eingetragen. Für jedes Objekt wird eine Methode erstellt, die diejenigen Objekte auflistet, die auf diesem Objekt liegen dürfen. Methoden, deren Objektliste sich gleichen, werden zusammengefaßt. Soll nun ein Tupel in die *A-on-B*-Relation eingefügt werden, so wird dann, anhand des ersten Elements des Tupels, die entsprechende Methode für das Objekt aufgerufen. Diese überprüft nun das zweite Element des Tupels auf Typkorrektheit und fügt ggf. das Tupel in die *A-on-B*-Relation ein. Nachfolgend sind die Definition einer *Aktion* und einer *Methode* aufgeführt.

Mit `(defaction pose (?a ?b))` wird eine Aktion namens *pose*, mit den zwei Parametern *?a* und *?b* definiert. Sie ist sozusagen nur das Grundgerüst. Sie benötigt deshalb wenigstens eine Methode, die bei Aufruf der Aktion ausgeführt werden kann. Anhand der Typen der Parameter *?a* und *?b* wird entschieden, welche Methode zu rufen ist. In jeder Methode werden die zulässigen Typen von *?a* und *?b* in *:situation* festgelegt. Da es in unserem Fall mehrere Methoden gibt, die sich nur in den zulässigen Typen von *?a* und *?b* unterscheiden, käme das, analog zur Objektorientierung, einer Operatorüberladung gleich. Am nachfolgenden Beispiel wird die Definition einer Methode erläutert.

Nun erzeugen wir für die Aktion *pose* eine Methode namens *put a on b 1*. Sie besagt, das Objekte vom Typ *Stift, Flasch, Glas, Mouse, Buch_liegend, CD, Diskette, Papier_A4* und *Venus* auf genau einem Objekt vom Typ *Buch_liegend, CD, Diskette, Papier_A4, Tisch, Stuhl, Rollcontainer, Projektor, Schrank, Waschbecken, Boden, Sockel* und *Regal* liegen dürfen.

```
; Stift, Glas, Flasche, Mouse, Buch_liegend, CD, Diskette, Papier_A4
(defmethod pose (?a ?b)
  :title "put a on b 1"
  :overrides "put a on b 1"
  :situation (and
    (or (Stift ?a) (Flasche ?a) (Glas ?a) (Mouse ?a) (Buch_liegend ?a)
      (CD ?a) (Diskette ?a) (Papier_A4 ?a)
      (Venus ?a))
    (or (Buch_liegend ?b) (CD ?b) (Diskette ?b) (Papier_A4 ?b)
      (Tisch ?b) (Stuhl ?b) (Rollcontainer ?b)
      (Projektor ?b) (Schrank ?b)
      (Waschbecken ?b) (Boden ?b)
      (Sockel ?b) (Regal ?b)))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))
```

Ist der Ausdruck bei *:situation* erfüllt, so werden die unter *:response* stehenden Operationen ausgeführt. Die Operation *(tell (a-on-b ?a ?b))* fügt ein Tupel in die *A-on-B*-Relation ein und *(format ...)* gibt die Rückmeldung an den Benutzer, daß das Tupel eingefügt werden konnte. Der Ausdruck bei *:situation* ist genau dann wahr, wenn Parameter *?a* eine Instanz eines Konzepts aus der Menge {*Stift, Flasch, Glas, Mouse, Buch_liegend, CD, Diskette, Papier_A4, Venus*} ist und *?b* eine Instanz eines Konzepts aus der Menge {*Buch_liegend, CD, Diskette, Papier_A4, Tisch, Stuhl, Rollcontainer, Projektor, Schrank, Waschbecken, Boden, Sockel, Regal*} ist.

Eine vollständige Liste der Methoden ist in Abschnitt *8.4 LOOM: Modell der Büroumgebung - TBox* gegeben.

Damit sind alle nötigen Eigenschaften beschrieben und wir können endlich das Modell mit Leben füllen und einige Objekte der Bürowelt anlegen:

```
(tell (create floor Boden) (about floor
(name "Bueroboden") (rgba "250 220 200 0") (position "0 0")
(direction "0.0") (scale "1.0")))
```
Definiert den Boden des Büros. Die Ausdehnung, des Bodens ist für alle Böden gleich.

```
(tell (create ob2 Tisch) (about ob2
(name "Tisch1") (rgba "211 148 23 0") (position "75 40")
(direction "0.0") (scale "1.0")))
```
Definiert einen hellbraunen Tisch an Position x=75 und z=40.

```
(tell (create c1 Stuhl) (about c1
(name "Stuhl") (rgba "220 170 70 0") (position "-30 50")
(direction "0.0") (scale "1.0")))
```
Definiert einen hellbraunen Stuhl an Position x=-30 und z=50.

```
(tell (create pen1 Stift) (about pen1
(name "Stift1") (rgba "20 170 70 0") (position "-30 20")
(direction "0.0") (scale "1.0")))
```
Definiert einen Stift.

```
(tell (create ob1 Glas) (about ob1
(name "Glas1") (rgba "172 229 229 0") (position "-20 10")
(direction "0.0") (scale "1.0")))
```
Definiert ein Glas.

Diese Objekte der realen Welt sind in der ABox zusammengefaßt.

Vorerst ist aber noch unklar, welche Objekte sich auf welchen befinden. Folglich ist unser Wissen noch unvollständig. Das Büro selbst ist der Ausgangspunkt für alle weiteren Objekte. Sie sollen sich darin bzw. auf den Boden befinden, wobei uns im Augenblick die Position der Objekt in der x, z-Ebene nicht kümmert[30]. Also befördern wir die vier Objekte ins Büro:

```
(perform (pose (get-instance 'ob1) (get-instance 'floor)))
(perform (pose (get-instance 'c1) (get-instance 'floor)))
(perform (pose (get-instance 'pen1) (get-instance 'floor)))
(perform (pose (get-instance 'ob2) (get-instance 'floor)))
```

Damit befinden sich alle Objekte im Büro auf dem Boden. Folgende Anfrage liefert uns genau diese Information:

```
(retrieve ?p (a-on-b ?p floor))
```

und liefert das Ergebnis:

```
-> (|I|OB1 |I|C1 |I|PEN1 |I|OB2)
```

Nun sollen der Stift und das Glas auf dem Tisch stehen:

```
(perform (pose (get-instance 'pen1) (get-instance 'ob2)))
(perform (pose (get-instance 'ob1) (get-instance 'ob2)))
```

Stellen wir erneut obige Anfrage, so erhalten wir als Ergebnis:

```
(retrieve ?p (a-on-b ?p floor))
-> (|I|OB2 |I|C1)
```

Die Objekte *PEN1* und *OB1* erscheinen nicht mehr im Ergebnis, da sie sich nicht mehr direkt auf dem Boden des Büros befinden. Um alle Objekte, die sich im Büro befinden, sei es direkt oder indirekt, zu erhalten, benötigen wir die transitive Hülle der *A-on-B*-Relation:

```
(defrelation A-on-B* :is
  (:satisfies (?x ?z)
    (:or (A-on-B ?x ?z)
         (:for-some ?y (:and (A-on-B ?x ?y) (A-on-B* ?y ?z))))))
```

Mit *A-on-B** haben wir nun die gewünschte Relation. Stellen wir nun die Anfrage mit der neuen Relation, so erhalten wir:

```
(retrieve ?p (a-on-b* ?p floor))
-> (|I|OB1 |I|C1 |I|PEN1 |I|OB2)
```

In Tabelle 10 sind weitere wichtige Anfragen mit Beschreibung aufgeführt.

Anfrage	**Beschreibung**
`(retrieve ?p (and (object ?p) (a-on-b* ?p ob2)))`	Was steht alles auf dem Tisch (ob2)?
`(ask (move-obj ob2))`	Ist der Tisch ein bewegliches Objekt? *T*, ja er ist; *NIL* sonst;

30 Die Position der Objekte wird erst mit der Einführung der Graphikkomponente (siehe Abschnitt 5) relevant.

Anfrage	Beschreibung
`(retrieve ?p (and (object ?p) (a-on-b ob1 ?p)))`	Auf welchem Objekt steht das Glas (ob1)?
`(ask (a-on-b pen1 ob2))`	Liegt der Stift (pen1) auf dem Tisch (ob2)?

Tabelle 10: Anfragen

Mit diesen Grundlagen läßt sich nun ein Büro beschreiben. Das nächste Ziel ist eine geeignete Schnittstelle für die Wissensrepräsentationskomponente zu entwickeln, über die schließlich die Graphikkomponente angebunden werden kann. Da dies über ein Netzwerk geschehen soll, ist eine entsprechende Netzwerkkomponente erforderlich, die im nächsten Kapitel vorgestellt wird.

4 Kommunikation

Dieses Kapitel behandelt die Erstellung einer Schnittstelle, mit entsprechendem High-Level-Protokoll, zwischen der Wissensrepräsentationskomponente und der Graphikkomponente. Wie zu Beginn erwähnt, sollen beide Komponenten netzwerktransparent miteinander kommunizieren. Als Netzwerkkomponente war SMARTSOFT[31] eingeplant. Dabei sollte jeder Komponente ein entsprechendes Modul von SMARTSOFT zukommen, über das dann die Kommunikation abgewickelt werden sollte. Leider kam SMARTSOFT aufgrund einiger Komplikationen nie zum Einsatz. Folglich mußte ein ausreichender Ersatz, basierend auf TCP/IP, implementiert werden. Dieser „Ersatz“ und das High-Level-Protokoll werden nun vorgestellt.

Zunächst wird anhand der technischen Details von TCP und IP gezeigt, daß sich dieses Basis-Kommunikationsprotokoll für unseren Fall gut eignet. Danach wird ein auf TCP/IP aufsetzendes Protokoll, über das sich die beiden Komponenten verständigen, festgelegt. Die Implementierung des LISP- und C++-Netz-Moduls bildet den Schluß des Kapitels.

4.1 Kommunikation über TCP/IP

Die Aufgabe von TCP/IP ist die Übermittlung von Daten über ein Netzwerk, von einem Rechner zu einem anderen. In unserem Fall läuft auf einem Rechner die WR-Komponente und die Graphikkomponente evtl. auf einem anderen Rechner, d.h. es ist egal, ob beide Komponenten auf einem Rechner oder auf verschiedenen Rechnern, die durch Netzwerk miteinander verbunden sind, laufen. Außerdem ist es notwendig, über die Art der Verbindung, die zwischen den beiden Rechnern bestehen soll, Kenntnis zu haben. Denn die Daten, die übertragen werden, sollen auf dem Empfängerrechner vollständig und in richtiger Reihenfolge ankommen.

Wenn wir nun das ISO OSI-Schichtenmodell für Netzwerkkommunikation betrachten, ist IP in der Vermittlungs-/Netzwerkschicht (Schicht 3) ([Web 98], [Tan 95], [Bro 94]) angesiedelt. Diese Schicht übernimmt die verbindungslose Kommunikation nach dem Prinzip des „best try“, d.h. es wird keine Garantie dafür übernommen, daß die Daten ankommen. Hier findet eine, von der darunterliegenden Netzhardware abhängige, Fragmentierung und ein Zusammenbau von Datagrammen[32] statt. Ebenso übernimmt IP das Routing der Pakete im Internet. Dabei werden mit Hilfe des *Address Resolution Protocol* (ARP) IP-Adressen auf Ethernetadressen abgebildet. Fehler, die während der Übertragung auftraten, werden erkannt und an die darüberliegenden Schicht weitergeleitet.

Diese darüberliegende Schicht ist die Transportschicht (Schicht 4), deren Aufgabe die Bereitstellung einer zuverlässigen Verbindung ist. Um die Zuverlässigkeit einer Verbindung einzustufen, wurden von der ISO die Klassen TP0 bis TP4 definiert. Bei TP0 werden Übertragungsfehler erkannt, aber nicht behoben, und TP4 beinhaltet volle Fehler-

31 SMARTSOFT ist ein Softwarerahmen, der innerhalb des SFB 527 „Integration symbolischer und subsymbolischer Informationsverarbeitung in adaptiven sensomotorischen Systemen“ entwickelt wird [SW 98]. SMARTSOFT selber ist in C++ implementiert und bietet Schnittstellen zu C++ und LISP.

32 Ein Datagramm kann als Header plus Benutzerdaten eines Netzwerkprotokolls verstanden werden.

und Flußkontrolle. Die bekanntesten Implementierungen sind UDP[33] und TCP[34], wobei UDP in etwa TP0 und TCP TP4 entspricht. Im Gegensatz zu UDP bietet TCP eine Verbindung, bei der die Daten immer vollständig und stets in der richtigen Reihenfolge beim Empfänger ankommen. Der Grund dafür liegt in der verbindungsorientierten Kommunikation[35]. Denn zur Übermittlung der Daten wird eine echte Verbindung aufgebaut, die während der Übertragung bereitgehalten und anschließend wieder abgebaut wird. Während die Verbindung besteht, findet auf der gesamten Strecke eine Fehlererkennung und Fehlerbehebung statt. Verlorengegangene Daten werden erneut angefordert und eingegangene Daten werden in der richtigen Reihenfolge zusammengestellt. Die geforderte fehlerfrei Verbindung steht damit zur Verfügung.

4.2 Protokoll

Weiter müssen wir uns darüber im Klaren sein, welche Daten und wie die Daten zwischen WR-Komponente und Graphikkomponente auszutauschen sind. Die WR-Komponente soll als Server fungieren und die Graphikkomponente als Klient, d.h. die WR-Komponente, der Server, läuft ständig und wartet darauf, daß sich ein Klient bei ihm meldet. In unserem Fall bedient der Server immer nur genau einen Klienten, d.h. wurde bereits eine Verbindung von einem Klienten zum Server aufgebaut, so kann kein weiterer Klient mehr den Server ansprechen. Sie müssen warten, bis der erste Klient seine Verbindung[36] mit dem Server beendet.

Die Daten, die zwischen der WR-Komponente und der Graphikkomponente ausgetauscht werden, sind LOOM-Ausdrücke, also kodierbar durch ASCII-Ketten. Man kann demnach sagen, daß das verwendete Protokoll direkt dem Interaktionsprotokoll von LOOM entspricht, mit dem der LOOM-Benutzer arbeitet. Verschickt werden die Daten mittels Sockets, die auf TCP/IP aufsetzen. Der grobe Ablauf einer Sitzung, wobei davon ausgegangen wird, daß der Server bereits läuft, sieht dann folgendermaßen aus:

1. der Klient meldet sich, sofern möglich, beim Server an; andernfalls muß er warten bis der Server frei ist;
2. der Server läßt ab sofort keine weiteren Klienten mehr zu und wartet auf Anfragen des Klienten;
3. der Klient läßt sich alle Objekte, die in der Wissensbasis enthalten sind, mittels der Anfrage *(retrieve ?p (object ?p))*, geben;
4. Der Klient läßt sich die spezifischen Daten eines jeden Objekts geben, also Farbe, Typ (z.B. Tisch, Stuhl, Stift, Tastatur usw.), Ausrichtung usw.; die entsprechenden

33 UDP: User Datagram Protokoll. Ein verbindungsloses Transportprotokoll des amerikanischen Verteidigungsministeriums.
34 TCP: Transmission Control Protokoll. Ein verbindungsorientiertes Transportprotokoll des amerikanischen Verteidigungsministeriums.
35 Bei einer verbindungslosen Kommunikation, wie bei UDP, kann jedes Datenpaket einen unterschiedlichen Weg zum Empfänger nehmen. Fehler in der Übertragung werden zwar erkannt und gemeldet, aber nicht behoben.
36 Eine Verbindung zwischen Klienten und Server, von Beginn der Verbindung, bis zu ihrem Ende, wird auch als Sitzung (engl. Session) bezeichnet.

Anfragen lauten:
*(pprint-object (get-instance '***obj***) nil)*
(ask (translate-obj **obj***))*
(ask (rotate-obj **obj***))*;

5. für jedes Objekt berechnet der Klient die Höhe (y-Wert), indem beim Server nachgefragt wird, welche Objekte sich alle unter diesem Objekt befinden; die Höhen dieser Objekte und jeweils ein kleiner Offset[37] werden addiert und ergeben den y-Wert für das Objekt; die Anfrage:
   (retrieve ?p (a-on-b **obj** *?p))*
   wird solange wiederholt, bis alle darunter befindlichen Objekte erfaßt wurden;
6. anschließend werden alle Objekte mit Hilfe der Kollisionserkennung so nah wie möglich zusammengerückt;
7. nun, da alle Informationen aus der Wissensbasis akquiriert wurden, wartet der Server auf benutzerinitiierte Änderungen der Objekte und reagiert entsprechend; alle Anfragen, die sich in diesem Fall ergeben können, sind in Tabelle 14 aufgelistet;
8. wurden alle Änderungen vom Benutzer vorgenommen und soll der Klient, die Graphikkomponente, beendet werden, so wird dem Server mittels *'end* signalisiert, daß der Klient die Sitzung beendet und die Verbindung gelöst werden kann; ein anderer Klient hat nun die Möglichkeit, eine Verbindung zum Server aufzunehmen;

Die Punkte 1. bis 6. dienen dem Aufbau der Szene aus den in der Wissensbasis gespeicherten Informationen. In Punkt 7. findet die Interaktion mit dem Benutzer statt. Hier kann der Benutzer Objekte anlegen, löschen und bearbeiten. Es wird solange in Punkt 7. verweilt, bis der Benutzer seine Sitzung mit der WR-Komponente beendet (Punkt 8.).

Beim Aufbau der Szene tritt, bzgl. der Stapelrelation, ein Problem auf, das nun ausführlich beschrieben wird: wie schon erwähnt, wird die Höhe eines jeden Objekts durch die Höhen (und jeweils einen kleinen Offset) der darunterliegenden Objekte bestimmt. Nun, nehmen wir an, der Benutzer hat bei seiner letzten Sitzung einen Stift auf die Sitzfläche des Stuhles gelegt. Da die exakte Form des Stuhles der Wissensbasis nicht bekannt ist, kann nicht gesagt werden, ob der Stift auf der Sitzfläche, auf der Armlehne oder Rükkenlehne liegt. Deshalb bleibt beim Aufbau der Szene aus der Wissensbasis nur noch die Aussage „der Stift liegt auf dem Stuhl“ übrig, doch das ist zu ungenau. Denn der Stift wird nun auf die absolute Höhe des Stuhles, relativ zu seinem Untergrund, positioniert. Abbildung 18 zeigt schematisch diese Situation.

37 Würden direkt die Höhen der Objekte genommen werden, so berührten sich die Objekte. Da sich aber die Objekte bei der Kollisionserkennung gerade nicht berühren dürfen, wird hier pro Objekt noch ein kleiner Offset zur Höhe addiert.

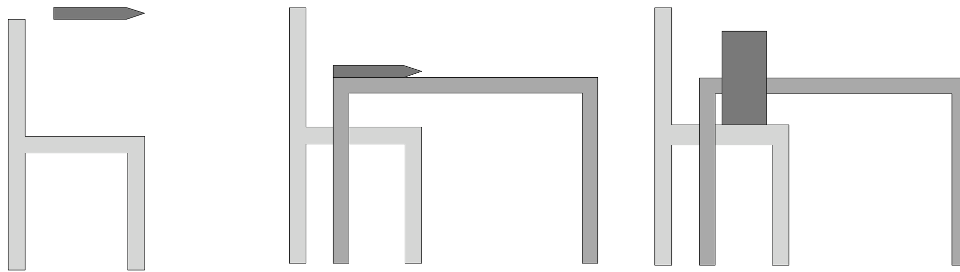

Abbildung 18: Stift liegt auf Höhe des Stuhls *Abbildung 19 : Stift wird fallen gelassen* *Abbildung 20: unmögliche Situation*

Wie man sieht „schwebt“ also der Stift über der Sitzfläche. Eine schnelle, aber, wie sich zeigen wird, nicht vollständige Lösung des Problems sieht wie folgt aus: man läßt den Stift einfach solange fallen, bis er mit dem Stuhl kollidiert und hat damit seine richtige Höhe bestimmt. Ist aber der Stuhl mit dem Stift auf der Sitzfläche ganz an den Tisch geschoben, so funktioniert obige Lösung nicht mehr, da nun der Stift zuerst mit dem Tisch kollidiert und darauf liegen bleibt, wie in Abbildung 19 dargestellt. Eine erweiterte Lösung, die im Gesamtsystem implementiert wurde, sieht vor, für die Kollisionserkennung nur den Stuhl und den Stift zu betrachten. Alle anderen Objekte nehmen nicht an der Kollisionserkennung teil. Das Problem tritt aber nur beim Aufbau der Szene auf, da sonst nur der Benutzer, über die graphische Oberfläche, die Objekte positioniert und, Dank der Kollisionserkennung, keine unerlaubten Zustände auftreten können. Ebenso wenig ist die in Abbildung 20 gezeigt Situation möglich. Denn der Benutzer hätte hierfür in seiner letzten Sitzung den Block auf den Stuhl legen und den Stuhl mit dem Block ganz an den Tisch bzw. den Block in den Tisch schieben müssen. Da dies aber die Kollisionserkennung verhindert, kann dieser Zustand nie auftreten. Abschnitt 6.2.1 befaßt sich mit der Erweiterung des Modells und zeigt eine allgemeinere Lösung für dieses Problem.

Nun aber wieder zurück zur Kommunikation.

In Abbildung 21 wird der Kommunikationsablauf graphisch dargestellt. Die Punkte 1. bis 6. wurden zu „Szene aus den Objekten aufbauen und zeichnen“ zusammengefaßt.

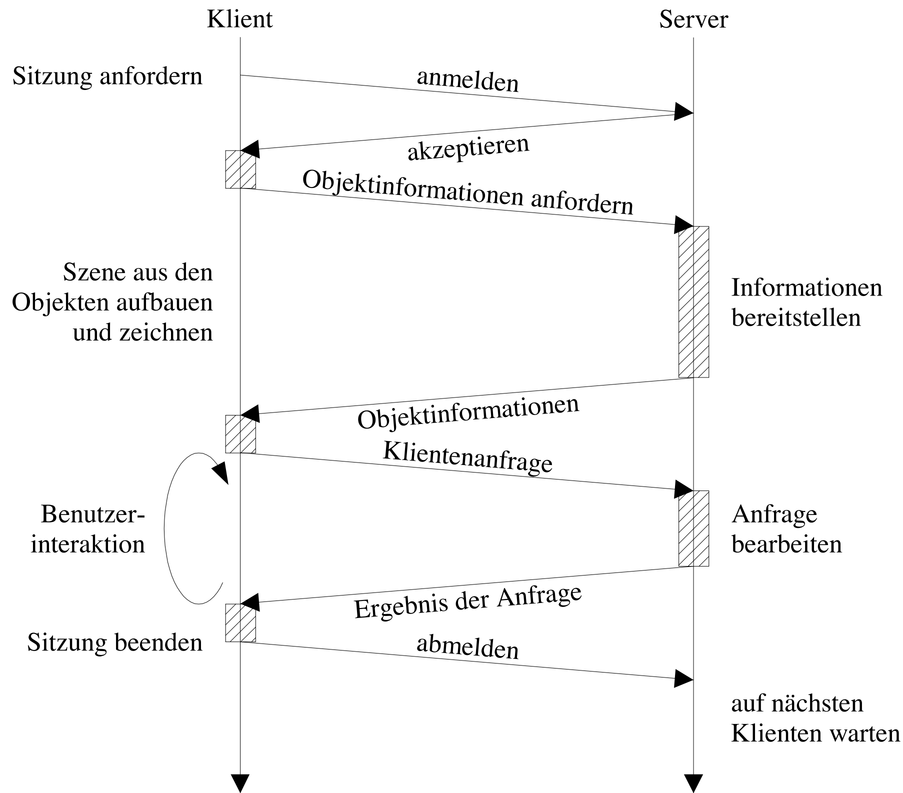

Abbildung 21: Kommunikationsablauf

In Tabelle 11 sind alle Anfragen, die von Klientenseite aus möglich sind, mit den entsprechenden Serverantworten gezeigt.

Klient	**Server**
alle Objekte in der Wissensbasis: *(retrieve ?p (object ?p))*	*(|I|***obj1** *|I|***obj2** usw.*)* so Objekte existieren, sonst *nil*
alle spezifischen Daten eines Objekts: *(pprint-object* *(get-instance* '**obj***) nil)*	*(tell (:about* **obj** *(:create* **type(obj)***)* *(a-on-b* **obj2***)* *(position* "**pos(obj)**"*)* *(direction* "**dir(obj)**"*)* *(name* "**nam(obj)**"*)* *(rgba* "**rgb(obj)**"*)* *(scale* "**sca(obj)**"*)))* so **obj** vorhanden, sonst *nil*

<table>
<tr><th>Klient</th><th>Server</th></tr>
<tr><td>Beweglichkeit des Objektes in x-,y- und z-Richtung:

(ask (translate-obj obj))</td><td>t
so obj ein translate-obj ist, sonst
nil</td></tr>
<tr><td>darf das Objekt gedreht werden?

(ask (rotate-obj obj))</td><td>t
so obj ein rotate-obj ist, sonst
nil</td></tr>
<tr><td>obj1 auf obj2 stellen:

(perform (pose
(get-instance'obj1)
(get-instance 'obj2)))</td><td>t
so die Operation zulässig ist, sonst
nil</td></tr>
<tr><td>auf welchem Objekt steht obj?

(retrieve ?p (a-on-b obj ?p))</td><td>(|I|obj1)
so die Anfrage erfolgreich war, sonst
nil</td></tr>
<tr><td>welche Objekte stehen direkt auf obj?

(retrieve ?p (a-on-b ?p obj))</td><td>(|I|obj1 |I|obj2 usw.)
so die Anfrage erfolgreich war, sonst
nil</td></tr>
<tr><td>welche Objekte stehen auf obj (transitive Hülle)?

(retrieve ?p (a-on-b* ?p obj))</td><td>(|I|obj1 |I|obj2 usw.)
so die Anfrage erfolgreich war, sonst
nil</td></tr>
<tr><td>Objektdaten ändern:

(tell (:about obj
(position "pos(obj)")
(direction "dir(obj)")
(name "nam(obj)")
(scale "sca(obj)")))</td><td>t
so die Operation zulässig ist, sonst
nil</td></tr>
<tr><td>neues Objekt obj anlegen:

(tell (:about obj
(:create type(obj))
(position "pos(obj)")
(direction "dir(obj)")
(name "nam(obj)")
(rgba "rgb(obj)")
(scale "sca(obj)")))</td><td>t
so die Operation zulässig ist, sonst
nil</td></tr>
<tr><td>Objekt obj löschen:

(forget-all-about 'obj)</td><td>t
so die Operation zulässig ist, sonst
nil</td></tr>
</table>

Klient	**Server**
Sitzung beenden: *'end*	keine Antwort vom Server

Tabelle 11: Anfragen und Ergebnisse

Damit ist der Rahmen für den Ablauf der Kommunikation der beiden Komponenten gegeben. Nun können wir uns das Kommunikationsaufkommen, das in der Beispielsitzung „stelle den Stuhl auf den Tisch“ (Abschnitt *1.1 Konzeption*) entsteht, ansehen.

Im ersten Schritt, Anwählen des Stuhls (*CHAIR1*), wird folgende Anfrage von der Graphikkomponente an die WR-Komponente gestellt:

```
(RETRIEVE ?P (AND (OBJECT ?P) (A-ON-B* ?P CHAIR1)))
```

„Welche Objekte liegen direkt oder indirekt auf dem Stuhl?“

Als Ergebnis liefert die WR-Komponente:

```
(|i|PEN1 |i|BOOK2)
```

Stift *PEN1* und Buch *BOOK2* liegen auf dem Stuhl.

Jetzt wird der Stuhl auf bzw. über den Tisch positioniert.

Im zweiten Schritt wird der Stuhl losgelassen, also erneutes Anklicken mit der Maus, und er fällt auf den Tisch, was zur Anweisung „stelle den Stuhl auf den Tisch (*OB2*)“ führt:

```
(PERFORM (POSE (GET-INSTANCE 'CHAIR1) (GET-INSTANCE 'OB2)))
```

die von der Wissensbasis mit

```
"put object |i|CHAIR1 on object |I|OB2"
```

bestätigt wird. Andernfalls würde `nil` zurückgegeben.

Jetzt müssen nur noch die neue Position des Stuhles, Buches und des Stiftes durch nachstehende Anweisungen aktualisiert werden:

```
(TELL (:ABOUT CHAIR1 (POSITION "0.187891 52.074043") (DIRECTION "139.486755")
 (NAME "Stuhl 1") (RGBA "220 200 127 0")))
```

Antwort: `|i|CHAIR1`

```
(TELL (:ABOUT PEN1 (POSITION "9.950440 53.861771") (DIRECTION "79.499664")
       (NAME "Stift 1") (RGBA "0 170 20 0")))
```

Antwort: `|i|PEN1`

```
(TELL (:ABOUT BOOK2 (POSITION "6.160450 48.424431") (DIRECTION "210.699234")
       (NAME "Buch 1") (RGBA "220 120 120 0")))
```

Antwort: `|i|BOOK1`

Damit ist die Aktion der Beispielsitzung abgeschlossen und wir können uns der Implementierung der entsprechenden Module widmen.

4.3 Implementierung

Die WR-Komponente und die Graphikkomponente benötigen noch jeweils ein weiteres Modul. Über diese Module werden sie miteinander verbunden. Die Einordnung der beiden Module bzw. der Netzwerkkomponente im Gesamtsystem ist in Abbildung 1 zu sehen. Beide Module setzen auf TCP/IP-Sockets[38] auf. Jeder Socket benötigt zwei zusätzliche Informationen, den Hostnamen und die Portnummer. Auf Serverseite ist der Hostname der DNS-Name des Rechners/Servers. Die Portnummer bestimmt den Port, den der Server für eine Verbindung mit einem Klienten bereitstellt. Auf Klientenseite wird eine Verbindung mit dem Server aufgebaut. Der Klient benötigt noch die „Adresse“ des Servers. Der DNS-Name bzw. der Port des Servers ist in Hostname bzw. in Portnummer angegeben. Abbildung 22 zeigt den Hostnamen und die Portnummer aus Sicht des Servers und Klienten.

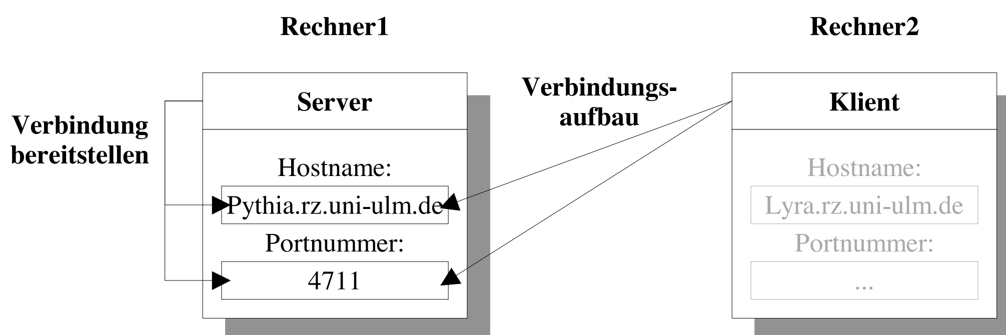

Abbildung 22: Server stellt über Hostnamen und Portnummer einen Kommunikationsendpunkt zur Verfügung. Klient baut eine Verbindung zum Server, über dessen Hostnamen und den Port mit der Nummer 4711, auf.

In den beiden folgenden Abschnitten werden zwei Module und ihre Schnittstellen vorgestellt, die einen Server-Socket bzw. einen Klienten-Socket erzeugen. Der erste Abschnitt behandelt das Modul für die WR-Komponente und im zweiten wird das Modul für die Graphikkomponente vorgestellt. Die Entwicklungsumgebung bzw. Implementierungsplattform ist in Kapitel 5 aufgeführt.

4.3.1 LISP-Netz-Modul

Da die Wissensbasis, LOOM, unter LISP läuft, und LISP standardmäßig keine Sockets bietet, müssen neue Funktionen, die die Handhabung von Sockets erlauben, definiert werden. Dies geschieht über sog. *foreign functions* ([ACL 96]). Das sind Funktionen, die auf Funktionen von Bibliotheken, die zur Laufzeit zu LISP dynamisch hinzugebunden werden, zugreifen. Für die vorliegende Arbeit mußte unter C eine Bibliothek namens *mesg.so* erstellt werden. Der Quellcode ist in *8.5 Netzwerkkomponente: LISP-Netz-Modul* abgedruckt. Um nun die Funktionen der Bibliothek LISP zugänglich zu machen, muß sie unter LISP mittels *(load “mesg.so“)* geladen werden. Schließlich

38 Sockets sind sog. Kommunikationsendpunkte [Bro 94], die von den meisten Betriebssystemen zur Verfügung gestellt werden. Entsprechend enthalten die Systembibliotheken Funktionen für den Umgang mit Sockets.

müssen noch die nötigen Funktionen für LISP definiert werden. In Tabelle 12 sind die Funktionsdefinitionen und ihre Bedeutung gezeigt.

Funktionsdefinition	Beschreibung
(ff::defforeign 'createsock :arguments '(integer integer) :return-type :void)	Hier wird die Funktion *createsock* definiert, die das Erstellen eines TCP/IP-Sockets erlaubt. Ihr erstes Argument ist ein Zeiger auf eine Zeichenkette, der Hostname des Servers (z.B. peregrin.informatik.uni-ulm.de), und das zweite Argument gibt den Port an, an dem sich ein Klient anmelden kann. Die Funktion liefert keinen Rückgabewert.
(ff::defforeign 'destroysock :return-type :void)	Ein zuvor angelegter Socket wird mit der Funktion *destroysock* freigegeben. Sie benötigt weder Argumente noch wird ein Wert zurückgegeben.
(ff::defforeign 'getmesg :return-type :integer)	Mit der Funktion *getmesg* wird solange am Socket gewartet, bis eine Nachricht eintrifft. So die Nachricht vollständig erhalten wurde, wird ein Zeiger auf die Nachricht, die als Zeichenkette vorliegt, zurückgegeben. Die Funktion benötigt keine Argumente.
(ff::defforeign 'putmesg :arguments '(integer) :return-type :void)	Die Funktion *putmesg* ermöglicht das Versenden von Nachrichten. Als Argument erhält sie einen Zeiger auf die Zeichenkette, die versendet werden soll; kein Rückgabewert.

Tabelle 12: Funktionsdefinitionen zur Socketsteuerung unter LISP

Die Nachrichtengröße, die empfangen und gesendet werden kann, ist hier auf 200000 Bytes begrenzt. Die Integration des LISP-Moduls in die WR-Komponente wird in Abschnitt *5.2.1 WR-Komponente* beschrieben.

4.3.2 C/C++-Netz-Modul

Das Modul für die Graphikkomponente wurde, wie die Graphikkomponente selber, in C++ gehalten. Es kann deshalb einfach zur Graphikkomponente hinzugenommen werden. Eine neue Klasse, die Netzwerkklasse, samt Methoden steht nun zur Verfügung. Die Headerdatei des Moduls ist in Anhang *8.6 Netzwerkkomponente: C/C++-Netz-Modul* aufgeführt. Nachfolgend werden die Methoden zur Socketsteuerung beschrieben.

Mit **net (char *, short unsigned int);** wird ein neues Netzwerkobjekt angelegt. Der erste Parameter ist ein Zeiger auf eine Zeichenkette, die den Hostnamen des Server beinhaltet (z.B. peregrin.informatik.uni-ulm.de). Der zweite Parameter gibt den Port des Servers an, zu dem eine Verbindung aufgebaut werden soll.

virtual ~net (); gibt ein Netzwerkobjekt vom Typ **net** frei und entläßt damit auch den verbundenen Socket.

Nachrichten werden mit **void putmesg (char *);** versendet. Der Parameter ist ein Zeiger auf die Nachricht, die als Zeichenkette vorliegen muß. Die Methode liefert keinen Rückgabewert.

Mit **char *getmesg (void);** werden Nachrichten empfangen. Es wird solange gewartet, bis eine Nachricht eintrifft. Diese wird dann vollständig gelesen und anschließend wird ein Zeiger auf die Nachricht, die als Zeichenkette vorliegt, zurückgegeben. Die Methode benötigt keine Parameter.

Ebenso, wie im LISP-Modul, ist die Nachrichtengröße hier auf 200000 Bytes begrenzt. Das Zusammenspiel des Moduls mit der Graphikkomponente wird in Abschnitt *5.2.2 Graphikkomponente* erläutert.

Im folgende Kapitel *5. Erstellung des Gesamtsystems* werden die Netzwerkkomponente, die WR-Komponente und die Graphikkomponente zu einem Prototypen, dem Gesamtsystem, zusammengefügt. Dabei wird auf die zugrunde liegende Architektur des Gesamtsystems und seiner Implementierung eingegangen. Die Implementierung gliedert sich in zwei Abschnitte auf, deren Themen die WR-Komponente und die Graphikkomponente sind.

5 Erstellung des Gesamtsystems

Die vorangegangenen Kapitel befaßten sich mit der Bestimmung eines adäquaten Graphikwerkzeugs, mit Wissensrepräsentation und der Netzwerkkomponente. Diese werden nun zu einem Gesamtsystem zusammengefügt (siehe Abbildung 1). Dabei ist das Graphikwerkzeug Bestandteil der Graphikkomponente.

In diesem Kapitel werden zunächst die Zielsysteme und Rahmenbedingungen zur Implementierung des Gesamtsystems abgesteckt. Danach werden die Module der Graphikkomponente betrachtet. Anschließend werden die Schnittstellen der WR-Komponente aufgezeigt. Hierfür wird anhand einiger Ausschnitte aus dem Quellcode die Vorgehensweise der Implementierung untermauert.

5.1 Zielsysteme

Die zwei wesentlichen Komponenten des Gesamtsystems sind die Graphikkomponente und die WR-Komponente. Da sie nur über die Netzwerkkomponente verbunden sind, können beide Komponenten auf unterschiedlichen Rechnern eingesetzt werden, was dazu führt, daß jeweils notwendige Hard- und Software-Voraussetzungen für beiden Komponenten getrennt betrachtet werden können. Die Netzwerkkomponente wird von jedem netzwerkfähigem System unterstützt. Da dies für die meisten Systeme zutrifft, ist dies keine wirklich ernsthafte Einschränkung.

Das Gesamtsystem kommt vorrangig auf den Systemen SUN Solaris, auf SUN Ultra-Maschinen, und Linux, auf Intel-PCs, zum Einsatz, benötigt aber auf jeden Fall die graphische Oberfläche XWindows, Version X11R5 oder höher. Beide Systeme bieten die notwendigen Software-Voraussetzungen, die im nächsten Abschnitt vorgestellt werden.

5.1.1 Entwicklungsumgebung und Plattform

Als Implementierungsplattform diente ein Intel-basierter Linux-PC. Als C-Compiler bzw. C++-Compiler wurde gcc bzw. g++ und als LISP-Umgebung Allegro Common Lisp 4.3 (ACL), von Franz, Inc., gewählt. Zusätzlich verwendete Bibliotheken sind OpenGL, GLUT, GLUI, RAPID und VCollide. Desweiteren werden noch einige Standardbibliotheken des X11-Systems benötigt, die nicht aufgeführt werden. Compiler, Bibliotheken und ACL sind ebenso auf SUN Solaris verfügbar. Zusätzlich ist für SUN Solaris Rechner mit spezieller Ausstattung ein Hardware-unterstütztes OpenGL erhältlich.

Als Entwicklungsumgebungen kamen als Editor NEdit[39], als C/C++-Debugger DDD[40] und ACL zum Einsatz.

39 NEdit: Fermi National Accelerator Laboratory. Universities Research Association, Inc., Nirvana Project http://www-pat.fnal.gov/nirvana/nedit.html

40 DDD (Data Display Debugger): Technische Universität Braunschweig, Abteilung Softwaretechnologie http://www.cs.tu-bs.de/softech/ddd

5.1.2 Wissensrepräsentationskomponente

Für die WR-Komponente ist eine LISP-Umgebung erforderlich, da die Wissensbasis, LOOM, in LISP implementiert ist.

Die WR-Komponente besteht aus der Wissensbasis LOOM, die unser Modell der Büroumgebung beinhaltet, dem LISP-Netz-Modul der Netzwerkkomponente aus Abschnitt 4.3.1 und einer Schicht die zwischen Wissensbasis und LISP-Modul vermittelt. Die Vermittlungsschicht wurde unter ACL entwickelt. Abbildung 23 zeigt den Ausschnitt des Gesamtsystems aus Abbildung 1, der die WR-Komponente mit ihren drei Modulen umfaßt.

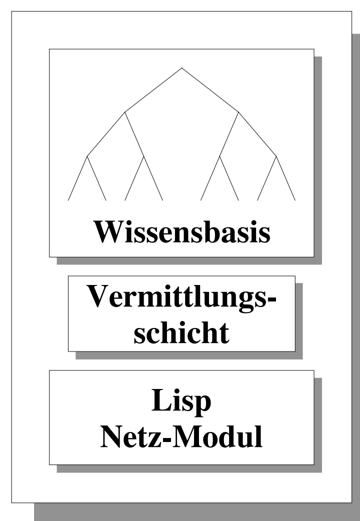

Abbildung 23: Module der WR-Komponente

5.1.3 Graphikkomponente

Die Graphikkomponente ist in C++ implementiert und benötigt die Graphikbibliotheken OpenGL, GLUT und GLUI. Eine Hardware-Unterstützung für OpenGL ist empfehlenswert, aber für das Funktionieren des Gesamtsystems nicht erforderlich. Für die Kollisionserkennung kommen die Bibliotheken RAPID und VCollide zum Einsatz. Schließlich muß noch das C/C++-Netz-Modul aus Kapitel 4 hinzugebunden werden. Über eine Kontrollschicht werden GLUT, GLUI, OpenGL, VCollide und das Netz-Modul verwaltet. Die Kontrollschicht ist in einzelne Module aufgeteilt, deren Aufgaben in Abschnitt 5.2.2 zusammengefaßt werden. Abbildung 24 zeigt die Graphikkomponente des Gesamtsystems. Zu sehen sind die einzelnen Bibliotheken, das Netzmodul und deren Zusammenhang.

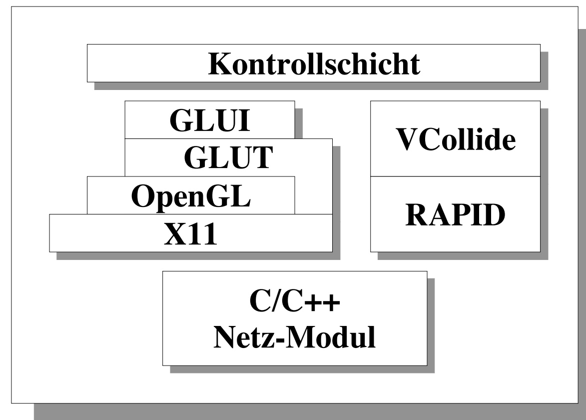

Abbildung 24: Graphikkomponente

5.2 Modulbeschreibung

In den folgenden Abschnitten wird nun für jedes Modul die Aufgabe und Funktion innerhalb des Gesamtsystems aufgezeigt. Dabei werden die wichtigsten Schnittstellen aufgeführt und, so es sich anbietet, mit Codeausschnitten untermauert.

5.2.1 WR-Komponente

Die WR-Komponente besteht, wie Abbildung 23 zeigt, aus den Modulen Wissensbasis, das in Kapitel 3 ausführlich betrachtet wurde, Netz-Modul, das in Abschnitt 4.3.1 beschrieben wurde, und der Vermittlungsschicht, die nun vorgestellt wird. Sie besteht aus beiden Funktionen *read_from_sock ()* und *server ()*. *server ()* ist die Hauptfunktion, sie legt einen Socket an und ruft die Funktion *read_from_sock ()*. Diese Funktion wartet nun bis sich ein Klient meldet. Hat ein Klient eine Verbindung aufgebaut, so leitet sie die Anfragen direkt an die Wissensbasis weiter und sendet deren Ausgabe unverzüglich an den Klienten zurück. Dies wiederholt sich solange, bis sich der Klient abmeldet. Dann wird diese Funktion beendet und es wird zur aufrufenden Funktion *server ()* zurückgekehrt. Dort wird nun der aktuelle Socket beendet, ein neuer angelegt und anschließend wird wieder *read_from_sock ()* gerufen. Die Funktion *server ()* ist als Endlosschleife implementiert und kann nur von der LISP-Konsole aus mit der Tastenkombination **ctrl-c** unterbrochen werden. Die beiden Funktionen haben folgende Gestalt:

```
(defun read_from_sock ()
  (loop
    (let* ((expr (ff:char*-to-string (getmesg)))
; wartet auf Nachrichten des Klienten
      (read-expr (read-from-string expr)) ; Nachricht in String umwandeln
      (eval-expr (eval read-expr))) ; String von Wissensbasis auswerten lassen
    (format T "~S~%" read-expr) ; String auf LISP-Konsole ausgeben
    (putmesg (ff:string-to-char* (format NIL "~S" eval-expr)))
; Ergebnisstring in Nachricht umwandeln und zurückschicken
    (when (equal expr "'end") ; Klient hat 'end gesendet, Sitzung beenden
      (return)))))

(defun server ()
  (loop
    (createsock (ff:string-to-char* "Dracula.extern.uni-ulm.de") 2722)
; Socket anlegen und warten, bis sich ein Klient meldet
```

```
    (read_from_sock) ; auf Klienten warten
    (destroysock) ; Socket beenden
    (sleep 1) ; noch kurz warten bevor der nächste Klient bedient wird
  ))
```

5.2.2 Graphikkomponente

Die Graphikkomponente besteht aus dem Hauptprogramm *ai*, das die Module *ai_object3D, ai_control, ai_scene, ai_face* und *ai_net* verwendet, wobei hier nur die relevanten Module aufgezählt wurden. *ai_net* ist das in Abschnitt 4.3.2 vorgestellte C/C++-Netz-Modul. Alle weiteren Module und das Hauptprogramm werden nun beschrieben.

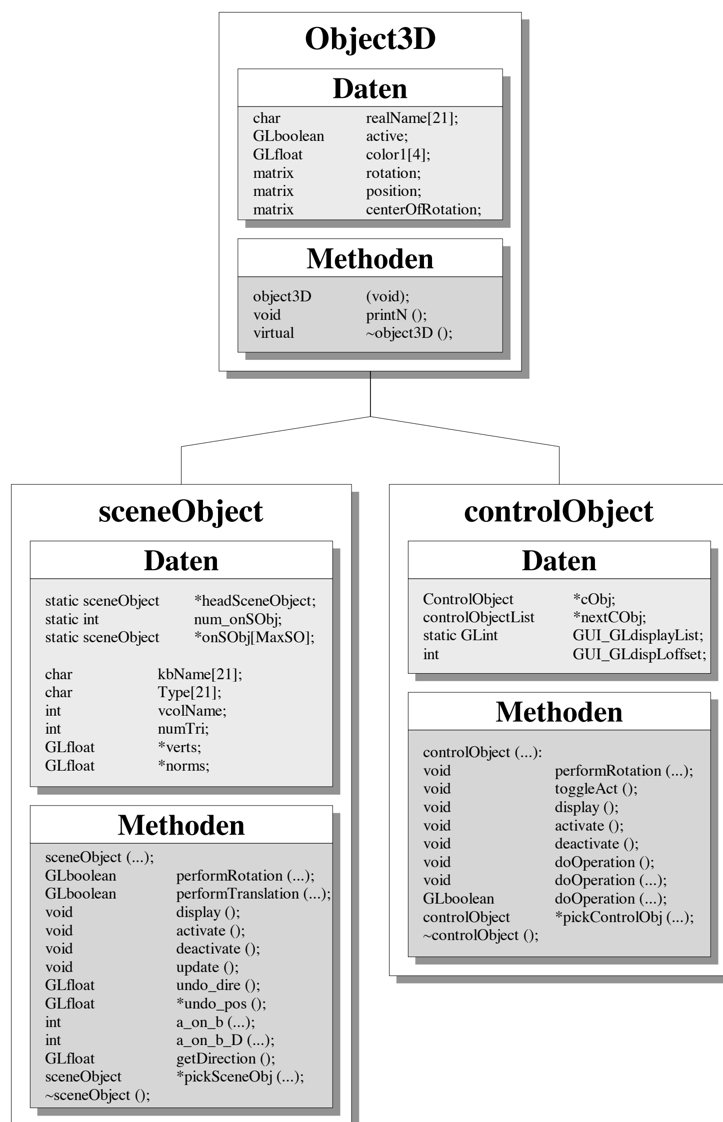

Abbildung 25: Klassenbaum der 3D-Objekte

5.2.2.1 Das Modul *ai_object3D*

Diese Modul dient der Verwaltung und Bearbeitung von 3D-Objekten. Das sind zum einen die 3D-Objekte der Kontrolleiste und zum anderen die 3D-Objekte der Büroumgebung (siehe Abbildung 2). Für jedes wird eine eigene Klasse angelegt, die von der gemeinsamen Basisklasse *object3D* abgeleitet wird. In Abbildung 25 ist der Klassenbaum für die drei Klassen zu sehen.

Die nächsten drei Abschnitte zeigen die einzelnen Klassen und beschreiben die wichtigsten Daten und Methoden.

5.2.2.1.1 Klasse *object3D*

Daten:

- char realName[21];
  Name des Objekts;
- GLboolean active;
  zeigt an, ob das Objekt aktive ist, also mit der Maus angewählt wurde;
- GLfloat color1[4];
  bestimmt die Farbe des Objekts;
- matrix rotation;
  diese Matrix gibt die Ausrichtung des Objekte bzgl. x-, y- und z-Achse an;
- matrix position;
  diese Matrix bestimmt die Position des Objekts im Raum;
- matrix centerOfRotation;
  diese Matrix bestimmt den Schwerpunkt des Objekts;

Methoden:

- *object3D (void);*
  erzeugt ein neues Objekt vom Typ *object3D*;
- *void printN ();*
  gibt den Namen des Objekts auf Konsole aus;
- *~object3D ();*
  löscht ein Objekt vom Typ *object3D*;

5.2.2.1.2 Klasse *sceneObject*

Diese Klasse erbt die Datenstrukturen und eine Methode der Klasse *objeck3D*. Hier werden entsprechend für die Objekte der Wissensbasis 3D-Objekte angelegt. Form, Position, Drehung bzw. Ausrichtung und Farbe sind Bestandteile eines 3D-Objektes und werden in diesem Datenobjekt verwaltet. Die Klasse bietet Methoden ein 3D-Objekt aktiv zu setzten, zu verschieben und zu drehen. Nun werden die wichtigsten Daten und

Methoden dieser Klasse aufgeführt.

Daten:

- *static sceneObject *headSceneObject;*
  erstes Objekt der Liste der 3D-Objekte der Szene;
- *static int num_onSObj;*
  Anzahl Objekte auf einem Objekt;
- *static sceneObject *onSObj[MaxSO];*
  Liste der Objekte auf einem Objekt;
- *char kbName[21];*
  Name des Objekts in der Wissensbasis;
- *char Type[21];*
  zugehörige Klasse des Objekts in der Wissensbasis;
- *int vcolName;*
  Name des Objekts in der Kollisionserkennung;
- *int numTri;*
  Anzahl Dreiecke des Objekts;
- *GLfloat *verts;*
  Koordinaten der Dreieckspunkte des Objekts;
- *GLfloat *norms;*
  Normalenvektoren der Dreiecke des Objekts

Methoden:

- ***sceneObject** (char *rName, char *kbN, char *ty, GLfloat *col, Glfloat *actCol, matrix rot, matrix pos, matrix centerOfRot, int nTri, GLfloat *tVerts, GLfloat *tNorms, GLboolean *apm);*

  legt ein 3D-Objekt für ein Objekt in der Wissensbasis an. Der Name des Objekts wird in *rName*, die Bezeichnung des Objekts in der Wissensbasis in *kbN* angegeben, in *ty* der Typ des Objekts bzw. die zugehörige Klasse, in *col* seine Farbe, in *actCol* die Farbe die das angewählte 3D-Objekt erhält, in *pos* seine Position, in *rot* seine Rotationsmatrix, in *centerOfRot* sein Schwerpunkt, in *nTri* die Anzahl Dreiecke, aus denen das 3D-Objekt besteht, in *tVerts* die 3D Koordinaten der Dreiecke, in *tNorms* die Normalenvektoren der Dreiecke, beiden (*tVerts*, *tNorms*) werden von OpenGL zur Beschreibung der Form des 3D-Objekts benötigt und in *apm*, ob es sich um frei bewegliches, verschiebbares und/oder drehbares Objekt handelt. Automatisch beim Anlegen wird ein Zeiger auf dieses 3D-Objekte in eine lineare Liste eingefügt, d.h. die Zeiger aller 3D-Objekte werden in dieser Liste verwaltet.

  Die Form, die als Menge von Dreiecken mit entsprechenden Normalenvektoren vorliegt, sowie die Rotationsmatrix und die Position, werden von OpenGL benötigt. Die

Dreiecke alleine, mit der Rotationsmatrix und der Position werden der Kollisionserkennung zur Verfügung gestellt.

Nachfolgende Tabelle 13 zeigt die Zuordnung der in der Wissensbasis gespeicherten Daten eines Objekts zu den Daten in diesem Methodenaufruf.

Objekt in der Wissensbasis	**Objekt als *sceneObjekt***
(tell (:about **obj**	**obj** -> *kbN*
(:create **type(obj)***)*	**type(obj)** -> *ty*
(position "**pos(obj)**"*)*	*pos* wird aus **pos(obj)** und den darunter befindlichen Objekte berechnet;
(direction "**dir(obj)**"*)*	**dir(obj)** -> *rot*
(name "**nam(obj)**"*)*	**nam(obj)** -> *rName*
(rgba "**rgb(obj)**"*)*	**rgb(obj)** -> *col*
(scale "**sca(obj)**"*)))*	wird nicht unterstützt
	*nTri, *tVerts, *tNorms* werden entsprechend dem Typ bzw. Klassen in der Wissensbasis gesetzt; *tVerts* und *tNorms* beschreiben die Form des 3D-Objekts und werden OpenGL zur Verfügung gestellt;
	apm spiegelt die Eigenschaften des Objekts in der Wissensbasis wider;

Tabelle 13: Zuordnung: von Wissensbasis nach C++ Klasse

- *GLboolean* ***performRotation*** *(GLfloat dx, GLfloat dy, GLfloat dz);*

  führt für ein 3D-Objekt eine Drehung um die x-Achse um *dx* Grad, für y um *dy* Grad und für z um *dz* Grad durch; bei Erfolg liefert die Methode *GL_TRUE*, sonst *GL_FLASE*;

- *GLboolean* ***performTranslation*** *(GLfloat dx, GLfloat dy, GLfloat dz);*

  bewegt ein 3D-Objekt in x-Richtung um *dx*, in y um *dy* und in z um *dz*-Einheiten; bei Erfolg liefert die Methode *GL_TRUE*, sonst *GL_FLASE*;

- *void* ***display*** *();*

  zeichnet das angegeben 3D-Objekt in OpenGL;

- *void* ***activate*** *();*

  versetzt ein 3D-Objekt in den aktiv-Statuts, es wird nun mit kleinen Quadraten an seiner Oberfläche, die die Farbe aus *actCol* besitzen, versehen;

- *void* ***deactivate*** *();*

  versetzt das 3D-Objekt wieder in den Normalzustand, es werden keine kleinen Quadrate mehr an seiner Oberfläche gezeichnet;

- *void* ***update*** *();*

  sendet die aktuellen Daten eines Objekts zur Wissensbasis;

- *GLfloat* ***undo_dire*** *();*

  erfragt die Ausrichtung des Objekts in der Wissensbasis; zurückgegeben wird die Ausrichtung im Gradmaß;

- *GLfloat ****undo_pos*** *();*

  erfragt die Position des Objekts in der Wissensbasis; liefert die Werte für x, y und z;

- *int* ***a_on_b*** *(sceneObject **);*

  liefert die Anzahl der 3D-Objekte und eine Liste der 3D-Objekte, vom Typ *sceneObjekt*, die sich direkt oder indirekt auf diesem 3D-Objekt befinden;

- *int* ***a_on_b_D*** *(sceneObject **);*

  liefert die Anzahl der 3D-Objekte und eine Liste der 3D-Objekte, vom Typ *sceneObjekt*, die sich direkt auf diesem 3D-Objekt befinden;

- *GLfloat* ***getDirection*** *();*

  extrahiert aus der Rotationsmatrix eines 3D-Objekts seine Drehung um die y-Achse, also seine Ausrichtung;

- *sceneObject ****pickSceneObj*** *(GLint x, GLint y);*

  liefert das dem Benutzer nächste 3D-Objekt bzgl. der Bildschirmkoordinaten *x* und *y*; wird für das Anwählen der 3D-Objekte mit der Maus gebraucht;

- ***~sceneObject*** *();*

  löscht ein 3D-Objekt und löscht den Zeiger auf dieses 3D-Objekt aus der Liste aller 3D-Objekte;

5.2.2.1.3 Klasse *controlObject*

Die Klasse *controlObject* verwaltet die Elemente der Kontrolleiste. Elemente können bestimmten Gruppen zugeordnet werden, die ein Verhalten von Radiobuttons und Checkboxen, wie sie von üblichen graphischen Oberflächen her bekannt sind, ermöglichen. Dieses Ein/Ausschaltverhalten wird über zwei Methoden *activate ()* und *deactivate ()* geregelt. Ebenso werden Methoden zur Positionierung und Rotation bereitgestellt. Die wichtigsten Daten und Methoden werden nun aufgelistet.

Daten:

- *controlObject *cObj;* und *controlObjectList *nextCObj;*
  bilden die Liste der Objekte der Kontrolleiste;

- ***controlObject*** *(char *rName, GLfloat *col, GLfloat *actCol, matrix rot, matrix pos, matrix centerOfRot, ButtonType bt, int offset);*

  legt ein neues 3D-Objekt vom Typ *controlObject* an. Wie die 3D-Objekte vom Typ *sceneObject* werden diese in einer linearen Liste verwaltet. In *rName* wird der Name des 3D-Objekts angegeben, in *col* die Farbe, in *actCol* die Farbe für das aktivierte 3D-Objekt (wird aber nicht verwendet), in *rot* seine Rotationsmatrix, in *pos* seine Position, in *centerOfRot* sein Schwerpunkt, in *bt* der Typ des Elements, Radiobutton oder Checkbox. *offset* ist für die interne Verwaltung und nicht weiter von Bedeutung.

- *void* ***performRotation*** *(GLfloat dx, GLfloat dy, GLfloat dz);*

  führt für ein 3D-Objekt eine Drehung um die x-Achse um *dx* Grad, für y um *dy* Grad und für z um *dz* Grad durch;

- *void* ***display*** *();*

  zeichnet das angegeben 3D-Objekt in OpenGL;

- *void* ***activate*** *();*

  versetzt ein 3D-Objekt in den aktiv-Statuts, es wird nun bzgl. der Farbe teilweise invers dargestellt, d.h. die Farbe aus *actCol* findet keine Verwendung; entsprechend des zugehörigen Buttontyps wird ggf. ein zuvor aktives Element bzw. 3D-Objekt mittels der folgenden Methode ***deactivate*** wieder normal dargestellt (z.B. will man genau diesen Effekt bei Radiobuttons);

- *void* ***deactivate*** *();*

  versetzt das 3D-Objekt wieder in den Normalzustand, es wird normal gezeichnet;

- *controlObject* ****pickControlObj*** *(GLint x, GLint y);*

  liefert das 3D-Objekt bzgl. der Bildschirmkoordinaten *x* und *y*; wird für das Anwählen der Elemente bzw. 3D-Objekte mit der Maus benötigt;

- ***~controlObject*** *();*

  löscht ein 3D-Objekt und löscht den Zeiger auf dieses 3D-Objekt aus der Liste aller 3D-Objekte;

5.2.2.2 Das Modul *ai_scene*

Diese Modul ist für die dreidimensionale Darstellung der Büroumgebung bzw. der entsprechenden 3D-Objekte zuständig. Hier werden der Ausschnitt des Bereichs des Fensters (siehe Abbildung 2), in den die Szene gezeichnet wird, und die Kollisionserkennung initialisiert. Die Objekte werden aus der Wissensbasis ausgelesen, in der

Kollisionserkennung abgelegt und für jedes Objekt wird seine Form und seine Höhe, also der y-Wert, ermittelt. Ebenso werden Funktionen zur Verfügung gestellt, mit denen sich die Szene drehen und verschieben läßt. Die wichtigsten Funktionen werden nun im einzelnen beschrieben.

- *extern **VCollide** collEng; // create collision detection engine*

  erzeugt ein Datenobjekt der Kollisionserkennung für die 3D-Objekte der Szene;

- *void **SceneViewport** (GLenum mode, GLint x, GLint y); // create viewport*

  legt den Ausschnitt im Fenster (siehe Abbildung 2) fest, in welchen die Szenen gezeichnet wird; die Übergabeparameter haben folgende Bedeutung: der Wert von *mode* kann *GL_RENDER* oder *GL_SELECT* sein; *GL_RENDER* besagt, daß die Szene zum Zwecke der Darstellung gezeichnet werden soll, *x* und *y* haben hier keine Funktion; *GL_SELECT* rechnet die Szene die im Fenster gezeigt wird nur für den Pixel im Fenster mit der Koordinate *(x, y)*; benötigt wird dies für die Selektion von 3D-Objekten mit der Maus; anschließend wird in Erfahrung gebracht, welche Objekte sich „hinter“ diesem Pixel verbergen, gestaffelt in der räumlichen Tiefe bzgl. der Sicht des Benutzers;

- *void **SceneInit** (void); // get scene objects from KB*

  liest die Objekte aus der Wissensbasis, legt dann für jedes Objekt ein 3D-Objekt, also ein Datenobjekt vom Typ *sceneObject* an, damit verbunden die Form usw. und fügt diese Form mit Rotationsmatrix und errechneter Position in die Kollisionserkennung ein; sie ruft weitere Funktionen, die nun angegeben werden, auf:

- *void **get_objects** (void);*

  liest aus der Wissensbasis die Liste der Objektnamen aller Objekte aus;

- *sceneObject ***get_objects_cr** (char *kbN, sceneObject *last);*

  liest zu einem, in *kbN*, der eben erfragten Objektnamen, die spezifischen Daten aus der Wissensbasis, legt dafür ein Datenobjekt vom Typ *sceneObject* an, fügt dies in die Liste der 3D-Objekte der Szene hinter das Objekt *last* ein und liefert einen Zeiger auf das neue Objekt zurück; folgende drei Funktionen werden hier noch benötigt, wobei die beiden ersten essentiell sind;

- *GLfloat ****ext_shape** (char *m, int *nT, matrix *cOfR);*

  entsprechend des Typs des ausgelesenen Objekts wird hier seine Form bestimmt; dazu ist der Typ des Objekts in *m* zu übergeben, zurückgegeben werden zwei Zeiger, einer auf die Liste der Dreieckskoordinaten, der andere auf die Liste der zugehörigen Normalenvektoren; in *nT* ist die Anzahl Dreiecke, aus denen das 3D-Objekt besteht, und in *cOfR* sein Schwerpunkt gegeben;

- *GLfloat **get_y** (char *m);*

  hier wird die y-Position eines 3D-Objekts berechnet, indem die Höhen aller unter

ihm befindlichen Objekt und ein kleiner Offset addiert werden; übergeben wird der Wissensbasisname *m* des Objekts

- *void **vColNewObj** (sceneObject *sO);*

  fügt ein 3D-Objekt vom Typ *sceneObject* in die Kollisionserkennung ein; dazu werden die Form, also die Dreiecke, die Rotationsmatrix und die Position benötigt;

- *void **SceneDisplay** (void); // display scene objects*

  veranlaßt das Zeichnen aller 3D-Objekte der Szene;

- *void **performWholeScnRot** (GLfloat dx, GLfloat dy, GLfloat dz, matrix *m);*

  dreht die gesamte Szene um die angegebenen Winkel; die Drehung der Szene ist in der Matrix *m* gespeicher;

- *void **performWholeScnTrans** (GLfloat dx, GLfloat dy, GLfloat dz, GLfloat *vek);*

  bewegt die Szene um die Distanzen *dx, dy* und *dz* bzgl. der alten Position, die in *vek* gespeichert ist; im Gegensatz zu den 3D-Objekten liegt hier die Position als Vektor mit drei Elementen, und nicht als Matrix, vor;

5.2.2.3 Das Modul *ai_face*

In diesem Modul wird die 2D Benutzerschnittstelle aus Abbildung 3 definiert. Dieses Modul basiert zum größten Teil auf der Bibliothek GLUI, die die 2D Bildschirmelemente, wie Textfelder, Radiobuttons sowie Checkboxen usw. beinhaltet. Sie setzt vollständig auf GLUT auf und ist somit völlig plattformunabhängig. Die wichtigsten Funktionen werden nun vorgestellt.

- *void **initGLUIwindow** (int mainwin);*

  initialisiert das Fenster in das die 2D Benutzerschnittstelle gezeichnet werden soll; übergeben wird die ID des Hauptfensters, in unserem Fall ist dies das Fenster in Abbildung 2; GLUI benötigt diese ID für interne Zwecke; desweiteren werden die fünf Buttons *Create, Delete, Update, Save-KB* und *Quit*, die auf der linken Seite des Fenster in Abbildung 3 zu sehen sind, angelegt; damit verbunden wird beim Anklikken, eines entsprechenden Buttons, eine dedizierte Funktion gerufen; jede dieser fünf Funktionen wird anschließend beschrieben; ebenso werden hier die Textfelder, die auf der rechten Seite des Fenster zu sehen sind, definiert; sie spiegeln die Attribute, Farbe, Position usw. eines Objekts wider;

- *void **quitPrg** (int);*

  diese Funktion ist mit dem Button-*Quit* verbunden und beendet das Programm;

- *void **createObj** (int);*

  entsprechende der Attribute, die sich auf der rechten Seite des Fensters befinden, wobei der y-Wert gleich Null sein muß, wird versucht ein Objekt anzulegen; schlägt

dies fehl, so wird am unteren Rand des Fenster, in *Message*, eine entsprechende Fehlermeldung ausgegeben;

- *void **deleteObj** (int);*

  löscht ein Objekt in der Wissensbasis, dessen Namen im Feld *KB-Name* angegebenen ist;

- *void **updateObj** (int);*

  aktualisiert die Attribute des Objekts in der Wissensbasis, dessen Namen im Feld *KB-Name* angegebenen ist;

- *void **saveKB** (int);*

  speichert die aktuelle Szene in der Wissensbasis ab; hier handelt es sich um eine Datei, die alle notwendigen Beschreibungen der Szene enthält; der Pfad und der Name der Datei sind in der Wissensbasis definiert;

5.2.2.4 Das Modul *ai_control*

Dieses Modul legt den Ausschnitt des Bereichs des Fensters (siehe Abbildung 2) fest, in den die Kontrolleiste gezeichnet wird. Ebenso werden die Elemente der Kontrolleiste angelegt und eine Funktion für deren Darstellung bereitgestellt.

- *void **GUIviewport** (GLenum mode, GLint x, GLint y)*

  legt den Ausschnitt im Fenster (siehe Abbildung 2) fest, in welchen die Kontrolleiste gezeichnet wird; die Übergabeparameter haben folgende Bedeutung: der Wert von *mode* kann *GL_RENDER* oder *GL_SELECT* sein; *GL_RENDER* besagt, daß die Szene zum Zwecke der Darstellung gezeichnet werden soll, *x* und *y* haben hier keine Funktion; *GL_SELECT* berechnet die Kontrolleist nur für den Pixel mit der Koordinate *(x, y)*; benötigt wird dies für die Selektion eines Kontrolleistenelements mit der Maus; anschließend wird in Erfahrung gebracht, welches Elemente bzw. Objekt sich „hinter" diesem Pixel verbirgt;

- *void **GUIinit** (void); // create control objects*

  die Elemente der Kontrolleiste werden hier definiert;

- *void **GUIdisplay** (void); // display control objects*

  veranlaßt die Berechnung und Darstellung der Kontrolleiste im Fenster (siehe Abbildung 2);

- *void **GUIupdateRot** (matrix *); // update the gyroscope*

  wurde ein 3D-Objekt der Szene angeklickt, so wird dessen Ausrichtung auf das Element am linken Rand des Fenster übertragen; es zeigt jeweils die Ausrichtung des aktuell aktiven 3D-Objekts der Szene; ist keines aktiv, so zeigt dieses Element die Drehung der gesamten Szene an;

5.2.2.5 Das Hauptprogramm *ai*

Dieser Teil faßt die oben angeführten Module - zur Graphikkomponente - zusammen. Durch die Aufteilung in Module bleibt das Programm übersichtlich und kompakt. Der Ablauf der Initialisierung der Graphikkomponente, die in der Hauptfunktion ***main*** *(int argc, char** argv)* erfolgt, wird nun betrachtet. Dabei werden die wichtigsten Funktionen herausgegriffen:

- ***glutInitDisplayMode*** *(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);*

  mit *GLUT_DOUBLE* wird GLUT angewiesen für das Zeichen der Szene Double Buffering zu verwenden; *GLUT_RGB* besagt, daß Szenen in 24 bzw. 32 Bit Farbtiefe gerechnet werden; mit *GLUT_DEPTH* wird für das Zeichnen der Szene ein z-Buffer bzw. ein Depth-Buffer verwendet;

- ***glutInitWindowSize*** *(WIN_SIZE, WIN_SIZE);*

  gibt die initiale Ausdehnung des Fensters (Abbildung 2) in x und y Richtung an; in diesem Fall ist das Fenster (ohne Fensterrahmen) quadratisch;

- *mainwin =* ***glutCreateWindow*** *("AI");*

  erzeugt das Fenster mit angegebener Größe und Namen „AI“ auf dem Bildschirm;

- ***init*** *(); // Fensterinitialisierung*

  nimmt einigen Initialisierungen für OpenGL vor; hier werden Lichtquellen definiert bzw. angeschaltet, die Art der Schattierung gewählt, die Form der Lichtdurchlässigkeit definiert bzw. angeschaltet usw.

- ***GUIinit*** *(); // Kontroll-Objekte anlegen*

  legt die Elemente der Kontrolleiste an und zeichnet diese;

- ***SceneInit*** *(); // Szene aus den Daten der WB erstellen*

  baut eine Verbindung zur WR-Komponente auf, liest die Objektdaten aus, generiert daraus die Szene der Büroumgebung und zeichnet diese;

- ***glutDisplayFunc*** *(display);*

  mit dieser Funktion werden die Kontrolleiste und die Szene auf dem Bildschirm angezeigt;

- ***glutReshapeFunc*** *(reshape);*

  zeichnet den Fensterinhalt erneut, so dessen Größe geändert oder des Fenster von einem anderen überlagert wurde;

- ***glutMouseFunc*** *(mouse_press);*

  verwaltet Benutzereingaben bzgl. Mausklicks, d.h. 3D-Objekte der Szene bzw. 3D-Objekte aus der Kontrolleiste können angewählt werden;

- ***glutMotionFunc** (mouse_move);*

  verwaltet Benutzereingaben bzgl. Mausbewegung; ermöglicht, über die Maus 3D-Objekte der Szene bzw. die gesamte Szene zu drehen und zu verschieben;

- ***glutKeyboardFunc** (keyboard);*

  verwaltet Benutzereingaben bzgl. Tastatureingaben; hier wird nur die <ESC>-Taste abgefragt, mit der das Programm beendet werden kann;

- ***initGLUIwindow** (mainwin);*

  veranlaßt das Zeichnen des Fenster der 2D Benutzerschnittstelle (Abbildung 3);

- ***updateAllObjs** ();*

  rückt die 3D-Objekte der Szene so nah wie möglich zusammen;

- ***glutMainLoop** ();*

  alle Initialisierungen wurden vorgenommen; Benutzereingaben werden nun erkannt und bearbeitet; folglich wird bei Bedarf die Szene neu gezeichnet; dies betrifft 3D-Objekte der Szene und Elemente der Kontrolleiste; damit verbunden werden auch die Attribute in der 2D Benutzerschnittstelle aktualisiert;

6 Bewertung

In diesem Kapitel wird das Gesamtsystems bewertet. Dabei wird im ersten Teil des Kapitels auf die Performanz der WR-Komponente und der Graphikkomponente, im Vergleich auf unterschiedlichen Plattformen, eingegangen. Der zweiten Teil des Kapitels widmet sich den Perspektiven des Gesamtsystems. Hier werden die Erweiterungsmöglichkeiten des Gesamtsystems bzw. der einzelnen Komponenten untersucht. Unter dem Aspekt der Erweiterungen werden im dritten Teil mögliche Einsatzgebiete vorgestellt.

6.1 Messungen

Die Performanz des Gesamtsystems teilt sich in drei Bereiche auf:

- Wissensbasis
  Wie schnell kann die Wissensbasis Anfragen bearbeiten, in Abhängigkeit von der Objektzahl?
- Zeichnen bzw. Berechnen der Szene
  Wieviel Zeit benötigt das Graphikwerkzeug zur Darstellung der 3D-Objekte bei steigender Anzahl an Dreiecken?
  In der Testszene (Abbildung 30) und der Beispielszene (Abbildung 2) besteht ein Objekt im Schnitt aus ca. 50 Dreiecken.
- Kollisionserkennung
  Wie schnell kann, bzgl. steigender Objektzahl bzw. Anzahl Dreiecke, eine Kollisionserkennung durchgeführt werden?

Für die Messung der Performanz dieser drei Bereiche ist die Kenntnis der Meßumgebung notwendig. In unserem Fall kommen für die Wissensbasis und die Kollisionserkennung zwei und für das Zeichnen sogar drei Umgebungen in Frage.

Folgende Plattformen kommen zum Einsatz:

1. Intel Pentium II-266, 128MB Hauptspeicher, Betriebssystem: Linux;
   hier wird die Performanz der drei genannten Bereiche gemessen;
2. Sun Ultra-2, 384MB Hauptspeicher, Betriebssystem: Solaris 2.5.1;
   alle drei Bereiche werden der Messung unterzogen;
3. Sun Ultra-2, 384MB Hauptspeicher, Hardware-unterstütze 3D Graphik: Sun Creator3D, Betriebssystem: Solaris 2.5.1;
   hier erfolgt eine weitere Messung der Zeit für das Zeichnen bzw. Berechnen der Szene;

Da beim Bearbeiten der Szene, Verschieben, Drehen usw. der Objekte, die Kollisionserkennung und das Zeichnen der Szene einhergehen, werden sie im weiteren Verlauf unter dem Begriff Bildschirmdarstellung zusammengefaßt. Ein Vergleich der Plattformen wird für jeden Bereich der Messung erstellt. Unter diesem Aspekt behandeln die nächsten beiden Abschnitte die Wissensbasis und die Bildschirmdarstellung.

6.1.1 Bearbeitungszeiten der Wissensbasis

Die Messungen für die Wissensbasis erstrecken sich über 1 bis 501 Objekte, in 50er Schritten. Bei den Objekten handelt es sich um das Büro und 500 Bücher (siehe Abbildung 30). Für jeden Schritt der Messung ist ein Katalog von Anfragen von der Wissensbasis zu bearbeiten. Dabei wird die reine Bearbeitungszeit, in Millisekunden (ms), der Wissensbasis gemessen. Der aufkommende Netzwerkverkehr wird nicht in die Messung miteinbezogen, da es sich nur um Datenmenge geringer Größe handelt und die Anfragen nur relativ selten gestellt werden. Dies gilt nicht für den Aufbau der Szene aus der Wissensbasis, wie wir später sehen werden. Tabelle 14 zeigt die Anfragen, die von der Wissensbasis zu beantworten sind.

	Anfrage	**Beschreibung**
1)	(retrieve ?p (object ?p))	liefert alle Instanzen vom Typ *object*;
2)	(retrieve ?p (and (object ?p) (a-on-b ?p FLOOR)))	gibt alle Objekte zurück, die sich direkt auf dem Boden im Büro befinden;
3)	(retrieve ?p (and (object ?p) (a-on-b* ?p FLOOR)))	gibt alle Objekte aus, die sich im Büro befinden;
4)	(perform (pose (get-instance 'B0_0) (get-instance 'FLOOR)))	stellt das Objekt *B0_0*, ein Buch, „auf“ das Objekt *FLOOR*, das Büro;
5)	(tell (:about B0_0 ...))	aktualisiert die Daten des Objekts *B0_0*;

Tabelle 14: Anfragen

Die Messung wurden unter LOOM auf Allegro Common Lisp auf den beiden Plattformen Sun Ultra-2 und Linux, wie oben unter Punkt 1. und 2. spezifiziert, durchgeführt. Die Ergebnisse der einzelnen Anfragen sind in Abbildungen 26 bis 28 angegeben. Auf Abbildungen zu den Anfrage 4) und 5) wurde verzichtet, da sich die Bearbeitungszeit für beide Plattformen bei Anfrage 4) konstant auf 2 ms und bei Anfrage 5 konstant auf 1 ms beläuft.

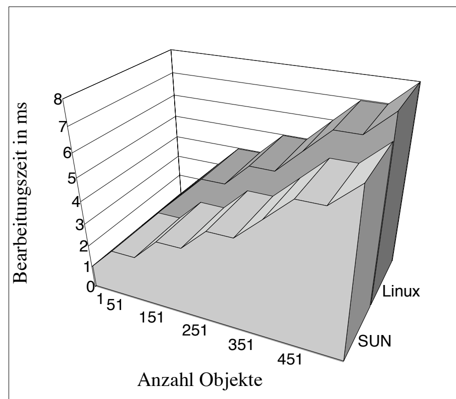

Abbildung 26: Anfrage 1)

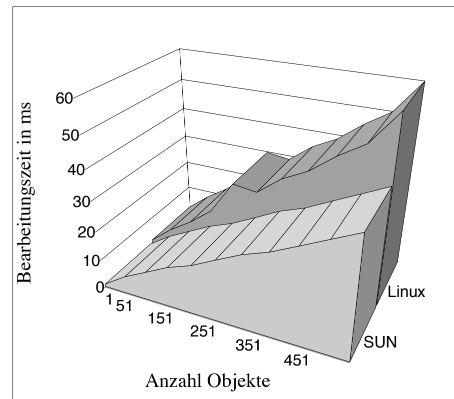

Abbildung 27: Anfrage 2)

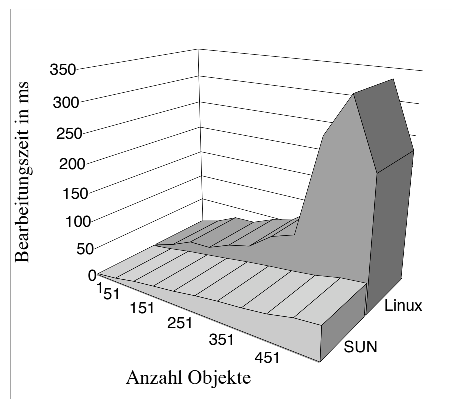

Abbildung 28: Anfrage 3)

Wie in den Abbildungen 26 bis 28 zu sehen ist, steigt die Bearbeitungszeit, mit stetig zunehmender Objektzahl, bei Anfrage 2) merklich (bis zu 60 ms) und bei Anfrage 3) deutlich an (bis zu 350 ms). Auf dem Linuxsystem benötigt Anfrage 3) sogar beträchtlich mehr Zeit als vergleichsweise auf Sun Ultra-2 Solaris 2.5.1. Dies wird beim Linuxsystem bei sehr vielen Objekten zum Flaschenhals.

Da für die Umgebung eines Serviceroboters ein Raum nicht einen derart hohen Detailgrad aufweist, der 500 oder mehr Objekte benötigt, wird die Bearbeitungszeit von Anfrage 3) unser Gesamtsystem nur wenig beeinträchtigen. So muß die Wissensbasis z.B. für das Büro, in Abbildung 2, gerade mal 30 Objekte verwalten. Da eine Fülle von Anfragen nur beim Aufbau der Szene, aus den Objekten der Wissensbasis, auftritt, genügt die Performanz der Wissensbasis unseren Ansprüchen. Für den Aufbau (das Büro und die 500 Bücher) benötigt das Gesamtsystem unter Linux einmalig ca. 60 s, unter Sun Ultra-2 Solaris 2.5.1 ca. 37 s. Während der Arbeit mit dem Gesamtsystem erfolgt aber nur beim Anwählen eines Objekts ein Anfrage, wie sie 3) zeigt.

6.1.2 Bearbeitungszeiten der Bildschirmdarstellung

Der nächste Schritt führt uns zur Bildschirmdarstellung. Da die Szene bzw. Objekte

gedreht, verschoben und bearbeitet werden können, ist hier eine schnelle Darstellung der Szene notwendig, andernfalls wäre das Arbeiten mit dem Gesamtsystem nicht zumutbar. Die Bildschirmdarstellung umfaßt in unserem Fall das erneute Zeichnen der Szene und die Kollisionserkennung zwischen den Objekten. Denn gerade das Bearbeiten der Szene, also das Drehen oder Verschieben der Objekte, bedarf für jede noch so kleine Änderung einer Kollisionserkennung. Die Bildrate setzt sich hierfür aus der Zeit für die Kollisionserkennung und der Zeit für das Zeichnen der Szene zusammen.

Wie im vorherigen Abschnitt werden in 50er Schritten 1 bis 501 Objekte (Büro und 500 Bücher) gezeichnet und eine Kollisionserkennung durchgeführt. Die Testszene, das Büro und die 500 Bücher, ist in Abbildung 30 zu sehen. Da die Bildschirmdarstellung nicht von der Objektanzahl, sondern von der Anzahl an Dreiecken abhängt, zeigt Abbildung 29 den Zusammenhang zwischen Dreieckszahl und Objektzahl für unsere Testszene. In diesem Fall besteht ein linearer Zusammenhang zwischen der Anzahl Objekte und der Anzahl Dreiecke. Allgemein besteht kein Zusammenhang zwischen Anzahl Objekte und Anzahl Dreiecke, da z.B. komplexere Objekte, wie die Säule in Abbildung 2, aus mehreren Dreiecken aufgebaut sind, als einfache Objekte, z.B. ein Buch, wie in Abbildung 30. Für jede Plattform beträgt die Auflösung des Fensters, in das die Ausgabe erfolgt, 800*800 Bildpunkte bei einer Farbtiefe von 24 Bit.

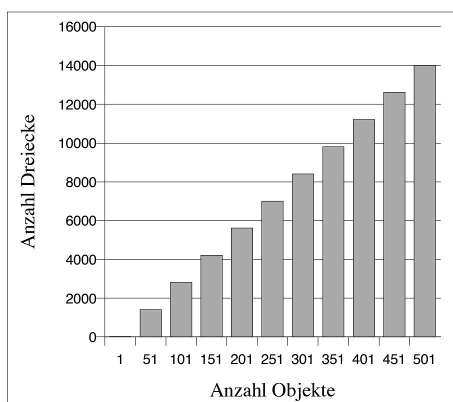

Abbildung 29: Zusammenhang: Objekte - Dreiecke

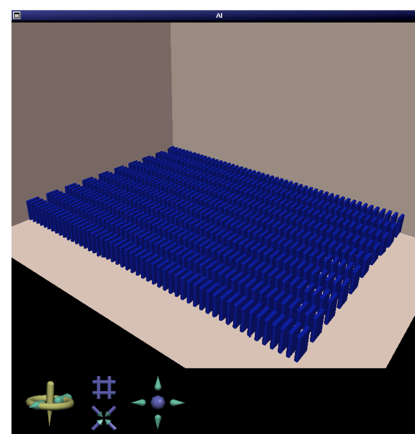

Abbildung 30: Testszene

Die Messungen für das Zeichnen der Szene werden auf allen drei Plattformen durchgeführt, einmal unter Sun Ultra-2 mit und ohne Hardware-Unterstützung und unter Linux, ohne Hardware-Unterstützung. Da sich die Hardware-Unterstützung nur auf das Zeichnen der Szene bezieht, werden für die Kollisionserkennung nur die Plattformen Sun Ultra-2 und Linux, Punkt 1. und 2., in Betracht gezogen. Gemessen wird jeweils die Zeit in ms für das Zeichnen der Szene bzw. die Zeit in ms für die Kollisionserkennung. Ein Vergleich der drei Plattformen bzgl. des Zeichnens der Szene ist in Abbildung 31 aufgezeigt.

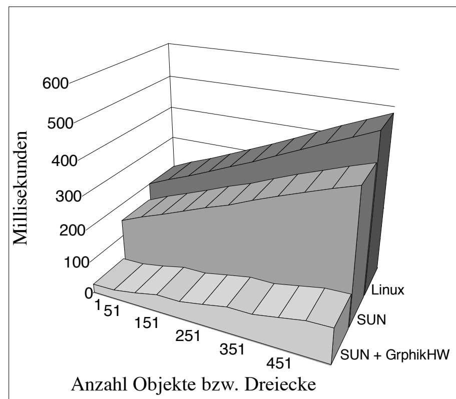

Abbildung 31: Zeitmessung für das Zeichnen der Testszene

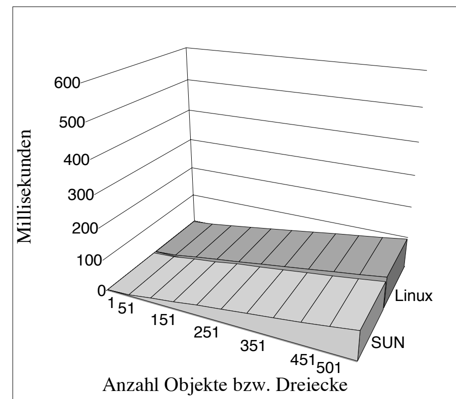

Abbildung 32: Zeitmessung für die Kollisionserkennung

Die gemessenen Zeiten für die Kollisionserkennung in Abhängigkeit von der Objekt- bzw. Dreieckszahl sind in Abbildung 32 dargestellt.

Die Zeit für die Bildschirmdarstellung ergibt sich somit als Summe aus der Zeit für das Zeichnen und aus der Zeit für die Berechnung der Kollisionserkennung. In den Abbildungen 33 und 34 sind Berechnungszeiten für das Zeichnen und für die Kollisionserkennung in ihrer Addition aufgezeigt. Abbildung 33 beschreibt die Zeit für die Bildschirmdarstellung unter der Plattform Linux und Abbildung 34 unter Sun Ultra-2 mit Hardware-Unterstützung.

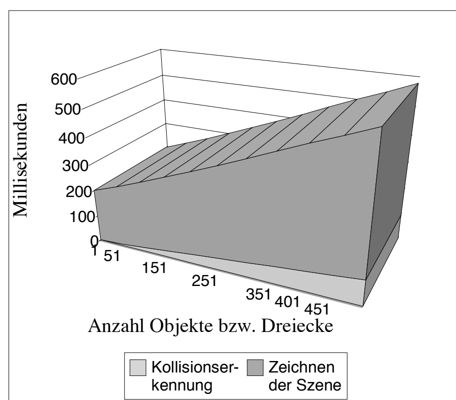

Abbildung 33: Berechnungszeiten für die Bildschirmdarstellung unter Linux

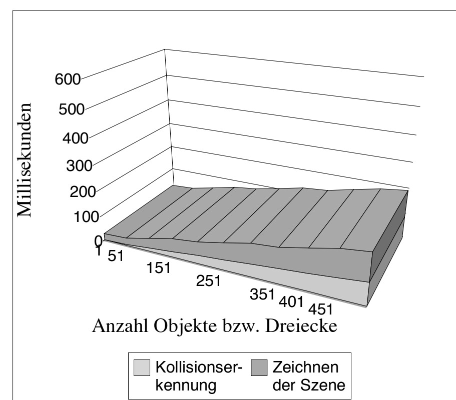

Abbildung 34: Berechnungszeiten für die Bildschirmdarstellung unter Sun Ultra-2 mit Hardware-Unterstützung

Für flüssiges Arbeiten sollten wenigstens 10 Bilder pro Sekunde dargestellt werden, d.h. für die Bildschirmdarstellung stehen maximal 100 ms zur Verfügung. Folglich ergibt sich aus den Meßwerten, daß für unser Gesamtsystem nur Plattform 3) Sun Ultra-2 mit Hardware-Unterstützung in Frage kommt. Das größte Problem bereitet aber immer noch das Zeichnen bzw. Berechnen der Szene, trotz Hardware-Unterstützung. Die Praxis zeigt aber, daß mit einer geringeren Auflösung, z.B. 500*500 Bildpunkten auch unter Linux, ohne Hardware-Unterstützung, noch akzeptabel gearbeitet werden kann.

6.1.3 Vergleich der Bearbeitungszeiten

In diesem Abschnitt werden die Bearbeitungszeiten für die Bildschirmdarstellung und für die Wissensbasisanfragen in Hinsicht auf ihr Vorkommen, bezogen auf eine benutzerinitiierte Aktion, verglichen.

Beim Aufbau der Szene aus den Daten der Wissensbasis treten Anfragen extrem gehäuft auf. Für das Arbeiten mit dem Gesamtsystem treten Anfragen nur noch gelegentlich auf, z.B. wenn der Benutzer Objekte verschiebt, löscht oder neu anlegt. Für die Beispielszene und die Testszene fallen die Anfragen, zeitlich gesehen, kaum ins Gewicht. Bei größeren Modellen ist es aber nicht abzusehen, wie stark und in welcher Form die Bearbeitungszeiten für die Anfragen steigen werden.

Die Bildschirmdarstellung hingegen wird ständig benötigt, z.B. für das Bewegen einzelner Objekte oder der gesamte Szene. Das Verhältnis zwischen Anfragen an die Wissensbasis und der erneuten Bildschirmdarstellung soll die Aktion *„Der Benutzer möchte Objekt A von Position x nach Position y bewegen“* zeigen. Folgende Schritte werden dabei durchlaufen:

1. Objekt A wird angewählt und die Anfrage *„welche Objekte befinden sich auf diesem?“* an die Wissensbasis gestellt. Das Objekt A wird nun mit kleinen roten Quadraten auf seiner Oberfläche gezeichnet.

2. Nun bewegt der Benutzer das Objekt A an Position y. Für jeden Pixel, den er mit der Maus zurücklegt, wird eine Kollisionserkennung und das erneute Zeichnen der Szene durchgeführt.

3. Objekt A wird deaktiviert. Die Kollisionserkennung wird nun verwendet, um herauszufinden, auf welchem Objekt B das Objekt A zu stehen kommt. Anschließend wird die Anfrage *„Objekt A steht auf Objekt B“* an die Wissensbasis geschickt, wobei hier davon ausgegangen wird, daß Objekt A auf Objekt B stehen darf. Objekt A hat nun seine neue Position y erreicht und die Szene wird erneut gezeichnet.

Wie man sieht, werden bei einer zulässigen Aktion dieser Art, genau zwei Anfragen an die Wissensbasis gestellt, aber die Bildschirmdarstellung wird unzählige Male ausgeführt.

Die Komplexität der Berechnung der Bildschirmdarstellung ist, im Gegensatz zur Wissensbasis, linear und damit handhabbar.

6.2 Perspektiven

Da in dieser Arbeit „nur“ ein exemplarisches Gesamtsystem implementiert wurde, bietet die Visualisierung von Wissensbasen noch viel Spielraum für Verbesserungen und Erweiterungen. In diesem Abschnitt wird die Funktionalität des Gesamtsystems, nicht nur aus Benutzersicht, betrachtet. Anhand der Funktionalität der Einzelkomponenten (WR-Komponente und Graphikkomponente) sollen Möglichkeiten zur Erweiterung und Verbesserung aufgezeigt werden. Anschließend werden noch einige denkbare Einsatzgebiete diskutiert.

6.2.1 Erweiterung des Modells

Im Augenblick bietet das Modell nur die Möglichkeit, Objekte zu stapeln. Dabei wird lediglich darauf geachtet, daß sich der Schwerpunkt des Objekts über dem anderen Objekt befindet. Dies wirft aber spätestens bei einem Stuhl, so wie er in Abbildung 2 zu sehen ist, Probleme auf. Er kann dann z.B. mit zwei Beinen auf einem anderen Objekt (z.B. auf dem Tisch) stehen, was im Sinne der Physik instabil ist. Der Stuhl müßte aus kleineren Komponenten modelliert werden, z.B. vier Beine, Sitzfläche und Rückenlehne. Für alle vier Beine überprüfte man nun, ob sie auf einem anderen Objekt stünden. Bei unterschiedlichen Höhen für die Objekte unter den Stuhlbeinen würde der Stuhl entsprechend gekippt werden usw. Dies führt dazu, daß entschieden werden muß wie detailliert die Gesetze der Physik modelliert werden sollen.

Ein anderes Problem der Stapelrelation läßt sich ebenso am Stuhl veranschaulichen. Die Aussage „der Stift liegt auf dem Stuhl“ ist zu ungenau, wie sich in Abschnitt 4.2, beim Aufbau der Szene aus den Daten der Wissensbasis, herausstellte. Die Lösung, die dort gezeigt wurde, ist zwar ausreichend, aber nicht universell. Denn jedes konvexe Objekt birgt dieses Problem, siehe Abbildung 35. Eine allgemeinere Lösung kann nur durch ein erweitertes, detaillierteres Modell gefunden werden. Objekte müssen aus kleineren, konvexen Objekten (siehe Abbildung 35) zusammengesetzt werden, d.h. das Modell sollte eine Gruppierung von Objekten zu einem neuen Objekt unterstützen. So könnte man z.B. den Stuhl durch seine Bestandteile, vier Beinen, Sitzfläche, Rückenlehne usw. in der Wissensbasis definieren. Damit ließe sich in der Graphikkomponente der Stuhl aus kleineren konvexen Objekten zusammensetzen und die Aussage „der Stift liegt auf dem Stuhl“ zu „der Stift liegt auf der Sitzfläche des Stuhles“ verfeinern.

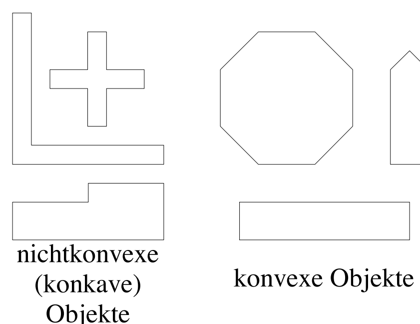

Abbildung 35: nicht-konvexe und konvexe Objekte

Desweiteren können Objekte, die z.B. an der Wand hängen oder Deckenbeleuchtung nicht dargestellt werden. Dies läßt sich über zusätzliche Relationen realisieren. Die Objekte benötigten folglich ein weiteres Attribut, das die Position des Objekts, relativ zum anderen Objekt widerspiegelt (z.B. Position des Bildes an der Wand relativ zur Wand).

Das aktuelle Modell beschränkt ein Objekt darauf, daß es nur auf genau einem anderen Objekt stehen darf. Ein Modell, das diese Einschränkung aufhebt, wäre wünschenswert, d.h. ein einzelnes Objekte dürfte auf mehreren Objekten liegen.

Die Funktionalität von Behältern könnte eine weitere Bereicherung des Modells darstellen. Objekte könnten z.B. *in* einen Schrank, Kiste usw. abgelegt werden.

Objekte können nicht frei gedreht werden, sondern nur die Ausrichtung kann variiert werden. Eine entsprechende Erweiterung wirft viele Fragen auf, z.B.:

- wenn auf dem gedrehten Objekt ein anderes Objekt liegt, fällt es dann herunter?
- kann auf dem gedrehten Objekt noch etwas abgelegt werden?
- liegt das gedrehte Objekt im Sinne unserer Wahrnehmung stabil, d.h. steht es nur auf einer Ecke und müßte eigentlich kippen?

Die Stapelrelation wird im Augenblick über Methoden realisiert. Für jede neue Objektklasse müssen die Methoden wieder geändert werden. Mit erweiterten Objekteigenschaften, könnte die Stapelrelation verallgemeinert werden und die Methoden fielen weg.

Die Möglichkeit, weitere Räume, ein Stockwerk, bis hin zum gesamten Gebäude, zu erstellen, sollte vom Modell noch umfaßt werden.

Wie man sieht, gibt es geradezu eine Flut von möglichen Erweiterung für das Modell. Die Schwierigkeit besteht darin, einen Kompromiß zu finden, so daß die reale Welt so gut wie möglich modelliert wird und zugleich das Modell nicht zu komplex und nur noch schwer berechenbar wird.

6.2.2 Erweiterung der Graphikkomponente

Die im vorigen Abschnitt genannten Erweiterungen des Modells ziehen eine entsprechende Angleichung der Graphikkomponente nach sich. Die Erweiterung um zusätzliche Räume usw. soll nun im Mittelpunkt der Diskussion stehen.

Eine wesentliche Schwäche der 3D Graphik liegt in der Beschränkung der Anzahl Dreiecke, die pro Sekunde dargestellt werden kann. Bezogen auf unsere Szene in Abbildung 2, die aus ca. 2600 Dreiecken besteht, hätte ein 7 stöckiges Gebäude mit ca. 20 Räumen pro Stockwerk 364000 Dreiecke. Wie das Meßergebnis in Abbildung 34 zeigt, liegt die Bildschirmdarstellung bereits bei 250 Objekten, ca. 7000 Dreiecke, bei 100 ms (ca. 10 Bilder pro Sekunde). Bei 364000 Dreiecken würde die Bildschirmdarstellung ungefähr 5200 ms benötigen, was für den Benutzer völlig unzumutbar ist. Deshalb muß dynamisch eine Dreiecksreduktion vorgenommen werden. Dies kann mit einem, von der Sicht des Betrachters abhängigen, Detailgrad geschehen, der nun erklärt wird.
Soll z.B. das gesamte Gebäude betrachtet werden, so werden nur die Außenwände der Räume gezeichnet, die das Gebäude beschreiben. Das gleiche gilt für die Betrachtung eines jeden Stockwerks, d.h. es wird nur das gewählte Stockwerk bzw. die Wände seiner Räume gezeichnet. Beim Detailgrad „Raumansicht“ werden die Objekte eines speziellen Raums gezeichnet. Die schematischen Zeichnungen in Abbildungen 36 bis 38 veranschaulichen die drei Abstufungen des Detailgrads.

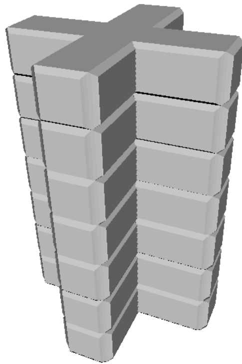

Abbildung 36: Detailgrad - Gebäudeansicht

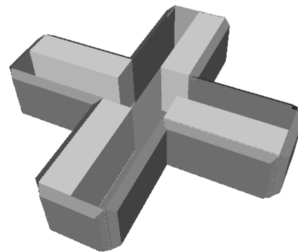

Abbildung 37: Detailgrad - Stockwerkansicht

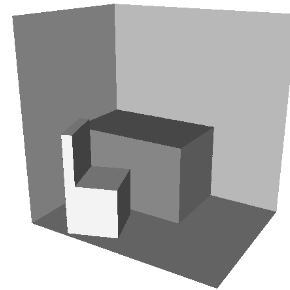

Abbildung 38: Detailgrad - Raumansicht

6.3 Einsatzgebiete

Wie schon zu Beginn erwähnt wurde, steht in dieser Arbeit die Service-Robotik im Vordergrund. Ein realer Roboter benötigt, um sich in seiner Umgebung zurechtzufinden, Wissen über diese Umgebung. Ein Wissensbasis ist daher erforderlich. Sie beinhaltet ein theoretisches Modell bzgl. der Eigenschaft der Objekte, die die Umgebung beschreiben. Die in dieser Arbeit vorgestellte Wissensbasis bietet zwar bereits einen hohen Grad der Abstraktion für die Bedienung der Wissensbasis, dies ist aber nicht intuitiv genug um schnell Wissen über die Umgebung modellieren zu können. Desweiteren verliert man bei einem größeren Modell der Umgebung, mehrere Räume usw., bald den Überblick. Genau hier soll mit dieser Arbeit Erleichterung geschaffen werden. Durch eine graphische Darstellung des Modells der Umgebung, mit der Möglichkeit der Manipulation, kann recht einfach die Umgebung für den Roboter erstellt, bearbeitet und gewartet werden. Dadurch, daß Graphikkomponente und WR-Komponente getrennt sind, kann unsere WR-Komponenten relativ einfach durch die WR-Komponente, auf die der Roboter zurückgreift, ausgetauscht werden.

Eine andere Möglichkeit wäre, das Gesamtsystem als Simulator für Roboter einzusetzen. Der Roboter ist dabei nur ein weiteres Objekt in der Wissensbasis, der sich in der Visualisierung widerspiegelt. Die Kollisionserkennung der Graphikkomponente, wie sie hier vorliegt, müßte durch eine Kollisionserkennung mit zusätzlicher Abstandsberechnung ausgetauscht werden. Abstandsberechnungen zwischen Roboter und allen weiteren Objekten, innerhalb eines definierten Radius um den Roboter, könnten als Sensordaten bzgl. der Umgebung fungieren. Der Roboter würde sich anhand dieser Daten durch das Modell seiner Umgebung bewegen. Dadurch könnten Wegeplanungsalgorithmen direkt am Simulator getestet werden. Dies hätte den Vorteil, daß man ständig sehen kann, wie sich der Roboter bewegt und wo er sich gerade befindet, ohne seinen Rechner verlassen zu müssen. Durch aktives Eingreifen bzw. dynamisches Ändern der Umgebung könnten auch Algorithmen für reaktives Planen getestet werden.

Ein völlig anderer, nicht wissenschaftlicher Aspekt ist im Bereich der Computerspiele angesiedelt. Immer mehr Spiele verwenden 3D Graphik, die es dem Spieler erlaubt sich

in einer virtuellen Welt zu bewegen und Aktionen ausführen. Dadurch, daß die Rechner immer schneller werden und 3D Graphik nahezu vollständig von der Hardware berechnet wird, bliebe genug Rechenzeit um eine Wissensbasis zu unterhalten, die die 3D Welt und die möglichen Aktionen enthält. Außerdem könnte das Verhalten der Computergegner (Wegeplanung usw.) entscheidend verbessert werden, da entsprechende Algorithmen ebenso von der Wissensbasis profitieren könnten.

Wie wir sehen, eignet sich die graphische Darstellung der Umgebung (3D-Welt), basierend auf Wissensbasen, für viele Gebiete.

7 Zusammenfassung

Diese Arbeit beschäftigte sich mit der Erstellung eines exemplarischen Gesamtsystems zur dreidimensionalen graphischen Darstellung und Manipulation von Wissensbasen.

Eine Motivation der Arbeit wurde zu Beginn von Kapitel 1 gegeben. Nach der Betrachtung der Anforderungen an das System, wurde das Konzept des Gesamtsystems vorgestellt. Der nächste Abschnitt zeigte zwei Arbeiten, denen ebenfalls die Visualisierung von Wissensbasen zugrunde liegt, die sich aber vom Ansatz her von dieser Arbeit signifikant unterscheiden. Der Überblick über die weiteren Kapitel dieser Arbeit bildete den Schluß des Kapitels.

Im zweiten Kapitel wurden drei Graphikwerkzeuge vorgestellt. Unter dem Aspekt der Anforderungen an die Graphikkomponente, erfolgte ein wertender Vergleich der drei Werkzeuge. Dabei zeigte sich, daß sich OpenGL am besten eignet. Da keines der Werkzeuge Kollisionserkennung beinhaltet, wurde das Software-Paket VCollide hinzugenommen und seine Arbeitsweise verdeutlicht.

Kapitel 3 widmete sich ganz der Wissensrepräsentation. Zuerst wurden die Ideen und die Einsatzgebiete der Wissensrepräsentation kurz beleuchtet. Nach einer grundlegenden Betrachtung der terminologischen Logiken und LOOM wurde, darauf aufbauend, ein Modell einer Büroumgebung erstellt.

Die Beschreibung der Netzwerkkomponente erfolgte in Kapitel 4. Dabei wurde auf einige Netzwerkgrundlagen eingegangen. Für die Kommunikation zwischen WR- und Graphikkomponente mußten zwei Fragen geklärt werden. Zum einen sollte ein adäquates Protokoll zum Datenaustausch und zum anderen Schnittstellen für beide Komponenten definiert werden. Basierend auf dem Ergebnis erfolgte die Implementierung der Netz-Module für die beiden Komponenten.

In Kapitel 5 fand die Beschreibung der Implementierung des Gesamtsystems statt. Dabei kamen die in den vorhergehenden Kapiteln erarbeiteten Teilbereiche zum Einsatz. Die Wissensrepräsentationskomponente wurde aus der Wissensbasis LOOM, aus Kapitel 3, dem Lisp-Netz-Modul, aus Kapitel 4, und einer, in Kapitel 5 erarbeiteten, Vermittlungsschicht zusammengesetzt. Aus dem C/C++-Netz-Modul, aus Kapitel 4, dem Graphikwerkzeug OpenGL, einigen weiteren Bibliotheken und einer Kontrollschicht wurde die Graphikkomponente geformt. Anschließend folgte die Betrachtung der wichtigsten Module. Mit einer Erläuterung der relevanten Funktionen, Klassen und Methoden konnte die Struktur der Graphikkomponente verdeutlicht werden.

Im abschließenden Kapitel 6 zeigten Messungen und Analysen, daß mit heutigen Rechnern, mit Hardware-Unterstützung für die Graphik, ein einzelnes Büro mit zugehörigen Objekten durchaus zügig dargestellt und manipuliert werden kann. Die Messungen verdeutlichten auch, daß bei komplexeren Umgebungen, z.B. ein Stockwerk, die Bildschirmdarstellung linear mit der Anzahl Dreiecke steigt, aber handhabbar bleibt. Bei der Wissensbasis ist es nicht abzusehen, wie sich die Laufzeit bei vielen Objekten verhält, da dies von der jeweiligen Implementierung der Wissensbasis abhängt. Im zweiten Teil des Kapitels, Perspektiven, war die Erweiterbarkeit der Graphikkomponenten, in

Hinsicht auf ein detaillierteres, umfassenderes Modell der Büroumgebung, zum Mittelpunkt der Diskussion geworden. Für das Modell ergab sich eine Fülle möglicher Erweiterung, von einem detaillierteren Modell, bis zur erweiterten Funktionalität der modellierten Objekte. Im letzten Teil des Kapitels wurden mögliche Einsatzgebiete des Gesamtsystems diskutiert.

8 Anhang

8.1 VRML 2.0 Beispiel

VRML Skript, das vier Symbole und einen Würfel anlegt.

```
#VRML V2.0 utf8

# Standardviewpoint
Viewpoint
{
  description "Standard"
  position -5.0 0.0 50
}

DEF Info NavigationInfo
{
  type ["EXAMINE"]
  headlight FALSE
}

Background
{
  skyColor 0.0 0.0 0.0
}

PointLight
{
  location 8.0 10.0 34.0
}

DEF timeSens TimeSensor
{
  cycleInterval 20
  loop TRUE
}

DEF rot OrientationInterpolator
{
  key [0, 0.5, 1]
  keyValue [1 1 1 0, 0 1 1 3.141592654, 1 1 1 6.283185307]
}

DEF rot_ao OrientationInterpolator
{
  key [0, 0.5, 1]
  keyValue [0 1 0 0, 0 1 1 3.141592654, 0 1 0 6.283185307]
}

Transform
{
  children
  [
    DEF sens1 SphereSensor
    {
    }
    DEF box Transform
    {
      translation -15 -5 0
      children
      [
        Shape
        {
          appearance Appearance
          {
```

```
              material Material
              {
                diffuseColor 1.0 0.0 0.0
                diffuseColor 1.0 0.117647 0.392157
                specularColor 1.0 1.0 1.0
#                shininess 100.0
              }
            }
            geometry Box
            {
              size 5 5 5
            }
          }
        ]
      }
    ]
}


#AI
DEF ai Transform
{
  children
  [
    #a oben
    DEF a_oben Transform
    {
      translation 3 -3 0
      children
      [
        Shape
        {
          appearance Appearance
          {
            material DEF std_blue Material
            {
              diffuseColor 0.117647 0.392157 1.0
              specularColor 1.0 1.0 1.0
#              shininess 100.0
            }
          }
          geometry IndexedFaceSet
          {
            coord DEF ai_verts Coordinate
            {
              point
              [
                -2.500 5.000 0.700, -1.000 5.000 0.700, -1.000 1.000 0.700,
-4.500 1.000 0.700, -5.500 2.000 0.700,
                1.000 5.000 0.700, 4.000 5.000 0.700, 5.000 4.000 0.700, 5.000
1.000 0.700, 2.000 1.000 0.700, 1.000 2.000 0.700,
                1.000 -1.000 0.700, 5.000 -1.000 0.700, 5.000 -4.000 0.700,
4.000 -5.000 0.700, 1.000 -5.000 0.700,
                -8.500 -1.000 0.700, -1.000 -1.000 0.700, -1.000 -5.000 0.700,
-10.500 -5.000 0.700, -11.500 -4.000 0.700,

                -2.995 5.700 0.000, -0.300 5.700 0.000, -0.300 0.300 0.000,
-4.995 0.300 0.000, -6.695 2.000 0.000,
                0.300 5.700 0.000, 4.495 5.700 0.000, 5.700 4.495 0.000, 5.700
0.300 0.000, 1.505 0.300 0.000, 0.300 1.505 0.000,
                0.300 -0.300 0.000, 5.700 -0.300 0.000, 5.700 -4.495 0.000,
4.495 -5.700 0.000, 0.300 -5.700 0.000,
                -8.995 -0.300 0.000, -0.300 -0.300 0.000, -0.300 -5.700 0.000,
-10.995 -5.700 0.000, -12.695 -4.000 0.000
              ]
            }
            coordIndex
```

```
            [
              0 4 3 -1, 0 3 2 -1, 0 2 1 -1, 0 1 22 -1, 0 22 21 -1, 1 23 22 -1,
1 2 23 -1, 2 24 23 -1, 2 3 24 -1, 3 4 24 -1, 24 4 25 -1, 21 4 0 -1, 25 4 21
-1, 21 22 23 -1, 21 23 24 -1, 21 24 25 -1
            ]
          }
        }
      ]
    }

    #a unten
    Transform
    {
      children
      [
        Shape
        {
          appearance Appearance
          {
            material USE std_blue
          }
          geometry IndexedFaceSet
          {
            coord USE ai_verts
            coordIndex
            [
              6 5 10 -1, 6 10 9 -1, 6 9 7 -1, 7 9 8 -1, 27 26 5 -1, 5 6 27 -1,
28 27 6 -1, 6 7 28 -1, 29 28 7 -1, 7 8 29 -1, 30 29 8 -1, 8 9 30 -1, 31 30 9
-1, 9 10 31 -1, 26 31 10 -1, 10 5 26 -1, 26 27 31 -1, 31 27 30 -1, 30 27 28
-1, 30 28 29 -1
            ]
          }
        }
      ]
    }


    #i oben
    Transform
    {
      children
      [
        Shape
        {
          appearance Appearance
          {
            material USE std_blue
          }
          geometry IndexedFaceSet
          {
            coord USE ai_verts
            coordIndex
            [
              11 13 12 -1, 11 14 13 -1, 11 15 14 -1, 33 32 11 -1, 11 12 33 -1,
34 33 12 -1, 12 13 34 -1, 35 34 13 -1, 13 14 35 -1, 36 35 14 -1, 14 15 36 -1,
32 36 15 -1, 15 11 32 -1, 32 33 34 -1, 32 34 35 -1, 32 35 36 -1
            ]
          }
        }
      ]
    }

    #i unten
    Transform
    {
      children
      [
```

```
        Shape
        {
          appearance Appearance
          {
            material USE std_blue
          }
          geometry IndexedFaceSet
          {
            coord USE ai_verts
            coordIndex
            [
              16 18 17 -1, 16 19 18 -1, 16 20 19 -1, 38 37 16 -1, 16 17 38 -1,
39 38 17 -1, 17 18 39 -1, 40 39 18 -1, 18 19 40 -1, 41 40 19 -1, 19 20 41 -1,
37 41 20 -1, 20 16 37 -1, 37 38 39 -1, 37 39 40 -1, 37 40 41 -1
            ]
          }
        }
      ]
    }
  ]
}

ROUTE sens1.rotation_changed TO ai.set_rotation
ROUTE sens1.rotation_changed TO box.set_rotation
ROUTE timeSens.fraction_changed TO rot.set_fraction
ROUTE timeSens.fraction_changed TO rot_ao.set_fraction
#ROUTE rot.value_changed TO box.set_rotation
#ROUTE rot.value_changed TO ai.set_rotation
ROUTE rot_ao.value_changed TO a_oben.set_rotation
```

8.2 Open Inventor: Beispiel zu C++

Kegel, der mit der Maus gedreht werden kann.

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/manips/SoTrackballManip.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main (int, char **argv) {
  // Fenster öffnen
  Widget myWin = SoXt::init(argv[0]);
  if (myWin == NULL) exit (1);

  SoSeparator *root = new SoSeparator;
  root->ref();

  // Kamera, Lichtquelle und Manipulator anlegen
  SoPerspectiveCamera * myCam = new SoPerspectiveCamera;
  root->addChild (myCam);
  root->addChild (new SoDirectionalLight);
  root->addChild (new SoTrackballManip);

  // Oberflächeneigenschaften für den Kegel definieren
  SoMaterial *myMat = new SoMaterial;
  myMat->diffuseColor.setValue (1.0, 0.0, 0.0);
  root->addChild (myMat);
  // Kegel anlegen
  root->addChild (new SoCone);

  SoXtRenderArea *myArea = new SoXtRenderArea (myWin);
  myCam->viewAll (root, myArea->getViewportRegion());
```

```
    myArea->setSceneGraph (root);
    myArea->setTitle (“Trackball“);
    myArea->show();

    SoXt::show (myWin);
    SoXt::mainLoop ();
  }
```

8.3 Open Inventor: Beispiel zum Dateiformat

Datei, die eine Windmühle, deren Flügel sich drehen, beschreibt.

```
#Inventor V2.0 ascii  Separator {
 Separator {
 RotationXYZ {
 axis Z
 angle 0 =
 ElapsedTime { # Bewegung der Flügel definieren
 speed 0.4
 }
 . timeOut # Ausgabe an angle weiterleiten
 }
 Transform {
 translation 0 0 0.5
 }
 Separator { # Befestigung für die Flügel definieren
 Material {
 diffuseColor 0.05 0.05 0.05
 }
 Transform {
 rotation 1 0 0 1.5708
 scaleFactor 0.2 0.5 0.2
 }
 Cylinder {
 }
 }
 DEF Blade Separator { # Flügel definieren
 Transform { # Geometrie und Eigenschaften festlegen
 translation 0.45 2.9 0.2
 rotation 0 1 0 0.3
 }
 Separator {
 Transform {
 scaleFactor 0.6 2.5 0.02
 }
 Material {
 diffuseColor 0.5 0.3 0.1
 transparency 0.3
 }
 Cube {
 }
 }
 Separator { # Form des Flügels definieren
 # .....
 }
 }
 Separator { # zweiter Flügel
 RotationXYZ {
 axis Z
 angle 1.5708
 }
 USE Blade
 }
 # dritten und vierten Flügel definieren
 }
 Separator { # Windmühle, Gebäude definieren
```

```
      # ...
    }
  }
```

8.4 LOOM: Modell der Büroumgebung - TBox

Hier sind alle Konzepte, Relationen und Methoden des Modells der Büroumgebung:

```
; Toplevel Objekt
(defconcept all-objects)

(defconcept object :is-primitive
(and all-objects (exactly 1 name)
(exactly 1 rgba) ; define color
(exactly 1 position) ; define position in x, z-plane
(exactly 1 direction) ; define rotation about y-axis
(exactly 1 scale)) ; define scalefactor
)
(defrelation name :domain object :range string)
(defrelation rgba :domain object :range string)
(defrelation position :domain object :range string)
(defrelation direction :domain object :range string)
(defrelation scale :domain object :range string)

; abstrakte Klassen
(defconcept translate-obj :is-primitive object)
(defconcept rotate-obj :is-primitive object)
(defconcept move-obj :is-primitive and translate-obj rotate-obj))

; konkrete Klassen
; stapelbare Objekte (move-obj)
(defconcept Stift :is-primitive (and move-obj))
(defconcept Glas :is-primitive (and move-obj))
(defconcept Flasche :is-primitive (and move-obj))
(defconcept Tischlampe :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Tastatur :is-primitive (and move-obj))
(defconcept Monitor :is-primitive (and move-obj))
(defconcept Mouse :is-primitive (and move-obj))
(defconcept Buch_stehend :is-primitive (and move-obj))
(defconcept Regal :is-primitive (and move-obj))
(defconcept Venus :is-primitive (and move-obj))
(defconcept Sockel :is-primitive (and move-obj))
(defconcept Papierkorb :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Zimmerpflanze :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Buch_liegend :is-primitive (and move-obj))
(defconcept CD :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Diskette :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Papier_A4 :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Tisch :is-primitive (and move-obj))
(defconcept Stuhl :is-primitive (and move-obj))
(defconcept Rollcontainer :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Projektor :is-primitive (and move-obj)) ; kein 3D-Modell
(defconcept Schrank :is-primitive (and move-obj)) ; kein 3D-Modell
; feststehende Objekte (object)
(defconcept Waschbecken :is-primitive (and object)) ; kein 3D-Modell
(defconcept Boden :is-primitive (and object))
(defconcept Saeule :is-primitive (and object)) ; kein 3D-Modell
(defconcept Tuerstock :is-primitive (and object))
(defconcept Wand :is-primitive (and object))
; drehbare Objekte
(defconcept Tuer :is-primitive (and rotate-obj))

; Objekte stapeln
(defrelation A-on-B
  :domain move-obj
  :range object
```

```
  :characteristics (:single-valued :closed-world))
(defrelation A-on-B* :is
  (:satisfies (?x ?z)
    (:or (A-on-B ?x ?z)
         (:for-some ?y (:and (A-on-B ?x ?y) (A-on-B* ?y ?z))))))

; über diese Funktion werden Objekte gestapelt
(defaction pose (?a ?b))

; ******************************************************************
; nachfolgende Methoden zeigen (ein) Objekt(e), das (die) auf versch.
; anderen Objekten liegen darf
; ******************************************************************
; Stift, Glas, Flasche, Mouse, Buch_liegend, CD, Diskette, Papier_A4
(defmethod pose (?a ?b)
  :title "put a on b 1"
  :overrides "put a on b 1"
  :situation (and
    (or (Stift ?a) (Flasche ?a) (Glas ?a) (Mouse ?a) (Buch_liegend ?a)
      (CD ?a) (Diskette ?a) (Papier_A4 ?a)
      (Venus ?a))
    (or (Buch_liegend ?b) (CD ?b) (Diskette ?b) (Papier_A4 ?b)
      (Tisch ?b) (Stuhl ?b) (Rollcontainer ?b) (Projektor ?b) (Schrank ?b)
      (Waschbecken ?b) (Boden ?b)
      (Sockel ?b) (Regal ?b)))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))

; ******************************************************************
; Tischlampe, Tastatur, Monitor, Buch_stehend, Projektor, Zimmerpflanze
; ******************************************************************
(defmethod pose (?a ?b)
  :title "put a on b 2"
  :overrides "put a on b 2"
  :situation (and
    (or (Tischlampe ?a) (Tastatur ?a) (Buch_stehend ?a)
      (Monitor ?a) (Projektor ?a) (Zimmerpflanze ?a))
    (or (Tisch ?b) (Stuhl ?b) (Rollcontainer ?b) (Schrank ?b) (Boden ?b)
      (Sockel ?b) (Regal ?b)))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))

; ******************************************************************
; Papierkorb
; ******************************************************************
(defmethod pose (?a ?b)
  :title "put a on b 3"
  :overrides "put a on b 3"
  :situation (and
    (Papierkorb ?a)
    (Boden ?b))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))

; ******************************************************************
; Stuhl
; ******************************************************************
(defmethod pose (?a ?b)
  :title "put a on b 4"
  :overrides "put a on b 4"
  :situation (and
    (Stuhl ?a)
    (or (Tisch ?b) (Boden ?b)))
  :response (
```

```
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))

; *********************************************************************
; Tisch, Rollcontainer, Schrank, Waschbecken, Saeule, Tuerstock, Tuer
; *********************************************************************
(defmethod pose (?a ?b)
  :title "put a on b 5"
  :overrides "put a on b 5"
  :situation (and
    (or (Tisch ?a) (Rollcontainer ?a) (Schrank ?a) (Waschbecken ?a)
      (Saeule ?a) (Tuerstock ?a) (Tuer ?a)
      (Sockel ?a) (Regal ?a))
    (Boden ?b))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))

; *********************************************************************
; Regal - einzelne Komponente
; *********************************************************************
(defmethod pose (?a ?b)
  :title "put a on b 6"
  :overrides "put a on b 6"
  :situation (and
    (Regal ?a)
    (Regal ?b))
  :response (
    (tell (a-on-b ?a ?b))
    (format nil "put object ~S on object ~S" ?a ?b)))
```

8.5 Netzwerkkomponente: LISP-Netz-Modul

Das ist das Netzwerkmodul, in ANSI-C, für die WR-Komponente. Der Programmcode enthält kein Funktion namens *main*, da das Kompilat nicht als ausführbares Programm, sondern als Bibliothek vorliegt. Die Bibliothek wird folgendermaßen erstellt:

1. Erstellen der Objektdatei *mesg.o*:
   gcc –o mesg.o mesg.c
2. Erstellen der Bibliothek *mesg.so*:
   gcc -c -fPIC mesg.c
   ld -G -o mesg.so mesg.o
   für Sun-Solaris:
   ld -G -o mesg.so mesg.o –lnsl -lsocket

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdlib.h>

void destroysock (void);

char mesg[200000];
int sock;
```

```
int ret, fd = 0;

void createsock (char *hostname, short int hport) {
  int client_len;
  struct sockaddr_in server, client;
  struct hostent *hptr;

  if (fd != 0) destroysock ();

  client_len = sizeof (struct sockaddr_in);
  sock = socket (AF_INET, SOCK_STREAM, 0);
  hptr = gethostbyname (hostname);
  if (hptr == NULL) { printf("Cannot resolve host: %s\n", hostname); exit(1);
}
  server.sin_family = AF_INET;
  server.sin_port = htons(hport); // Portnumber: default 2722
  bcopy (hptr->h_addr, &server.sin_addr, hptr->h_length);
  ret = bind (sock, (struct sockaddr *) &server, sizeof (struct sockaddr_in));
  listen (sock, 1);
  fd = accept (sock, (struct sockaddr *) &client, &client_len);
  if (fd == -1) { printf ("Cannot accept client\n"); exit (1); }
}

void destroysock (void) {
  close (fd);
  ret = close(sock);
}

/* receive TCP packets */

char * getmesg (void) {
  memset(mesg, 0, 200000);
  while (recv (fd, mesg, 200000, 0) < 0) {
    if (errno !=EINTR) exit (4);
  }
  return (mesg);
}

/* send TCP packets */

void putmesg (mesg) char *mesg; {
  while (send (fd, mesg, strlen(mesg), 0) < 0) {
    if (errno != EINTR) exit (4);
  }
}
```

8.6 Netzwerkkomponente: C/C++-Netz-Modul

Die ist das Netzwerkmodul, in C++, für die Graphikkomponente.

```
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include "ai_glob.H"

#ifndef ai_net
#define ai_net

// class for network operations

class net {
private:
  char mesg[200000];
  int  sock;
```

```
public:
  int ret;

  net (char *, short unsigned int); // creates a socket
  virtual ~net (); // destroys socket
  void putmesg (char *); // send TCP packets
  char *getmesg (void); // receive TCP packets
};

extern net *netpoint; // pointer to network-object

#endif
```

9 Literaturverzeichnis

ACL 96 **Allegro Common LISP User Guide Version 4.3**, Franz Inc., 1996

BBH 90 F. Baader, H.-J. Bürckert, J. Heinsohn, B. Hollunder, J. Müller, B. Nebel, W. Nutt, H.-J. Profitlich, **Terminological Knowledge Representation: A Proposal for a Terminological Logic**, Deutsches Forschungszentrum für Künstliche Intelligenz, 1990

BBH 92 F. Baader, H.-J. Bürckert, B. Hollunder, A. Laux, W. Nutt, **Terminologische Logiken**, Künstliche Intelligenz, 3:23-33, 1992

BL 94 F. Baader, A. Laux, **Terminological Logics with Modal Operators**, Deutsches Forschungszentrum für Künstliche Intelligenz, 1994

BPS 94 A. Borgida, P.F. Patel-Schneider, **A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic**, Journal of Artificial Intelligence Research 1, S. 277-308, 1994

Bri 95 D. Brill, **Loom Reference Manual Version 2.0**, Information Sciences Institute, University of Southern California, 1995

Bro 94 C. Brown, **Programmieren verteilter UNIX-Anwendungen**, Prentice Hall, 1994

BS 85 R.J. Brachman, J.G. Schmolze, **An overview of the KL-ONE knowledge representation system**, Cognitive Science, S. 171-216, 1985

CLM 95 J. Cohen, M. Lin, D. Manocha, M. Ponamgi,
I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments, Department of Computer Science, Univerity of North Carolina, 1995

Fie 99 S. Fiedler, **Visualisierung von User Vicinities im WWW auf der Basis von VRML**, Diplomarbeit an der Universität Ulm, Fakultät für Informatik, Abteilung Verteilte Systeme, 1999

Fin 99 D. Finkenzeller, **Visualisierung und Manipulation von Wissensbasen: Benutzerdokumentation**, Universität Ulm, Fakultät für Informatik, Abteilung Künstliche Intelligenz, 1999

HKN 94 J. Heinson, D. Kudenko, B. Nebel, H.-J. Profitlich, **An empirical analysis of terminological representation systems**, Artificial Intelligence, 68(2):367-398, 1994

HLC 98 T. Hudson, M. Lin, J. Cohen, S. Gottschalk, D. Manocha,
V-COLLIDE: Accelerated Collision Detection for VRML, Department of Computer Science, Univerity of North Carolina, 1998
http://www.cs.unc.edu/~geom/collide.html

KGL 98 S. Krishnan, M. Gopi, M. Lin, D. Manocha, A. Pattekar **Rapid and Accurate Contact Determination between Spline Models using ShellTrees**, Department of Computer Science, Univerity of North Carolina, 1998

Kil 94 M. Kilgard, **OpenGL and X, Part 1: An Introduction**, The X Journal, SIGS Publications, 1994

Kil 96 M. Kilgard, **The OpenGL Utility Toolkit (GLUT) Programming Interface**, Silicon Graphics, Inc., 1996

Mac 91 R.M. MacGregor, **Using a Description Classifier to Enhace Deductive Inference**, in Proceedings Seventh IEEE Conference on AI Applications, S. 141-147, 1991

Neb 90 B. Nebel, **Reasoning and Revision in Hybrid Representation Systems**, Springer-Verlag, 1990

Neb 91 B. Nebel, **Terminological Cycles: Semantics and Computational Properties**, in [Sow 91], S. 331-361

OLN 93 B. Owsnicki-Klewe, K. v. Luck, B. Nebel, **Wissensrepräsentation und Logik**, in G. Görz *et al.*, **Einführung in die Künstliche Intelligenz**, Addison-Wesley, Bonn 1993

Rad 98 P. Rademacher, **A GLUT-Based User Interface Library (GLUI)**, 1998

SA 94 M. Segal, K. Akeley, **The Design of the OpenGL Graphics Interface**, Silicon Graphics Computer Systems, 1994

Sow 91 J.F. Sowa, **Principles of Semantic Networks**, Representation and Reasoning, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991

Ste 90 G. Steele, **Common Lisp²**, Digital Press, 1990

SW 98 C. Schlegel, R. Wörz, **Der Softwarerahmen SmartSoft zur Implementierung sensomotorischer Systeme**, FAW Ulm, 1998, http://www.uni-ulm.de/SMART/Projects/c3.html

Tan 95 A. Tanenbaum, **Verteilte Betriebssysteme**, Prentice Hall, 1995

VRL 97 **VRML 97 (VRML 2.0)**, ISO/IEC 14772-1:1997, http://www.vrml.org/Specifications

Web 98 M. Weber, **Verteilte System**, Spektrum Verlag, 1998

Wer 94 J. Wernecke, **The Inventor Mentor**, Addison-Wesley, 1994

WND 97 M. Woo, J. Neider, T. Davis, **OpenGL Programming Guide²**, 1996