- Diplomarbeit -

Qualitatives räumliches Schließen zur Objektverfolgung in Bildfolgen

Enrico Ehrich

1.Gutachter: Prof. Dr. J. Heinsohn
2.Gutachter: Dipl.-Inform. I. Boersch

Fachhochschule Brandenburg
Fachbereich Informatik & Medien
Studiengang Informatik
D-14470 Brandenburg(Havel)

20. November 2002

Kurzfassung

Zielstellung des Themas ist die Konzeption eines Systems zur Verfolgung von Objekten in Bildfolgen. Dieses System gibt bei Anfragen an die Wissensbasis Auskunft über die möglichen Positionen der Objekte, auch wenn diese zum aktuellen Zeitpunkt von anderen Objekten verdeckt sind. Systembedingt (z.B. Verdeckung) kann die Position eines Objektes mehrdeutig sein. Hierbei sind alle möglichen Positionen des Objektes unter bestimmten Randbedingungen zu verfolgen.

In der Arbeit sollen die Grundlagen für die Wissensrepräsentation räumlicher Relationen sowie des Inferenzprozesses untersucht, als auch eine Systemarchitektur für das Gesamtsystem entwickelt werden. Die Funktionalität wird über eine prototypische Implementierung der Architektur (ohne Bildaufnahme, -verarbeitung) nachgewiesen. Die Umsetzung soll besonderen Wert auf Modularisierung und Wiederverwendbarkeit legen.

Abstract

Objective of the topic is to plan a system for tracking objects in frame sequences. The system provides information about the possible locations of the objects by inquiries to the knowledgebase even if the objects are covered by other objects at this moment. Depending on the system (i.e. covering), the location of an object can be ambiguous. Therefore the possible locations of the object have to be followed under certain side conditions.

The thesis shall examine the basics for the representation of knowledge of spatial relations as well as the inference process. Furthermore a system architecture for the complete system shall be developed. The functionality will be proved by a prototype implementation of the architecture (excluding image capturing and image processing). The implementation shall put a special emphasis on modulation and reusing.

Inhaltsverzeichnis

Kurzfassung i
Abbildungsverzeichnis iv
Tabellenverzeichnis vi

1 Einleitung **1**
1.1 Motivation 1
1.2 Aufgabenstellung 2
1.3 Gliederung 3

2 Analyse der Aufgabenstellung **4**
2.1 Überblick 4
2.2 Problemklassen 6
2.2.1 Formen 6
2.2.2 Transformation 14
2.2.3 Deformation 15
2.2.4 Weitere Attributsänderungen 16
2.2.5 Zusammenfassung 16
2.3 Randbedingungen 18
2.4 Problemfälle 23

3 Theoretische Grundlagen **26**
3.1 Einführung 26
3.2 Allens Temporallogik 27
3.3 Räumliches Wissen 32
3.3.1 Relationen in Bezug auf die Überlappung 32
3.3.2 Orientierungsrelationen 37
3.4 Szenegraph 40
3.5 Auswahl 44

4 Konzeption **45**
4.1 Tiefenrelation 45
4.2 Wahl des Aufbaus der Objektdaten 48

4.3 Wissensbasis 54
4.3.1 Datenfluss 54
4.3.2 Aufbau 56
4.3.3 Ablauf 62
4.3.4 Vermuten der Objektbewegung 63
4.3.5 Zuweisung von Objekten 68
4.3.6 Bilden der Relationen 71
4.3.7 Anfragen 72
4.4 Zusammenfassung 73

5 Umsetzung **74**
5.1 Einführung 74
5.2 Prolog, XPCE 75
5.2.1 SWI-Prolog 75
5.2.2 XPCE 85
5.3 Implementierung 89
5.3.1 Systemarchitektur 90
5.3.2 Tools-Bibliothek 91
5.3.3 BV-Simulator 102
5.3.4 Wissensbasis 111
5.3.5 Anwendung 115
5.4 Zusammenfassung der Umsetzung 123

6 Fazit und Ausblick **124**
6.1 Zusammenfassung der Arbeit 124
6.2 Ungenauigkeiten und Fehler 125
6.3 Ausblick 126

Anhang **A**
Literaturverzeichnis . . . . . A
Verzeichnisstruktur . . . . . D

Abbildungsverzeichnis

2.1 Überblick über das Gesamtsystem 5
2.2 Überlauf von Kreisen 7
2.3 Aktionsradius relativ zur Zeit 8
2.4 Aktionspfad relativ zur Zeit 8
2.5 Bedingungen beim Überlaufen von Kreisen 9
2.6 Kombination von mehr als 3 Objekten 9
2.7 Bedingungen beim Überlaufen von Rechtecken 10
2.8 Verdeckte konvexe Objekte 11
2.9 Schnittline bei konvexen Objekten 11
2.10 3 konvexe Objekte, eines davon ist von Anfang an verdeckt 12
2.11 Verdeckte konkave Objekte 12
2.12 Mehrere Übergänge bei konkaven Objekten 13
2.13 Löcher 13
2.14 Deformationstoleranz 16
2.15 Unterteilung der Objektformen 17
2.16 Unterteilung der Objektformveränderung 17
2.17 Randbedingung: Maximalgeschwindigkeit 19
2.18 Randbedingung: Objekte dürfen die Szene nicht verlassen 20
2.19 Randbedingung: Auftauchen von Objekten 20
2.20 Randbedingung: verdeckte Objekte 21
2.21 Durchdringen eines Objektes in der z-Achse 21
2.22 Problemfall: Einfacher Objekt Überlauf 23
2.23 Pflicht der Einbeziehung 24
2.24 Problemfall: Objektgruppe 24
2.25 Problemfall: Wechseln der Ebene 25
2.26 Problemfall: Objektbewegung 25

3.1 Beispiel einer Nachbarschaftsbeziehung 30
3.2 Verschiedene Netze für Nachbarschaftsbeziehungen 31
3.3 Beispiel eines Constraint-Netzes 31
3.4 RCC5-Relationen 33

3.5 Begrenzungsabhängigkeit und Begrenzungsunabhängigkeit 34
3.6 RCC8-Relationen 35
3.7 Anordnungsabhängigkeit und Anordnungsunabhängigkeit 36
3.8 Unterschiede zwischen $\mathcal{ROA}$ und $\mathcal{CDA}$ 38
3.9 Verschiedene Einteilungen der Relationen ($\mathcal{CDA}$) 39
3.10 Das $\mathcal{ROA}$-Modell 39
3.11 2D-PIR Beispiel 41
3.12 Kodierung der Richtungen 41
3.13 Beispiel Szenegraph 1 42
3.14 Beispiel Szenegraph 2 43

4.1 Tiefenrelation 46
4.2 Kodierung der 8 Richtungen 51
4.3 Die verschiedenen Kommunikationswege zur Wissensbasis 55
4.4 Ablauf beim Einfügen eines neuen Frames in die Wissensbasis 55
4.5 ER-Modell der Datenstruktur 56
4.6 Locations 57
4.7 Beispielszene: interne Repräsentation der Locations 59
4.8 Beziehungen zwischen Locations und Objekte 60
4.9 Objekthierarchie der Beispielszene 61
4.10 Überblick über den Ablauf 62
4.11 Beispiel eines Verbindungsgraphen 64
4.12 Veranschaulichung des Pfades von x nach y 64
4.13 Übergangslinien als Gewichtung am Verbindungsgraph 65
4.14 Übergang bei Gruppierung von Objekten 66
4.15 Zusammenfassen der Objekte zu Gruppen und deren Übergänge . . . 66
4.16 Zusammenfassen der Objekte in Gruppen 67
4.17 Das Lochproblem bei Kreisgruppen 67
4.18 Zuweisung von Objekten 69
4.19 Algorithmus zur Zuweisung von Objekten 70
4.20 Problem bei der Bestimmung von RCC-Relationen 72

5.1 SWI-Prolog Windows-Version 76
5.2 Emacs-Version von XPCE 86
5.3 Das XPCE-Beispielprogramm 88
5.4 Systemarchitektur 91
5.5 Die Oberfläche des Simulators 103
5.6 Beispiel C++ Anwendung 118
5.7 Prolog Anwendung mit XPCE-Oberfläche 119
5.8 Die Einstellungsmöglichkeiten 122
5.9 Kommandos, die die Datenbank manipulieren. 122

Tabellenverzeichnis

2.1 Objekttransformation und Objektveränderung 15

3.1 Die 13 Basisrelationen von Allen 27
3.2 Allens Basisrelationen 28
3.3 Kompositionstabelle der Allenschen Basisrelationen 29
3.4 Die RCC5 Basisrelationen 32
3.5 RCC5 Hasse Diagramm 33
3.6 RCC5 Kompositionstabelle 34
3.7 Die RCC8-Basisrelationen 35
3.8 RCC8 Hasse Diagramm 36
3.9 Unterscheidung der verschiedenen RCC-Verfahren 36
3.10 $\mathcal{CDA}$ Kompositionstabelle 39

4.1 Tiefenrelationen 46
4.2 Kompositionstabelle der Basisrelationen 46
4.3 Alle 49 möglichen Kompositionen der Tiefenrelation 47
4.4 Vergleich der einzelnen Formate nach ausgewählten Kriterien 53

Kapitel 1

Einleitung

1.1 Motivation

Der Passant beobachtet den Spieler genau, der drei gleiche Hütchen auf dem Tisch hin und her schiebt. Er konzentriert sich auf das Hütchen, unter dem sich die Kugel befindet. Nachdem der Spieler die Hütchen mehrmals gewendet hat, hält er inne und wartet darauf, dass der Passant auf ein Hütchen tippt.

Mit ausreichender Konzentration kann der Passant das Hütchen eine Zeit lang verfolgen, unter dem sich die Kugel befindet. Die Konzentrationsfähigkeit lässt aber nach einer bestimmten Zeit stark nach. Auch wenn sich die Hütchen zu schnell bewegen oder gleichzeitig mehrere bewegt werden, kann er nicht mehr genau sagen, unter welchem Hütchen sich die Kugel befindet. Ein naiver Mensch wird auch nicht vermuten, dass sich die Kugel nicht mehr unter einem der Hütchen befindet, sondern z.B. in der Hand der Spielers, der versucht den Passanten zu betrügen. Der Mensch schließt also nicht unbedingt immer alle Möglichkeiten mit ein.

Verallgemeinert gesagt geht es darum, Objekte zu verfolgen, auch wenn diese verdeckt sind, und alle Möglichkeiten über deren Position in Erwägung zu ziehen.

Nun stellt sich die Frage: **Kann ein System Objekte automatisch und vielleicht sogar zuverlässiger als der Mensch verfolgen ?**

Die Vorteile liegen klar auf der Hand:

1. Eine Szene könnte auch ohne Anwesenheit eines Menschen automatisch verfolgt werden.

2. Man ist nicht von der Konzentrationsfähigkeit des Menschen abhängig.

3. „Überraschungen“ sind somit nicht möglich, da das System alle Möglichkeiten in Betracht zieht.

Ein automatisches System könnte die Szene beispielsweise mit einer Kamera in Echtzeit aufnehmen und analysieren. Es könnten aber auch Videodaten analysiert werden. Auch andere Quellen sind denkbar, wie z.B. eine sprachliche Beschreibung der Szene oder eine Simulation, in der die Szenebeschreibung erstellt wird.

Die Anwendungsgebiete sind vielfältig. Beispielsweise in der Robotik, wo ein Roboter versucht sich in einer natürlichen Umgebung zurechtzufinden [Sch93], in der Auswertung und Suche von Videodaten [NNS01] und Sequenzen von Satellitenbildern [PMT01], aber auch die Anwendung in der Industrie oder bei elektronischen Spielzeugen, bis hin zu speziellen Überwachungssystemen sind denkbar.

1.2 Aufgabenstellung

Zielstellung des Themas ist die Konzeption eines Systems zur Verfolgung von Objekten in Bildfolgen. Dieses System gibt bei Anfragen an die Wissensbasis Auskunft über die möglichen Positionen der Objekte, auch wenn diese zum aktuellen Zeitpunkt von anderen Objekten verdeckt sind. Systembedingt (z.B. Verdeckung) kann die Position eines Objektes mehrdeutig sein. Hierbei sind alle möglichen Positionen des Objektes unter bestimmten Randbedingungen zu verfolgen.

In der Arbeit sollen die Grundlagen für die Wissensrepräsentation räumlicher Relationen sowie des Inferenzprozesses untersucht, als auch eine Systemarchitektur für das Gesamtsystem entwickelt werden. Die Funktionalität wird über eine prototypische Implementierung der Architektur (ohne Bildaufnahme, -verarbeitung) nachgewiesen. Die Umsetzung soll besonderen Wert auf Modularisierung und Wiederverwendbarkeit legen.

Die Bildaufnahme und Bildanalyse (Bildverarbeitung) soll durch einen Simulator ersetzt werden. Dieser Simulator besitzt eine Benutzeroberfläche, mit der verschiedene Szenen modelliert werden können. Außerdem ist eine Beispielanwendung zu erstellen, die die Möglichkeiten des Systems nutzt. Desweiteren sind sinnvolle Randbedingungen zu wählen, um die Umsetzung eines ersten Prototypen zu ermöglichen.

Die Arbeit ist als Basis für weitere aufbauende Arbeiten zu verstehen, die das System Schritt für Schritt verbessern und erweitern sollen.

1.3 Gliederung

Im Anschluss werden erste Überlegungen zu dem Thema diskutiert, einige Problemfälle dargelegt und die Komplexität durch die Definition von Randbedingungen eingegrenzt.

In dem Kapitel zu den theoretische Grundlagen werden wissenschaftliche Arbeiten, die für das Thema von Interesse sind, zusammenfassend vorgestellt und die wichtigsten Konzepte erläutert. Auf Basis der Allen-Zeitrelationen werden die räumlichen Beziehungen zwischen Objekten abgeleitet.

Es folgt der konzeptionelle Teil, in dem die Aufgabe theoretisch ausgearbeitet wird, um einen Lösungsweg zu finden. Dabei wird zuerst eine neue Objektbeziehung eingeführt, die die Beziehung der Objekte zueinander in der Tiefe (auf der z-Achse) beschreibt. Es werden Schnittstellen diskutiert und die Funktionsweise und der Ablauf der Wissensbasis erläutert.

Das Kapitel Umsetzung beschäftigt sich mit der Implementierung des Prototypen. Zuerst werden die verwendeten Entwicklungsumgebungen mit deren Besonderheiten vorgestellt. Die Benutzung der verwendeten Bibliotheken wird anhand von Beispielen erklärt, sodass ein Nachvollziehen des Quellcodes möglich ist. Es werden alle Komponenten des Prototypen erläutert.

Die erreichten und nicht erreichten Ergebnisse werden im Anschluss zusammengefasst und einige Vorschläge und Ausblicke für die weitere Entwicklung gegeben.

Im Anhang befinden sich das Literaturverzeichnis sowie die Liste der Projektdateien als Überblick und zur Hilfestellung bei der Weiterentwicklung des Prototypen.

Auf der beigefügten CD befindet sich der Prototyp inklusive Quellcode und einige der verwendeten Literaturquellen als druckbare Dokumente.

Kapitel 2

Analyse der Aufgabenstellung

2.1 Überblick

Das System, das die Objekte automatisch verfolgen soll, besteht aus 3 Teilen:

1. Ein Modul, welches die Szenenbeschreibung liefert. Im Folgenden wird dieses immer als Bildverarbeitung bezeichnet. Dieses Modul kann entweder ein Echtzeitsystem sein, welches reale Bilder mit einer Kamera aufnimmt und diese auswertet. Anstatt der Kamera können aber auch gespeicherte Videodaten als Quelle dienen, die dann analysiert werden.

   Im Rahmen der Arbeit wurde ein Simulator entwickelt, der eine Kamera mit Bildverarbeitung simuliert und eine idealisierte Szenebeschreibung erstellt (siehe 5.3.3).

2. Eine Wissensbasis (WiBa), die die Szene auswertet, die die Daten in geeigneter Form speichert und Schlüsse daraus zieht, wo sich welche Objekte befinden könnten.

3. Eine Anwendung, die diese Daten nutzt, indem es Anfragen an die Wissensbasis stellt.

Die Beschreibung der Szene ist immer zweidimensional, d.h. es handelt sich um ein projiziertes Abbild der realen Szene. Ein Bild von einer Kamera beispielsweise ist ein solches Abbild und es wird davon ausgegangen, dass es sich um Bitmapdaten handelt, also einer Matrix mit Farbinformationen. Aus diesen Daten erstellt die Bildverarbeitung eine Beschreibung der Szene, indem es Objekte separiert, rekonstruiert und in Klassen einstuft (Objektklassifikation). Diese Daten werden an die Wissensbasis übermittelt, die daraus eine interne Repräsentation der Szene erstellt und Anfragen zur Verfügung stellt, um diese Daten auszuwerten.

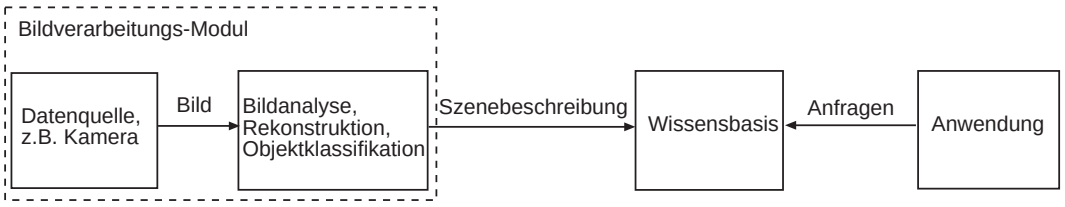


Abbildung 2.1: Überblick über das Gesamtsystem

Die Wissensbasis muss mindestens folgende Fragen beantworten können:

1. Welche Objekte sind bekannt?
2. An welchen Positionen bzw. unter welchen Objekten könnten sich diese befinden?

Allgemeine Probleme dabei sind:

1. Erkennen des Objektes. Handelt es sich um das bekannte Objekt oder ist es ein anderes oder gar ein neues Objekt.
2. Rekonstruktion der Objektform, speziell bei Überlappung.
3. Erkennung der Überlappung. Das bedeutet, man muss erkennen, ob sich ein Objekt gerade unter ein anderes schiebt oder ob sich die Objektform gerade ändert. Es sind also Probleme, bei dem Mehrdeutigkeiten vorkommen.
4. Bewerten, ob ein Objekt sich zu einer bestimmten Position bewegen kann. Darunter fällt auch, dass sich das Objekt unter ein anderes hinweg bewegen kann und dabei ggf. auch vollständig verdeckt wird (oder bleibt). Oder auch, ob sich das Objekt unter ein Objekt hindurch bewegen kann. Wieviel Zeit t es dafür minimal braucht, um zu einem Objekt zu gelangen.
5. Vermutung in der Voraussage der Objektbewegung, Formveränderung und weiterer Attributsänderung.

2.2 Problemklassen

Zur Analyse der Probleme, die bei der Beobachtung von mehreren Objekten auftreten, ist es angebracht, die einzelnen Szenen in Problemklassen zu unterteilen. Diese Problemklassen unterscheiden sich meist in der Form und dem Verhalten der Objekte.

2.2.1 Formen

Zuerst sollte man mit Objekten arbeiten, die eine statische Form besitzen. Genauer gesagt, die Objektform, die von der 3-dimensionalen Szene auf ein 2-dimensionales Bild projiziert wird, besitzt nur einen sehr kleinen Toleranzbereich, in dem es sich verändern darf. Dieser Toleranzbereich ist notwendig wegen der begrenzten Auflösung, die das Bild bzw. die Kamera hat, und diversen Störfaktoren, die Einfluss auf die Qualität des Bild haben.

Kreise

Da wir in der Projektion ($\Re^2$) arbeiten und es damit Punkte und Linien nicht gibt, ist die einfachste Objektform der Kreis. Dieser kann in Wirklichkeit also auch eine Kugel, Zylinder o.ä. sein. Er lässt sich einfach durch seine Position (x, y) und dessen Radius r beschreiben. Die Objekte können sich überlappen oder ganz verdecken.

Kreise haben keine Richtung, wodurch die Rotation der Objekte keine Auswirkung auf deren Form hat. Wir bekommen also von der Bildverarbeitung die Objektbeschreibung in Form der Position und dessen Radius. Durch die bekannte Form sind auch Überlappungen einfach erkennbar.

1 Kreis Eine Szene mit nur einem Kreis ist trivial. Eine Überlappung ist nicht möglich und die Position ist eindeutig bestimmbar, da der Kreis immer sichtbar ist.

2 Kreise Wenn es um die Positionsbestimmung geht, ist die Szene mit 2 Kreisen auch trivial. Das Objekt befindet sich entweder unter dem anderen Kreis oder ist sichtbar. Allerdings kann hier gut getestet werden, ob die Erkennung der Überlappung funktioniert.

Bei zwei gleichen Objekten kann allerdings eine Verwechselung stattfinden, wenn sie zwischen zwei Bildern die Positionen getauscht haben bzw. nicht eindeutig zuweisbar ist, welches Objekt welches ist.

3 Kreise Wir gehen davon aus, dass die Kreise die sichtbare Region nicht verlassen können, also sich immer innerhalb des Bildes befinden. Außerdem wird eine Maximalgeschwindigkeit v_{max} und eine maximale Beschleunigung a_{max} angenommen. Diese Werte sind den physikalischen Gegebenheiten angepasst und dürfen von den

Objekten nicht überschritten werden, weil es sonst zu falschen Ergebnissen kommen kann (Objekt taucht an einer Position auf, an der es nicht vermutet wird).

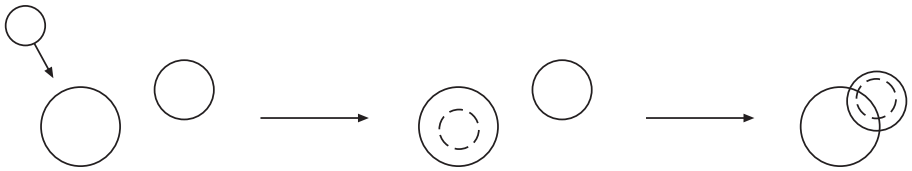


Abbildung 2.2: Überlauf von Kreisen: Das verdeckte Objekt kann auf das andere Objekt unbemerkt überlaufen.

Wenn z.B. nur zwei Objekte sichtbar sind, muss davon ausgegangen werden, dass sich das dritte Objekt unter einem der anderen befindet. Dies wird durch Beobachtung herausgefunden. Wenn allerdings das gesuchte Objekt größer ist, als eines der sichtbaren, kann natürlich schon ausgeschlossen werden, dass es sich nicht unter diesem befinden kann.

In Abbildung 2.2 sieht man gut, dass es notwendig ist, alle Möglichkeiten zu berücksichtigen und es nicht immer klar ist, unter welchem Objekt sich das gesuchte Objekt gerade befindet.

Durch die Definition von Randbedingung v_{max} und a_{max} (was im nächsten Abschnitt näher erläutert wird) kann man allerdings einige Varianten ausschließen. Demzufolge gibt es einen gewissen Aktionsradius (Abbildung 2.3), der im Laufe der Zeit, wo das Objekt nicht sichtbar ist, größer wird. Eigentlich ist es kein Aktionsradius, sondern eine Menge von Pfaden, deren Länge durch die Randbedingungen berechnet werden können (Abbildung 2.4).

Also ist eine unbemerkte Position (x, y) nur möglich, wenn der direkte Pfad dorthin erstens nicht länger als die berechnete Maximallänge, und zweitens dieser Pfad dorthin überhaupt existiert, auf dem das Objekt hätte (von der Kamera unbemerkt) dorthin gelangen können. Das Problem hierbei ist nur die Berechnung des kürzesten Pfades zwischen den sich überlappenden Objekten.

Ein Objekt kann auch nur dann zu einem anderen Objekt unbemerkt *überlaufen*, wenn dieses sich dicht genug an diesem befindet und die Schnittlinie größer als der Durchmesser des gesuchten Objektes ist (Abbildung 2.5). Eine Toleranz berechnet sich aus v und der Framerate *fps*, da die Kamera die Szene nicht kontinuierlich beobachten kann, sondern nur Einzelbilder in bestimmten Zeitabständen aufnimmt. Das System muss durch Algorithmen den kompletten Ablauf vervollständigen.

Ein Objekt A kann sich unter Objekt B unbemerkt (verdeckt) bewegen, wenn:

1. dorthin ein Pfad existiert,
2. der Pfad nicht Länger ist als die maximal zulässige Strecke, die sich aus der vergangen Zeit Δt, der Maximalgeschwindigkeit v_{max} und der Maximalbeschleunigung a_{max} berechnet
3. und die Objektverbindungen (Übergänge) es zulassen, dass das Objekt sich dahin verdeckt bewegt haben kann, ohne sichtbar zu werden.

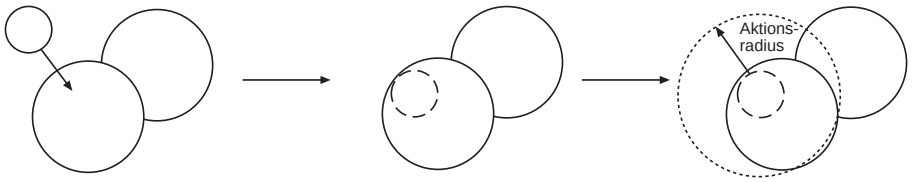


Abbildung 2.3: Mit der Zeit steigt auch der mögliche Aktionsradius des verdeckten Objektes

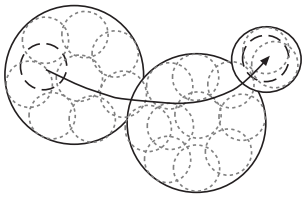


Abbildung 2.4: Richtige Darstellung des Aktionsspielraumes des Objektes mittels Pfade: Der kürzeste mögliche Pfad muss ermittelt werden aus der Menge der möglichen Pfade. Hier angedeutet durch die möglichen Positionen des Kreises.

3 Kreise, einer davon von Anfang an verdeckt Unbekannte Objekte werden hiermit eingeführt. D.h. das System muss automatisch erkennen, dass es sich bei dem unerwartet auftauchenden Objekt um ein unbekanntes Objekt handelt und es in die Wissensbasis dynamisch aufnehmen.

viele Kreise, zufällig verteilt Mit vielen zufällig verteilten Objekten kann man das System gut auf Fehler testen. Auch die vorläufige Ausführungsgeschwindigkeit der Wissensbasis kann damit gut beurteilt werden. Man sieht aber auch, dass bei

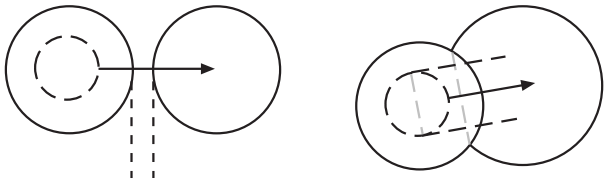


Abbildung 2.5: Bedingungen beim Überlaufen von Kreisen. links: Toleranzbereich; rechts: Vergleich der Schnittlinie mit dem Radius

mehr als 3 Objekten andere Kombinationsmöglichkeiten entstehen (Abbildung 2.6), das die Bestimmung des „Kann Objekt *a* sich unbemerkt unter Objekt *b* bewegen ?“ komplizierter macht.

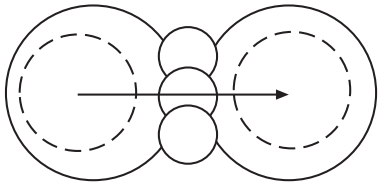


Abbildung 2.6: Neue Kombinationen bei mehr als 3 Objekten. Hier kann sich das Objekt trotzdem unbemerkt dorthin bewegen, da aus der Kombination von Überlappungen der anderen Objekte ein *Übergang* entstanden ist.

Hiermit wird eine einfache Szene mit Kreisen beschrieben. Wie diese Szene intern dargestellt wird, ist weitaus komplizierter. Gleiches gilt für die Umsetzung z.B. der geometrischen und logischen Operatoren. Als erste Versuchsszene eignet sie sich jedoch sehr gut.

Rechtecke

Rechtecke sind eine spezielle Form der konvexen Objekte und stellen einen Zwischenschritt zwischen Kreisen und allgemeinen konvexen Objekten dar. Ein Vorteil gegenüber allgemeinen konvexen Objekten ist die bekannte und einfache Form und damit die einfacheren geometrischen Operationen, die durchzuführen sind. Die Randbedingungen sind immer noch die gleichen wie bei den Kreisen. Eine zusätzliche Schwierigkeit stellt hier die Erkennung der Überlappung dar.

3 Rechtecke Der mehr oder weniger einfachste Fall mit Rechtecken sind 3 Rechtecke. Diese verhalten sich ähnlich wie 3 Kreise.

Die Funktion zur Bestimmung, ob sich ein Objekt unter einem Objekt hinweg bewegen kann, ist sehr einfach. Man vergleicht einfach die Kantenlänge des verdeckten Objektes mit der Kantenlänge des Schnittrechtecks, welches sich bei der Überlappung ergibt (siehe Abbildung 2.7).

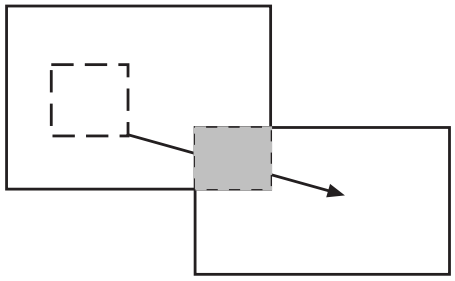


Abbildung 2.7: Bedingungen beim Überlaufen von Rechtecken. Das graue Rechteck muss mindestens in einer Kantenlänge so groß sein wie die entsprechende Kante des überlaufenden Rechtecks.

3 Rechtecke, eines davon von Anfang an verdeckt Diese Szene ist nicht so einfach wie die mit den Kreisen. Es ist nämlich noch herauszufinden, wie groß das unbekannte Rechteck ist. D.h. wenn das unbekannte Rechteck sichtbar wird, muss man feststellen, wann die Überlappung aufhört.

2 Rechtecke + 2 Kreise Die Schwierigkeit hierbei ist, die beiden Objektformen miteinander zu kombinieren.

Konvexe Objekte

Mit allgemeinen konvexen Objekten kann der Schritt von den von Anfang an bekannten Formen hin zu unbekannten Formen gewagt werden. Außerdem sind die Algorithmen komplizierter, z.B. zur Klärung der Frage, ob ein Objekt vollständig von einem darüber liegenden verdeckt werden kann (siehe Abbildung 2.8).

> Konvexe Objekte sind Objekte, bei denen die Kurve $y = f(x)$ des Umrisses konvex ist. Die Bestimmung erfolgt über das Vorzeichen der 2. Ableitung f''.
>
> $$f'' = \begin{cases} > 0 & \text{konkave Kurve (von oben)} \\ < 0 & \text{konvexe Kurve (von oben)} \end{cases} \tag{2.1}$$

Kreise und Rechtecke sollten nun auch als allgemeine konvexe Objekte eingestuft werden und kein Spezialfall mehr sein. Zur Vereinfachung der Berechnungen bzw. zur Steigerung der Berechnungsgeschwindigkeit kann diese Annahme ggf. weiterhin sehr hilfreich sein.

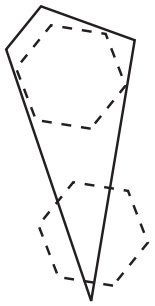


Abbildung 2.8: Bei konvexen Objekten kann es passieren, dass das Objekt nicht mehr an jeder Position vollständig von einem anderen Objekt verdeckt wird.

2 Kreise + 1 konvexes Objekt Die erste Schwierigkeit ist hierbei das Ermitteln der verdeckten Teile des konvexen Objektes. Die Szene *1 Kreis + 1 konvexes Objekt* wurde Übersprungen, kann aber trotzdem getestet werden.

1 Kreis + 2 konvexe Objekte Hier müssen jetzt die beiden konvexen Objekte gegeneinander getestet werden. Dies erfordert sicherlich neue Algorithmen (Abbildung 2.9). Mit dem *1 Kreis* soll hier nur angedeutet werden, dass dieser jetzt auch als allgemeines konvexes Objekt behandelt werden muss.

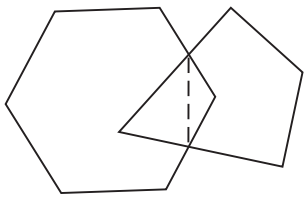


Abbildung 2.9: Die Schnittlinie ist bei allgemeinen konvexen Objekten nicht mehr einfach zu bestimmen.

3 konvexe Objekte, eines davon ist von Anfang an verdeckt Die Schwierigkeit besteht hier darin, dass die Form des verdeckten Objektes dynamisch angepasst werden muss, da dieses nicht von Anfang an bekannt ist (Abbildung 2.10).

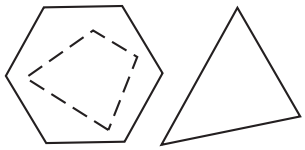


Abbildung 2.10: 3 konvexe Objekte, eines davon ist von Anfang an verdeckt

Die Kombinationen von noch mehr konvexen Objekten, mit den verschiedensten extremen Formen sollten noch getestet werden.

Konkave Objekte

Konkave Objektformen sind alle Objektformen, die einen geschlossenen Umriss haben und keine Löcher besitzen. Dies beinhaltet auch die bisher angesprochenen konvexen Objekte. Bei konkaven Formen ist es schwieriger, Überlappungen zu erkennen. Somit wird die Berechnung, ob das Objekt von einem anderen verdeckt werden kann, drastisch erschwert. Eine Lösung wäre z.B. die Unterteilung in *Primitive Unterobjekte* wie z.B. Dreiecke, Kreise oder Rechtecke (Abbildung 2.11).

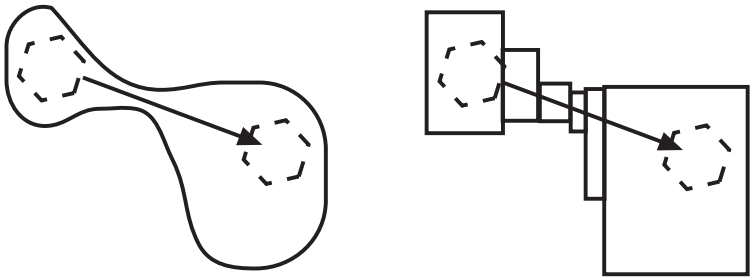


Abbildung 2.11: Bei konkaven Objekten wird die Bestimmung des möglichen Pfades eines Objektes schwieriger (links). Eine Lösung wäre die Unterteilung des konkaven Objektes in beispielsweise Rechtecke (rechts).

1 Kreis + 1 konkaves Objekt Damit werden erstmals konkave Objekte in die Szenenbeschreibung aufgenommen. Die besonderen Eigenschaften der konkaven Form müssen dabei berücksichtigt werden.

3 konkave Objekte Die konkaven Objekte können sich nun gegenseitig überlappen. Dabei können mehrere Schnittlinien auftreten, die es zu berücksichtigen gilt.

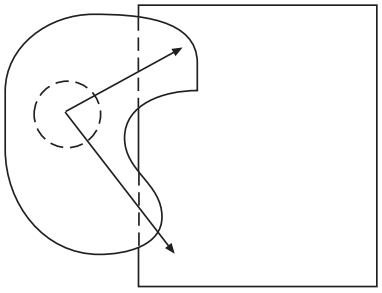


Abbildung 2.12: Es können mehrere Überlappungen entstehen und damit auch mehrere Übergänge zu einem Objekt.

3 konkave Objekte, eines davon von Anfang an verdeckt Das unbekannte Objekt muss dynamisch in die Wissensbasis aufgenommen werden. Dies sollte keine weitere Schwierigkeit sein und ähnlich funktionieren wie mit den konvexen Objekten.

Löcher

Löcher in Objekten jeder Art sind noch eine Steigerung der konkaven Objekte. Die Besonderheit besteht darin, das man durch die Löcher hindurchsehen kann. Das Objektformat muss evtl. daraufhin angeglichen werden, da es nicht nur einen Objektumriss gibt, sondern auch die Umrisse dieser Löcher.

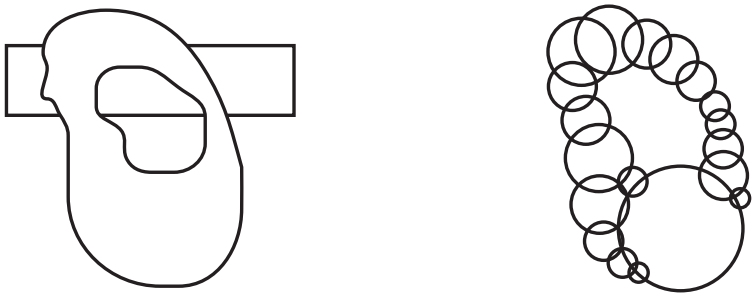


Abbildung 2.13: Objekte mit Löchern (links). Darstellung mit Primitiven, in diesem Fall (als anderes Beispiel) mit Kreisen (rechts).

2 Kreise + 1 Objekt mit einem Loch Die erste Szene, wo Objekte mit Löchern vorkommen, dient zum Testen, ob das evtl. neue Objektformat so gut funktioniert wie das bisherige. Die Berechnung, ob ein Objekt an Position (x,y) unter ein anderes passt, muss vermutlich auch angeglichen werden.

3 Objekte mit Löcher Die Objekte mit den Löchern können sich nun untereinander überlappen. Außerdem kann hier ein Objekt mehrere Löcher haben.

16 konkave Objekte mit und ohne Löcher, zufällig verteilt Die Szene ist gut geeignet zum Testen der Geschwindigkeit des Systems. Hier kann erkannt werden, ob auch konvexe Objekte als konkave Objekte betrachtet werden können.

Objekte aus mehreren Segmenten

Ein Objekt kann auch aus mehreren Teilen bestehen. Beispielsweise ein Objekt, welches an unterschiedlichen Stellen verschiedene Texturen aufweist. Die Bildverarbeitung würde jedes Teil als eigenständiges Objekt erkennen. Es ist zu bestimmen, ob es sich um ein einziges zusammenhängendes Objekt handelt.

2.2.2 Transformation

Unter Transformation eines Objektes ist die Neupositionierung des Objektes im Raum gemeint. Andere Arten der Transformation, wie Skalierung, fallen nicht in diese Kategorie, sondern werden als allgemeine Deformation betrachtet (siehe 2.2.3). Man kann die Transformation des Objektes im Raum in 3 Schwierigkeitsstufen einordnen:

1. Verschiebung des Objektes in der Bildebene (x,y-Achse). Dies ist der Fall, der keine Formveränderung in der Projektion auslöst, und damit der einfachste Fall. Diesen Fall haben wir schon von Anfang an einbezogen und muss nicht mehr extra erklärt werden.

2. Verschiebung entlang der z-Achse (in der Tiefe) und Rotation des Objektes um die z-Achse. Hierbei wird die projizierte Objektform skaliert bzw. sie rotiert. Die relative Kantenreihenfolge der Objektform bleibt gleich. Dadurch ist ein Formenvergleich in der Bewegung möglich.

3. Rotation des Objektes um die x,y-Achse. Dies ist der schwierigste Fall. Die Objektform ändert sich u.U. sehr stark und ist nicht einfach voraussagbar.

Eine Kombination aus allen Transformationen muss berücksichtigt werden. Man kann diese letzte Stufe der Transformation für den Anfang auch als allgemeine Deformationen betrachten, jedoch mit großem Toleranzbereich. Allerdings sind mit einer Unterscheidung von Transformation und Deformation genauere Aussagen möglich. Beispielsweise kann damit erkannt werden, ob sich ein Objekt gerade unter ein anderes schiebt oder sich deformiert. Objekte können damit besser unterschieden werden.

Objekttransformation in der Wirklichkeit	Objektveränderung im aufgenommenen Bild
t_x	t_x
t_y	t_y
t_z	$s_{x,y}$
r_x	d
r_y	d
r_z	r

Tabelle 2.1: Objekttransformation$\Longrightarrow$Objektveränderung.
t=Translation, r=Rotation, s=Skalierung, d=Deformation

Allgemeine-Beispiele Allgemein ist zu sagen, dass mit der einfachsten Transformation angefangen werden sollte, wie Stufe 1 und Stufe 2. Die Objektform ist davon unabhängig. Ob die schwierige Stufe 3 überhaupt implementiert werden sollte, ist fraglich, oder ob diese als allgemeine Deformation betrachtet werden sollte.

2.2.3 Deformation

Hiermit ist gemeint, dass sich die Objektform auch in der Wirklichkeit verändert und nicht nur im projizierten Bild. Es spielt aber eigentlich keine Rolle, ob sich das Objekt wirklich ändert. Wie schon im vorherigen Abschnitt erwähnt, kann auch die Transformation als Deformation betrachtet werden.

Die Deformation ist meist nicht genau voraussehbar, sondern pro Frame kann nur ein Toleranzbereich festgelegt werden, in dem sich das Objekt verändern darf. Dieser Toleranzbereich ist notwendig, um die Objekte zu identifizieren und das Verhalten bestimmen zu können. Wird der Toleranzbereich zu groß definiert, entstehen u.U. zu viele Kombinationsmöglichkeiten, wo sich ein verdecktes Objekt befinden kann.

Es sollten mindestens zwei Toleranzwerte vergeben werden. Einmal w_{abs}, welcher beschreibt, wie weit sich ein Punkt absolut von der zuerst bekannten Position entfernen darf und w_{rel}, welcher beschreibt, wie weit sich der Punkt pro Zeiteinheit, also Frame, bewegen darf (Abbildung 2.14). Dies ist vergleichbar mit den allgemeinen Randbedingungen der Geschwindigkeit v_{max} und Beschleunigung a_{max}. Noch besser wäre es, wenn man minimale und maximale Werte getrennt für x,y vergibt, oder minimale und maximale Objektformen angibt, die das Objekt nicht überschreiten darf.

Gummiball-Simulation Ein Kreis deformiert sich mit einer minimalen und maximalen Ausdehnung, die relativ gering ist.

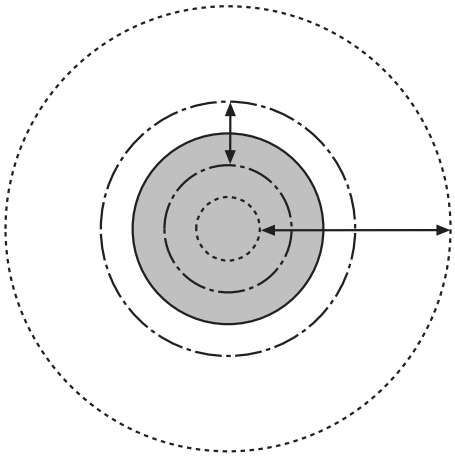


Abbildung 2.14: Toleranzbereiche bei der allgemeinen Deformation von Objekten. Der äußere und innere Kreis beschreibt die absolute maximale und minimale Objektausdehnung; die anderen beiden Kreise, die Objektveränderung, die pro Frame auftreten darf. Der graue Bereich ist das Objekt in seiner aktuellen Form.

Umfallen eines Bleistiftes Rotation eines Objektes um die x,y-Achse. Da der Bleistift relativ lang ist, ist auch eine starke Deformation in dem projizierten Bild zu erwarten.

2.2.4 Weitere Attributsänderungen

Als weiterführende Schwierigkeit können weitere Attributsänderungen berücksichtigt werden. Beispielsweise kann sich die Farbe und Textur der Objekte ändern, wie sie auch in der Wirklichkeit vorkommen; so bei der Änderung der Beleuchtungssituation (Licht und Schatten). Hauptsächlich wird damit die Qualität des Bildverarbeitungs-Moduls verbessert, hat aber so gut wie keinen Einfluss auf die Wissensbasis, nur dass die Attribute zur Laufzeit geändert werden müssen.

2.2.5 Zusammenfassung

Abbildung 2.15 zeigt noch einmal zusammenfassend die verschiedenen Objektformen, die es zu unterscheiden gilt. Abbildung 2.16 zeigt die unterschiedlichen Klassen der Formveränderungen.

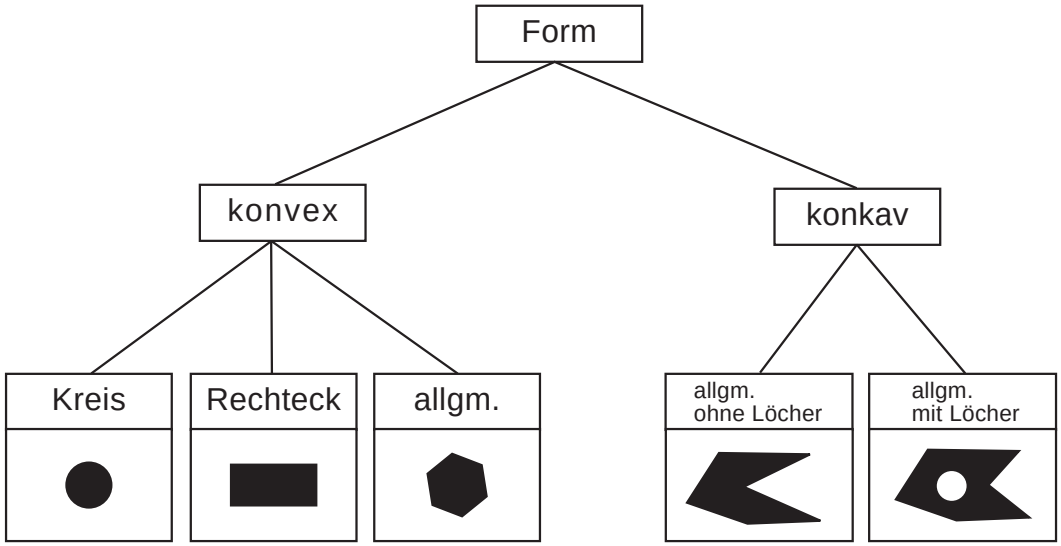


Abbildung 2.15: Unterteilung der Objektformen

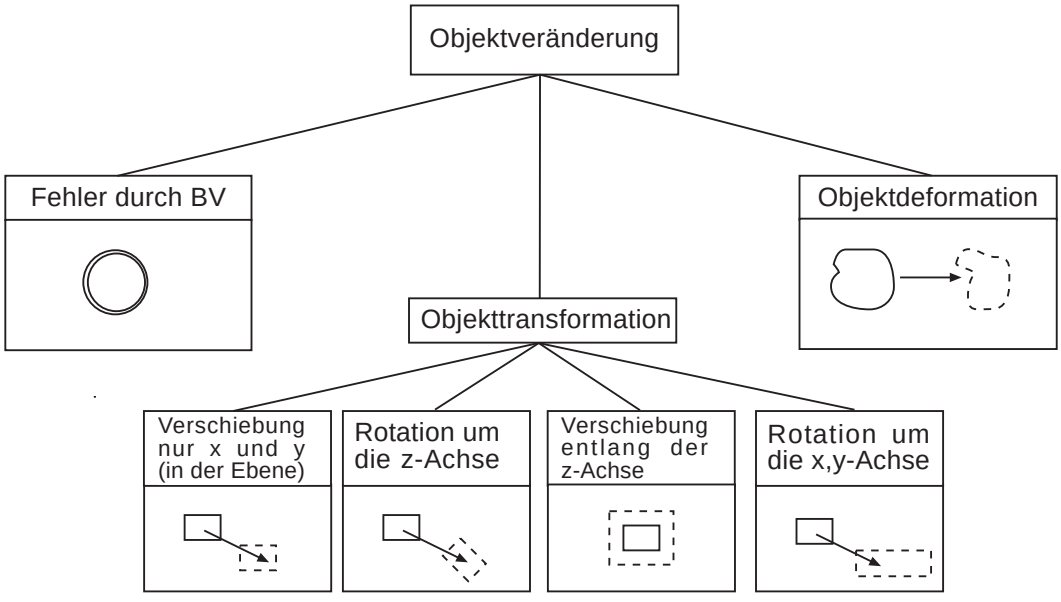


Abbildung 2.16: Unterteilung der Objektformveränderung

2.3 Randbedingungen

Man kann nicht alle Möglichkeiten, wie Objekte sich in einer Szene verhalten könnten, im System berücksichtigen. Wenn man das machen würde, wäre das System unnütz, da die Aussagen, die das System dann trifft, zu mehrdeutig sind. Außerdem ist es unsinnig, physikalisch unmögliche Verhalten mit zu berücksichtigen, sowie Fälle zu berücksichtigen, die sehr selten auftreten, aber die Mehrdeutigkeit stark beeinflussen (wie im Weiteren noch an Beispielen erläutert wird). Um Probleme auszulagern, werden oftmals Vereinfachungen vorgenommen.

All dies wird hier in den Randbedingungen zusammengefasst, die die Objekte der Szene nicht verletzten dürfen, da die Wissensbasis sonst von falschen Annahmen ausgeht und damit mit hoher Wahrscheinlichkeit auch falsche Aussagen trifft.

Es wird nun im Einzelnen erläutert, warum die entsprechende Randbedingung definiert wurde.

Wie aus den Problemklassen zu entnehmen ist, wird das Problem bei komplexeren Objektformen sehr schwer lösbar. Daher wurde im Rahmen dieser Arbeit nur die einfachste Objektform, der *Kreis*, berücksichtigt. Wie sich im weiteren herausstellt, ist das Problem für n Kreise schon relativ schwer zu lösen. In nachfolgenden Arbeiten kann das System auf andere Objektformen erweitert werden.

Eine weitere Vereinfachung ist die Voraussetzung, dass ein Objekt, insofern es teilweise sichtbar ist, die verdeckten Teile von der Bildverarbeitung wiederhergestellt werden (rekonstruiert). Also auch wenn ein Kreis nur teilweise sichtbar ist, erkennt die Bildverarbeitung die Eigenschaften zu dem Objekt wie Klasse, Radius, Position und Farbe. Völlig verdeckte Objekte, also nicht sichtbare Objekte, sind der Bildverarbeitung nicht bekannt und werden nicht an die Wissensbasis übermittelt.

Objekte dürfen in einer gewissen Zeit Δt nur einen gewissen Weg zurücklegen. Dies lässt sich über die Geschwindigkeit ausdrücken: $v = s/t$. Die physikalische Grenze ist natürlich die Lichtgeschwindigkeit c, mit der sich ein Körper (der eine Masse besitzt) bewegen kann. In realen Szenen sind die Geschwindigkeiten allerdings sehr viel geringer (Beispielsweise $0.5m/s$[1]). Daher wird für eine Szene eine Maximalgeschwindigkeit v_{max} definiert. Um Mehrdeutigkeiten bei der Identifizierung der Objekte einzuschränken, sollte man diese Maximalgeschwindigkeit so gering wie möglich halten. In Abbildung 2.17 ist dargestellt, dass Objekte von einem aufgenommen Bild zum nächsten nur eine gewisse Strecke zurücklegen dürfen.

[1]Da mit Bitmapbildern gearbeitet wird, kann auch in Pixel/s gerechnet werden. Bei der Umsetzung des Prototypen wurde allerdings mit physikalischen Einheiten gerechnet und ein einstellbarer m/Pixel Umrechnungsfaktor gewählt.

Anders ausgedrückt, Objekte bewegen sich kontinuierlich, wobei die Kamera nur in gewissen Zeitabständen den Status der realen Szene erfassen kann und somit die Wissensbasis Vermutungen über die Bewegungen der Objekte aufstellen muss. Damit ist sie auf die maximale Strecke angewiesen, die ein Objekt in der vergangenen Zeit zurückgelegt haben kann.

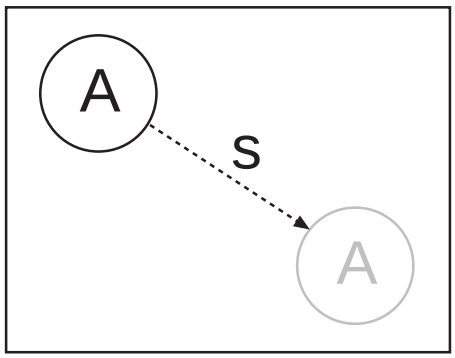


Abbildung 2.17: Randbedingung der Maximalgeschwindigkeit, Objekte dürfen nicht „springen“

Es kann als Erweiterung der Randbedingung der Maximalgeschwindigkeit gesehen werden, dass man zusätzlich eine Maximalbeschleunigung a_{max} definiert, um die Mehrdeutigkeiten weiter einzuschränken. Allerdings macht es nur in Szenen Sinn, wo Objekte stark von der physikalischen Trägheit beeinflusst werden. Bei einer großen Beschleunigung (Beispielsweise durch fremde Krafteinwirkungen) macht das Definieren dieser Randbedingung dagegen keinen Sinn.

Objekte dürfen den sichtbaren Bereich der Kamera nicht verlassen, da sonst die Bewegung von Objekten außerhalb des Erfassungsbereiches mit berücksichtigt werden müsste, und somit die Mehrdeutigkeit der Objektposition gefördert wird (Abbildung 2.18).

Da Objekte erst von der Wissensbasis identifiziert werden, ist somit die Verwechslungsgefahr sehr hoch, wenn mehrere Objekte der gleichen Klasse in einer Szene auftauchen. Ganz besonders dann, wenn zugelassen wird, dass Objekte plötzlich auftauchen oder verschwinden können (Abbildung 2.19). Die sicherste Methode wäre die, dass alle Objekte am Anfang der Szene sichtbar sind und nicht plötzlich verschwinden können. Bei einer solchen Randbedingung würde die Wissensbasis in jedem Fall korrekt arbeiten.

Da es aber flexibler ist, von Anfang an verdeckte Objekte zuzulassen, muss diese Randbedingung etwas gelockert werden. Das plötzliche Verschwinden von Objekten ist weiterhin nicht zulässig, da die Logik der Wissensbasis damit nicht in Einklang zu

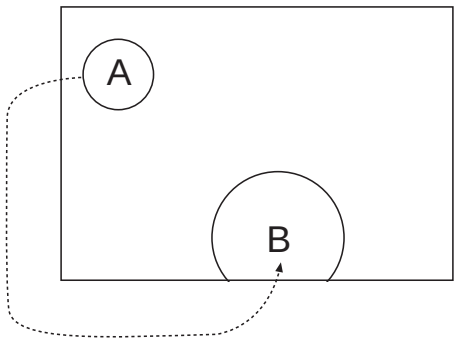


Abbildung 2.18: Objekte dürfen die Szene nicht verlassen.

bringen wäre. Das plötzliche Auftauchen von Objekten muss so geschehen, dass eine Verwechslung nicht möglich ist. D.h., Objekte die verwechselt werden können, dürfen am Anfang der Szene nicht verdeckt sein (Abbildung 2.20). Plötzlich auftauchende Objekte dürfen keine anderen Objekte vollständig verdecken (Abbildung 2.19).

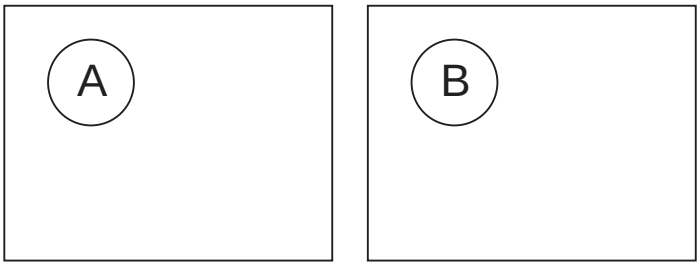


Abbildung 2.19: Das Auftauchen bzw. das Verschwinden eines Objektes, führt u.U. zu nicht korrekten Ergebnissen. In dieser Szene taucht das Objekt B genau über Objekt A auf und verdeckt dieses, beide Objekte haben die selbe Klasse. Die Wissensbasis nimmt jedoch an, es handelt sich immer noch um Objekt A, da es die beiden Objekte nicht voneinander unterscheiden kann.

Die letzte Randbedingung, die die Objekte in ihrem Verhalten zu erfüllen haben, ist die, dass Objekte (speziell in der z-Achse) nicht andere Objekte durchdringen können. Dies würde dann der Fall sein, wenn Objekte in der Projektion plötzlich vor ein anderes Objekt springen würde obwohl es vorher von diesem verdeckt war, oder umgekehrt (Abbildung 2.21). Dies ist in der realen Welt normalerweise auch nicht der Fall. Eine Ausnahme wäre, wenn das Objekt elastisch oder von nicht festen Aggregatzustand wäre.

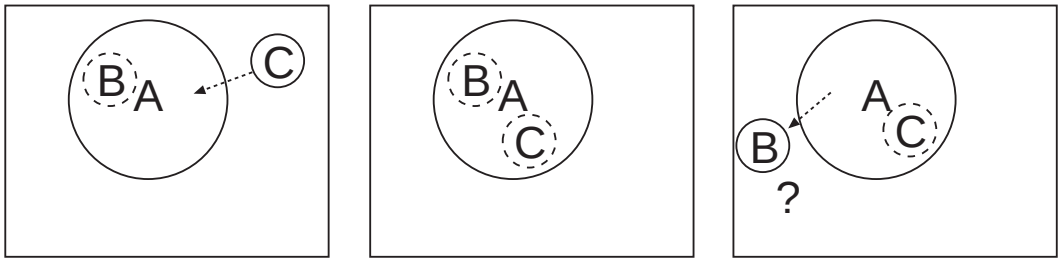


Abbildung 2.20: Objekt B ist von Anfang an von Objekt A verdeckt (a); Objekt C, welches die gleiche Klasse wie Objekt B hat, bewegt sich unter Objekt A (b); die Wissensbasis kennt Objekt B nicht und geht davon aus, dass es sich um Objekt C handelt, welches unter Objekt A wieder auftaucht (was aber eine falsche Annahme ist (c)).

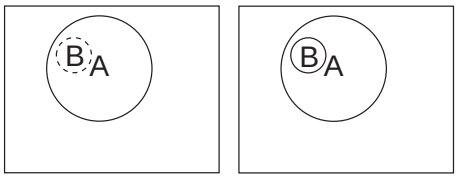


Abbildung 2.21: Das Objekt B durchdringt das Objekt A in der z-Achse. Das ist laut Randbedingung nicht zulässig.

Zusammenfassung der Randbedingungen

1. Es existiert nur der Objekttyp *Kreis*. Die Form ist statisch, d.h. es finden keine Deformationen statt. Die Transformation der Objekte ist nur in der Ebene (Projektion) erlaubt.

2. Teilweise verdeckte Objektteile werden von der Bildverarbeitung rekonstruiert.

3. Es muss eine *Maximalgeschwindigkeit* v_{max} vorgegeben werden, mit der sich ein Objekt in der Szene bewegen darf. Je geringer diese gewählt wird, um so eindeutiger sind die Ergebnisse der Wissensbasis.

4. Die Definition einer *Maximalbeschleunigung* a_{max} fördert die Eindeutigkeit. Die Maximalbeschleunigung ist allerdings nicht unbedingt notwendig, da es auch Szenen gibt, wo Objekte sehr stark durch Fremdeinwirkung beschleunigt oder abgebremst werden.

5. Objekte dürfen den sichtbaren Bereich der Szene nicht verlassen. Damit ist der rechteckige Erfassungsbereich der Kamera gemeint.

6. Objekte, die plötzlich auftauchen, dürfen keine anderen Objekte verdecken. Desweiteren müssen Objekte, die verwechselt werden können, am Anfang sichtbar sein, und Objekte dürfen nicht plötzlich verschwinden.

7. Objekte können andere Objekte nicht durchdringen. Das kann auch in der realen Welt bei festen Körpern nicht eintreten.

Es sollte bei den Versuchen oder in der praktischen Anwendung darauf geachtet werden, dass die Objekte der Szene diese definierten Randbedingungen nicht verletzten. Werden diese verletzt, wird die Wissensbasis mit großer Wahrscheinlichkeit falsche Annahmen über den Zustand der Welt vermuten, da die Verletzung der Randbedingungen nicht wahrgenommen werden kann.

2.4 Problemfälle

Es gibt einige spezielle Fälle, die zu beachten sind. Auch wenn manche Situationen nicht oft auftreten, sollten sie einbezogen werden.

Die Kamera, und damit die Wissensbasis, erfasst nur in gewissen Zeitabständen (Beispielsweise alle 50ms) den Zustand der Welt. Daher muss die Wissensbasis vermuten, was in der Zwischenzeit passiert sein könnte. Ein Beispiel hierfür ist in Abbildung 2.22 veranschaulicht. Im Bild (a) befindet sich Objekt A noch unter Objekt B, im Verlaufe der Zeit bewegt es sich allerdings unter Objekt C, ohne dass die Kamera dies registriert, da das Objekt A die ganze Zeit verdeckt ist.

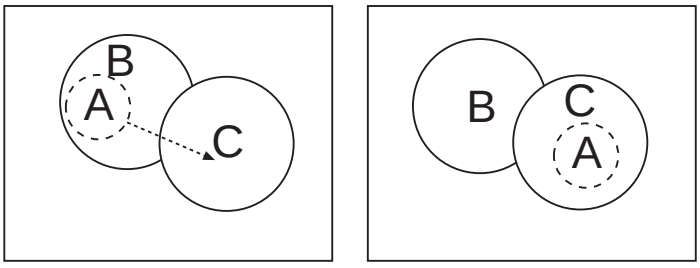
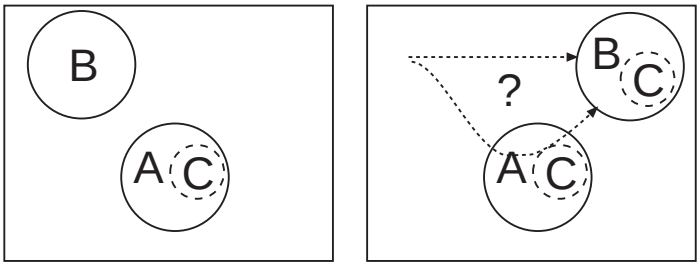


Abbildung 2.22: Objekt A wandert *verdeckt* von Objekt B (a) auf Objekt C über (b)

In der Szene in Abbildung 2.23 sieht man, dass das Objekt B nicht unbedingt den geradlinigen Weg genommen haben muss, sondern auch den Umweg über Objekt A genommen haben könnte, um Objekt C mitzunehmen; natürlich unter der Voraussetzung, dass die Zeit zwischen den beiden Bildern ausreichend war, damit das Objekt (nach den Randbedingungen) auch diesen Weg zurückgelegt haben kann.

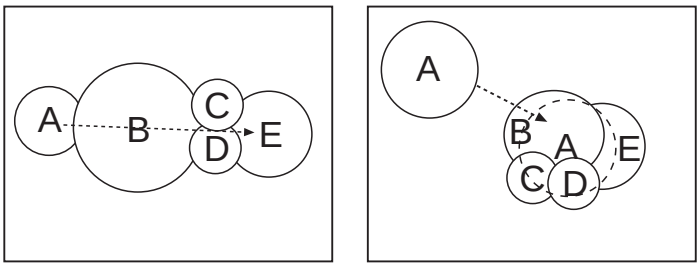
Die Abbildung 2.24 zeigt (wie bei den Problemklassen schon dargelegt), dass Objekte sich unter Objektgruppen bewegen können. Das Problem ist nicht mehr trivial und erfordert einen höheren Aufwand, um es zu lösen. Objekt A in Szene (a) kann sich unter allen Objekten zu Objekt E bewegen, ohne sichtbar zu werden. In Szene (b) ist die Situation ähnlich. Es wird nochmals verdeutlicht, dass Objekt A sich unter der Objektgruppe befinden kann und dabei verdeckt bleibt, obwohl jedes Einzelobjekt der Gruppe kleiner als das Objekt A ist.

Ein weiterer Problemfall ist, dass ein Objekt innerhalb von zwei Bildern auch die Ebene wechseln kann. In Abbildung 2.25 wird das veranschaulicht. Bild (a) ist die zuletzt von der Wissensbasis erfasste Situation. Bild (c) ist das nächste erfasste Bild.



a) b)

Abbildung 2.23: Es müssen alle Möglichkeiten mit einbezogen werden.



a) b)

Abbildung 2.24: Bewegen des Objektes A unter einer Objektgruppe hindurch (a), Objekt A kann sich unter der Objektgruppe befinden, obwohl die Einzelobjekte kleiner sind als das Objekt A (b)

Bild (b) wurde ausgelassen, da das Echtzeitsystem es nicht erfassen konnte. Da die Wissensbasis nur Bild (a) und (c) erfasst hat, muss es davon ausgehen, das der Zwischenschritt b stattgefunden hat, wobei ein Ebenenwechsel legitim ist und damit die Randbedingungen nicht verletzt wurden.

Wenn sich ein Objekt A unter Objekt B befindet (Abbildung 2.26), kann es von diesem mitgenommen werden, und zusätzlich kann es auch die Position unter dem Objekt B verändern. Also beide Objekte bewegen sich eigenständig und alle möglichen Positionen von Objekt A unter Objekt B müssen angenommen werden.

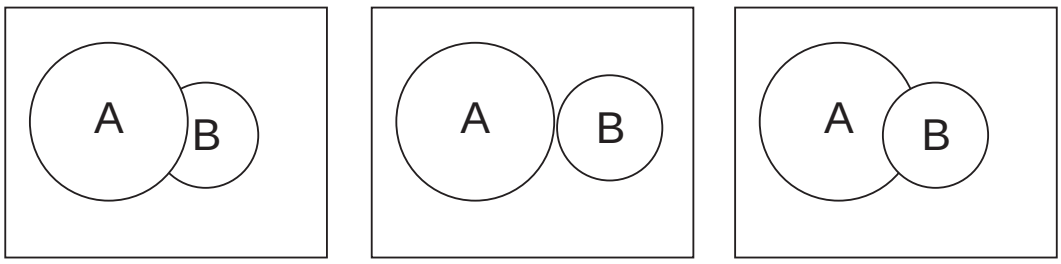


Abbildung 2.25: Wechseln der Ebene

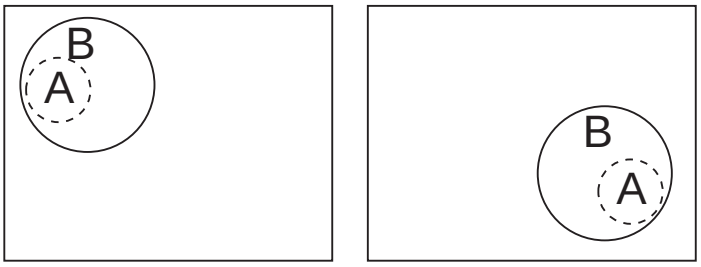


Abbildung 2.26: Das verdeckte Objekt A bewegt sich von Bild (a) zu Bild (b) mit dem Objekt B mit und verändert auch unter dem Objekt B seine Position

Kapitel 3

Theoretische Grundlagen

3.1 Einführung

Der Mensch denkt und kommuniziert oft nicht in absoluten Zahlen, sondern bewegt sich auf einer hohen Abstraktionsebene, um schnell und einfach zu einer Aussage zu kommen. Er stellt sich auch mit relativen Aussagen zufrieden. Beispielsweise: „der Wald ist grün“ oder „Frank sitzt links neben Lisa im Auto“.

Genau hier setzt nun die KI-Forschung an, um hohe Berechnungszeiten zu umgehen, mit abstrakten Daten arbeiten zu können oder eine natürliche Kommunikation zwischen Mensch und Maschine zu ermöglichen. Seit längerem beschäftigt sich die KI-Forschung mit der Entwicklung von Modellen, um diese Daten formal zu repräsentieren und Schlußfolgerungen daraus zu ziehen. Die meisten Modelle wurden dabei in Hinblick auf Zeit und Raum erstellt. Die Teilbereiche der KI werden demzufolge als *Qualitative Temporal Reasoning* und *Qualitative Spatial Reasoning* (QSR) bezeichnet.

In diesem Kapitel werden einige Modelle vorgestellt, die für die Arbeit von Interesse sind. Zuerst werden die Allen-Zeitrelationen vorgestellt und aus diesen werden die verschiedensten räumlichen Relationen abgeleitet. Damit werden Beziehungen zwischen Objekten beschrieben und Schlüsse daraus gezogen. Zum Schluss wird noch eine möglicherweise geeignete Datenstruktur vorgestellt, der Szenegraph, und diskutiert, welche Modelle für dieses System am geeignetsten erscheinen.

Es gibt sicherlich auch andere Methoden und Erweiterungen, mit denen graphische Daten ausgewertet werden können. Allerdings wird auf diese hier nicht näher eingegangen, da sie für dieses Projekt nicht als geeignet erscheinen. Als Beispiel sei [PE97] erwähnt, wo sich mit hierarchischen Methoden auseinandergesetzt wird.

3.2 Allens Temporallogik

Wir teilen die Zeitachse auf verschiedene Art und Weise ein, z.B. in Minuten, Stunden, Tage, usw. Wenn wir über die Relationen zwischen Ereignissen und Vorgängen sprechen, machen wir das oft durch die Positionierung dieser auf der Zeitachse. Beispiel: „während der Studienzeit“, „vor der Studienzeit“.

Viele Theorien in Bezug auf Qualitative (Temporal) Reasoning beziehen sich auf J. F. Allen, der sein Modell zur Darstellung und Schlussfolgerung von (Zeit-)Intervallen vorgestellt hat [All83]. Allen hat eine Menge aus 13 Basisrelationen definiert, mit der man alle Zustände beschreiben kann, die zwischen zwei Intervallen auftreten können: *before, meets, overlaps, starts, during, finishes, equal, is-started-by, contains, is-finished-by, is-overlapped-by, is-met-by, after*. In Tabelle 3.1 sind die Basisrelationen definiert und veranschaulicht [Bre].

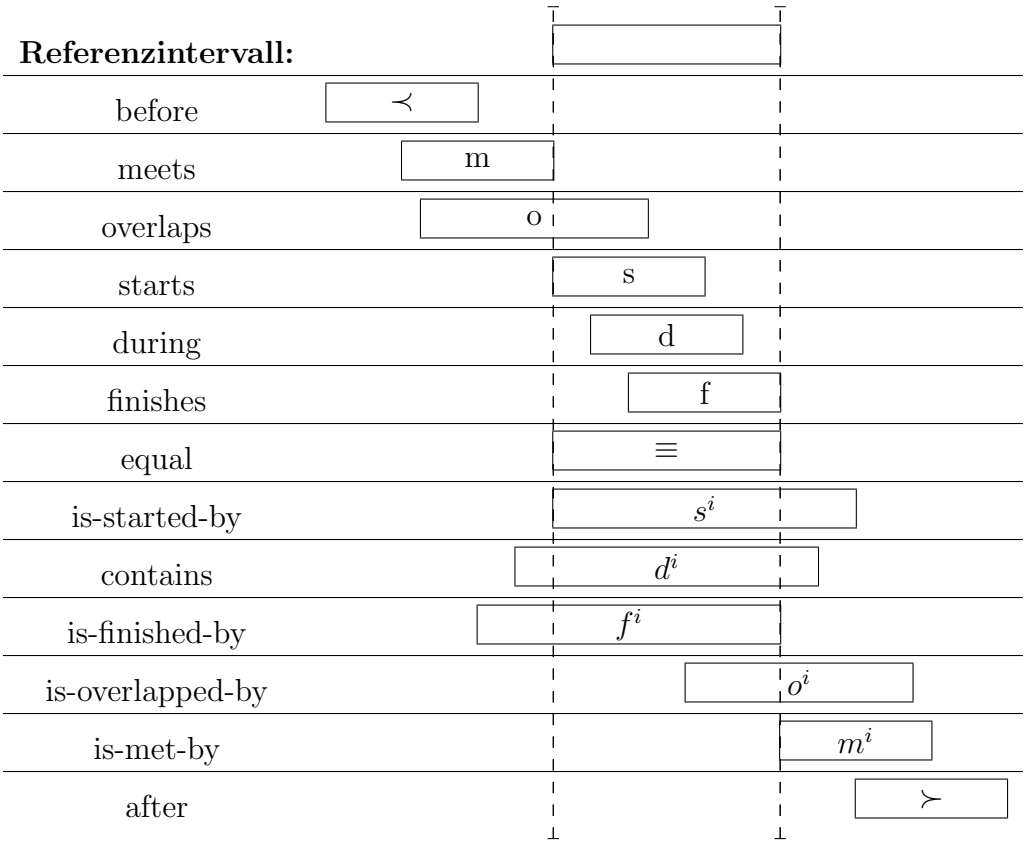


Tabelle 3.1: Die 13 Basisrelationen von Allen mit deren Symbole

Sei X und Y ein Intervall und R die Relation zwischen X und Y, so schreibt man

XRY[1]. Wobei R eine Teilmenge aus der Menge der Basisrelationen β ist. $R \subseteq \beta, \beta : \{\prec, \succ, o, o^i, s, s^i, f, f^i, d, d^i, m, m^i, \equiv\}$. Als Beispiel: $X\{\prec, s\}Y$.

Basisrelation	Symbol	Inverse	Endpunktrelation
X before Y	$\prec$	$\succ$	$X^- < X^+ < Y^- < Y^+$
X meets Y	m	m^i	$X^- < X^+ = Y^- < Y^+$
X overlaps Y	o	o^i	$X^- < Y^- < X^+ < Y^+$
X starts Y	s	s^i	$X^- = Y^- < X^+ < Y^+$
X during Y	d	d^i	$Y^- < X^- < X^+ < Y^+$
X finishes Y	f	f^i	$Y^- < X^- < X^+ = Y^+$
X equal Y	$\equiv$	$\equiv$	$Y^- = X^- < X^+ = Y^+$

Tabelle 3.2: Allens Basisrelationen

Die Inverse Relation R^i wird durch Invertieren der einzelnen Basisrelationen von R gebildet (siehe Tabelle 3.2): $R^i = \{B^i | B \in R\}$.

Komposition von Relationen Das Interessante daran ist, dass unbekannte Relationen durch Komposition der bekannte Relationen gebildet werden können. Beispielsweise, wenn $X\{\succ\}Y$ und $Y\{\succ\}Z$ bekannt sind, kann daraus geschlossen werden, dass auch $X\{\succ\}Z$ zutreffen muss. Allgemein ist zu schreiben $XRY \wedge YSZ \rightarrow XTZ, T = R \circ S$. Das Symbol $\circ$ steht dabei für die Komposition der Relationen. Die Komposition zweier Basisrelationen $B_1 \circ B_2$ kann mit einer Kompositionstabelle ermittelt werden (Tabelle 3.3).

$$X(B_1 \circ B_2)Y \; gdw. \exists Z(XB_1Z \wedge ZB_2Y)$$

Die Komposition zweier Relationen $R \circ S$, die eine Menge von Basisrelationen sind ($R \subseteq \beta, S \subseteq \beta$), erfolgt durch die Vereinigung der Komponentweisen Komposition der Basisrelationen:

$$R \circ S = \bigcup \{B \circ B' | B \in R, B' \in S\} \tag{3.1}$$

Außerdem können noch folgende Aussagen getroffen werden in Bezug auf die Inverse, des Schnittes und der Komposition von Relationen: [NB95]

$$\begin{array}{lccc} \forall X, Y: & XR^iY & \leftrightarrow & YRX \\ \forall X, Y: & X(R \cap S)Y & \leftrightarrow & XRY \wedge XSY \\ \forall X, Y: & X(R \circ S)Y & \leftrightarrow & \exists Z : (XRZ \wedge ZSY). \end{array}$$

[1]Im folgenden wird die Infixschreibweise für Intervallrelationen benutzt

$B_1 \circ B_2$	$\prec$	$\succ$	d	d^i	o	o^i	m	m^i	s	s^i	f	f^i	$\equiv$
$\prec$	$\prec$	β	$\prec o$ md s	$\prec$	$\prec$	$\prec o$ md s	$\prec$	$\prec o$ md s	$\prec$	$\prec$	$\prec o$ md s	$\prec$	$\prec$
$\succ$	β	$\succ$	$\succ o^i$ $m^i d$ f	$\succ$	$\succ o^i$ $m^i d$ f	$\succ$	$\succ o^i$ $m^i d$ f	$\succ$	$\succ o^i$ $m^i d$ f	$\succ$	$\succ$	$\succ$	$\succ$
d	$\prec$	$\succ$	d	β	$\prec o$ md s	$\succ o^i$ $m^i d$ f	$\prec$	$\succ$	d	$\succ o^i$ $m^i d$ f	d	$\prec o$ md s	d
d^i	$\prec o$ md^i f^i	$\succ o^i$ $m^i d^i$ s^i	$oo^i \equiv$ $dd^i f$ $ss^i f^i$	d^i	o d^i f^i	o^i d^i s^i	o d^i f^i	o^i d^i s^i	o d^i f^i	d^i	o^i d^i s^i	d^i	d^i
o	$\prec$	$\succ o^i$ $d^i m^i$ s^i	o d s	$\prec o$ md^i f^i	$\prec$ o m	$oo^i \equiv$ $dd^i f$ $ss^i f^i$	$\prec$	o^i d^i s^i	o	d^i f^i o	d s o	$\prec$ o m	o
oi	$\prec o$ md^i f^i	$\succ$	o^i d f	$\succ o^i$ $m^i d^i$ s^i	$oo^i \equiv$ $dd^i f$ $ss^i f^i$	$\succ$ o^i m^i	o d^i f^i	$\succ$	o^i d f	o^i $\succ$ m^i	o^i	o^i d^i s^i	oi
m	$\prec$	$\succ o^i$ $m^i d^i$ s^i	o d s	$\prec$	$\prec$	o d s	$\prec$	f f^i $\equiv$	m	m	d s o	$\prec$	m
m^i	$\prec o$ md^i f^i	$\succ$	o^i d f	$\succ$	o^i d f	$\succ$	s s^i $\equiv$	$\succ$	d f^i o^i	$\succ$	m^i	m^i	m^i
s	$\prec$	$\succ$	d	$\prec o$ md^i f^i	$\prec$ o m	o^i d f	$\prec$	m^i	s	s s^i $\equiv$	d	$\prec$ m o	s
s^i	$\prec o$ md^i f^i	$\succ$	o^i d f	d^i	o d^i f^i	o^i	o d^i f^i	m^i	s s^i $\equiv$	s^i	o^i	d^i	s^i
f	$\prec$	$\succ$	d	$\succ o^i$ $m^i d^i$ s^i	o d s	$\succ$ o^i m^i	m	$\succ$	d	$\succ$ o^i m^i	f	f f^i $\equiv$	f
f^i	$\prec$	$\succ o^i$ $m^i d^i$ s^i	o d s	d^i	o	o^i d^i s^i	m	s^i o^i d^i	o	d^i	f f^i $\equiv$	f^i	f^i
$\equiv$	$\prec$	$\succ$	d	d^i	o	o^i	m	m^i	s	s^i	f	f^i	$\equiv$

Tabelle 3.3: Kompositionstabelle der Allenschen Basisrelationen [HSA99]

Nachbarschaftsbeziehungen Die 13 Basisrelationen stehen in Nachbarschaftsbeziehungen (conceptual neighbors) zueinander. Ein Ereignis kann direkt in eines seiner Nachbarn umgewandelt werden, wenn eine Deformation (im topologischem Sinne) der Ereignisse stattfindet. Die *Verkürzung*, *Verlängerung* und *Verschiebung* von Intervallen sind solche Deformationen.

Beispielsweise sind die Basisrelationen *before* $\prec$ und *meets* m Nachbarn, die bei einer Veränderung ineinander umgewandelt werden können (Abbildung 3.1). Je nach Art und Weise der Deformation (bzw. Anwendung) gibt es verschiedene Nachbarschaftsnetzwerke, aus denen die Nachbarn einer Relation zu entnehmen sind (Abbildung 3.2). Die Interpretation der Nachbarn ist sehr liberal und im Prinzip sind alle 3 Netze zulässig.

Mit Hilfe dieser Nachbarschaftsbeziehungen können weitere Schlussfolgerungen aus den zeitlichen Relationen gezogen werden. Beispielsweise kann die Nachbarschaftskette $\{\prec, m, o, f^i, d^i\}$ als *older* bezeichnet werden. Diese neuen Relationen werden wie Basisrelationen behandelt und daraus können wiederum Kompositionstabellen aufstellt werden. Näheres kann in [Fre96] nachgelesen werden, wo intensiv auf sogenannte Semi-Intervalle eingegangen wird. Hier sollte nur der Begriff der Nachbarschaftsbeziehung eingeführt werden.

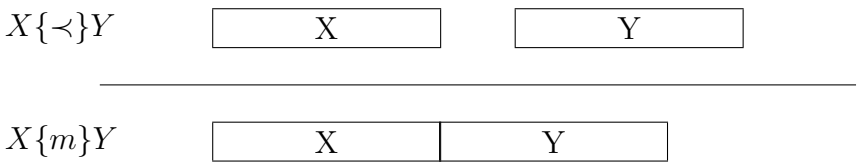


Abbildung 3.1: Beispiel einer Nachbarschaftsbeziehung

Constraint Netze Gegeben sei ein gerichteter Graph (Constaint Netz), dessen Knoten die Intervalle darstellen. An jeder Kante ist eine Gewichtung R angetragen, die eine Teilmenge der Basisrelationen β ist. Für unbekannte Relationen werden alle Basisrelationen angenommen (β).

Unter Verwendung dieses Graphen kann durch einen Constraint-Propagation-Algorithmus herausgefunden werden, ob der Graph konsistent ist. Dadurch werden auch die Mengen der Relationen minimiert.

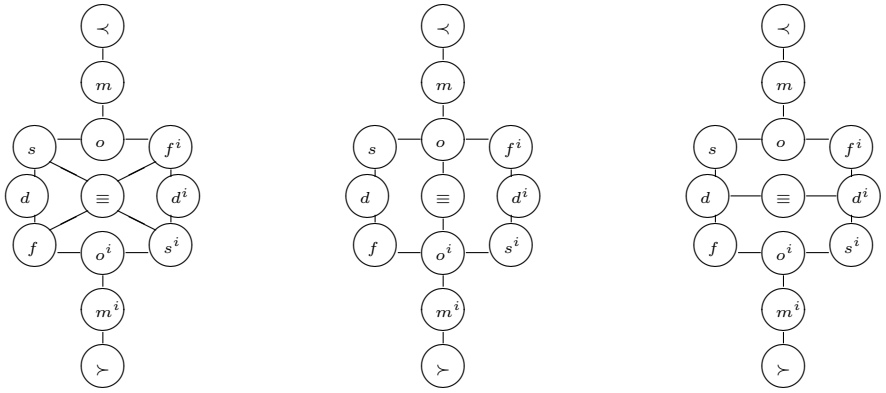


Abbildung 3.2: Verschiedene Netze für Nachbarschaftsbeziehungen (nach [Fre96])

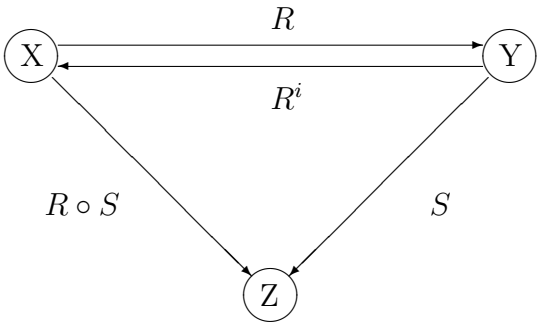


Abbildung 3.3: Beispiel eines Constraint-Netzes sowie die graphische Darstellung der Komposition und der inversen Relation

Definition 3.1: 3-Kanten-Konsistenz: Ein Graph ist 3-Kanten-konsistent, wenn jedes Knoten-Tripel aus dem Graphen konsistent ist. Ein Knoten-Tripel ist konsistent, wenn durch Berechnung der Komposition und dem Schnitt mit den vorgegebenen Kantenbewertungen nicht die leere Menge $\emptyset$ entsteht.

Pfadkonsitenz-Algorithmus Allen hat einen Pfadkonsistenz-Algorithmus entwickelt (3C-Algorithmus), welcher ein Backtracking-Verfahren ist. Wenn dabei irgendwo die leere Menge $\emptyset$ entsteht, bedeutet das, dass der Graph inkonsistent ist, also die Relationen zwischen den Intervallen falsch sind. Dieser Algorithmus ist allerdings nicht vollständig. Er stellt nur lokale Konsistenzen her (3-Kanten-Konsistenz) und „übersieht“ u.U. globale Inkonsistenzen. In der Praxis führt dies allerdings selten zu Problemen. Ein Beispiel für die Unvollständigkeit ist in [HSA99] beschrieben.

Der 3C-Algorithmus besitzt einen Aufwand von $O(n^3)$. Es gibt auch optimierte

Algorithmen wie der rekursive Pfad-Algorithmus IC'' oder diverse ACx Algorithmen (Variationen des Forward Checkings) [Tre97].

3.3 Räumliches Wissen

Abgeleitet von der Allenschen Intervalllogik wurde das Verfahren von Zeitintervallen auf den Raum übertragen und dabei verschiedene Mengen von Relationen definiert, die für die verschiedensten Aufgabenbereiche als geeignet erscheinen. Hier werden zwei Varianten vorgestellt, die sich auf die Überlappung zwischen zwei Regionen bzw. auf die Richtung von Punkten zueinander beziehen.

Es sei auch auf den konzeptionellen Teil (Abschnitt 4.1) verwiesen, wo beschrieben wird, wie eine Relationsmenge in Bezug auf die Anordnung der Objekte in der Tiefe erstellt wird.

3.3.1 Relationen in Bezug auf die Überlappung

Region Connection Calculus (RCC) ist ein topologisches Verfahren, um Relationen zwischen zwei Regionen in Bezug auf deren Überlappung zu beschreiben und daraus Schlussfolgerungen zu ziehen. Dabei wird die absolute Distanz vernachlässigt sowie wird die Orientierung der Regionen zueinander nicht berücksichtigt. Es ist also ein Modell, um zu beschreiben, ob und wie sich zwei Objekte überlappen (überschneiden). Hier werden zwei Modelle vorgestellt, die am häufigsten verwendet werden.

RCC5-Relationen

Das erste Verfahren, als RCC5 bezeichnet, kommt mit nur 5 Basisrelationen aus (daher die Zahl 5). Die Basisrelationen sind: *disjoined, partial-overlapped, proper-part, inverse-proper-part* und *equal* (siehe Tabelle 3.4). Abbildung 3.4 zeigt die topologischen Relationen zwischen zwei Regionen X und Y mit den jeweiligen RCC5-Relationen [Bit01].

Basisrelation	Symbol	Inverse
X *disjoint* Y	*dr*	*dr*
X *partial-overlapped* Y	*po*	*po*
X *proper-part* Y	*pp*	pp^i
X *equal* Y	*eq*	*eq*

Tabelle 3.4: Die RCC5 Basisrelationen

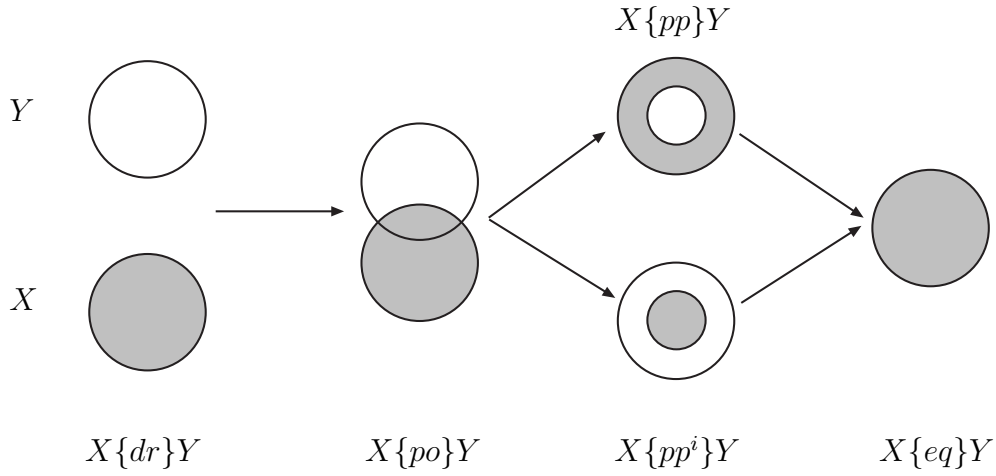


Abbildung 3.4: Definition der RCC5-Relationen.

Gegeben seien zwei Regionen X und Y. Jede RCC5-Relation kann mit folgendem Tripel ermittelt werden [Bit01]:

$(X \wedge Y \neq \perp, X \wedge Y = X, X \wedge Y = Y)$, $\perp$ sei die leere Menge

Die zu den Werten zugehörigen Relationen sind in Tabelle 3.5 aufgeführt.

$X \wedge Y \neq \perp$	$X \wedge Y = X$	$X \wedge Y = Y$	RCC5
F	F	F	*dr*
T	F	F	*po*
T	T	F	*pp*
T	F	T	pp^i
T	T	T	*eq*

Tabelle 3.5: Hasse Diagramm der RCC5-Relationen. T=***T****rue*, F=***F****alse*

Die Abbildung 3.4 zeigt auch die Nachbarschaftsbeziehungen zwischen den RCC5-Relationen. Wie bei den Allenschen Zeitrelationen können diese bei Deformationen direkt ineinander umgewandelt werden. Die Komposition der Basisrelationen bzw. Relationsmengen $R \circ S$ geschieht mit der Kompositionstabelle 3.6.

RCC8-Relationen

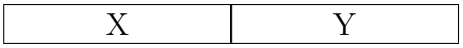
RCC8-Relationen sind die Erweiterung der RCC5-Relationen in Bezug auf die Begrenzungsabhängigkeit. Damit ist gemeint, dass es auch die topologischen Beziehungen im Inneren und bei den Begrenzungen der Regionen beachtet werden. In Abbildung 3.3.1

$R \circ S$	dr	po	pp	pp^i	eq
dr	β	dr, po, pp	dr, po, pp	dr	dr
po	dr, po, pp^i	β	po, pp	dr, po, pp^i	po
pp	dr	dr, po, pp	pp	β	pp
pp^i	dr	dr, po, pp^i	β	pp^i	pp^i
eq	dr	po	pp	pp^i	eq

Tabelle 3.6: RCC5-Kompositionstabelle, $\beta : \{dr, po, pp, pp^i, eq\}$

wird anhand eines Beispiels veranschaulicht, worin der Unterschied zwischen begrenzungsabhängig und begrenzungsunabhängig besteht. Die Aussage *X partial overlapped Y* beschreibt diese Szene mit den RCC5-Relationen. Die gleiche Szene wird mit den RCC8-Relationen als *X externally-connected Y* beschrieben.

Das RCC8-Modell unterscheidet zwischen 8 Basisrelationen (Abbildung 3.6), die aus den RCC5-Relationen hervorgehen (Tabelle 3.7). Die Relation *dr* wurde neu definiert zu *dc* und *ec*, welche die Begrenzungsrelationen beschreiben. Desweiteren wurde *pp* unterteilt in *tpp* und *ntpp* bzw. pp^i in tpp^i und $ntpp^i$. Diese beschreiben die inneren Relationen näher. Das *t* steht dabei für *tangency*, was mit *angrenzend* zu übersetzen wäre.



Begrenzungsabhängig (RCC8): X externally-connected Y
Begrenzungsunabhängig (RCC5): X partial-overlapped Y

Abbildung 3.5: Der Unterschied zwischen begrenzungsabhängig und begrenzungsunabhängig.

Die Tabelle 3.8 enthält die dazugehörigen Werte. Wobei *M* für *meets* steht, welches der Grenzfall ist.

Es kann wieder eine Kompositionstabelle für $R \circ S$ der 8 Basisrelationen β aufgestellt werden. Dies soll hier allerdings aus Platzgründen entfallen. Die Kompositionstabelle kann in [Bit01] nachgeschlagen werden.

Anordnungsabhängige Modelle

Kurz erwähnt seien die RCC_1^9 - und RCC_1^{15} -Relationen, welche die begrenzungsabhängigen und begrenzungsunabhängigen Relationen zwischen eindimensionalen Re-

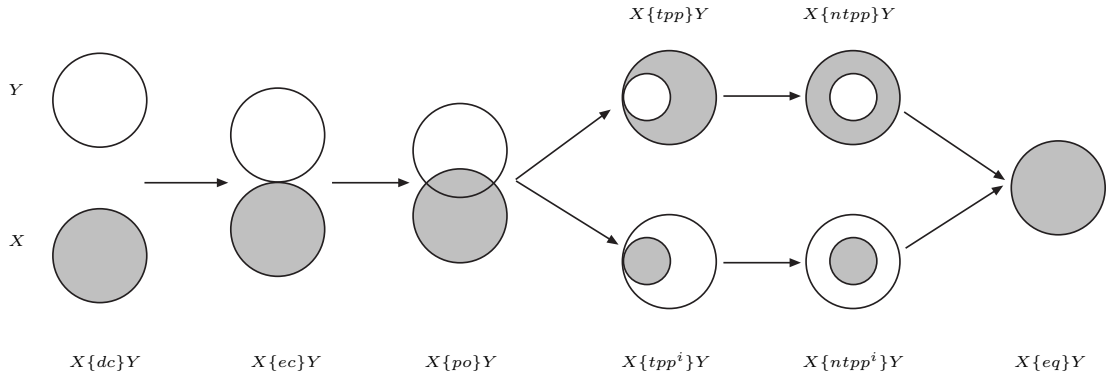


Abbildung 3.6: Definition der RCC8-Relationen.

Basisrelation	Symbol	Inverse
X *disconnected* Y	*dc*	*dc*
X *externally connected* Y	*ec*	*ec*
X *partial overlapped* Y	*po*	*po*
X *tangential proper part* Y	*tpp*	tpp^i
X *non-tangential proper part* Y	*ntpp*	$ntpp^i$
X *equal* Y	*eq*	*eq*

Tabelle 3.7: Die RCC8-Basisrelationen

gionen (Intervallen) repräsentieren. Die hochgestellte Zahl gibt die Anzahl der Relationen wieder und die tiefergestellte Zahl die Anzahl der Dimensionen (hier eindimensional). Im Unterschied zu den RCC5- und RCC8-Relationen wird aber die Anordnung der Regionen zueinander berücksichtigt (Abbildung 3.7). RCC_1^9 ist dabei aus den RCC5-Relationen abgeleitet, wobei 9 Basisrelationen entstehen und RCC_1^{15} wurde aus den RCC8-Relationen abgeleitet; es entstehen 15 Basisrelationen.

Dies sollte hier nur zur Vollständigkeit erwähnt werden und Weiteres kann in [Bit01] ausführlich nachgelesen werden.

Zusammenfassung RCC-Relationen

Die RCC-Verfahren werden dazu benutzt, um Objekte und Intervalle in Bezug auf deren Überlappung zu beschreiben. Dabei werden, wie auch bei den Allen-Zeitrelationen, durch Komposition unbekannte Relationen ermittelt (insofern es möglich ist), die sich aus der Schlussfolgerung bekannter Relationen ergeben.

Bei RCC-Verfahren wird hauptsächlich zwischen *begrenzungsabhängigen* und *be-*

$X \wedge Y \neq \perp$	$X \wedge Y = X$	$X \wedge Y = Y$	RCC8
F	F	F	*dc*
M	F	F	*ec*
T	F	F	*po*
T	M	F	*tpp*
T	T	F	*ntpp*
T	F	M	tpp^i
T	F	T	$ntpp^i$
T	T	T	*eq*

Tabelle 3.8: Hasse Diagramm der RCC8-Relationen ($F < M < T$)

Anordnungsabhängig (RCC_1^{15}): X before Y
Anordnungsunabhängig (RCC8): X disconnected Y

Abbildung 3.7: Der Unterschied zwischen anordnungsabhängig und anordnungsunabhängig.

grenzungsunabhängigen Verfahren (boundary sensitivity) unterschieden [Coh96] [Ben94]. Daraus resultieren die beiden am häufigsten angewendeten Verfahren. Die RCC5-Relationen sind begrenzungsunabhängig und RCC8-Relationen sind begrenzungsabhängig, welches auch als Verfeinerung der Granularität gesehen werden kann. RCC5 ist eine Teilmenge von RCC8. $RCC5 \subset RCC8$. RCC_1^9 - und RCC_1^{15} -Relationen (analog zu RCC5- und RCC8-Relationen) sind anordnungsunabhängig, bei der die Reihenfolge der Regionen (Intervalle) berücksichtigt wird.

	Begrenzung	Anordnung
RCC5	nein	nein
RCC8	ja	nein
RCC_1^9	nein	ja
RCC_1^{15}	ja	ja

Tabelle 3.9: Unterscheidung der verschiedenen RCC-Verfahren

3.3.2 Orientierungsrelationen

Beziehungen zwischen zwei Punkten, welche die Richtungen beschreiben, werden als *Orientierungrelationen* bezeichnet. Es wurden seit Anfang der 90er Jahre mehrere Modelle entwickelt, wobei sich zwei Modelle etabliert haben. Das Kompass-Modell (calculus of cardinal directions), entwickelt von Frank [Fra91] und das Modell mit relativen Orientierungen (calculus of relative orientations), entwickelt von Freska [Fre92].

Beispiel 3.1: Folgende Aussagen werden über die 4 Städte *Berlin, Hamburg, London* und *Paris* getroffen:

1. Von Hamburg aus gesehen, liegt Berlin links von Paris, Paris liegt links von London und Berlin liegt links von London.
2. Von London aus gesehen liegt Berlin links von Paris.
3. Hamburg liegt nördlich von Paris und nordwestlich von Berlin.
4. Paris liegt südlich von London.

Die ersten zwei Sätze beschreiben die Orientierungen der 4 Städte zueinander mit $\mathcal{ROA}$ (Relative Orientation Algebra) Komponenten, wobei die anderen beiden Sätze mit $\mathcal{CDA}$ (Cardinal Directions Algebra) Komponenten beschrieben wurden. Abbildung 3.8 stellt die Szene bildlich in den beiden Modellen dar. Anhand des Beispiels sind die Unterschiede der beiden Modelle gut zuerkennen. Das $\mathcal{CDA}$-Modell bezieht sich, wie bei der üblichen Kartographie, auf die Himmelsrichtungen. Das $\mathcal{ROA}$-Modell geht von einem Elternobjekt aus, auf welches sich das Referenzobjekt bezieht.

Beim $\mathcal{CDA}$-Modell werden die bekannten Himmelsrichtungen, Norden, Süden, Osten und Westen verwendet. Zur Verfeinerung der Granularität werden Zwischenstufen gewählt wie Nordwest, Südost, usw.

Die Abbildung 3.9 zeigt zwei mögliche Modelle, wie die Himmelsrichtungen von absoluten Werten in Relationen umgewandelt werden können. Im Allgemeinen wird von 8 Richtungen ausgegangen sowie von der Gleichheit. Dabei entstehen folgende 9 Basisrelationen: *north(n), south(s), east(e), west(w), north-east(ne), north-west(nw), south-east(se), south-west(sw) und equal(eq).*

$$\beta : \{n, ne, e, se, s, sw, w, nw, eq\}$$

Auch hier können wiederum durch Komposition fehlende Relationen gebildet werden (Tabelle 3.10). Die Tabelle ist abhängig vom verwendeten Modell. In diesem Fall wurde sie für das projektionsbasierten $\mathcal{CDA}$-Modell aufgestellt. Die Komposition

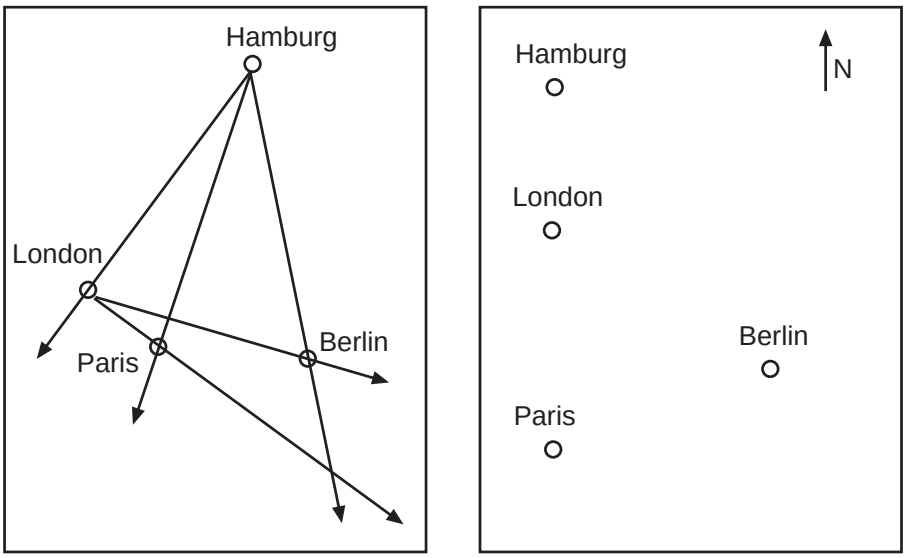


Abbildung 3.8: Das Beispiel 3.1 dargestellt im $\mathcal{ROA}$-Modell (links) und im $\mathcal{CDA}$-Modell (rechts)

erfolgt genau nach dem Schema wie bei den Allens Zeitrelationen und den RCC-Relationen.

Freksa hat bei seinen relativen Orientierungen ($\mathcal{ROA}$-Modell) den 2D-Raum in 15 Regionen unterteilt ausgehend von einem Elternobjekt A und einem Referenzobjekt B [Fre92]. In Abbildung 3.10 wird dies dargestellt. Die Einteilung erfolgt in *left/straight/right* und *front/neutral/back* sowie *front/neutral/back* bei getauschten Rollen des Eltern- und Referenzobjektes. Bei Kombination dieser drei Teile entstehen die erwähnten 15 Regionen, die wie normale Relationen behandelt werden.

Um die Vorteile beider Modelle zu nutzen, können beide miteinander kombiniert werden (um beispielsweise alle 4 Aussagen in Beispiel 3.1 in ein Modell zu bringen). Eine Möglichkeit ist in [IHM] sehr gut beschrieben, sowie die Umwandlung und Interaktion zwischen den einzelnen Modellen.

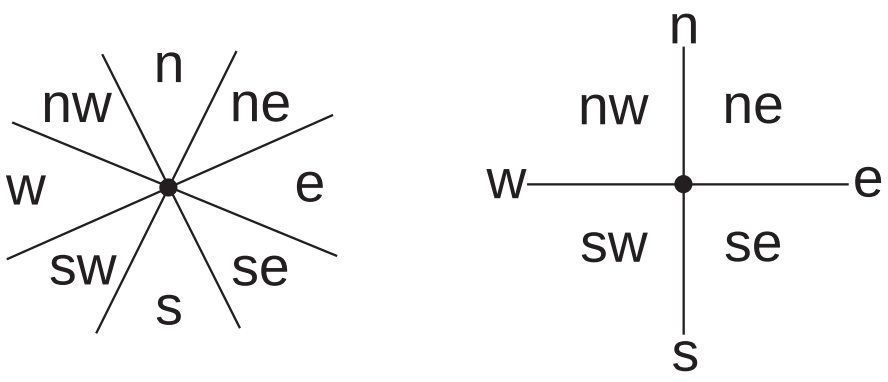


Abbildung 3.9: Verschiedene Einteilungen der Relationen ($\mathcal{CDA}$), konisch (links) und projektionsbasiert (rechts)

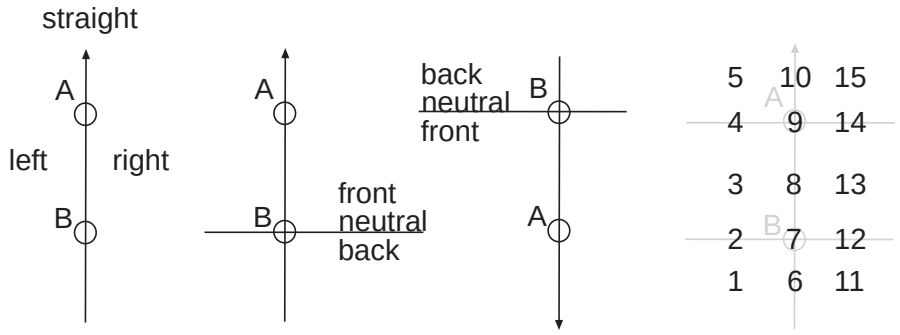


Abbildung 3.10: Die Einteilung des 2D-Raumes nach dem $\mathcal{ROA}$-Modell.

$R \circ S$	n	s	e	w	ne	nw	se	sw	eq
n	n	s, n	ne	nw	ne	nw	se, ne	sw, nw	n
s	s, n	se	sw	nw	se, ne	sw, nw	se	sw	s
e	ne	se	e	we, e	ne	nw, ne	se	sw, se	e
w	nw	sw	we, e	we	nw, ne	nw	sw, se	sw	w
ne	ne	se, ne	ne	nw, ne	ne	nw, ne	se, ne	β	ne
nw	nw	sw, nw	nw, ne	nw	nw, ne	nw	β	sw, nw	nw
se	se, ne	se	se	sw, ne	se, ne	β	se	sw, ne	se
sw	sw, nw	sw	sw, se	sw	β	sw, nw	sw, se	sw	sw
eq	n	s	e	w	ne	nw	se	sw	eq

Tabelle 3.10: $\mathcal{CDA}$ Kompositionstabelle mit 8 Richtungen (projektionsbasiert), $\beta : \{n, ne, e, se, s, sw, w, nw, eq\}$

3.4 Szenegraph

Da wir nicht mit statischen Daten arbeiten, muss ein Datenmodell gefunden werden, welches dynamische Objekte beschreibt. Ein guten Ansatz liefert Nabil [NNS01], wo er ein Datenmodell und eine Anfragesprache vorstellt. Er benutzt diese Technik, um aus Videodaten (Bildfolgen) Objekte und deren räumlichen und zeitlichen Beziehungen zu modellieren. Damit möchte er später Anfragen stellen, wie „Finde ein Bild, auf dem ein roter Apfel auf einem Tisch liegt“. Es folgt eine zusammenfassende Beschreibung des Datenmodells.

Um eine statische Szene zu beschreiben, werden in diesem Modell drei Relationen benötigt. Die topologische Relation δ in Bezug auf die Überlappung, wobei das RCC8-Modell benutzt wird[2]. Die anderen beiden Relationen χ und ψ sind Intervallrelationen (Allen), die die Lage der Objekte zueinander beschreiben. Genauer ausgedrückt, beschreibt es die Objektintervalle in der x- bzw. y-Ausdehnung. Die Relation χ bezieht sich auf die x-Achse und ψ auf die y-Achse. Allens Zeitintervalle sind also nicht nur auf zeitliche Beziehungen anwendbar, sondern allgemein auf Intervalle.

Diese drei Relationen werden in einem Tripel zusammengefasst: (δ, χ, ψ), welches als *2D-PIR* bezeichnet wird(2-Dimensional Projection Intervall Relationship). In Abbildung 3.11 ist dies an einem Beispiel dargestellt sowie ein daraus erstellter Graph, an dessen Kanten die entsprechenden Tripel angetragen wurden.

2D-PIR dient als Basis für ein Modell, welches auch Objektbewegungen einschließt. Der Unterschied besteht darin, dass auch der Faktor Zeit in das Modell einfließen muss. Das 2D-PIR-Tripel wird um ein Zeitintervall I erweitert, sodass ein 4-Tupel entsteht: (δ, χ, ψ, I). Das Zeitintervall I ist die Zeit, an dem die Relationen gültig sind. Dieses Tupel wird *TS-2D-PIR* genannt. Das *TS* steht dabei für *temporal sequence*.

Es wird eine Sequenz der Form:

$$[(\delta_1, \chi_1, \psi_1, I_1), (\delta_2, \chi_2, \psi_2, I_2), \ldots, (\delta_n, \chi_n, \psi_n, I_n)].$$

erstellt. Jedes Tupel stellt also einen Status dar, der in einem gewissen Zeitintervall gilt. Das Intervall definiert sich aus der Sequenz der Tupel.

$$ende(I_1) = start(I_2), ende(I_2) = start(I_3), \ldots, ende(I_n) = start(I_{n-1})$$

Ein Graph, wo die Objekte die Knoten sind und die Kantengewichtungen die TS-2D-PIR Sequenzen, wird *Szenegraph* genannt.

[2]Nabil verwendet eine andere Notationsweise für die Relationen. In dieser Arbeit wurde diese auf die RCC8-Notationsweise übertragen.

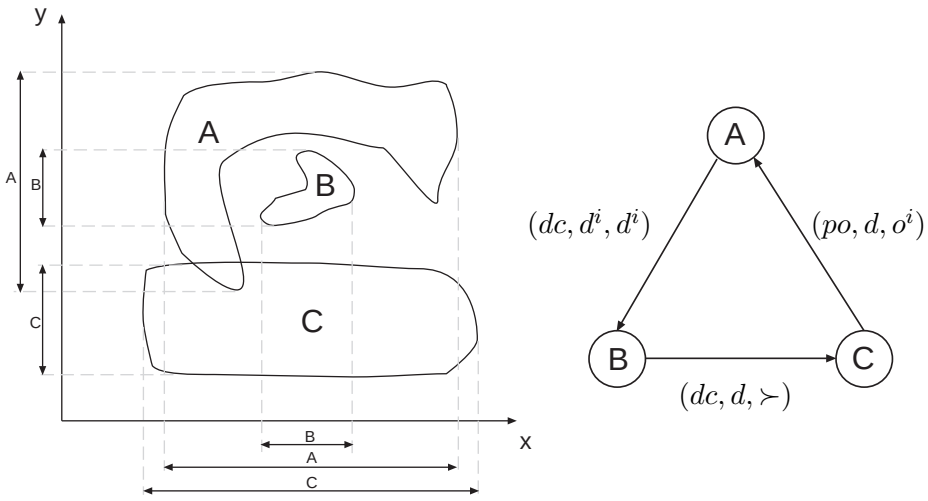


Abbildung 3.11: Beispiel Szene mit den Projektionen (links) und der dazugehörige 2D-PIR-Graph (rechts)

Definition 3.2: Ein Szenegraph ist ein gerichteter, gewichteter Graph der Form $S(M, R)$, wobei M eine Menge von Objekten der Szene ist und R eine Menge von Kanten, an denen eine Sequenz von TS-2D-PIR's zwischen den jeweiligen Objektpaaren angetragen ist. $M \neq \emptyset$

Wie und wohin sich ein Objekt bewegt, wird durch ein Tripel (d, r, I) dargestellt. Der Wert d ist die Entfernung, die das Objekt zurückgelegt hat, r die Richtung und I ist das Zeitintervall. Die Richtung wird in Zahlen kodiert. In Abbildung 3.12 ist die Kodierung graphisch erläutert. Die 8 Himmelsrichtungen wurden in Zahlen kodiert, um mit ihnen einfacher arbeiten zu können.

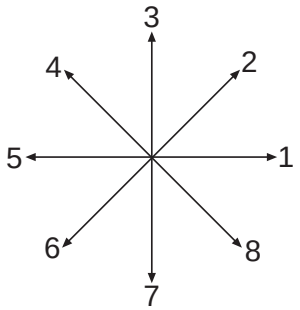


Abbildung 3.12: Kodierung der Richtungen

Ein Beispiel eines Szenegraphen eines einzelnen Objektes, welches sich in der Szene bewegt, ist in Abbildung 3.13 dargestellt. Es wird immer nur ein neues Intervall begonnen, wenn eine Veränderung der Relationen bzw. Bewegungsdaten stattfindet.

Ein komplexeres Beispiel wird in Abbildung 3.14 gezeigt. An der Kante zwischen den Objekten A und B steht die Sequenz für die Veränderung der Objektbeziehungen in dem Zeitraum $I_1 - I_4$. Das Start-Tupel $(d_1, 4, I_1)$ sagt aus, dass im Zeitintervall I_1 die Objekte A und B sich nicht überlappen und grob auf der selben x-Achse liegen. In anderen Worten: B ist direkt über A. Das letzte Tupel $(d_4, 2, I_4)$ ist der Endzustand, bei dem sich Objekt A in Objekt B befindet.

Die Schleifen über den Knoten A und B geben die Objektbewegung wieder. Objekt A bewegt sich nach Norden und Objekt B nach Süden, und zwar über die Distanz d_a bzw. d_b in dem Intervall I_a bzw. I_b. Zu beachten ist: $I_a = (start(I_1), ende(I_5)), solange I_b = (start(I_1), ende(I_4))$. Also Objekt A bewegt sich, zeitlich gesehen, etwas länger als Objekt B.

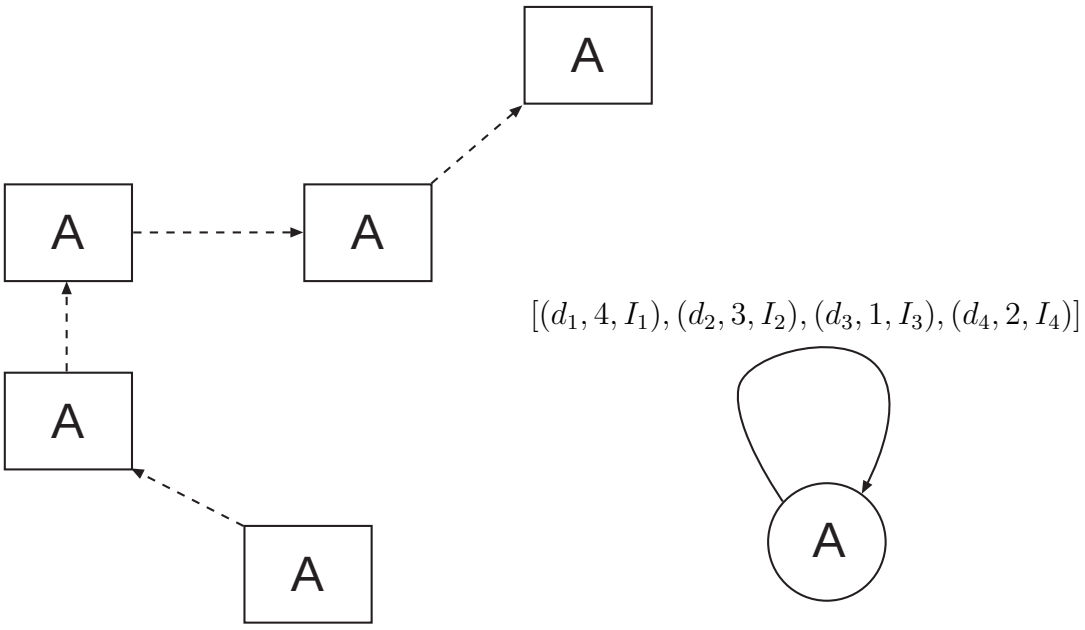


Abbildung 3.13: Ein sich bewegendes Objekt (links) und der dazugehörige Szenegraph (rechts)

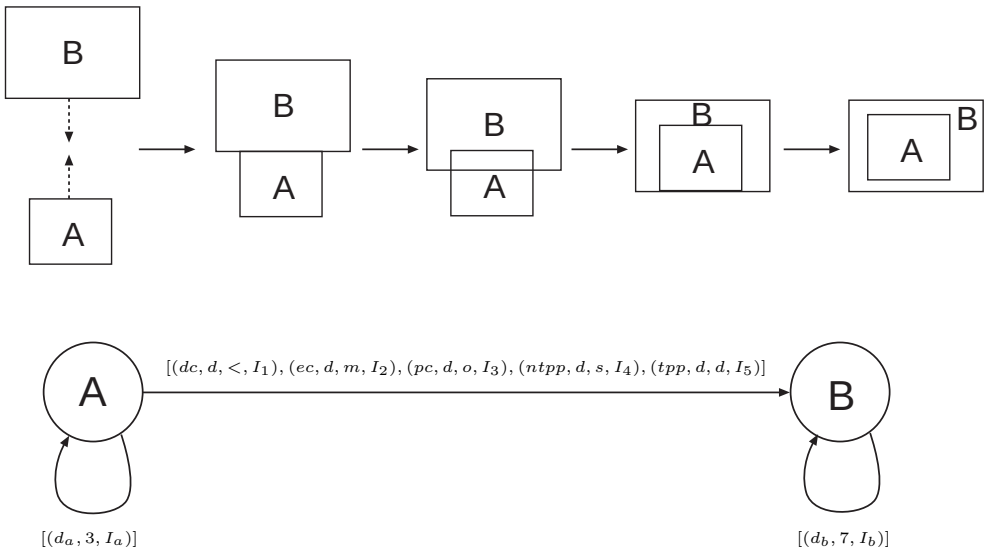


Abbildung 3.14: Zwei Objekte, die sich aufeinander zu bewegen (oben), und der dazugehörige Szenegraph (unten)

3.5 Auswahl

Es ist zu entscheiden, inwieweit die vorhandenen Grundlagen in diesem Projekt verwendet werden können oder ob völlig neue Modelle entwickelt werden müssen.

Es hat sich herausgestellt, dass sich mit den RCC-Relationen Beziehungen zwischen Objekten gut beschreiben lassen. Allerdings sind die RCC8-Relationen zu detailliert. Die Grenzbeziehungen treten in der Praxis zwar auf, wenn z.B. zwei Objekte aneinander stoßen, aber es brachte keinen Vorteil, diese Grenzbeziehungen in das Modell einzubeziehen. Deshalb erscheint das RCC5-Modell als geeigneter, um zu beschreiben, wie sich Objekte überlappen.

Die Nachbarschaftsbeziehungen können nicht angewendet werden, da durch niedrige Frameraten der Bildverarbeitung oder durch schnelle Bewegungen von Objekten die Nachbarschaftsbeziehungen nicht mehr gültig sind.

Die Orientierungsrelationen haben sich als nicht geeignet herausgestellt, da die Orientierung der Objekte zueinander bei der Auswertung, wo sich welche Objekte befinden, keine Rolle spielt. Es ist unerheblich, ob sich Objekt A nördlich oder südlich von Objekt B befindet.

Der Szenegraph erscheint als sehr gute Datenstruktur, um den Verlauf von Objektbewegungen zu repräsentieren. Bei den aktuellen Randbedingungen ist es allerdings nicht notwendig, Auswertungen über einen längeren Zeitraum anzustellen. Es wird immer nur die aktuelle Situation der Szene betrachtet. Aber vielleicht kann es in einer Weiterentwicklung Verwendung finden, wenn mit Wahrscheinlichkeiten gearbeitet wird oder vorausgesagt werden soll, wohin sich ein Objekt bewegen wird.

Es ist noch zu klären, wie Mehrdeutigkeiten der Position der Objekte dargestellt werden können und wie die Anordnung der Objekte hintereinander repräsentiert werden kann. Die RCC-Relationen sagen zwar aus, wie sich zwei Objekte überlappen, aber es lässt sich daraus nicht ableiten, in welcher Reihenfolge die Objekte in der Tiefe zueinander liegen. Hierfür muss demzufolge jeweils ein Modell entwickelt werden.

Fazit Also ist zusammenfassend zu sagen, dass derzeit nur das RCC5-Modell für das Projekt angewendet werden kann. Der Szenegraph sollte bei möglichen Weiterentwicklungen des Systems mit in Betracht gezogen werden. Desweiteren sind zwei neue Modelle zu entwickeln, womit Mehrdeutigkeiten dargestellt werden können und die Anordnung der Objekte in der Tiefe beschrieben werden kann.

Kapitel 4

Konzeption

Es hat sich bei der Untersuchung der vorhandenen Modelle ergeben, dass sich das RCC5-Modell gut dafür eignet, um Beziehungen zwischen Objekten in der Ebene zu repräsentieren. Allerdings wurde auch festgestellt, dass sich mit den untersuchten Modellen, keine Mehrdeutigkeiten abbilden lassen. Weiterhin ist es notwendig, Objekte in der Tiefe (die Anordnung der Objekte entlang der z-Achse) zu beschreiben, um zu untersuchen, ob es einem Objekt überhaupt möglich ist, sich unter ein anderes zu schieben.

Zuerst wird die eigens entwickelte Tiefenrelation vorgestellt, die die Reihenfolge der Objekte in der Tiefe, durch topologische Relationen, beschreibt. Im Anschluss wird diskutiert, mit welchem Format sich Objektdaten optimal, über eine Schnittstelle, von der Bildverarbeitung zur Wissensbasis übertragen lassen.

Der Hauptteil dieses Kapitels besteht aus der Beschreibung von Aufbau und Ablauf der Wissensbasis. Dazu gehört, mit welcher Datenstruktur sich Mehrdeutigkeiten darstellen lassen und welche Algorithmen notwendig sind, um das Problem (für Kreise) zu lösen.

4.1 Tiefenrelation

Es reicht nicht aus, nur Relationen in Bezug auf die Objektposition entlang der x,y-Achse zu definieren, sondern auch entlang der z-Achse (also die Reihenfolge der Objekte in der Tiefe). Es handelt sich hierbei allerdings nicht um Intervalle, sondern nur um Punkte, da wir als Voraussetzung gegeben haben, dass die Objekte keine Tiefe besitzen, also keine Ausdehnung in z-Richtung. Wir gehen als Erstes davon aus, dass Objekte immer in einer Ebene liegen. Damit existieren nur 3 mögliche Relationen, in der die Objekte zueinander liegen können (Tabelle 4.1). Als Basis dient eine Sub-

klasse von Allen's Basisrelationen (siehe 3.2). Es sind nicht alle 13 Basisrelationen notwendig, da die Intervalle wegfallen und es sich um Punkte handelt.

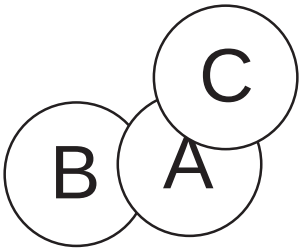


Abbildung 4.1: Beispiel der Anwendung für die Tiefenrelation. Objekt A liegt über Objekt B, Objekt C liegt über Objekt A. Daraus folgt, dass Objekt C auch über Objekt B liegen muss. $A\{\prec\}B \wedge C\{\prec\}A \rightarrow C\{\prec\}B$

Basisrelation	Bezeichnung	Abkürzung	Symbol	Inverse
X hinter Y	behind	bh	$\succ$	$\prec$
X vor Y	in front	fr	$\prec$	$\succ$
X gleich Y	equal	eq	$\equiv$	$\equiv$

Tabelle 4.1: Tiefenrelationen

Die Vorgehensweise ist ähnlich wie bei den Allenschen Relationen. Die Kompositionstabelle der Basisrelationen ist einfach nachzuvollziehen (Tabelle 4.2). β ist dabei die Menge aller Basisrelationen. $\beta : \{\prec, \succ, \equiv\}$

$B_1 \circ B_2$	$\succ$	$\prec$	$\equiv$
$\succ$	$\succ$	β	$\succ$
$\prec$	β	$\prec$	$\prec$
$\equiv$	$\succ$	$\prec$	$\equiv$

Tabelle 4.2: Kompositionstabelle der Basisrelationen

Da nur 3 Basisrelationen existieren, kann eine Tabelle mit allen Kombinationen aufstellt werden (Tabelle 4.3).

Unter den Bedingungen $(R, S) \subseteq \beta, (R, S) \neq \emptyset$ und Formel 3.1 kann folgenden Formeln aufstellt werden, die die Kompositionstabelle wiederspiegelt:

$R \circ S$	$\{\succ\}$	$\{\prec\}$	$\{\equiv\}$	$\{\succ, \prec\}$	$\{\succ, \equiv\}$	$\{\prec, \equiv\}$	β
$\{\succ\}$	$\{\succ\}$	β	$\{\succ\}$	β	$\{\succ\}$	β	β
$\{\prec\}$	β	$\{\prec\}$	$\{\prec\}$	β	β	$\{\prec\}$	β
$\{\equiv\}$	$\{\succ\}$	$\{\prec\}$	$\{\equiv\}$	$\{\succ, \prec\}$	$\{\succ, \equiv\}$	$\{\prec, \equiv\}$	β
$\{\succ, \prec\}$	β	β	$\{\succ, \prec\}$	β	β	β	β
$\{\succ, \equiv\}$	$\{\succ\}$	β	$\{\succ, \equiv\}$	β	$\{\succ, \equiv\}$	β	β
$\{\prec, \equiv\}$	β	$\{\prec\}$	$\{\prec, \equiv\}$	β	β	$\{\prec, \equiv\}$	β
β	β	β	β	β	β	β	β

Tabelle 4.3: Alle 49 möglichen Kompositionen. $\beta = \{\succ, \prec, \equiv\}$

$$
\begin{array}{lcll}
R \circ S & \leftrightarrow & S \circ R & \text{(Symmetrie)} \\
R \circ R & \rightarrow & R & \text{(Reflexivität)} \\
R \circ \beta & \rightarrow & \beta & \text{(Vereinigung)} \\
R \circ \{\equiv\} & \rightarrow & R & \text{(Gleichheit)} \\
\succ \circ \prec & \rightarrow & \beta & \text{(Widerspruch)}
\end{array}
$$

4.2 Wahl des Aufbaus der Objektdaten

Zum Austausch der Objektdaten zwischen Bildverarbeitung und der Wissensbasis muss eine Schnittstelle definiert werden. Es gibt mehrere Möglichkeiten, diese zu realisieren. Es gilt ein geeignetes Format zu finden, welches einfach zu handhaben, flexibel ist, nicht zu viel Speicherkapazität beansprucht und sinnvoll für den späteren Gebrauch ist.

Als erstes ist zu klären, welche Daten zwischen Bildverarbeitung und Wissensbasis ausgetauscht werden müssen:

1. Objektdaten pro Objekt, beispielsweise:
   - Objekt ID. Zum Identifizieren des Objektes.
   - Die Objektgruppe mit den entsprechenden Eigenschaften.
   - Typ des Objektes, z.B. Kreis, Rechteck, Hand, Allgemeines Objekt.
   - Beschreibung der Objektform und die vom Objekt einnehmende Fläche.
   - Subdividiertes Objekt in primitive Unterobjekte, wie z.B. Rechtecke. Dies ist eine Serviceleistung der Bildverarbeitung, um spätere Berechnungen zu vereinfachen.
   - Weitere Attribute wie Farbe, Textur, Name, ...
2. Regionale Objektdaten, entweder pro Pixel oder Pixelgruppen:
   - Absolute (x,y)-Position des Pixel in der Szene.
   - Ist der Pixel sichtbar oder verdeckt? Genauer gesagt, wurde der Pixel sichtbar wahrgenommen oder (nach Annahmen) rekonstruiert ?
   - Welche Objekte verdeckt der Pixel bzw. von welchen Objekten wird der Pixel verdeckt?
   - Evtl. weitere Attribute wie z.B. Farbe.

Die Darstellung der primitiven Objekte ist unabhängig von den hier angesprochenen Formaten. Die primitiven Objekte sind nur eine Liste von Rechtecken, die die vereinfachte Objektform beschreiben, um darauf aufbauend schneller geometrische Operationen anzuwenden. Es ist dabei auch zu markieren, welches Rechteck verdeckt ist und welches nicht.

Ziel ist die Beschreibung der Objektform und der regionalen Objektattribute. Allgemeine Objektattribute, wie ID, Name, etc. werden einfach als Datenstruktur mit übermittelt.

Matrix pro Objekt

Für jedes Objekt wird eine Matrix der Größe des Bildes erstellt (4.1), in der die Teile markiert (gefüllt) werden, welche sich auf dieses Objekt beziehen. Pro Pixel werden die entsprechenden Attribute vergeben, wie u.a. das Attribut *verdeckt* (4.2).

$$M(m,n) = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} \tag{4.1}$$

$$a_{m,n} := \{Attribut_1, Attribut_2, \dots, Attribut_x\} \tag{4.2}$$

Ein großer Nachteil ist der unnötig hohe Speicherverbrauch, da immer eine Matrix so groß wie das Bild pro Objekt erstellt werden muss. Bei vielen Objekten kann die Grenze der Speicherkapazität hierdurch schnell erreicht werden. Eine Objektanzahl < 100 und einer Bildgröße $< 640 * 480$ (zusammen $\sim 60MB$) sollte es bei heutigen Rechnern aber noch zu bewältigen sein. Allerdings ist diese Verfahrensweise unwirtschaftlich. da der Speicherbedarf sehr hoch ist.

Pixelliste

Die Pixelliste ist ähnlich des o.g. Verfahrens, der *Matrix pro Objekt*. Der wesentliche Unterschied besteht darin, dass nicht Matrizen verwendet werden, sondern nur Daten zu den Pixel gespeichert werden, die zu dem Objekt gehören. Dadurch wird der Nachteil des durchschnittlich hohen Speicherbedarfs eliminiert. Allerdings ist die Handhabung um einiges komplizierter, da die ganze Liste (die sehr groß sein kann) immer durchsucht werden muss, wenn ein Pixel an der Position (x,y) zu finden ist.

Die Objektdaten sehen folgendermaßen aus:

Pixelliste=$\{p_1,p_2,\dots,p_n\}$
p_n:=(x,y,a)
a:=$\{Attribut_1,Attribut_2,\dots,Attribut_x\}$

Die Struktur der Daten in der Programmiersprache C könnte folgendermaßen aussehen:

```
struct  Objektpixel
{
        int x,y;                // Position des Bildpunktes im Bild
        bool ist_verdeckt;      // Flag, ob der Pixel verdeckt ist oder nicht
};
```

```
struct Objekt
{
        int ID;                 // eindeutige Objekt ID
        Objektpixel *pixel;     // Alle Pixel zu dem Objekt, Ende=0
        Color farbe;            // Objektfarbe
};
```

Eine Matrix für alle Objekte

Um nun die Vorteile aus beiden Formaten zu vereinigen und die Nachteile gering zu halten, gibt es folgende Möglichkeit. Man erstelle eine Matrix, die so groß wie das Bild ist. Jedes Matrixelement ist ein Array mit unterschiedlicher Größe. Jeder Eintrag in diesem Array ist nun eine Referenz (Objekt ID) auf das Objekt, das diesem Pixel zugeordnet ist mit den dazugehörigen Pixelattributen.

Also ähnlich dem *Matrix pro Objekt*-Format, mit dem Unterschied, dass nur Speicher für die wirklich zugeordneten Pixel belegt wird. Und in Anlehnung an das *Pixelliste*-Format, muss hier nicht lange nach dem Pixel gesucht werden. Jeder Pixel ist direkt adressierbar und das Array ist oft sehr klein ($0 \leq Arrayelemente \leq Objektanzahl$).

Die Matrix ist folgendermaßen aufgebaut:

$$M(m,n) = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}$$

$$a_{m,n} := \{o_1, o_2, \dots, o_x\}$$

$$o_x := \{ObjektID, \{Attribut_1, Attribut_2, \dots, Attribut_x\}\}$$

Polygonliste

Als die wohl im Durchschnitt speichersparenste Methode ist die, nur die Umrisse des Objektes wiederzugeben, und zwar in Form von Polygonen. Damit ist gemeint, dass aus einer Menge von Bildpunkten ein geschlossenes Polygon gebildet wird. Dieses erreicht man, indem man den Objektrand „abtastet“ und über einen Schwellwert festlegt, wann die Kante des Polygons beendet ist und eine neue Kante begonnen werden muss, indem man ein neuen Punkt im Polygon setzt. Durch den notwendigen Schwellwert ist auch ein (leichter) Verlust von Objektdaten wahrscheinlich. Wenn man ohne Schwellwert arbeiten würde, hätte man in den meißten Fällen eine Unmenge

von Polygonkanten und der Vorteil der Speicherersparnis ginge zum großteil verloren. Selbst bei Objekten mit geraden Kanten, da die Bildverarbeitung keine 100%ig exakten Bilder liefern kann.

Für unseren Fall ein großer Nachteil der Polygonlisten ist, dass diese nur den Objektumriss beschreibt und wenn man mit den Pixeln innerhalb des Objektumrissesü arbeiten möchte, ist dies schwierig, da diese erst berechnet werden müssen und dies ist relativ schwer. Ein weiterer Nachteil ist, dass man die verdeckten Flächen (allgemeiner gesagt Attribute) auch wieder als extra Umriss beschreiben muss. Da diese verdeckten Flächen nicht immer zusammenhängend sind, muss pro Pixelgruppe ein Umriss erstellt werden. Wenn man beim Erstellen der Daten nicht aufpasst, kann es zu inkonsistenten Daten kommen oder zu anderen Ungereimtheiten.

Eine Polygonliste könnte folgendermaßen aussehen:

Polygonliste=$\{p_1,p_2,\ldots,p_n\}$
p_n:=(x,y)

Punktliste

Wie beschrieben, können bei den Polygonlisten Verluste auftreten. Um das zu verhindern gibt es eine andere Methode, und zwar die der Punktlisten die den genauen Umriss speichersparender beschreibt. Dabei wird anstatt der absoluten Position der Umrisspunkte nur die relative Richtung angegeben in der sich der nächste Nachbarpixel, der zu dem Umriss gehört, befindet. Da wir mit binären Bildern arbeiten, gibt es nur 8 Richtungen (siehe Abb. 4.2). Zu dem ersten Pixel des Umrisses gibt man noch die absolute Position an und dann verfolgt man die Kante (z.B. im Uhrzeigersinn) bis man wieder an diesem Ursprungspunkt angekommen ist. Jeder Pixel wird dann nur noch mit dem 3Bit-Wert der Richtungskodierung beschrieben.

0	1	2
3		4
5	6	7

Abbildung 4.2: Kodierung der 8 Richtungen, in der sich ein Nachbarpixel befinden kann.

Eine Punktliste für ein Quadrat mit der Kantenlänge = 3 Pixel könnte folgendermaßen aussehen:

Punktliste={1,1,1,4,4,4,6,6,6,3,3,3}

Zusammenfassung und Entscheidung für ein Format

Die Tabelle 4.4 zeigt einen Vergleich der einzelnen Formate nach ausgewählten Kriterien. Im Prinzip ist es fast irrelevant für welches Format man sich entscheidet, da ein Format in das andere umgewandelt werden kann. Allerdings können bei einigen Formaten Qualitätsverluste entstehen.

Die beiden Formate, die nur den Umriss des Objektes beschreiben, sollte man nicht zum Anfang verwenden, da diese relativ schwer zu erstellen sind und man damit nicht so einfach/schnell die bedeckte Fläche des Objektes bestimmen kann. Den Vorteil der einfachen geometrischen Operatoren kann man verkraften, da die Bildverarbeitung zusätzlich auch eine Liste mit primitiven Unterobjekten zurückgibt, auf die man dann die geometrischen Operatoren anwenden kann. Außerdem können viele Operatoren, die normalerweise geometrisch gelöst werden, auch (zum Teil sogar besser) Pixelweise gelöst werden; wie z.B. die Bestimmung der überlappten Fläche oder ob ein Objekt an einer Position (x,y) völlig verdeckt wird oder nicht.

Das Format **Matrix pro Objekt** ist zwar sehr einfach aufgebaut, verbraucht aber unnötig viel Speicher und sollte daher so nicht verwendet werden. Das Format **Pixelliste** ist zu umständlich und man kommt wohl nicht drumherum es doch wieder in eine Matrix umzuwandeln, da diese einfacher zu handhaben ist, ohne ständige Suche nach Objektpunkten an Position (x,y). Die beste Variante ist wohl **Matrix für alle Objekte**. Diese ist zwar im Aufbau komplizierter als die anderen, aber dafür relativ speichersparend und beschreibt gut alle notwendigen Daten. Den komplizierteren Formataufbau kann man durch Kapselung in OOP-Klassen stark vereinfachen. Der leichte Geschwindigkeitsverlust sollte nicht so schwer ins Gewicht fallen, wie der sonst so hohe Speicherverbrauch. Eine Alternative dazu gäbe es vielleicht noch, indem man bei den **Matrix pro Objekt**-Format die Matrizen nur so groß macht, wie die jeweilige Objektausdehnungen ist (Boundingbox) und damit viel Speicher spart. Allerdings ist dann der Aufwand wieder größer, da mit unterschiedlich großen Matrizen gearbeitet werden muss.

Also ist **Matrix für alle Obj** das Format, welches am besten geeignet ist und ein Kompromiss zwischen Aufwand und Speicherverbrauch.

Format	Verlust von Daten	Speicherverbrauch	Anzahl der Daten	Einfachheit des Formates
Matrix pro Objekt	nein	sehr hoch	sehr viel	sehr einfach
Pixellisten	nein	hoch	sehr viel	einfach
Matrix für alle Obj	nein	annehmbar	sehr viel	kompliziert
Polygonlisten	ja	sehr gering	sehr wenig	kompliziert
Punktlisten	nein	gering	wenig	kompliziert

Format	Bilden von Pixelgruppen	Anw. geom. Operatoren	Anw. pixelb. Operatoren	Format aus einem Bild bilden
Matrix pro Objekt	sehr einfach	sehr schlecht	sehr gut	sehr einfach
Pixellisten	umständlich	sehr schlecht	schlecht	sehr einfach
Matrix für alle Obj	einfach	sehr schlecht	gut	einfach
Polygonlisten	kompliziert	sehr gut	sehr schlecht	sehr schwer
Punktlisten	kompliziert	gut	sehr schlecht	schwer

Format	Attribute pro Pixel möglich	Beschreibt
Matrix pro Objekt	ja	Fläche
Pixellisten	ja	Fläche
Matrix für alle Obj	ja	Fläche
Polygonlisten	nein	Umriss
Punktlisten	nein	Umriss

Tabelle 4.4: Vergleich der einzelnen Formate nach ausgewählten Kriterien

4.3 Wissensbasis

Die Wissensbasis ist das zentrale Modul des Systems und hat mehrere Aufgaben zu bewältigen. Einerseits muss es die Objektdaten in einer Datenbank verwalten und die von der Bildverarbeitung gelieferten Daten verarbeiten, um damit den vermuteten Zustand der Welt zu aktualisieren. Andererseits muss es Anfragen vom Benutzer auf das aktuelle Wissen erlauben, sowie eine Schnittstelle zur Kommunikation mit den anderen beiden Modulen bereitstellen. Nicht zu vergessen ist, die Verwaltung und Konfiguration der Datenbanken, die es beispielsweise erlaubt, Datenbanken zu erstellen, zu löschen und deren Parameter zu verändern.

Die Aufgaben sind damit:

- Verwaltung der Daten.
- Neuberechnen des Zustandes aus dem bekannten Wissen und dem neuen Wissen aus der Bildverarbeitung.
- Erlauben von Anfragen auf das bekannte Wissen.
- Bereitstellen einer Kommunikationsschnittstelle zu den Modulen: Bildverarbeitung und Anwendung.
- Funktionen zur Verwaltung und Konfiguration der Datenbanken.

In diesem Abschnitt soll es hauptsächlich um die ersten drei Punkte diskutiert werden. Also wie die Daten am effektivsten gespeichert und verwaltet werden und wie ein neuer Zustand berechnet wird. Das beinhaltet auch, die Realisierung von Anfragen an die Wissensbasis.

Zu den Details der Schnittstellen, Datenbankverwaltung und Konfiguration sei auf das Kapitel Umsetzung verwiesen und auf den Abschnitt 4.2.

4.3.1 Datenfluss

In Abbildung 4.3 ist verdeutlicht, welche Kommunikationswege zur Wissensbasis existieren. In diesem Abschnitt geht es um den Datenfluss zwischen Bildverarbeitung und Wissensbasis, welcher einseitig von der Bildverarbeitung zur Wissensbasis verläuft. In diesem Datenstrom werden die Daten zu einem Bild übertragen.

Zu übertragen ist die Zeit, an dem das Bild (Frame) aufgenommen wurde, sowie die Objektdaten. Die Objekte wurden zuvor von der Objektklassifikation in Klassen eingestuft und fehlende Teile von der Bildverarbeitung rekonstruiert. Wie die Objektdaten aussehen können, ist im Abschnitt 4.2 beschrieben. Für diesen Fall, wo nur Kreise zulässig sind, reicht der Radius, die Farbe und die Position aus.

Da wir einen Simulator verwenden, ist die Einstufung in Klassen und die Rekonstruktion der verdeckten Objektteile trivial.

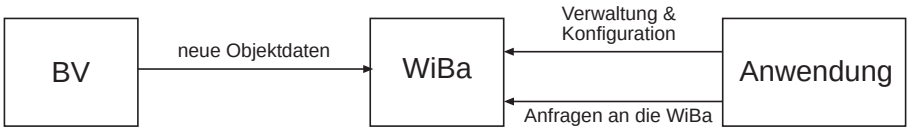


Abbildung 4.3: Die verschiedenen Kommunikationswege zur Wissensbasis

Der Ablauf gestaltet sich folgendermaßen: Zuerst wird bekanntgegeben, dass ein neuer Frame beginnt. Informationen zu dem Frame werden gleichzeitig übertragen. Es werden aber vom Simulator keine Bilddaten übertragen und damit ist nur der Zeitpunkt von Interesse, an dem das Bild aufgenommen wurde. Danach folgt eine Sequenz von Objekten und zum Schluss wird bekanntgegeben, dass das Übertragen des Frames abgeschlossen ist (Abbildung 4.4).

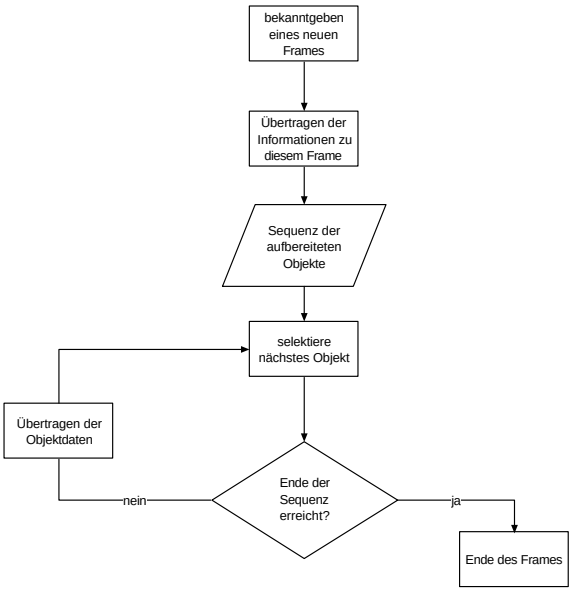


Abbildung 4.4: Ablauf beim Einfügen eines neuen Frames in die Wissensbasis

Dies ist der Ablauf, wie neue Objektdaten zur Wissensbasis gelangen. Wie diese Daten dort repräsentiert werden wird im folgenden Abschnitt erläutert.

4.3.2 Aufbau

Ein wichtiger konzeptioneller Teil dieser Arbeit war es, den Aufbau der Wissensbasis zu definieren. Es wurde ein geeignetes Datenmodell entwickelt, mit dem sich die mehrdeutigen Positionen und Relationen der Objekte optimal verwalten lassen. Zur Übersicht wurde ein vereinfachtes ER-Modell der Datenstruktur erstellt, welches in Abbildung 4.5 dargestellt ist.

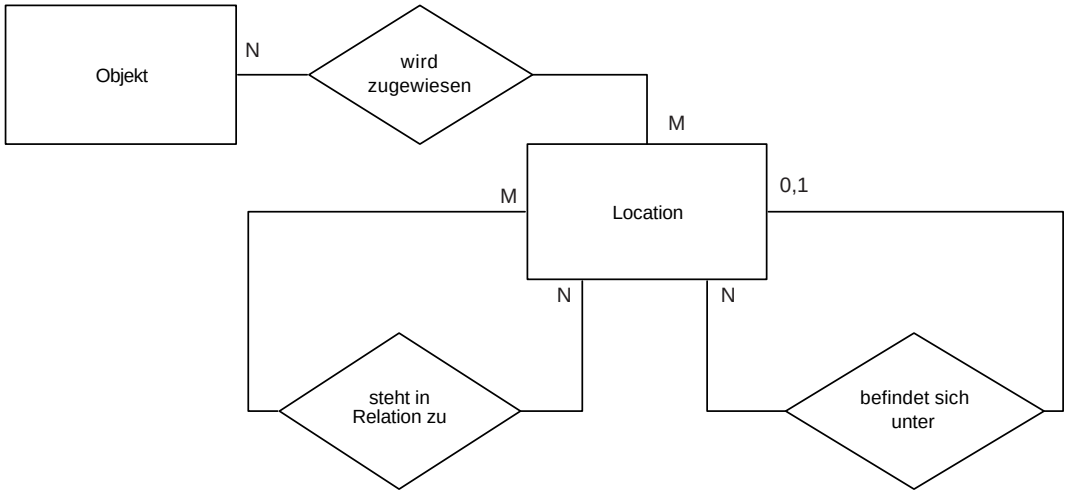


Abbildung 4.5: ER-Modell der Datenstruktur

Die Flexibilität der Wissensbasis lässt sich ohne großen Aufwand steigern, indem das System so aufbaut wird, dass es mehrere Datenbanken gleichzeitig verwalten kann. Dies ist zwar zum aktuellen Zeitpunkt nicht von Vorteil, ermöglicht aber in zukünftigen Anwendungen[1], Datenbanken zu wechseln und Teile von Szenen in unterschiedliche Datenbanken zu packen, um sie einzeln zu analysieren.

Der Aufwand besteht darin, dass zu jedem Datensatz noch zusätzlich die Datenbank-ID mit angegeben wird. D.h., wenn Beispielsweise der Datensatz: $obj(circle, 10, [100, 100])$ gespeichert werden soll, wird zusätzlich die ausgewählte Datenbank (hier *db1*) mit angegeben: $obj(db1, circle, 10, [100, 100])$.

Objektdaten Objekte werden beim ersten Auftreten neu in die Datenbank aufgenommen und bei Änderung aktualisiert. Es gibt verschiedene Arten von Objekten, die unterschiedliche Daten beinhalten. Ein Kreis-Objekt wird durch einen Radius und

[1] Hauptsächlich bei nicht Echtzeit-Anwendungen, wo gespeicherte Videodaten als Medium genutzt werden, um sie zu analysieren.

eine Farbe beschrieben, ein Rechteck durch Höhe, Breite und Farbe und allgemeine konvexe Objekte durch die Beschreibung des Umrisses.

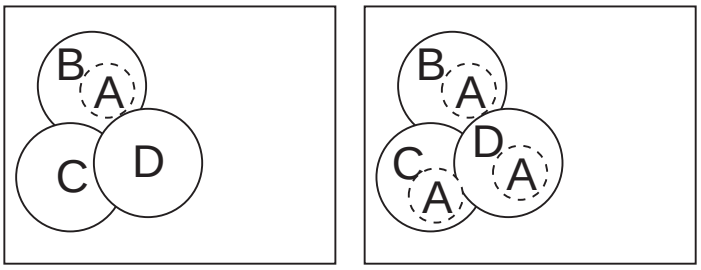
Da die Objektdaten für jeden Typ so unterschiedlich sind, sollten diese so allgemein wie möglich gehalten werden. Die Information $(Objekttyp, Klasse, Daten)$ ist dabei ausreichend und sehr flexibel. Der Objekttyp ist ein einfaches Atom, wie *circle* oder *rectangle*. Die *Klasse* ist eine Nummer, welche Objekte zu einer Klasse zusammenfasst. Objekte gleicher Klasse sind für die Wissensbasis Objekte von gleicher Form und Farbe.

Das Prädikat *obj* für ein Objekt o_x:

$$obj(db, o_x, type, class, data).$$

Locations Man muss einen Weg finden, Objektdaten sowie die mehrdeutigen Objektpositionen in der Datenbank zu speichern. Mehrdeutig deshalb, da ein Objekt sich an mehreren Positionen befinden kann. Natürlich ist dies in der Realität nicht der Fall, denn dort befindet es sich genau an einer Position. Hiermit sind die vermuteten Positionen gemeint. Beispielsweise könnte sich Objekt A an Position l_B, l_C oder l_D befinden (Abbildung 4.6).

Existiert zu einem Objekt nur eine Location, so ist seine Position eindeutig. Existiert dagegen keine Location, so ist das Objekt eine „Leiche“ in der Datenbank, da die Wissensbasis keine Referenz mehr auf das Objekt besitzt. Dies sollte nach Möglichkeit nicht eintreten und kann eigentlich nur passieren, wenn einer der Randbedingungen verletzt wurde, z.B. ein Objekt plötzlich verschwindet.



a) b)

Abbildung 4.6: Reale Position des Objektes A (a) und mehrdeutig vermutete Positionen (Locations) von der Wissensbasis (b)

Also eine Location l stellt eine Referenz auf ein Objekt o dar und beinhaltet eine mögliche Position. Eine Location kann allerdings auf mehrere Objekte verweisen, da auch eine Position nicht eindeutig sein muss. Dies tritt dann auf, wenn mehrere

Objekte der gleichen Klassen miteinander verwechselt werden, also sie nicht mehr eindeutig voneinander unterschieden werden können. Es ist am ER-Modell zu erkennen, dass es sich zwischen Objekten und Locations um eine N zu M Beziehung handelt.

Das Prädikat *loc* für eine Location l_x:

$$loc(db, l_x, l_{parent}, [x, y], t)$$

Das Prädikat *obj2loc* stellt die Verknüpfung zwischen Objekten und Locations her:

$$obj2loc(db, o_x, l_x).$$

Eine Location existiert nicht für jede mögliche Position eines Objekte. Es gibt zwei Voraussetzungen, damit eine neue Location angelegt wird:

1. Für jedes Objekt, was sichtbar ist, existiert eine Location. Dort ist die Position eindeutig und zwar der Schwerpunkt des Objektes. Genauer gesagt, wenn 8 Objekte zum aktuellen Zeitpunkt sichtbar sind, existieren mindestens 8 Locations in der Datenbank. Es wird markiert das sie sichtbar sind, indem keine LocationID l_{parent} angegeben wird (bzw. einen spezielle ID).

2. Es wird vermutet, dass ein Objekt unter einem anderen verschwunden ist (auch bei verdeckten Bewegungen). Dann wird eine neue Location angelegt, mit der möglichen Eintrittsposition (an welcher Position das Objekt verschwunden ist), sowie der Zeitpunkt, an dem es verschwunden ist und der Eltern-Location, unter der er verschwunden ist,

Damit sind die notwendigen Aussagen möglich, ob ein Objekt sichtbar ist, und wenn nicht, unter welchen Objekten es sich befinden könnte. Mehrdeutigkeiten werden somit auch berücksichtigt.

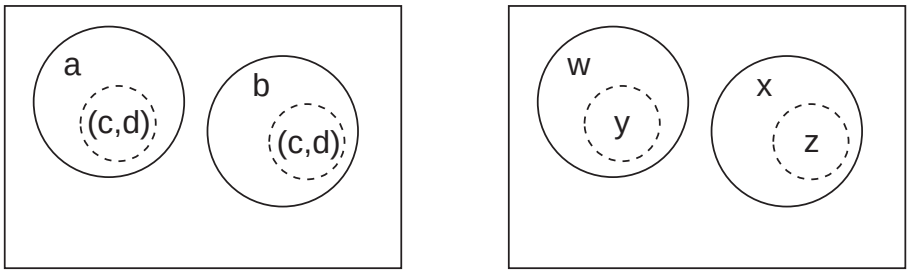
Relationen Relationen zwischen Objekten bzw. Locations werden einfach in der Form $rel(DB, Typ, l_a, l_b, R)$ repräsentiert. Relationen werden zwischen Locations aufgestellt. Der Typ kann entweder *rcc5* oder *depth* sein, welches die Art der Relation bestimmt. Unter Verwendung dieser Prädikate, wird eine Constraint-Graph erstellt, aus dem weiter Schlüsse gezogen werden können und für die Algorithmen der Wissensbasis von Bedeutung sind.

Das sind im wesentlichen die Bausteine der Datenbank. Weitere Prädikate zur Konfiguration der Datenbanken und um temporäre Daten zu halten, sind noch notwendig.

Beispieldatenbank Anhand eines kleinen Beispiels sollen nochmals die Elemente der Datenbank erläutert werden. Die Abbildung 4.7 stellt die Situation da, die in unserer Beispieldatenbank gespeichert ist.

Es existieren 4 Objekte, wobei zwei davon sichtbar sind (Objekt a und b) und zwei verdeckt (Objekt c und d). Die Wissensbasis kann nicht genau sagen, unter welchen Objekten sich die verdeckten Objekte befinden, da die Objekte c und d schon vorher miteinander verwechselt worden sind (sie sind von der selben Klasse). Sie weiß nur, dass sich jeweils ein nicht sichtbares Objekt unter einem sichtbaren befindet.

In der Abbildung 4.7 sieht man auf der linken Seite die Situation, wie sie hier beschrieben wurde. Auf der rechten Seite ist die Szene mit den Locations dargestellt, wie sie die Wissensbasis im Verlaufe des Geschehens angelegt hat. Dabei verweist Location w eindeutig auf das Objekt a und entspricht damit dem Objekt a. Die Location x verweist eindeutig auf Objekt b und entspricht damit diesem; $w \mapsto a, x \mapsto b$. Die Locations y und z sind mehrdeutig, d.h. sie verweisen auf mehr als ein Objekt. Beide verweisen auf die Objekte c und d; $y \mapsto \{c, d\}, z \mapsto \{c, d\}$.



a) b)

Abbildung 4.7: Darstellung der vermuteten Objekte (a) und die interne Repräsentation der Locations (b)

Die Beziehungen der Locations zu den Objekten zeigt der Überblick in Abbildung 4.8. In der linken Abbildung ist dargestellt, dass die Locations y und z auf jeweils zwei Objekte verweisen. Die Verweise der Locations auf andere Locations bedeutet, dass diese Location sich unter einer anderen befindet. Daraus entsteht eine Baumstruktur. Jede Location hat maximal einen Verweis auf eine andere. Existiert kein Verweis, ist die Location sichtbar, also befindet sich nicht unter einem anderen Objekt.

Die Verdeckungs-Hierarchie der Objekte ist in Abbildung 4.9 illustriert. Sie wurde erstellt durch verfolgen der Verweise der Locations auf die Objekte und veranschaulicht somit die möglichen Positionen der Objekte, welches die Lösung des Problems ergibt. Also auf eine Anfrage, wo sich ein Objekt zur aktuellen Situation befinden kann, ist das die Lösung.

Nun zur Erläuterung der Datenstruktur der Beispieldatenbank zur aktuellen Si-

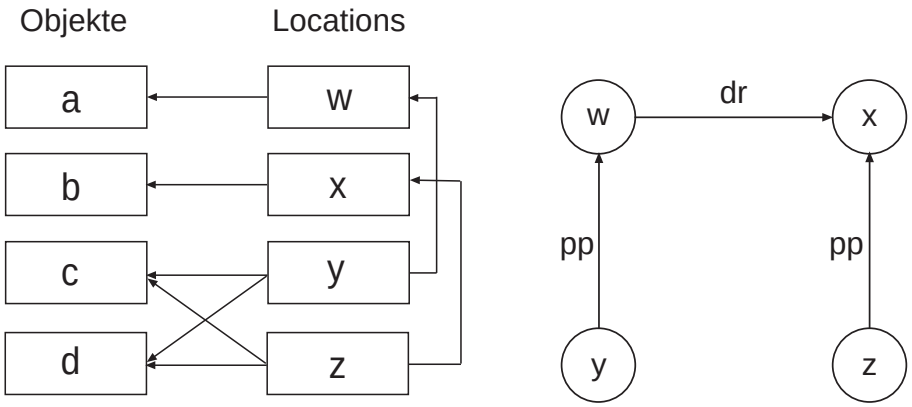


Abbildung 4.8: Beziehungen zwischen Locations und Objekte, sowie die Baumstruktur der Location (links) und das (unvollständige) Constraint-Netz der RCC5-Relationen (rechts)

tuation der Szene. Angemerkt sei, dass die Relationen weggelassen wurde, da sie hier nicht von Bedeutung ist.

Die Datenstruktur sieht folgendermaßen aus:

```
% bekannte Objekte
obj(db,a,circle,1,d_a)      % Objekt a
obj(db,b,circle,2,d_b)      % Objekt b
obj(db,c,circle,3,d_c)      % Objekt c
obj(db,d,circle,3,d_d)      % Objekt d (gleiche Klasse wie c)

% bekannte Locations
loc(db,w,∅,p_w,t_w)      % Location w ist sichtbar
loc(db,x,∅,p_x,t_x)      % Location x ist sichtbar
loc(db,y,w,p_y,t_y)      % Location y wird von w verdeckt
loc(db,z,x,p_z,t_z)      % Location z wird von x verdeckt

% Verknüpfungen
obj2loc(db,a,w)          % Objekt a befindet sich an Position w
obj2loc(db,b,x)          % Objekt b befindet sich an Position x
```

```
obj2loc(db,c,y)        % Objekt c könnte sich an Position y befinden
obj2loc(db,c,z)        % oder auch an Position z
obj2loc(db,d,y)        % Objekt d könnte sich an Position y befinden
obj2loc(db,d,z)        % oder auch an Position z
```

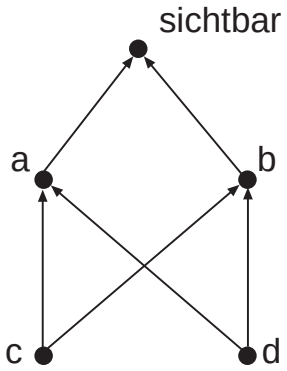


Abbildung 4.9: Hierarchie der Objekte in der Beispielszene. Die Objekte a und b sind sichtbar, die Objekte c und d befinden sich entweder unter Objekt a oder Objekt b.

4.3.3 Ablauf

Ein wichtiger Teil ist der Überblick über den Ablauf, wie die Daten in der Wissensbasis aufgenommen und neuberechnet werden. Diese Rechenschritte erfolgen immer, nachdem die Bildverarbeitung alle neuen Objektdaten aus dem aufgenommenen Bild an die Wissensbasis geliefert hat. Aus den bekanntem Wissen in der Wissensbasis und dem neuen Wissen, den die Bildverarbeitung liefert, wird der neue Zustand der Wissensbasis berechnet (Abbildung 4.10).

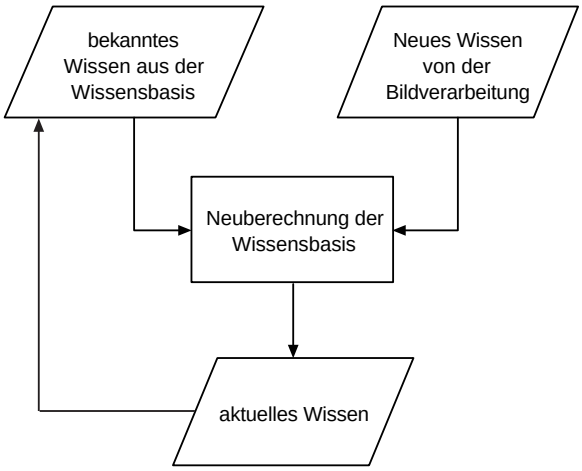


Abbildung 4.10: Überblick über den Ablauf

Diese Berechnung gliedert sich in mehrere Hauptteile, die in folgender Reihenfolge abgearbeitet werden:

1. Das Berechnen der möglichen Bewegungen, unabhängig von dem neuen Wissen.
2. Die anonymen Objekte, die die Bildverarbeitung liefert, den der bekannten zuweisen oder das Objekt neu in die Datenbank aufnehmen.
3. Vermutete Daten mit den reallen Daten abgleichen.
4. Die Datenbank der Wissensbasis auf Konsistenz prüfen, dabei werden alle inkonsistente Daten entfernen.
5. Die RCC und Tiefenrelationen erstellen.

Es gibt einige Probleme, die dabei gelöst werden müssen und für komplexere Objektformen, als dem Kreis, teilweise sehr schwer zu lösen sind.

Einige der Probleme sind:

- Kann Objekt A vollständig von Objekt B verdeckt werden ?
- Kann Objekt A nach einer Zeit Δt, sich unter Objekt B befinden ?
- Um welches bekanntes Objekt kann es sich bei dem Objekt A handeln, das von der Bildverarbeitung ermittelt wurde?
- In welcher Relation stehen die Objekte zueinander, und wie lässt sich diese bestimmen ?

4.3.4 Vermuten der Objektbewegung

Sichtbare, also von der Kamera wahrgenommene Objektbewegungen, müssen nicht vermutet werden, da alle sichtbaren Positionen eindeutig sind. Genauer gesagt, es gibt nicht mehr sichtbare Positionen von Objekten, als die, die von der Kamera ermittelt wurden. Das die Zuweisung zu einem bekannten Objekt nicht eindeutig ist, ist eine andere Frage.

Es geht in diesem Abschnitt also um die Vermutung, an welche sich nicht sichtbare Position sich ein Objekt bewegt haben kann. Ob dieses Objekt dabei vorher sichtbar war oder nicht, spielt keine Rolle. Einfacher ausgedrückt, es müssen die Locations neu berechnet werden.

Je größer der Zeitabstand zwischen den aufgenommen Bildern ist, umso mehr mögliche verdeckte Positionen (Locations) können entstehen. Deshalb arbeitet das System umso besser, je höher die Framerate der Bildverarbeitung und die Abarbeitungsgeschwindigkeit der Wissensbasis ist.

Verbindungsgraph

Der Wissensbasis sind zum aktuellen Zeitpunkt eine Anzahl Objekte mit deren Attributen bekannt. Um weiter Schlussfolgerungen ziehen zu können, wird ein Verbindungsgraph benötigt. Knoten des Graphen sind die Objekte. Wenn eine Verbindung im Graph zwischen 2 Objekten besteht, bedeutet es, dass diese Objekte auch in der Projektion miteinander verbunden sind oder sich überlappen. Dieser Graph ist normalerweise ungerichtet.

Unter Verwendung der RCC5-Relationen kann man leicht den Verbindungsgraphen herstellen. Wenn die Relationen nicht *dr* ist, besteht eine Verbindung zwischen den beiden Objekten. Die Relation ist eindeutig, da sie direkt aus dem rekonstruiertem Bild gewonnen wird.

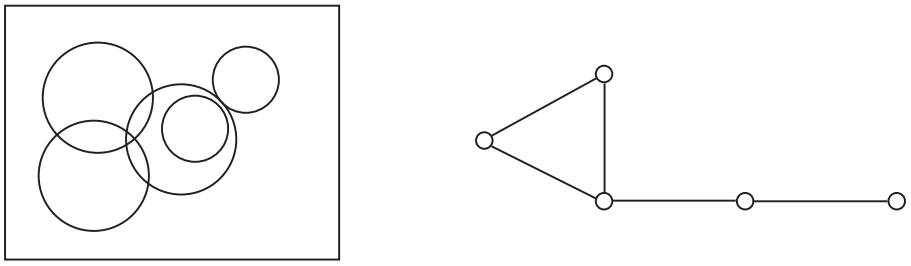


Abbildung 4.11: Beispiel eines Verbindungsgraphen

Kann Objekt x vollständig von Objekt y verdeckt werden?

Die Frage kann leicht beantwortet werden. Bei Kreisen werden einfach die Radien verglichen. Wenn der $r_x \leq r_y$ zutrifft, kann Objekt x von Objekt y verdeckt werden. Bei komplexeren Objekten ist dies weitaus schwieriger zu bestimmen.

Es ist hier die Frage nach der Ausdehnung bzw. Fläche der Objekte und nicht nach der Anordnung in der Tiefe. Dieses wird im Verlauf des Kapitels noch besprochen.

Finden des Weges von Objekt x nach Objekt y

Es wird hier ein Algorithmus vorgestellt, der dieses Problem zu lösen versucht. Abbildung 4.12 veranschaulicht das Problem. Es soll also die Frage geklärt werden: *Kann Objekt x bis zu Objekt y vordringen, wenn es sich innerhalb der verbundenen Objekte bewegt?* Die Aussage, welches Objekt mit welchem verbunden ist, hat man durch den Verbindungsgraph gegeben. Logischerweise kann ein Objekt nur zu diesen Objekten vordringen, zu dem auch ein Pfad im Verbindungsgraph besteht.

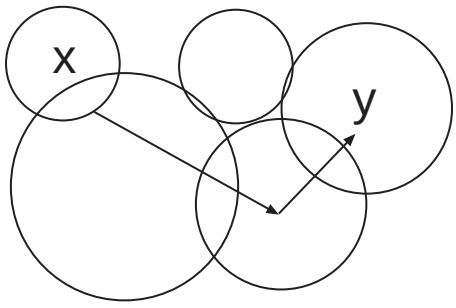


Abbildung 4.12: Veranschaulichung des Pfades von x nach y

Die Verbindungen zwischen 2 Knoten im Verbindungsgraphen werden hier als *Übergänge* bezeichnet. Dieser Verbindungsgraph ist damit ein gewichteter ungerichte-

ter Graph. Die Gewichtung, die angetragen wird, entspricht der Länge der Übergangslinie (siehe Abbildung 4.13). Diese Übergangslinie berechnet sich durch geometrische Operationen aus den 2 Kreisen, die sich überschneiden.

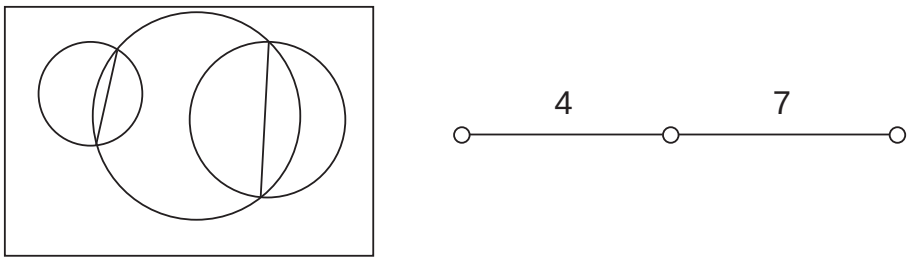


Abbildung 4.13: Übergangslinien als Gewichtung am Verbindungsgraph

Um nun zu ermitteln ob ein Objekt x bis zu Objekt y vordringen kann, ist folgendermaßen vorzugehen:

1. Basis ist der Durchmesser des Objektes x, d_x.
2. Nun wird der Graph unter Verwendung z.B. einer Breitensuche durchsucht und jeweils die Gewichtung g_{ab} mit dem Durchmesser d_x verglichen.
3. Wenn $d_x \leq g_{ab}$, dann wird der Pfad AB in der Suche freigegeben, ansonsten blockiert.
4. Wenn nun ein Pfad von Objekt x nach Objekt y gefunden wurde, kann geschlussfolgert werden, dass das Objekt x nach Objekt y vordringen kann und sich dabei innerhalb der Kreise bewegt.

Gruppieren von Objekten Nun kann es den Fall geben, dass der Pfad nicht gefunden wird. Es wurde bisher davon ausgegangen, das sich ein Objekt zu einem bestimmten Zeitpunkt immer nur unter ein anderes Objekt bewegen kann. Dies ist allerdings nicht immer der Fall. Ein Objekt kann sich auch unter mehrere Objekte bewegen und befindet sich dabei immer noch innerhalb der Kreise (Abbildung 4.14). Genauer gesagt, bilden Gruppen aus kleineren Objekten eine zusammenhängende Einheit, die ein Objekt, welches eigentlich zu groß für jedes einzelne Objekt wäre, passieren lassen kann.

Um dieses Problem in der Lösungsmenge mit zu erfassen, wird der Algorithmus um den Sachverhalt erweitert, dass Objekte zu Gruppen zusammengefasst werden. Also jeder Knoten in dem Graphen kann nun ein einzelnes Objekt oder eine Gruppe von Objekten sein. Die Übergänge zwischen den Objekten werden dann auch geometrisch

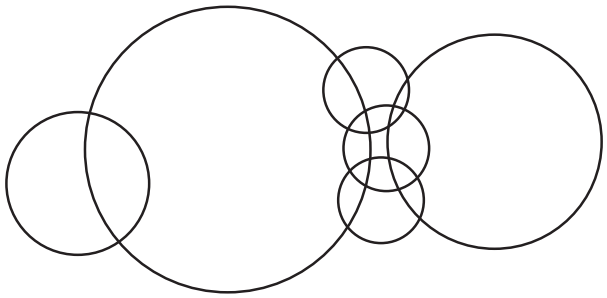


Abbildung 4.14: Durch Gruppierung von Objekten kann trotzdem ein Übergang geschaffen werden

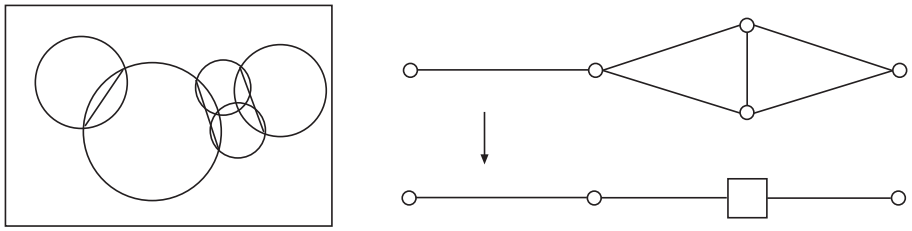


Abbildung 4.15: Zusammenfassen der Objekte zu Gruppen und deren Übergänge

zusammengefasst, um die Gewichtungen für die Verbindungen bei Objektgruppen zu berechnen (Abbildung 4.15).

Als Ausgangspunkt nehmen wir wieder den Graphen wie im original Algorithmus. Allerdings wird dieser zuerst so optimiert, dass wirklich nur Knoten im Graphen vorhanden sind, die auf einem der Pfade zum Zielobjekt liegen. Alle anderen spielen keine Rolle und können weggelassen werden. Nun bauen wir aus diesem einen zweiten Graphen auf, wo versucht wird, Objekte zu gruppieren. Objekte können zu einer Gruppe zusammengefasst werden, wenn diese im Graphen ein Dreieck bilden. Auch Objektgruppen können wiederum zusammengefasst werden (Abbildung 4.16). Der Algorithmus zum finden des Pfades ändert sich nicht. Komplizierter allerdings wird die Bestimmung der Gewichtungen(Übergangslänge) der Verbindungen. Hierbei müssen die beiden äußeren Schnittpunkte der jeweiligen Objekte/Gruppen bestimmt werden, welche die Punkte des Vektors darstellen. Die gesuchte Gewichtung ist folglich die Länge des Vektors.

Bei der Gruppierung von Kreisen kann ein Loch zwischen drei Kreisen entstehen, wo das Objekt theoretisch durchscheinen kann und somit eine Gruppierung eigentlich nicht möglich ist, da es keine zusammenhängende Fläche ergibt (Abbildung 4.17). In der Praxis tritt das allerdings sehr selten auf, und die Größe des Loches beträgt in den

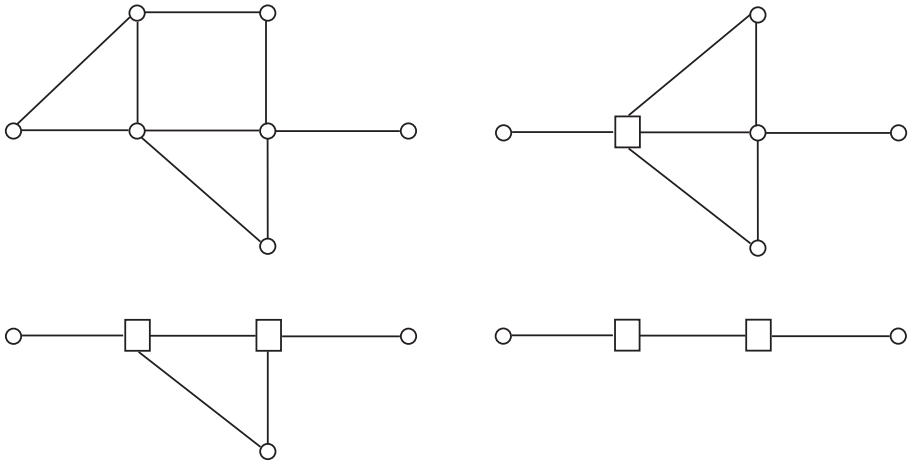


Abbildung 4.16: Zusammenfassen der Objekte in Gruppen

allermeisten Fällen nur einige Pixel, sodass diese Ungenauigkeit toleriert werden kann. Diese Ungenauigkeit ist zugunsten des Systems. D.h. die Anzahl der Möglichkeiten, wo sich ein Objekt befinden kann, erhöht sich.

Man kann dieses Problem beseitigen, indem die Größe dieses Loches geometrisch oder durch Auszählen der Pixel im Binärbild ermittelt wird. Wenn die Fläche dieses Loches mindestens ein Pixel beträgt, können diese Kreise somit nicht gruppiert werden.

Aber im Rahmen dieser Diplomarbeit wurde diese kleine Ungenauigkeit vernächlässigt, da es in den seltensten Fällen eine Rolle spielt.

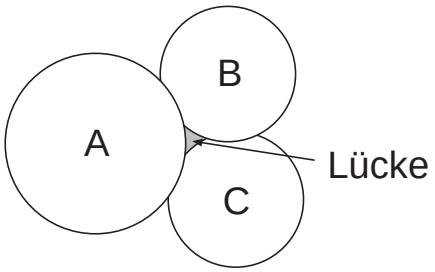


Abbildung 4.17: Das Lochproblem bei Kreisgruppen

Bestimmen des kürzesten Pfades

Eine gute Annäherung und die fürs erste einfachste Möglichkeit ist die Bestimmung aller möglichen Pfade und die Summierung der Distanzen der Objekte auf dem jeweiligen Pfad. Der Pfad mit der kleinsten Summe ist der kürzeste.

Eine genauere Bestimmung der Pfadlängen ist sogar für Kreise nicht einfach und erfordert Vektorrechnung mit Fallunterscheidung. Es ist anzunehmen, dass für komplexere Objekte der Aufwand für die genaue Bestimmung des kürzesten Pfades im exponentiellen Bereich liegt und deshalb sollte mit Näherungen gerechnet werden.

Die Pfadlänge ist wichtig um die minimale Zeit zu bestimmen, in der sich das Objekt zu diesem Punkt hin bewegen kann.

Anordnung der Objekte in der Tiefe

Hierbei wird auf den Abschnitt zur Tiefenrelation verwiesen. Diese wird dazu verwendet, um festzustellen in welcher Relation in Bezug auf die Tiefenanordnung die Objekte zueinander stehen.

Der Algorithmus zum Finden der Wege von x nach y wird dahingehend erweitert, dass er berücksichtigt, dass bei einem Übergang das Objekt *über* dem Objekt liegt oder die Relation unbekannt ist, also die Relation *fr* in der Relationsmenge vorhanden ist. Ansonsten wird die Suche blockiert und ein anderer Pfad muss gefunden werden. Grund hierfür ist, dass sich Objekte unbemerkt vom Beobachter natürlich nur unter anderen Objekten bewegen können.

4.3.5 Zuweisung von Objekten

Die Objektdaten, die die Wissensbasis von der Bildverarbeitung bekommt, sind anonyme Daten. Damit ist gemeint, dass die Objekte nicht identifiziert sind. Die Bildverarbeitung liefert nur die Objektklasse zu jedem Objekt mit. Objekte derselben Klasse sind gleiche Objekte für die Wissensbasis.

Einer der Hauptaufgaben der Wissensbasis besteht darin, diese Objekte zu identifizieren. Sie muss aus dem bekannten Wissen W und dem Wissen aus dem aktuellen Bild W_{frame}, neues Wissen W_{neu} bilden.

$$W_{neu} = W + W_{frame}$$

In dem Fall der Zuweisung der Objekte: aus den bekannten Objekten und den nicht identifizierten Objekten muss die Datenbank der bekannten Objekte geändert bzw. erweitert werden. Dies erfolgt entweder durch Zuweisung der unbekannten Objekte zu schon bekannten oder, wenn dies nicht möglich ist, wird es als neues Objekt in die Datenbank aufgenommen.

Es existiert also eine Menge von bekannten Objekten OA in der Wissensbasis, welche aus den vorhergehenden Frames erstellt wurde und eine Menge von sichtbaren

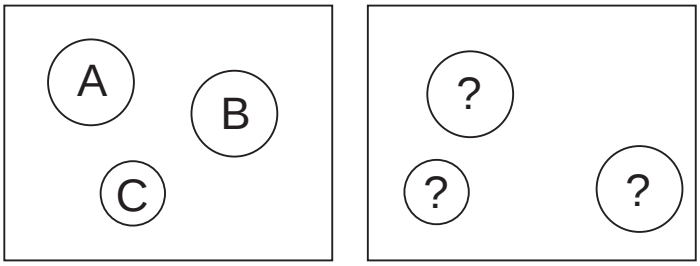


Abbildung 4.18: Unbekannte Objekte müssen aus bekannten Daten der Wissensbasis (a) zugewiesen werden (b)

Objekten OV, die uns die Bildverarbeitung liefert und die zu den bekannten Objekten zugewiesen werden müssen.

Die Zuweisung erfolgt durch schrittweises Entfernen der Möglichkeiten. Betrachten wir zunächst ein einzelnes Objekt a aus der Menge OV. Zuerst wird angenommen, dass jedes bekannte Objekt aus der Menge OA das Objekt a sein kann, insofern es die gleiche Klasse hat[2]. Diese Menge bezeichnen wir als O_a. Nun wird diese Menge O_a weiter eingeschränkt, indem jedes Element von O_a daraufhin geprüft wird, ob es nach den Randbedingungen mit dem Objekt a identifiziert werden kann. Alle Objekte, bei denen die Randbedingungen noch übereinstimmen, wenn sie mit dem zu identifizierende Objekt a gleichgesetzt werden, werden in die neue Menge Z_a aufgenommen. Diese Menge stellt die möglichen Zuweisungen für das Objekt a da.

Ist die Ergebnis-Menge Z_a leer, so handelt es sich um ein neues Objekt, da kein bekanntes zu diesem zugewiesen werden konnte. Ist genau ein Element vorhanden, so ist das Objekt eindeutig zu einem bekannten zuweisbar. Ist mehr als ein Element in der Menge Z_a vorhanden, so ist die Zuweisung mehrdeutig. Dies kann z.B. bei Verwechslungen passieren.

Das Ziel ist es, so wenig wie möglich Elemente in Z_n zu haben. Günstigerweise genau ein Element oder die Leere Menge. Um die Größe der Mengen weiter einzuschränken, werden für alle Objekte der Menge OV die Mengen Z_n abgeglichen. Dabei wird der Fakt ausgenutzt: Wenn ein Objekt eindeutig zuzuordnen ist, kann es aus den anderen mehrdeutigen Zuweisungen ausgeschlossen werden.

Beispiel 4.1: Es ist die Objektmenge $OA : \{a, b\}$ der Wissensbasis bekannt und beide Objekte haben die gleiche Klasse. Die Bildverarbeitung liefert die Objektmenge

[2]Objekt fremder Klassen können logischerweise nicht einander zugewiesen werden, da sie direkt unterscheidbar sind.

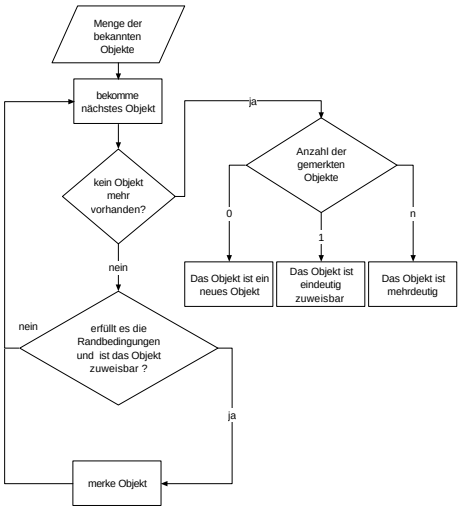


Abbildung 4.19: Algorithmus zur Zuweisung von Objekten

$OV : \{x, y, z\}$. Nach dem Zuweisungsalgorithmus ergeben sich die Zuweisungsmengen $Z_x : \{a\}, Z_y : \{a, b\}, Z_z : \emptyset$. Da das Objekt z eindeutig zu dem Objekt a zuweisbar ist, kann das Objekt a aus den anderen Mengen entfernt werden, sodass die endgültigen Zuweisungen entstehen: $Z_x : \{a\}, Z_y : \{b\}, Z_z : \emptyset$. Objekt x ist damit eindeutig Objekt a, Objekt y ist Objekt b und Objekt z ist ein neues Objekt.

Nun kann es passieren, dass mehrere Objekte zu dem gleichen bekannten Objekt eindeutig zugewiesen werden können. Beispiel: $Z_x : \{a\}, Z_y : \{a\}$. Dieser Fall kann nur auftreten, wenn die Randbedingungen verletzt wurden oder ein neu aufgetauchtes Objekt der gleichen Klasse mit einem schon vorhandenen Objekt verwechselt wurde. Die Lösung ist relativ einfach, eines der beiden Objekte wird als neues Objekt in die Datenbank aufgenommen. Um dies zu erreichen, wird wahllos eine Zuweisung erhalten und für die restlichen Zuweisungsmengen, die leere Menge $\emptyset$ angegeben: $Z_x : \{a\}, Z_y : \emptyset$.

Allerdings ist es nicht notwendig, sich explizit darum zu kümmern, da der vorhergehende Schritt, der die eindeutigen Zuweisungen entfernt, als Nebeneffekt dies schon erledigt. Die eindeutigen Zuweisungen aus den anderen Mengen werden entfernt und damit entsteht bei mehrfachem Vorkommen der selben eindeutigen Zuweisung die Leere Menge.

Da es mehrdeutige Positionen für Objekte gibt, ist abschließend anzumerken, dass Objekte nicht direkt zuzuweisen sind, sondern die Objekte werden zu den schon angesprochen *Locations* zugewiesen. Es werden also die sichtbaren Objekte, die die Bild-

verarbeitung liefert, zu bekannten *Locations* zugeordnet. Jedes bekannte Objekt kann mehrere dieser Locations besitzen, da es sich an verschiedenen Positionen befinden kann (Mehrdeutigkeiten). Eine Location kann wiederum auf mehrere Objekte (der selben Klasse) verweisen, aber nur sie besitzt die Positionsdaten.

4.3.6 Bilden der Relationen

Wenn der neue bekannte Zustand der Wissensbasis ermittelt wurde, werden die Relationen zwischen den Objekten neu gebildet. Die notwendigen Relationen sind RCC5 und Tiefenrelationen.

Zuerst werden die vorhanden Relationen gelöscht, dann die Tiefenrelationen erstellt und zuletzt die RCC5-Relationen, da dazu die Tiefenrelationen benötigt werden, um zwischen pp und pp^i zu unterscheiden.

Tiefenrelation Die Information, welches Objekt vor welchem liegt, muss von der Bildverarbeitung vorgegeben werden. Es kann nicht aus den rekonstruierten Objekten ermittelt werden, wenn keine Bilddaten mit übertragen werden.

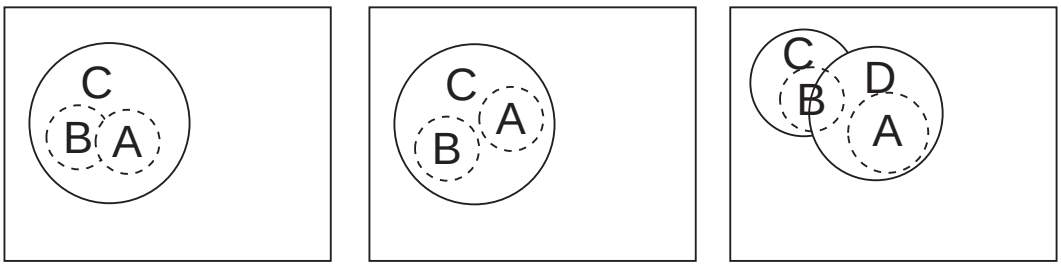
RCC5-Relation Durch geometrisches Vergleichen der einzelnen Locations zueinander, kann die RCC5-Relation gebildet werden. Es ist einfach zu prüfen, ob der Kreis A innerhalb, außerhalb oder auf dem Rand von Kreis B liegt. Wenn er innerhalb liegt, ist noch zu prüfen ob der Kreis A vor Kreis B liegt, um zwischen pp und pp^i zu unterscheiden.

Nachdem die bekannten Relationen gebildet wurden, wird zum Schluss der Constraint-Propagation Algorithmus für beide Relationstypen angewendet, um die Relationsmengen einzuschränken.

Zu beachten ist, dass verdeckte Locations, nicht eindeutige Relationen zueinander haben. Wenn die Objekte Objekt A und B sich unter Objekt C befinden , also dort eine Location existiert, und Objekt A unter Objekt B passt, aber Objekt B nicht unter Objekt A, dann sieht die RCC5-Relation zwischen A und B folgendermaßen aus: $A\{dr, po, pp^i\}B$. Die Basisrelation pp kann ausgeschlossen werden, da Objekt B nicht unter Objekt A passt. Man kann zusätzlich noch dr ausschließen, allerdings ist die Überprüfung dafür sehr kompliziert.

Das Problem, bei dem Ausschließen von dr ist, dass die Objekte, die sich über Objekt A und B befinden, mindesten die Ausdehnung von A+B haben müessen, was aber nicht einfach zu bestimmen ist und sollte daher nicht mit einfließen. Der Fehler ist immer zugunsten des Systems, wenn dr in der Relationsmenge mit vorhanden ist.

Außerdem beeinflusst es derzeit nicht die Lösungsmenge. Die ganze Szene ist nochmal in Abbildung 4.20 veranschaulicht.



a) b) c)

Abbildung 4.20: Problem bei der Bestimmung von RCC-Relationen zwischen verdeckten Objekten: Wie verdeckte Objekte sich gegenseitig überlappen, kann nicht eindeutig festgestellt werden. Im Bild (a) ist $A\{po\}B$ und im Bild (b) $A\{dr\}B$. Das Problem beim Ausschließen von dr (c) (siehe Text).

4.3.7 Anfragen

Die Anfragen gestalten sich sehr einfach. Es werden dazu die zuvor erklärten Datenstrukturen ausgewertet.

Welche Objekte sind der Wissensbasis bekannt?

Die Fakten der Objekte *obj* geben dies wieder. Jeder Datensatz entspricht einem bekannten Objekt.

Welche Daten gehören zu dem Objekt ?

Dazu werden die Objektfakten *obj* ausgewertet, wo die Objektdaten zu einer ObjektID angelegt sind.

Wo könnte sich das Objekt befinden ?

In der Location *loc* ist vermerkt, ob sie sichtbar ist oder von einer anderen Location verdeckt wird. Ist sie sichtbar, so kann direkt die Position ausgelesen werden, die in der Location gespeichert ist. Ist die Location verdeckt, so ist auch das Objekt verdeckt und befindet sich unter dieser Location.

Um an die entsprechenden Locations zu gelangen existiert die Verknüpfung *obj2loc*. Hier ist vermerkt, welche Locations zu einem Objekt zugeordnet sind. Sie kann auch benutzt werden, um herauszufinden welche Objekte zu einer Location zugeordnet sind und damit kann ermittelt werden, unter welche Objekten sich das gesuchte Objekt befinden könnte.

Wie die Daten genau aufgebaut sind, wie sie in dem Prototypen umgesetzt worden sind, ist dem Kapitel Umsetzung zu entnehmen.

4.4 Zusammenfassung

Wie man sieht, ist das Problem für *n Kreise* lösbar. Es hat sich aber auch gezeigt, dass eine 100%ig exakte Berechnung schwer umzusetzten ist (wenn man die Vorteile der einfachen geometrischen Form des Kreises beibehalten möchte). Deshalb wurde auf Vereinfachungen zurückgegriffen, die Ungenauigkeiten zur Folge haben, die aber immer zugunsten des Systems gehen. Durch Ungenauigkeiten werden keine möglichen Position ausgelassen, sondern die Optimalität des Systems leidet darunter. Die Lösungsmenge ist u.U. Größer als eigentlich möglich wäre.

Die Ungenauigkeiten zusammengefasst:

- Zusammenfassen von Kreisen (gruppieren):
  - Das resultierende Objekt ist zu Groß.
  - Kreise werden zusammengefasst, die eigentlich nicht zusammengefasst werden können. (Lochproblem)
- Berechnung des kürzesten Pfades ist nicht exakt, da die Berechnungszeiten dafür zu hoch wären.
- Das Ausschließen von dr, bei der Erstellung der RCC-Relationen zwischen verdeckten Objekten, wird nicht vorgenommen.

Kapitel 5

Umsetzung

5.1 Einführung

Aufgabe dieser Diplomarbeit war es nicht nur, das Problem theoretisch zu lösen, sondern auch einen Prototypen zu implementieren. Dieser Prototyp dient als Hilfe, um die theoretischen Algorithmen zu testen und die Richtigkeit zu zeigen. Er soll auch als Basissystem für weitere Entwicklungen dienen. Deshalb sollte der Prototyp erweiterbar und relativ leicht portierbar sein.

Mit Erweiterbarkeit ist gemeint, dass das System modular aufgebaut ist, d.h. es wird in einzelne Module aufgeteilt, welche ausgetauscht und erweitert werden können. Das heißt aber auch, dass die interne Struktur der Wissensbasis schon in Voraussicht auf komplexere Objektformen gewählt werden sollte. Dabei ist natürlich nicht genau absehbar, wie sich das System weiter entwickeln wird.

Portierbarkeit heißt, dass das System mit relativ wenig Aufwand auf verschiedenen Rechnersystemen lauffähig ist. Es soll nicht nur unter Windows laufen, sondern z.B. auch auf einem Linux-Betriebsystem. Dies ist notwendig, da Teile des Systems evtl. auf einer Roboterplatine untergebracht werden sollen, welche ein minimales Linux als Betriebsystem hat.

Desweiteren ist es angebracht, Bildverarbeitung, Wissensbasis und die Nutzer-Applikation nicht in einem einzigen Programm unterzubringen, sondern eine getrennte Architektur zu wählen. Die drei Einzelanwendungen kommunizieren über fest definierte Schnittstellen miteinander. Der Abstraktionsgrad soll so hoch sein, dass das Kommunikationprotokoll, über den die Einzelanwendungen miteinander kommunizieren, austauschbar ist.

Wenn alle drei Anwendungen auf dem selben Computer laufen, reicht eine speichergekoppelte Kommunikation aus. Es kann auch eine Netzwerkverbindung zwischen den Anwendungen gewählt werden, insofern man das Protokoll implementiert. Im Abschnitt 5.3.2 wird darauf näher eingegangen.

In diesem Kapitel wird zuerst aufgeführt, welche Programmiersprachen und Bibliotheken verwendet wurden. Diese werden kurz erläutert, sodass die späteren Ausführungen leichter verständlich sind. Im Anschluss wird eine Übersicht über das Gesamtsystem gegeben, wie es implementiert wurde. Danach folgt der Hauptteil mit den Details der Implementierung. Es wird dabei jedes Modul intensiv besprochen. Einerseits die Verwendung der Module einschließlich der Schnittstellen, andererseits auch die Beschreibung, wie die Funktionen umgesetzt wurden.

5.2 Prolog, XPCE

Wesentliche Teile des Systems wurden mit der Programmiersprache Prolog entwickelt. Es hat sich angeboten, bei der Umsetzung der Wissensbasis die Vorteile von Prolog zu nutzen. Man hätte das ganze Programm natürlich auch komplett in C++ oder Java entwickeln können, aber dies wäre mit sehr viel mehr Aufwand verbunden.

Prolog ist eine sehr alte Programmiersprache. Die erste Version wurde schon 1973 erstellt. Prolog ist nicht wie die meisten Programmiersprachen prozedual oder objektorientiert, sondern eine Programmiersprache, die auf logischen Formeln basiert[1]. Prolog wird immer noch weiterentwickelt und es gibt Standards (ISO), die verschiedene Prologsysteme erfüllen, und damit zueinander weitestgehend kompatibel sind. Aber der Kern des Systems ist im wesentlichen über die Jahre gleich geblieben.

Prolog basiert im Kern auf der Prädikatenlogik erster Stufe. Die Grundbausteine sind Terme und Aussagen. In Prolog existieren drei verschiedene Aussagen: Fakten, Regeln und Anfragen. Prolog kennt nur eine Datenstruktur, den logischen Term. In der beigefügten Bibliothek sind Funktionen vordefiniert, um z.B. mit Listen und Dateien zu arbeiten. Es ist üblich, in Prolog stark rekursiv zu programmieren, da es normalerweise nicht möglich (und notwendig) ist, Schleifen zu programmieren.

Es soll hier nun keine Einführung in die Programmiersprache Prolog stattfinden. Dies kann in entsprechender Fachliteratur viel besser nachgelesen werden [BK91]. Es wird nun kurz auf das verwendete SWI-Prolog eingegangen und der Oberflächen-Bibliothek XPCE für SWI-Prolog.

5.2.1 SWI-Prolog

SWI-Prolog wurde von Jan Wielemaker an der Universität von Amsterdam entwickelt und befindet sich zu dem Zeitpunkt der Diplomarbeit in der Version 5.0.9. Die Versionsnummer belegt, dass das Prologsystem schon seit einiger Zeit existiert und es

[1] Was aber nicht heißt, dass in Prolog nicht auch objektorientiert programmiert werden kann.

wird immer noch ständig weiterentwickelt. SWI-Prolog ist wohl eine der derzeit am häufigsten verwendeten Prologsysteme. Der Quellcode liegt offen (neuerdings LGPL) und kann frei verwendet werden. Das bedeutet auch, dass es möglich ist, es auf fast jedem Posix-kompatiblen System zu benutzen. Derzeit existieren offizielle Versionen für viele Unix-Betriebssysteme sowie MacOS und Windows. Im Rahmen dieser Diplomarbeit wurde nur die Version für Windows verwendet.

SWI-Prolog kann auf verschiedene Art und Weise genutzt werden. Erstens interaktiv, d.h. der Benutzer kommuniziert direkt mit dem Prologinterpreter. Das ist allerdings nicht sehr effektiv und nur zum Testen bzw. zur Fehlersuche sinnvoll. Desweiteren kann ein Prolog-Quellcode geladen werden, welcher interpretiert wird, also der Quellcode kann im internen oder externen Texteditor manipuliert werden. Es können auch externe Bibliotheken verwendet werden (foreign libraries), welche in einer anderen Programmiersprache geschrieben wurden. Als letzte Variante kann Prolog direkt in einem Programm eingebunden werden. Das bedeutet, dass die Prolog-Engine aus einer Anwendung heraus genutzt wird. Die Engine wird dabei statisch zu der Anwendung gelinkt und ist damit in dem Programm integriert (Embedded-Prolog), ohne Prolog zusätzlich starten zu müssen.

Um die Ausführungsgeschwindigkeit des Systems zu steigern, ist ein Compiler integriert. Und seit Version 3.3 erfüllt SWI-Prolog den ISO-Standard ([Cov93]).

Der Nachteil von SWI-Prolog ist der relativ hohe Speicherverbrauch (teilweise über 10MB bei der Windowsversion). Die Ausführgeschwindigkeit ist im Vergleich zu einigen kommerziellen Prologsystemen langsamer.

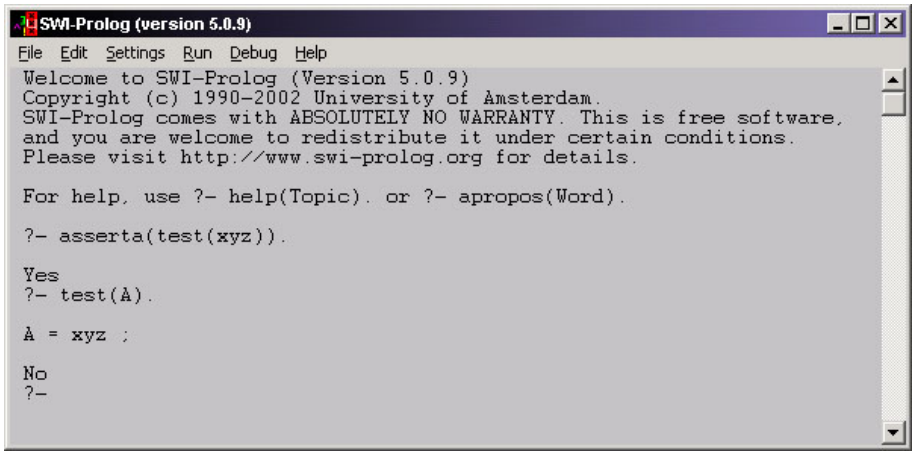


Abbildung 5.1: SWI-Prolog Windows-Version

Einbinden anderer Programmiersprachen in SWI-Prolog

Bei SWI-Prolog kann man aus Prolog heraus die Vorteile anderer Programmiersprachen nutzen. Dies geschieht mittels dynamischen Bibliotheken. In der MS-Windows-Welt heißen diese DLL's (Dynamic-Link-Libraries) und unter Unix-Betriebsystemen werden $*.so$ Dateien benutzt, welche ähnlich wie DLL's aufgebaut sind und auch so benutzt werden.

Die Vorteile, diese DLL's von Prolog aus zu nutzen, liegen klar auf der Hand. Es können Funktionen implementiert werden, die so in Prolog nicht möglich sind. Dies ist sinnvoll, wenn z.B. Betriebssystemfunktionen benutzt werden sollen, aber auch wenn Daten höher komprimiert gespeichert werden sollen als Prolog es kann oder die Ausführungsgeschwindigkeit von Prolog nicht ausreichend ist.

Beispielsweise kann zwar in Prolog ein Bild in einer Liste oder einem Array gespeichert werden, aber der Speicherverbrauch und die Zugriffsgeschwindigkeit wären nicht akzeptabel. Deshalb ist es angebracht, dieses Array und deren Zugriffsfunktionen in eine DLL auszulagern.

Auch besteht die Möglichkeit der bidirektionalen Kommunikation. Einmal kann man aus Prolog heraus DLL-Funktionen nutzen. Andererseits kann aus den DLL-Funktionen heraus auf Prolog-Regeln und Fakten zugriffen werden.

Im Weiteren wird insbesondere die Programmiersprache C bzw. C++ benutzt, um diese DLL-Funktionen zu erstellen. Es ist aber auch möglich, diese DLL's mit einer anderen Programmiersprache zu erstellen. Dies erfordert allerdings erheblich mehr Aufwand, da die SWI-Prolog-Headerdateien in die entsprechende Programmiersprache umgesetzt werden müssten. Daher ist es nicht zu empfehlen, eine andere Programmiersprache zu wählen.

Als Compiler wird VisualC++5 von Microsoft benutzt. Für Linux wird der mitgelieferte GCC genommen, am besten in der Version 2.95.x . Der Quellcode zwischen der Windows- und Linux-Version unterscheidet sich nicht, insofern keine compilerspezifischen Ausdrücke benutzt werden und sich an den C++ Standard gehalten wird.

Anhand eines kleinen Beispiels soll gezeigt werden, wie externe Prolog-Funktionen erstellt werden. Wie ein C-Compiler benutzt wird, als auch die C++ Programmiersprache soll hier nicht näher erläutert werden. Zum Thema C++ ist das Buch [Str98] zu empfehlen.

Beispiel 5.1: Es soll eine Funktion implementiert werden, die einfach 2 Zahlen addiert und das Ergebnis zurückgibt.

Der dazugehörige Prolog-Quellcode sieht so aus:

```
:-load_foreign_library(mylib).
:-addition(3,4,A),write('3+4='),writeln(A).
:-unload_foreign_library(mylib).
```

Als erstes wird unsere externe Bibliothek nachgeladen. Die neue Bibliotheks-Funktion *addition* wird aufgerufen und das Ergebnis ausgegeben. Zum Schluss wird die Bibliothek wieder entfernt.

Die Bibliothek, die erstellt wird, muss sich im gleichen Pfad wie der Prolog-Quellcode befinden oder direkt im Prolog-Programm-Verzeichnis, damit der Interpreter diese findet.

Die C-Headerdatei sieht folgendermaßen aus:

```
#ifndef     C_TEST_H
#define     C_TEST_H 1
#include    "SWI-Prolog.h"

#ifdef      __cplusplus
extern      "C"{
#endif

install_t     install();
install_t     uninstall();

#ifdef __cplusplus
}
#endif
#endif /* C_TEST_H */
```

Hier werden nur zwei Bibliotheksfunktionen definiert. Die Funktion *install()* wird vom Prologinterpreter aufgerufen, nachdem die Bibliothek geladen wurde und *uninstall()* wird aufgerufen, wenn die Bibliothek wieder entfernt wird. In der CPP-Datei werden nun die beiden Funktionen implementiert. In der Funktion *install()* werden alle Funktionen registriert, die später von Prolog aus benutzt werden können. Die *uninstall()* Funktion ist leer, da keine Aufräumaktionen stattfinden müssen.

Der Quellcode sieht folgendermaßen aus:

```
static foreign_t
pl_addition(term_t t_a, term_t t_b, term_t t_out)
{
     double a,b;
     if( PL_get_float(t_a, &a) && PL_get_float(t_b, &b) )
     {
          PL_unify_float(t_out, a+b);
          PL_succeed;
     }
     PL_failed;
}

install_t install()
{
     PL_register_foreign("addition", 3, pl_addition, 0);
}

install_t uninstall()
{}
```

Die Funktion *PL_register_foreign()* registriert eine C-Funktion und macht diese in Prolog sichtbar. Wichtig hierbei sind die ersten drei Parameter. Der erste ist der Name, unter welcher er in Prolog registriert wird. Dann folgt die Anzahl der Parameter und die C-Funktion selbst, die als Prolog-Regel fungieren soll. Wir haben hier die Regel *addition* definiert und zwar mit 3 Parametern.

Die zugeordnete C-Funktion ist *pl_addition()*. Wichtig ist es, den Funktionsrückgabetyp als *foreign_t* zu definieren und nicht einfach nur *BOOL* oder *int*, da im Makro *foreign_t* noch compilerspezifische Optionen benutzt werden, die u.U. wichtig sind, damit der Compiler weiß, wie er den lokalen Stack der Funktion zu handhaben hat.

Die 3 Funktionsparameter entsprechen den Parametern der Prolog-Regel. Es sind 3 Terme, die intern durch eine Zahl repräsentiert wird. Dies ist der Referenzindex des Terms auf dem lokalen Stack von Prolog. Zu beachten ist, dass die Terme nur solange existieren wie die Funktion. Danach werden sie zerstört. Die beiden Terme *t_a* und *t_b* müssen Zahlenwerte sein. Wenn der Typ des Terms nicht stimmt, wirft Prolog eine Exception. Und zwar bei dem Versuch die Werte mittels *PL_get_XXX()* auszulesen. Nachdem die Werte aus dem Term in *double*-Werte eingelesen wurden, wird das Ergebnis dem Term *t_out* zugewiesen, also unifiziert. Dies geschieht mit der Funktion *PL_unify_XXX()*. Ist der Term keine freie Variable, wirft Prolog wiederum eine Exception. Man kann den Typ des Terms auch mit den Funktionen *PL_is_XXX()* testen, also z.B. mit der Funktion *PL_is_variable(t_out)*.

Diesen C-Quellcode compiliert man mit der Option, eine DLL zu erstellen und linkt diesen mit den bei SWI-Prolog mitgelieferten Statischen-Link-Bibliotheken. Wie das genau geschieht, kann in der Dokumentation zu SWI-Prolog nachgelesen werden.

C++ Interface Wenn die Schnittstelle zu SWI-Prolog lieber objektorientiert sein soll, gibt es derzeit zwei Bibliotheken, die dies ermöglichen. Einmal das *SWI-cpp*, welches im SWI-Prolog-Paket enthalten ist, und das *swiprolog C++ 0.1.0*. Letzteres wurde entwickelt um das SWI-cpp in Sachen Objektorientierung und Funktionsvielfalt zu verbessern. Man kann es aber derzeit leider nicht mit VisualC++ verwenden, da es speziell für GCC 2.9.x erstellt wurde und deshalb beschränken wir uns auf das originale C++ Paket.

Das *SWI-cpp* ist ein Paket, welches den Zugriff auf die SWI-Funktionen mit Hilfe von Klassen regelt. Es ist allerdings nicht sehr gut umgesetzt worden (wie später noch gezeigt wird) und hat auch einige Einschränkungen. Trotzdem ist es sehr hilfreich und übersichtlicher mit diesen Klassen zu arbeiten.

Die Verwendung ist sehr einfach, da einfach die *SWP-cpp.h* Header-Datei eingebunden wird. Nun stehen alle Klassen zur Verfügung. Es braucht keine weitere Bibliothek hinzugelinkt werden, da alle Klassen-Methoden *inline* sind.

Unser Beispiel sieht mit der SWI-cpp wie folgt aus:

```
PREDICATE(addition, 3)
```

```
{
     double a = A1;
     double b = A2;
     A3 = (double) a+b;
     PL_succeed;
}

install_t install()
{}

install_t uninstall()
{}
```

Damit ist der Quellcode um einiges kürzer. Auch müssen in *install()* die Funktionen nicht mehr registriert werden. Das erledigt das Makro *PREDICATE*. Erster Parameter ist der Name und zweiter die Anzahl der Parameter (arity). Nachteilig ist, dass die Anzahl der Parameter auf 10 beschränkt ist, da nur A1-A10 definiert wurden, welche Term-Objekte darstellen und der Klasse *PlTerm* angehören. Dabei sind die Operatoren für diverse Typen-Casts und der Zuweisungsoperator überlagert, sodass in dem Beispiel aus A1 einfach ein double-Wert gemacht werden kann und umgekehrt.

Fehler(Ausnahmen) werden als C++-Exceptions gehandhabt, die mit try/catch behandelt werden können. Also würde hier z.B. die Cast-Operation eine *PlTypeError*-Exception erzeugen, wenn der Term nicht vom Typ *double* ist oder nicht in diesen konvertiert werden kann.

Nun ist es manchmal notwendig, Terme selbst zu erstellen, beispielsweise um eine Liste zu erzeugen. Auch dafür ist die Prolog API ausgelegt. Zuerst werden eine oder mehrere Term-Referenzen mittels *PL_new_term_ref()* bzw. *PL_new_term_refs()* erzeugt und kann diesen dann Werte zuweisen, vergleichen oder manipulieren.

```
term_t term1 = PL_new_term_ref();
term_t term2 = PL_new_term_refs(2);
PL_unify_nchars(term1,"Ein String");
PL_unify_long(term2,(long)10);
```

```
PL_unify_nchars(term2+1,"Zweiter Term");
```

Mit der C++ API würde das so aussehen:

```
PlTerm term1("Ein String");
PlTermv term2((long)10,"Zweiter Term");
```

Die Klasse *PlTermv* ist ein Term-Array mit den *PL_new_term_refs()* erzeugten Termreferenzen. Auf diese Terme kann mittels des []-Operators zugegriffen werden. Allerdings ist die Anzahl der Terme auf maximal 6 beschränkt, da nicht mehr Konstruktoren definiert wurden. Eine Bereichsprüfung beim Zugriff findet auch nicht statt.

Um eine Liste aufzubauen, wird die Klasse *PlTail* benutzt. Diese akzeptiert als Parameter entweder eine freie Variable oder eine schon bestehende Liste. Zum schrittweisen Durchgehen der Liste besitzt die Klasse die Methode *int next(PlTerm &)* und zum Aufbauen die Methoden *int append(const PlTerm &)* und *int close()*.

Beispiel 5.2: Erzeugen einer Liste:

```
PlTerm list;      // freie Variable = neuer Termref
PlTail t(list);   // Variable = neue Liste

// Etwas an die Liste anhängen
t.append((long)10);
t.append("Text");

t.close();        // Liste mit [] schließen.
```

Wie schon erwähnt, ist die API bidirektional, d.h. es kann auch von C aus Prolog aufgerufen werden. Dabei gibt es einige Dinge zu beachten, da Unzulänglichkeiten in der API vorhanden sind, denen man u.U. erst nach langem Suchen auf die Spur kommt.

Um Regeln und Fakten abzufragen, gibt es die Klasse *PlQuery*. Die Benutzung ist denkbar einfach. Man erstellt ein Objekt der Klasse und übergibt dabei eine Zeichenkette mit dem Namen und einem PlTermv mit den Argumenten, sowie optional den

Modulnamen. Um die Ausführung zu erreichen, wird die Methode *int next_solution()* aufgerufen. Der Rückgabeparameter ist dabei der Erfolg, wie die Prolog-Funktion ihn wiedergibt.

```
PlQuery query("writeln",PlTermv("hello world"));
query.next_solution();
```

Nun gibt es bei der Klasse aber ein Problem, und zwar wenn mehrere Querys benutzt werden sollen. Das folgende Beispiel erzeugt u.U. einen Fehler oder gar einen Absturz des Prolog-Interpreters:

```
PlQuery query1("writeln", PlTermv("eins"));
PlQuery query2("writeln", PlTermv("zwei"));
query1.next_solution();
query2.next_solution();
```

Das liegt daran, dass das SWI-Prolog verbietet, aktive Querys zu verschachteln und dies nicht prüft. Da wir nun 2 Instanzen der Klasse Query erzeugt haben, wurden auch zwei Querys in Prolog geöffnet. Dies ist soweit kein Problem. Allerdings werden diese nun mit *next_solution()* aktiviert und erst wieder mit *PL_cut_query()* im Destruktor der Klasse PlQuery geschlossen. Da der Destruktor logischerweise aber erst am Ende des C-Funktionsblockes aufgerufen wird, sind somit zwei Querys gleichzeitig aktiv. Folgender Quellcode berichtigt dieses:

```
PlQuery *query1 = new PlQuery("writeln",PlTermv("eins"));
PlQuery *query2 = new PlQuery("writeln",PlTermv("zwei"));
query1->next_solution(); delete query1;
query2->next_solution(); delete query2;
```

Wenn die volle Funktionalität von PlQuery nicht benötigt wird, existiert eine Hilfsfunktion *PlCall()*, wodurch dieser Fehler nicht auftreten kann.

Wenn aus C heraus Prolog benutzt wird, muss immer ein sogenannter *foreign frame* aktiv sein. Dieser erstellt u.a. einen lokalen Stack. Mit diesem Frame können auch

alle Veränderungen rückgängig gemacht werden (rewind). Dieser Frame wird automatisch erstellt, wenn Prolog eine C-Funktion aufruft; d.h. dass dort kein eigener Frame erstellt werden muss (aber kann). Wenn allerdings Prolog z.B. aus einem Thread heraus aufgerufen werden soll, muss vorher ein solcher Frame erstellt werden, da es sonst mit Sicherheit zu zufälligen Fehlern kommt, bis hin zum Absturz des Systems.

Dazu existiert die Klasse *PlFrame*, welche diese Aufgabe übernimmt. Wichtig hierbei ist es, dass Frames zwar verschachtelt werden können, aber zu jedem Erstellen muss auch ein Schließen folgen. Was wiederum zu dem Problem führt, wie es bei PlQuery angesprochen wurde. In dem aktuellen Frame können auch nur die Terme angesprochen werden, die in diesem Frame erstellt wurden, da ein neuer lokaler Stack existiert. Mit der Methode *rewind()* können alle Änderungen rückgängig gemacht werden.

Hier ein Beispiel zur Benutzung von Frames:

```
void mythread()
{
     for(;;)
     {
          PlFrame *frame = new PlFrame;
          try
          {
               PlTerm term;
               term = "hello world";
               PlCall("writeln", PlTermv(term));
          }
          catch(PlException &ex)
          {
               // Fehler
               ex.plThrow();
               frame->rewind(); // Alle Aktionen rückgängig machen
          }
          delete frame;
     }
}
```

Es gibt noch viel mehr Möglichkeiten, die mit der SWI-Prolog API durchgeführt werden können. Allerdings sind die vorgestellten Funktionen und Klassen für das Verstehen des Quellcodes, die in der Diplomarbeit verwendet wurden, ausreichend. Umfassendes Material ist in der zusätzlichen SWI-Prolog-Dokumentation nachzulesen, die auf der Homepage von SWI-Prolog heruntergeladen werden kann.

5.2.2 XPCE

XPCE ist eine programmiersprachenübergreifende, objektorientierte Umgebung zur Programmierung von Benutzeroberflächen. Das ursprünglich als PCE bekannte System wurde von Anjo Anjewierden entwickelt. Seit Version 3 arbeitet auch Jan Wielemaker (SWI-Prolog) an dem Projekt und XPCE wird derzeit hauptsächlich für Prolog weiterentwickelt. Die aktuelle Version ist 6.0.9 und ist im SWI-Prolog Paket enthalten. Wie auch bei SWI-Prolog ist der Quellcode von XPCE offen und unterliegt dem LGPL-Lizenzmodell.

Die Benutzung des Oberflächensystems wird hier kurz erläutert, da alle Benutzeroberflächen in Rahmen dieser Diplomarbeit mit dieser Bibliothek erstellt wurden.

XPCE ist nicht nur ein Oberflächensystem, sondern bereichert auch das Prolog-System. Somit wurde das interne Prolog-Hilfesystem auf XPCE umgestellt. Auch gibt es den *Emacs* als XPCE-Version, der mittels *emacs(dateiname)* aufgerufen werden kann. Er unterstützt Syntaxhighlighting, sowie komfortableres und übersichtlicheres Debuggen. XPCE bietet auch ein internes Hilfesystem, einen Klassenbrowser, Demoprogramme u.v.m.

Mit XPCE wurde ein gelungenes objektorientiertes System in Prolog gebracht, mit PCE Klassen und deren Methoden. Alle Objekte sind von der Basisklasse *object* abgeleitet. XPCE bietet nur Einfachvererbung und unterstützt keine Templates oder Interfaces, sowie kein Überladen von Operatoren. Ein Zugriffsschutz wie public, protected und private existiert ebenfalls nicht. Alle Methoden sind public, welche aber überladen werden können. Ausnahmebehandlungen (exceptions) werden unterstützt. XPCE unterscheidet streng zwischen Ein- und Ausgabemethoden, welche mit *send* und *get* angesprochen werden, indem Nachrichten übermittelt werden.

Beispiel 5.3: Grundfunktionalität von XPCE.

```
:- module(myclass, [test_myclass/0]).
:- use_module(library(pce)).

% Die Klasse beginnt hier
% hat den Namen 'myclass'
```

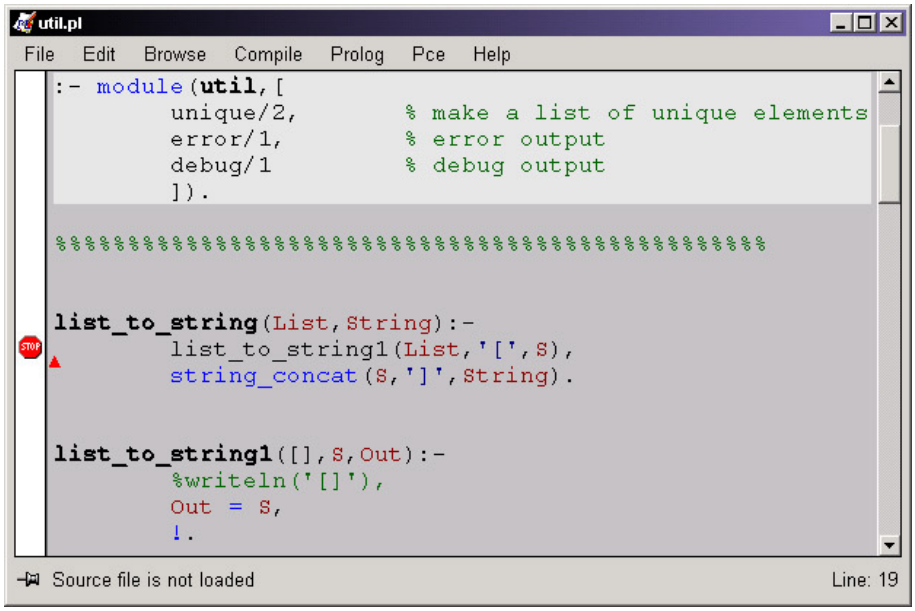


Abbildung 5.2: Emacs-Version von XPCE

```
% und ist von 'object' abgeleitet
:-pce_begin_class(myclass, object).

% Eine zu dem Objekt gehörende 'Variable' anlegen.
% Dieses Objekt wird erst angelegt, wenn es benötigt wird
:-pce_global(@number,new(number(0))).

% Konstruktor
initialise(Self,V):->
     % Supermethode aufrufen, in dem Fall den Konstruktor
     send_super(Self,initialise),
     send(@number,value,V).

% Eingabemethode um den Wert zu ändern
value(_,V):->
     send(@number, value ,V).
```

```
% Wert bekommen mittels Ausgabemethode
value(_,V):<-
     "Eine kurze Beschreibung für die Onlinehilfe"::
     get(@number, value ,V).

% Klassendefinition endet hier
:-pce_end_class.

% Die Klasse testen
test_myclass:-
     new(M, myclass(10)),     % neue Instanz der Klasse erzeugen
     send(M, value, 20),      % Eingabe Methode
     get(M, value, V),        % Ausgabe Methode
     writeln(V).
```

Die im Quellcode eingefügten Kommentare sollten als Erklärung ausreichend sein. Weiterhin ist zu bemerken, dass man mit dem „?“ die get-Aufrufe abkürzen kann:

```
get(F, device, D), get(D, member, M),
send(M, for_some, message(@prolog, do_something)).

% Dies bewirkt das gleiche wie das obere Quellcodestück
send(F?device?member, for_some, message(@prolog, do_something)).
```

Die OOP-Umgebung von XPCE bietet noch mehr Möglichkeiten, wie autoload, funktionale Gruppen und Klassenvariablen, sowie die Möglichkeit, schon bestehende Klassen zu erweitern.

Die Klassen zur Erstellung der Benutzeroberflächen

Die OOP-Umgebung dient als Grundlage für das GUI-System. Es existieren viele Klassen, um Oberflächenelemente zu programmieren. Das XPCE-Oberflächensystem ähnelt andereren Oberflächensystemen und bietet beispielsweise Fenster, Dialoge, Knöpfe, Textfelder, Schieberegler, Karteikarten, Listen u.v.m.

Eine druckbare Dokumentation ist leider im Paket nicht enthalten, sodass auf die Onlinedokumentation zurückgegriffen werden muss. Da die Beschreibungen manchmal recht dürftig sind, hilft oft nur ein Ausprobieren oder es sind Beispielprogramme

zur Hilfe zu nehmen.

Beispiel 5.4: Benutzung von XPCE.

```
test:-
     new(F, frame('Fenstertitel')),
     send(F, append, new(D, dialog)),
     send(D, append, button(exit1, message(F, destroy))),
     send(D, append, button(exit2, message(@prolog, pl_test, F))),
     send(F, open).

pl_test(F):-
     send(F, destroy).
```

Die Abbildung 5.3 zeigt die erzeugte Oberfläche. Die beiden Knöpfe führen dieselbe Aktion aus. Sie sind nur auf unterschiedliche Weise umgesetzt worden.

Abbildung 5.3: Das XPCE-Beispielprogramm

Die Elemente sind in XPCE hierarchisch angeordnet. Hier im Beispiel so:

- frame('Fenstertitel')
    - dialog()
        - button(exit1)
        - button(exit2)

Mittels der *append* werden die Elemente zu dem Elternobjekt hinzugefügt. Das Layout erfolgt hierbei automatisch. Man kann allerdings auch die Anordnung selbst bestimmen, indem man anstatt *append* einen der vier Anordnungsfunktionen verwendet.

Diese sind *left*, *right*, *above* und *below*. Allerdings werden diese nicht an das Elternobjekt gesendet, sondern an die Geschwisterobjekte.

Wenn beispielsweise die beiden Knöpfe untereinander angeordnet werden sollen, wird zuerst ein *button* zu dem *dialog* hinzugefügt und der zweite *button* wird dann mit *below* angeordnet. Er wird damit automatisch zum *dialog* hinzugefügt und XPCE versucht die Elemente nach den Vorgaben anzuordnen.

```
send(F, append, new(D, dialog)),
send(D, append, new(B1, button(exit1, message(F, destroy)))),
send(button(exit2, message(@prolog, pl_test, F)), below, B1),
```

Wenn ein Knopf gedrückt wird, wird das übergebene *message*-Objekt ausgewertet. Der erste Parameter von *message* ist ein Objekt und als zweiter Parameter wird die dazugehörige Methode angegeben. Alle weiteren Parameter sind Übergabeterme an die Methode. Es gibt spezielle vordefinierte Parameter. Dazu zählen z.B. *@arg1* oder *@receiver*. Das Atom *@receiver* ist wie der Name schon sagt, das Objekt selbst, also der Empfänger der Nachricht. In dem Beispiel wäre es der *button*. Die Atome *@arg1*,*@arg2*, etc. werden von den Klassen benutzt, um Werte an die Methode zu übergeben und haben deshalb in jeder Klasse unterschiedliche Bedeutungen. Bei einem *slider* beispielsweise ist *@arg1* der aktuelle Wert des Schiebereglers.

Wenn anstatt des Objektes das Atom *@prolog* übergeben wird, weiß damit XPCE, dass die Methode, die aufgerufen werden soll, eine Prolog-Regel ist. In unserem Beispiel ist das die Regel *pl_test*, die als Parameter den *frame* übergeben bekommt und die Methode *destroy* aufruft, also damit den Frame schließt.

Mit diesen Ausführungen sollte das Prinzip des XPCE-System ausreichend erklärt sein. Näheres kann man in der XPCE-Dokumentation nachlesen.

5.3 Implementierung

Nachdem die Grundlagen erläutert wurden, erfolgt nun in diesem Abschnitt die Erläuterung der Details zur Implementierung des Prototypen.

Zuerst wird ein Überblick zu dem Projekt gegeben. Die dazugehörigen Dateien sind auf der beigefügten CD zu finden. Im Anschluss daran wird auf die Tools-Bibliothek näher eingegangen, welche als Grundlage für die Erstellung der DLL's dient. Daran folgt die ausführliche Beschreibung der Module. Im Einzelnen sind diese: der Simulator, die Wissensbasis und die Beispielanwendungen.

5.3.1 Systemarchitektur

Das System gliedert sich in drei große Module:

1. Bildverarbeitung/Objektklassifikation
2. Wissensbasis
3. Applikation

Die Bildverarbeitung erzeugt ein Bild, wertet dieses aus und erzeugt daraus Objektdaten, welche die Objektklassifikation in Klassen einstuft. Die Objektdaten werden mit einem Zeitstempel versehen an die Wissensbasis weitergeleitet, welches daraus neue Fakten erzeugt. Die Applikation fragt nun die Wissensbasis über die bekannten Objekte und deren möglichen Positionen ab.

Nun können die drei Module in unterschiedlichen Prozessen laufen oder sogar auf verschiedenen Computersystemen. D.h. die drei Module müssen auf irgend einem Wege miteinander kommunizieren. Dafür wurden Bibliotheken entwickelt, um diese Kommunikation so transparent wie möglich zu halten.

Alle drei Module wurden in Prolog entwickelt. Nur die Kommunikation und die Schnittstellen wurden in C++ geschrieben und in Bibliotheken verpackt, um damit eine allgemeine von Prolog unabhängige Schnittstelle zu erhalten. Desweiteren wurde eine Beispielapplikation, die die Schnittstelle zur Wissensbasis nutzt, in C++ geschrieben, um zu belegen, dass auch wirklich eine von Prolog unabhängige Schnittstelle existiert, und um zu zeigen wie sie genutzt wird. Abbildung 5.4 gibt einen allgemeinen Überblick über das System.

Die C++ Programme wurden alle mit Microsoft VisualC++ (VC++) erstellt. Es kann u.U. vorkommen, wenn eine andere Version von VC++ verwendet wird, dass die Projektdateien und Bibliotheken neu erstellt werden müssen. Die Quellcodes sollten aber mit allen Versionen von VC++ ab 5.0 compilierbar sein.

Es wurde auch die STL (Standard Template Library) benutzt, die bei VC++ mitgeliefert wird. Wenn zu dem entsprechenden Compiler-Paket kein STL mitgeliefert wird, kann die *OpenSTL* benutzt werden, welche frei verwendbar ist (GPL). Der Umgang mit der STL vereinfacht stark die Verwendung von *strings*, *streams* und sogenannten *container*-Klassen. Als Dokumentation zur STL ist [Str98] zu empfehlen.

Im Anhang 6.3 befindet sich die Dateistruktur mit kurzen Kommentaren, die als Überblick dienen soll.

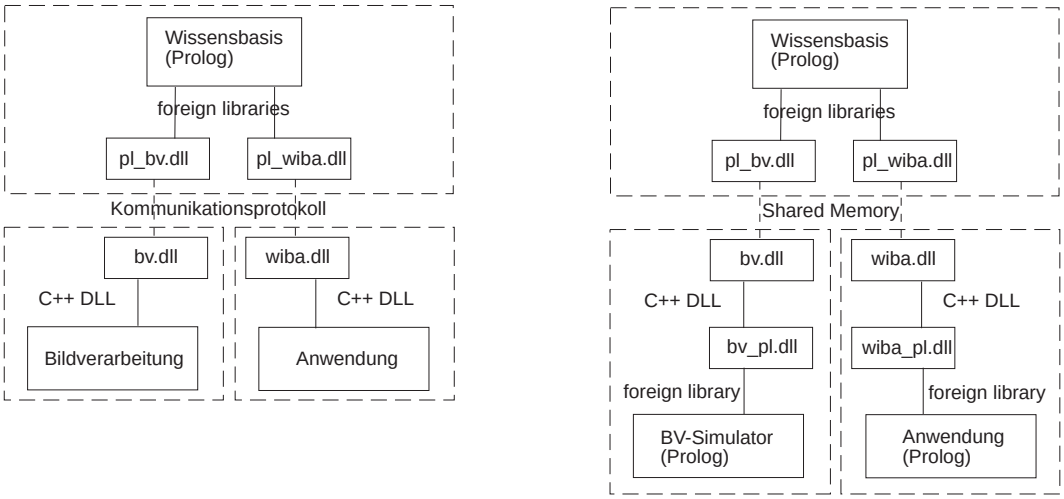


Abbildung 5.4: Allgemeine Systemarchitektur (links) und die des Prototypen (rechts)

5.3.2 Tools-Bibliothek

Als Grundlage für die DLL's wurde einen Statische-Link-Bibliothek erstellt, die diverse nützliche Funktionen und Klassen beinhaltet. Die Headerdateien sind dabei betriebssystemunabhängig programmiert worden. Nur die CPP-Dateien müssten bei einer Portierung auf ein anderes Betriebssystem neu programmiert werden.

Zuerst werden die Erweiterungen der SWI-cpp erläutert. Dann geht es um Threads und deren Hilfsmittel sowie um die Kommunikationsschnittstelle und insbesondere um die Sharedmem-Implementierung.

Erweiterungen der SWI-cpp

In der Headerdatei *SWI-cpp-ext.h* sind Erweiterungen zu der SWI-cpp hinzugekommen. Einmal die Klasse *PlVariable*, die eine freie Variable darstellt, und die Klasse *PlNil*, welches die leere Liste [] ist. Beide Klassen sind von PlTerm abgeleitet und stellen nur Erweiterungen dar, die die Lesbarkeit des Quellcodes erhöhen.

Zur Ausgabe von Fehlermeldungen wurde die Funktion *PlError(PlTerm &)* erstellt. Diese gibt die entsprechende Fehlermeldung über den Prolog-Ausgabestream aus. Dabei können alle Prologterme ausgegeben werden, insofern diese gültig sind. Ansonsten beendet sich das Programm selbständig mit einer *unexpected*-Exception.

Weitere Erweiterungen mussten nicht vorgenommen werden.

Thread, Mutex, Event

Um die Portabilität des Systems zu erhöhen, wurden die Windows-Thread-Funktionen in Klassen gekapselt. Dabei wird in der Headerdatei ganz auf windowsspezifische Ausdrücke verzichtet und es muss nur die CPP-Datei an das jeweilige Betriebsystem angepasst werden.

Es existieren drei Klassen:

1. cThread
2. cEvent
3. cMutex

cEvent und *cMutex* wurden in dem Projekt nicht verwendet, wurden aber trotzdem im Projekt belassen, da diese evtl. für weitere Entwicklungen sinnvoll sind.

cEvent Es handelt sich hierbei um ein Event, welches solange wartet, bis es benachrichtigt wird. Mit dem Aufrufen der Methode *wait()* tritt der Thread in den Wartezustand ein und mit *notify()* wird der Thread wieder aufgeweckt.

Es kann auch ein prozessübergreifender Event angelegt werden, wenn ein Event mit einer Zeichenkette zur Identifikation angelegt wird. Wird nun ein zweites Event-Objekt von demselben oder von einem anderen Prozess angelegt, welches dieselbe Identifikationszeichenkette verwendet, werden diese beiden Objekte wie ein Event-Objekt behandelt. Es ist darauf zu achten, dass es nicht zu Kollisionen kommt, wenn versehentlich die gleiche Identifikation genommen wird oder ein anderer Prozess diesen schon verwendet. Dies wird nicht geprüft. Man sollte daher die Zeichenkette so wählen, dass es sehr unwahrscheinlich ist, dass ein anderer Prozess diese schon verwendet.

Beispiel 5.5: Benutzung von cEvent.

```
#include <iostream.h>
#include "thread.h"

void mythread(cEvent &event)
{
     cout<<"Warte ..."<<endl;
```

```
    event.wait();
    cout<<"Wartezustand beendet"<<endl;
}

int main()
{
    cEvent event;
    // Hier wird ein Thread gestartet (symbolisch)
    start_thread(mythread,event);
    // etwas warten
    sleep(1000);
    // Thread wieder aufwecken
    event.notify();
    return 0;
}
```

cMutex Die Klasse *cMutex* ist im Prinzip ähnlich der *cEvent*-Klasse. Es handelt sich hierbei allerdings um eine Mutex-Variable zur Synchronisation von Resourcen bei Multithreaded-Systemen. Also um den Zugriffsschutz auf Programmteile, die nicht von mehreren Threads gleichzeitig abgearbeitet werden dürfen.

Die Verwendung erfolgt genau so wie bei der *cEvent*-Klasse. Die Methoden heißen hier allerdings *enter()* und *release()*. Es sind auch prozessübergreifende Mutex-Objekte möglich.

cThread Diese Klasse wurde in diesem Projekt ausgiebig verwendet, um Multithreading einigermaßen systemunabhängig zu ermöglichen. Multithreading war notwendig, um die Kommunikationsschnittstelle zwischen den einzelnen Modulen zu realisieren. Der Aufbau der Klasse ist der Java-Thread-Klasse (in stark vereinfachter Form) sehr ähnlich.

Die Methoden sind:

void start() Der Thread wird gestartet.

void stop() Der Thread wird wieder angehalten, falls er gestartet wurde.

void run() Diese Methode ist der eigentliche neue Thread. Diese muss überladen werden mit einer eigenen Thread Funktion.

Wird diese Funktion verlassen, wird der Thread automatisch beendet. Sie muss allerdings nicht verlassen werden. Bei einem *stop()* oder dem Zerstören des Objektes wird die Abarbeitung dieser Methode automatisch unterbrochen.

setPriority(THREAD_PRIORITY) Setzt vor dem Start oder zur Laufzeit die Priorität des Threades. In dem Aufzählung *THREAD_PRIORITY* sind einige Stufen definiert.

bool yield(int timeout=-1) Ein anderer Thread kann auf die Beendigung dieses Threads warten, indem er diese Methode aufruft. Er tritt dann solange in den Wartezustand ein.

Bei Verwendung dieser Methode ist jedoch erhöhte Aufmerksamkeit angebracht, da sie im Rahmen des Projektes nicht verwendet wurde und damit nicht ausreichend getestet wurde. Also die Funktionsweise ist nicht garantiert.

Der Parameter *timeout* ist die maximale Wartezeit in Millisekunden. Wird kein Parameter angegeben, ist die Wartezeit unbegrenzt.

Ist der Rückgabewert *false*, ist ein Timeout aufgetreten.

Die Methode darf logischerweise von Thread selbst nicht benutzt werden, da er sonst auf sich selbst warten würde.

bool sleep(int millies) Diese Methode darf nur von Thread selbst aufgerufen werden und versetzt den Thread für die angegebene Zeit in den Wartezustand. Die Zeit wird dabei in Millisekunden angegeben.

bool wait(int timeout=-1) Der Thread wartet auf eine Benachrichtigung. Dies ist ähnlich dem *cEvent*-Objekt, ist aber bereits in der Klasse verankert und mit einer Timeout-Option versehen.

Diese Methode darf nur vom Thread selbst aufgerufen werden.

Ist der Rückgabewert *false*, ist ein Timeout aufgetreten.

bool notify()

Ein anderer Thread kann den Wartezustand des Threads wieder aufheben. Und zwar dann, wenn dieser die *wait()* Methode aufgerufen hat.

Der Rückgabewert ist *true*, wenn kein Fehler aufgetreten ist.

Beispiel 5.6: Verwendung von Threads.

```
#include <iostream.h>
#include "thread.h"

class MyThread : public cThread
{
public:
     void run()
     {
          // 10x den Text aufgeben
          for(int i=0;i<10;i++)
          {
               this->sleep(100);
               cout<<"Thread Ausgabe"<<endl; cout.flush();
          }
     }
};

int main()
{
     MyThread thread;
     // Thread starten
     thread.start();
     // Warten bis der Thread sich beendet hat
     thread.yield();
     return 0;
}
```

Kommunikation zwischen den Modulen

Die Module müssen untereinander kommunizieren, d.h. Daten austauschen. Die Einzelmodule befinden sich allerdings nicht im selben Prozess. Also muss mindestens eine prozessübergreifende Kommunikation stattfinden. Um aber die Möglichkeit zu

gewährleisten, dass die einzelnen Programme auf unterschiedlichen Rechnersystemen laufen können, muss auch eine Netzwerkkommunikation in Betracht gezogen werden.

Dieses wurde so umgesetzt, indem mehrere abstrakte Klassen definiert wurden, die all diese Kommunikationsmöglichkeiten repräsentieren. Diese Headerdatei ist wiederum vom verwendeten Betriebsystem unabhängig und nur die CPP-Datei muss an das Betriebssystem angeglichen werden. Das System kann auch einfach um eine neue Übertragungsmöglichkeit erweitert werden, wie im Weiteren noch zu sehen ist.

Die Kommunikation ist so aufgebaut, dass Nachrichten zwischen Client und Server verschickt werden können. Es ist vorgesehen, dass mehrere Clients sich mit einem Server verbinden können. Dies ist allerdings noch nicht konsequent umgesetzt worden, sodass zur Zeit nur ein Client sich zu einem Server verbinden kann, was für dieses Projekt auch ausreichend ist. Es sei angemerkt, dass es Probleme gibt, wenn versucht wird mehr als eine Bildverarbeitung zu starten, oder mehrere Anwendungen versuchen, sich mit der Wissensbasis zu verbinden. Dies führt zu nicht vorhersehbaren Resultaten.

Bei allen Schnittstellen werden die Daten synchron übertragen. Wenn eine Zeitverzögerung, ausgelöst durch Wissensbasis oder Anwendung, vermieden werden soll, muss die jeweilige Seite mit einem zusätzlichem Thread ausgestattet werden. Das ist meistens bei Echtzeitsystemen der Fall, wo eine reale Kamera die Bildinformationen liefert. Beispielsweise, wenn die Anwendung die Daten, die die Wissensbasis liefert, nicht schnell genug auswerten kann, ist es angebracht, nicht jeden Frame auszuwerten. Sondern es sind einige zu überspringen, um gute Frameraten zu erhalten, mit der die Wissensbasis aktualisiert wird.

Zunächst werden die abstrakten Klassen erläutert und im Anschluss daran die Beispielimplementierung für die Kommunikationsschnittstelle.

In den Dateien *clientserver.h* und *clientserver.cpp* wurden folgende Klassen definiert:

- cServer
- cClient
- cServerHandler
- cClientHandler

cServer Diese Klasse stellt einen Server dar. Wenn dieser gestartet wird, akzeptiert er automatisch Verbindungsanfragen von Clients. Beim Anlegen des Server-Objektes wird die Identifkation und das Protokoll mit übergeben.

Die Methoden sind:

bool open() Startet den Server und erlaubt Verbindungsanfragen.

Bei Erfolg wird *true* zurückgegeben, ansonsten kann mit *getLastError()* die Fehlermeldung ausgewertet werden.

bool close() Beendet alle geöffneten Verbindungen und schließt den Server.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

int waitForConnection() Wartet darauf, bis sich mindestens ein Client verbunden hat.

Der Rückgabewert ist die Client-ID oder 0 bei einem Fehler.

int recv(int client, std::ostream &stream)

Wartet solange, bis Daten von dem Client mit der ID *client* empfangen wurden und schreibt die Daten in den Ausgabestrom *stream*.

Der Rückgabewert ist die Anzahl der Bytes, die empfangen wurden.

bool send(int client, const char *buf, int len)

Sendet Daten an den Client mit der ID *client*.

Zu beachten ist, dass solange gewartet wird, bis die Daten vollständig empfangen wurden.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

std::vector<int> getClients()

Bekommt die Liste der verbundenen Clients als Vektor.

std::string getLastError()

Gibt die letzte Fehlermeldung wieder.

Wenn eine Funktion einen Fehler produziert, wird die Meldung als *string* zurückgegeben und beispielsweise an den Benutzer weitergereicht.

cClient Dies ist das Gegenstück zu *cServer* und repräsentiert einen Client, der sich mit einem Server verbinden kann. Der Server muss dazu natürlich vorher gestartet werden, ansonsten ist eine Verbindung nicht möglich.

Die Methoden sind:

bool connect(const std::string &addr) Versucht den Server zu verbinden unter der Adresse *addr*. Diese Adresse ist von Protokoll zu Protokoll unterschiedlich. Es kann ein einfacher Name sein oder auch eine IP-Adresse.

Bei Erfolg wird *true* zurückgegeben, ansonsten kann mit *getLastError()* die letzte Fehlermeldung erhalten werden.

bool disconnect() Schließt eine geöffnete Verbindung.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bool isConnected() Signalisiert, ob der Client mit einem Server verbunden ist.

int recv(std::ostream &stream)

Wartet solange, bis Daten von dem Server empfangen wurden und schreibt die Daten in den Ausgabestrom *stream*.

Der Rückgabewert ist die Anzahl der Bytes, die empfangen wurden.

bool send(const char *buf, int len)

Sendet Daten an den Server.

Zu beachten ist, dass solange gewartet wird, bis die Daten vollständig empfangen wurden.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

std::string getLastError()

Gibt die letzte Fehlermeldung wieder.

Wenn eine Funktion einen Fehler produziert, wird die Meldung als *string* zurückgegeben und beispielsweise an den Benutzer weitergereicht.

cServerHandler, cClientHandler Diese Klassen besitzen dieselben Methoden wie *cServer* bzw. *cClient* und sind die Interfaces[2] für eine Implementierung der Server- bzw. Client-Klasse. Dies wird bei der Beispielimplementierung noch weiter erläutert. In der *clientserver.cpp* muss nur das neue Protokoll registriert werden, d.h. die neuen Klassen, die von diesen beiden Klassen abgeleitet wurden.

Client/Server Beispielimplementierung

Es standen mehrere Möglichkeiten zur Auswahl, die als Protokoll (Übertragungsmedium) geeignet waren.

[2]Mit Interface ist eine vollständig virtuelle Klasse gemeint.

RPC/COM/...

Das Windowssystem bietet mehrere Möglichkeiten an, prozessübergreifende Funktionen aufzurufen (auch netzwerkübergreifend). Es wurde allerdings nicht weiter betrachtet, da dies eine umfangreiche Einarbeitung bedeutet hätte.

TCP/IP-Netzwerk

Hiermit ist es sicherlich recht portabel, Daten zu übertragen und die Umsetzung ist auch nicht allzu schwierig. Allerdings, da ursprünglich geplant war, Bilddaten zu übertragen, war es fraglich, ob die Geschwindigkeit ausreichend ist.

Im Rückblick ist zu sagen, dass vielleicht doch dieses Protokoll hätte gewählt werden sollen, da die Datenmenge zum aktuellen Projektstatus recht klein ist und die Implementierung einfacher gewesen wäre.

Shared-Memory

Shared-Memory ist ein Speicherbereich, der von verschiedenen Prozessen genutzt werden kann. Da die heutigen Betriebssysteme mit Speicherschutzmechanismen arbeiten, ist es nicht möglich einfach ein Stück Speicher mit *malloc()* oder *new* anzufordern und den Zeiger an einen anderen Prozess weiterzugeben, damit er Zugriff auf den Speicher hat. Dies würde zu einer Fehlermeldung des Betriebsystems führen. Nun bietet Windows (auch Linux und andere Betriebssysteme) mit speziellen Funktionen an, genau solchen einen Speicherbereich zu erstellen.

Dieses Möglichkeit wurde genutzt, um damit Daten prozessübergreifend auszutauschen.

Die Klassen für die Sharedmem-Kommunikation sind in der Datei *clientserver.cpp* untergebracht und relativ komplex. Hier sollen daher nur die Grundfunktionen erläutert werden.

Wie schon erwähnt, lässt diese Client/Server Implementierung nur einen Client gleichzeitig pro Server zu. Daher ist die Bezeichnung Client/Server unglücklich gewählt. Die beiden Klassen heißen daher *cSMMaster*, welcher der Server ist, und *cSMSlave*, welches den Client repräsentiert. *SM* steht hierbei für SharedMem.

Auf die beiden Klassen kann man von außerhalb nicht zugreifen, sondern nur über die Klassen *cServer* bzw. *cClient*, indem man das Protokoll „sharedmem“ im Konstruktor mit übergibt.

Beispiel 5.7: Benutzung der Sharedmemübertragung.

```
#include <iostream.h>
#include <sstream>
#include "thread.h"
#include "clientserver.h"

class MyServer : public cServer, public cThread
{
public:
          MyServer(const std::string &addr,
               const std::string &type)
               : cServer(addr, type) {}

     // Überladen von open()
     // damit der Thread gestartet wird
     bool open()
     {
          bool ret = cServer::open();
          cThread::start();
          return ret;
     }

     // der Thread
     void run()
     {
          int client = waitForConnection();
          for(;;)
          {
               std::stringstream data;
               this->recv(client,data);
               cout<<data.str().c_str()<<endl;
               cout.flush();
          }
```

```
    }

};

int main()
{
    std::string id("meineserverid");
    MySever server("sharedmem", id);
    cClient client("sharedmem");

    // Server starten
    if(!server.open())
    {
        cerr<<server.getLastError().c_str()<<endl;
        cerr.flush();
        return 1;
    }

    // Mit dem Server verbinden und Nachricht senden
    if(!client.connect(id))
    {
        cerr<<client.getLastError().c_str()<<endl;
        cerr.flush();
        return 1;
    }

    std::string data("Hallo Server");
    client.send(data.c_str(),data.size());

    return 0;
}
```

Das Beispiel sollte anhand der Kommentare selbsterklärend sein. Hier nur eine kurze Zusammenfassung zum Ablauf.

Zuerst wird ein Server gestartet mit dem Sharedmem-Protokoll und einer ID. Dieser Server ist auch gleichzeitig ein Thread, der nach dem Öffnen auf Daten wartet und diese in der Console ausgibt. Dann wird ein Client gestartet, der sich zu dem Server verbindet und ihm eine Zeichenkette sendet.

Die Klassen für die Sharedmem-Übertragung nutzen die entsprechenden Windows-API-Funktionen für das Anlegen, Zugreifen und Entfernen eines Stücks gemeinsamen Speichers. Der Server legt einen solchen Speicher mit fester Größe an und der Client holt sich die Adresse des Speichers.

Beim Senden und Empfangen der Daten sowie beim Verbindungsaufbau und Verbindungsabbau werden Events verwendet, um Benachrichtigungen zu senden, um mitzuteilen, dass Daten im Puffer sind bzw. sich ein Client verbindet. Ist der zu übertragende Datenblock größer als der intern angelegte gemeinsame Speicher, werden die Daten in Blöcken übertragen und zwischengepuffert.

Es gibt genau zwei gemeinsame Speicher, einen zum Senden und einen zum Empfangen von Daten. Anders ausgedrückt, für jede Kommunikationsrichtung existiert ein Speicher, damit keine Blockierungen und Konflikte auftreten.

Notwendige kritische Bereiche wurden abgesichert mittels *CRITICAL_SECTION*. Prozessübergreifende Mutex-Variablen mussten nicht verwendet werden, da es wie erwähnt zwei getrennte Speicherbereiche gibt, auf die jeweils nur der Server bzw. Client Schreibzugriff hat.

Weitere Details und Kommentare stehen dazu im Quellcode selbst.

5.3.3 BV-Simulator

Zuerst soll kurz die Benutzung des Simulators erklärt werden. Im Anschluss folgt die Beschreibung der beiden DLL's und zum Schluss die des Prolog-Quellcodes.

Benutzung des Simulators

Das dazugehörige Prolog-Modul ist *simulator.pl* und ist gleichzeitig die Startdatei des Simulators. In diesem Modul wird zuerst die *bv_pl.dll* geladen, welches die Kommunikation zur Wissensbasis übernimmt. Dann öffnet sich eine XPCE-Oberfläche, welches die Benutzerschnittstelle (Abbildung 5.5) zum Simulator darstellt und in dem der Benutzer Kontrolle über die zu sendenden Objektdaten hat.

Mit dem *Add*-Knopf können Objekte hinzugefügt werden, wobei derzeit nur Kreisobjekte möglich sind. Jedes Objekt muss im Simulator seine eigene ID besitzen. Dies

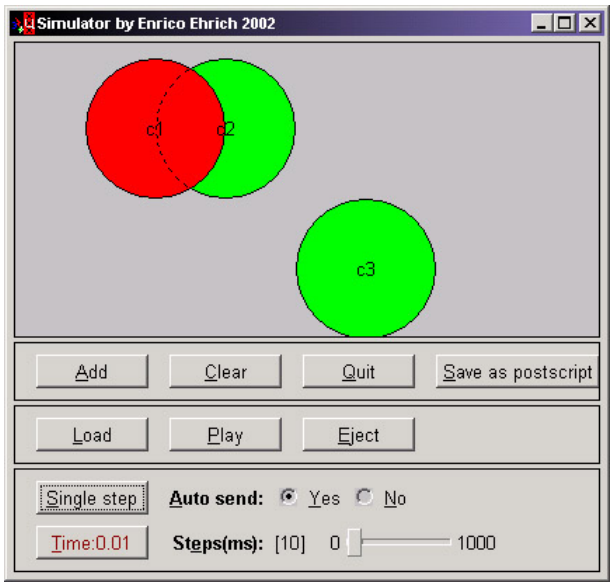


Abbildung 5.5: Die Oberfläche des Simulators

wird überprüft und wenn es bereits existiert, wird das Hinzufügen des Objektes verhindert. Zu dem Kreis wird Radius, Farbe und Position angegeben.

Die Veränderung der Objekte erfolgt einfach mit der Maus, mit der die Objekte verschoben werden. Wenn auf den Rand eines Objektes geklickt wird, kann so seine Größe geändert werden. Das Wechseln der Ebene erfolgt mit einem Doppelklick auf das Objekt. Linke Maustaste für nach hinten bewegen und rechte Maustaste für nach vorne bewegen.

Löschen kann man die Objekte derzeit nur alle auf einmal. Das erfolgt mit dem *Clear*-Knopf.

Es existieren diverse Steuerungselemente für die Frame-Zeiten und das Senden der Framedaten. Der Knopf *Single step* sendet den aktuellen Frame an die Wissensbasis und erhöht den Framezähler um die eingestellte Schrittweite. Mit dem *Time*-Knopf kann man die Zeit kontinuierlich ablaufen lassen. Wenn dabei die *Auto send* Option aktiviert wird, wird zu jedem Schritt auch ein Frame an die Wissensbasis gesendet.

Desweiteren können Skripte geladen werden, die sich hiermit abspielen lassen. Der Aufbau der Skripte wird im Anschluss besprochen.

Auch können hier Einstellungen vorgenommen werden und das simulierte Bild als Encapsulated-Postscript-Datei (*∗.eps*) gespeichert werden.

Die Skriptsprache Das Skript, das in den Simulator eingeladen werden kann, ist eine Prologdatei. In dieser ist nur eine Regel vorhanden mit dem Namen *script* und

hat einen Parameter, die man mit einer Liste unifizieren muss.

Diese Liste ist wie folgt aufgebaut:

$$[Objekt1, Objekt2, ...]$$

Ein Objekt ist wiederum eine Liste:

$$[Name, Typ, Daten, Veränderung]$$

Der Typ kann derzeit nur *circle* sein und der Aufbau der Daten ist vom Objekttyp abhängig. Bei einem Kreis ist das $[Farbe, Radius]$.

Die *Veränderung* ist eine Liste mit Bewegungsdaten $[M1, M2, ...]$. Jedes Element ist ein Zustand des Objektes zu einem bestimmten Zeitpunkt. Die Werte dazwischen werden interpoliert. Entweder linear oder für weichere Bewegungen kurvenförmig mittels Splines. Derzeit werden Splines noch nicht unterstützt und werden daher nicht weiter erläutert.

Der Aufbau eines Elementes (Keyframe) ist folgender:

$$[Zeit, [X, Y], Kommando, Spline]$$

Zeit gibt an, zu welcher Zeit dieser Zustand eintreten soll. $[X, Y]$ sind die Koordinaten der Position, zu der das Objekt bewegt werden soll. Wichtig ist, dass die Elemente nach den Zeiten aufsteigend sortiert sind, da der Skript-Interpreter ansonst den richtigen Keyframe nicht findet bzw. falsche Werte interpoliert.

Kommando ist ein Atom, welches eine spezielle Aktion mit dem Objekt zu diesem Zeitpunkt ausführt.
Derzeit sind folgende Kommandos vorhanden:

0	- Keine spezielle Aktion durchführen
fr	- Objekt nach vorne bewegen.
bh	- Objekt nach hinten bewegen.
show	- Objekt anzeigen.
hide	- Objekt ausblenden.

Spline ist ein Flag welches angibt, ob mit einem Spline-Interpolations-Algorithmus oder einen Linearen-Interpolations-Algorithmus interpoliert werden soll. Der Wert *0* bedeutet Lineare-Interpolation und *1* bedeutet Spline-Interpolation. Der Wert bezieht sich dabei immer auf den Bereich zwischen diesem Element und dem Vorgängerelement.

Beispiel 5.8: Ablaufskript

```
script(D):-
    writeln('Test Script'),
    D=[
        % Das Objekt c1: Kreis, Radius 50 und mit roter Farbe
        [c1,circle,[red,50],[
            % Keyframe 1
            % Objekt c1 befindet sich zum Zeitpunkt 0s
            % an Position [100,100]
            [0,[100,100],0,0]
            % Keyframe 2
            % Objekt c1 befindet sich zum Zeitpunkt 5s
            % an Position [200,100]
            [5,[200,100],0,0],
            % Keyframe 3
            % Objekt c1 befindet sich zum Zeitpunkt 10s
            % an Position [300,300]
            [10,[300,300],0,0]
        ]],

        % Das Objekt c2: Kreis, Radius 50 und mit grüner Farbe
        [c2,circle,[green,50],[
            % Keyframe 1
            % Objekt c2 befindet sich zum Zeitpunkt 0s
            % an Position [250,200]
            [0,[250,200],0,0],
            % Keyframe 2
            % Objekt c2 wird zum Zeitpunk 5s
            % ausgeblendet
            [5,[250,200],hide,0]
        ]]
    ].
```

bv.dll

Diese Bibliothek wird von einer beliebigen Bildverarbeitung (oder hier vom Simulator) benutzt, um Framedaten an die Wissensbasis zu übermitteln. Dabei wird mit Hilfe der Client/Server Klassen der Tools-Bibliothek eine Kommunikation zur *pl_bv.dll* hergestellt, welche die Wissensbasis zur Verfügung stellt. Diese enthält den entsprechenden Server.

Es werden folgende DLL Funktionen bereitgestellt:

void bv_free(char *obj)

Diese Funktion gibt ein Objekt wieder frei, welches die DLL angelegt hat. Dies ist nach dem Aufruf der Funktionen *bv_frame_begin* und *bv_frame_addobj* notwendig.

bool bv_select_database(const char *database)

Wählt eine Datenbank aus.

Rückgabewert ist *true*, wenn die Datenbank existiert und gewählt werden konnte.

bool bv_connect(const char *protokoll)

Stellt eine Verbindung zur *pl_bv.dll* her.

Diese Funktion muss vor allen anderen in dieser Bibliothek aufgerufen werden.

Der Parameter *protokoll* wählt das Protokoll aus, mit dem verbunden werden soll.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bool bv_disconnect()

Beendet eine bestehende Verbindung.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bool bv_frame_begin(double time, char **frame)

Startet die Übertragung neuer Framedaten.

Der Parameter *time* ist die Framezeit in Sekunden und *frame* ist die zurückgegebene Frameumgebung, die bei *bv_frame_end* und *bv_frame_addobj* angegeben werden muss.

Zu beachten ist, dass *bv_free(frame)* aufgerufen werden muss, wenn das *frame*-Objekt nicht mehr gebraucht wird.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bool bv_frame_end(const char *frame)

Beendet den Frame und startet die Berechnung in der Wissensbasis. Dies kann u.U. einige Zeit dauern.

Der Parameter *frame* ist wieder die erzeugte Frameumgebung.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bool bv_frame_addobj(char **frame, const cObjectData *object, const cPoint position)

Fügt ein Objekt der Frameumgebung hinzu.

Der Parameter *frame* ist die Frameumgebung, die hierbei jedoch geändert wird. Genauer gesagt, es wird die alte Frameumgebung genutzt und eine neue angelegt.

Der Parameter *object* ist das Objekt, das hinzugefügt wird. Die *cObjectData* Struktur ist wie folgt aufgebaut:

```
struct cObjectData
{
    OBJECT_TYPE     type;          // Type des Objektes
    long            o_class;       // Objektklasse
    char            *name;         // Objektname
    int             sizeof_data;   // Größe der Objektdaten
    char            *data;         // Objektdaten
};
```

OBJECT_TYPE ist ein enum, das derzeit nur *CIRCLE* als Typ enthält. Die Objektklasse ist eine von der Objektklassifizierung festgelegte Nummer, zu welcher Klasse dieses Objekt zuzuordnen ist. Der Objektname dient nur zur Information und Fehlersuche. Jedes Objekt besitzt einen Datenblock mit seinen Daten zur Form, Farbe, etc. Dies ist bei jedem Objekttyp anders und ist deshalb nur als einfaches Array definiert. Die Größe des Arrays muss natürlich mit angegeben werden, damit der Client weiß, wieviel Daten er übertragen muss.

Der Parameter *position* gibt die Position an, an dem sich das Objekt befindet. Die Struktur *cPoint* enthält nur einen x- und y-Wert.

Zu beachten ist, dass *bv_free(frame)* aufgerufen werden muss, wenn das *frame*-Objekt nicht mehr gebraucht wird.

Der Rückgabewert ist *true*, wenn der Ausführung erfolgreich war.

bv_pl.dll

Diese Bibliothek ist eine DLL, die in Prolog eingeladen werden kann und einige Funktionen zur Verfügung stellt. Diese DLL wird von der *simulator.pl* benutzt, um auf die *bv.dll* zuzugreifen. Es stellt also nur die DLL-Funktionen von der *bv.dll* in Prolog zur Verfügung. Alle dort enthaltenen Funktionen brauchen nicht näher erläutert werden, da sie denen der *bv.dll* entsprechen. Sie werden hier nur noch einmal aufgelistet:

- bv_select_database(DB)
- bv_connect('protokoll')
- bv_disconnect()
- bv_frame_begin(Time, Frame)
- bv_frame_end(Frame)
- bv_frame_addobj(Frame, Object, Position, NewFrame)

Um Objekte auszusortieren, die nicht sichtbar sind, wird mit Hilfe der Klasse *cDepthBuffer* ein Test durchgeführt, bei dem jeder Pixel in einem 2D-Array (mit dynamischer Größe) getestet und markiert wird, um festzustellen ob er sichtbar ist. Ist kein Pixel eines Objektes sichtbar, so ist das ganze Objekt nicht sichtbar und wird nicht an die Wissensbasis weitergeleitet.

In dem Simulator erfolgt zwar schon eine Aussortierung, allerdings auf geometrischem Wege, wo jedes Objekt mit jedem verglichen wird. Dies reicht aber nicht aus, da auch mehrere Objekte daran beteiligt sein können, ein Objekt vollständig zu verdecken. Dies ist wiederum geometrisch schwer lösbar. Daher ein weiterer Vergleich und Aussortieren der Objekte mit Hilfe dieser cDepthBuffer-Klasse.

Die Prologdatei

Die Datei *simulator.pl* ist der Prolog-Quellcode des Simulators. In ihm wird über XPCE die Benutzeroberfläche erstellt. Die Klasse *simulator* ist diese Oberfläche und ist von der Klasse *frame* abgeleitet. In diesem Frame befinden sich mehrere *dialog*-Objekte, welche die Steuerelemente beinhalten. Desweiteren existiert ein *picture*-Objekt, welches die Objekte darstellt.

Zu diesem *picture* können verschiedene grafische Objekte hinzugefügt werden. Unter anderem *circle*, *box*, *ellipse* und *text*. Die Attribute dieser Objekte sind veränderbar. Die Farbe beispielsweise wird über die Methode *fill_pattern* verändert.

```
new(O, circle(10)),
send(O, fill_pattern, color(red))
```

Objekte kann man in XPCE „gruppieren“, also dafür sorgen, dass diese immer an der selben Position sind wie das Geschwisterobjekt. Dafür bietet XPCE sogenannte *constraint*-Objekte.

Beispiel 5.9: Verwendung von *constrains*.

```
:-pce_global(@center, new(identity(center))).


constraint:-
    new(P, picture('Constraints')),
    send(P, display, new(B1, box(100,100))),
    send(P, display, new(B2, box(50,50))),
    send_list([B1, B2], recogniser,
        handler_group(
            new(resize_gesture),
            new(move_gesture))),
    new(_, constraint(B1, B2, @center)),
    send(P, open).
```

Das Beispiel erstellt ein *picture*-Objekt und fügt diesem zwei Rechtecke hinzu. Diesem fügt es eine Gruppe von *recogniser*-Objekten hinzu. Ein *recogniser* ist eine Klasse zur Bearbeitung von Benutzer-Ereignissen. Hier wird damit das Verschieben und das Ändern der Größe der Rechtecke ermöglicht. Mit dem *@center*-Objekt wird ein *constraint* angelegt, welches dafür sorgt, dass die beiden Rechtecke sich immer an der selben Position befinden.

Wie auch in diesem Beispiel wird in dem Simulator mittels *constraint* und *recogniser* dafür gesorgt, dass die Texte sich immer in der Mitte der Kreise befinden. Außerdem wird zu jedem Kreis an der gleichen Position noch ein gestrichelter Kreis gezeichnet. Der wird allerdings über allen Objekten gezeichnet. Somit ist die gestrichelte Linie des Kreises auch sichtbar, wenn sie von anderen Kreisen verdeckt wird, da der gestrichelte Kreis über dem Kreis gezeichnet wird, der den eigentlichen Kreis verdeckt.

Um sicherzustellen, dass der gestrichelte und der gefüllte Kreis immer genau übereinander liegen, wurde mit *constraint* zusätzlich zu der Position auch die Größe gleichgesetzt. Dies erreicht man mit *identity(area)*.

Nun musste noch ausgewertet werden, wenn ein Objekte doppelt angeklickt wurde oder verändert wurde. Dazu wurden die entsprechenden *recogniser*-Klassen erweitert und bei Auftreten eines Events zusätzliche Aktionen ausgeführt.

Beispiel 5.10: Ein *recogniser* zur Größenänderung.

```
:- pce_begin_class(change_resize_gesture, resize_gesture).

     drag(R, Ev:event):->
     (
          send(R, send_super, drag, Ev),
          gui_changed(Ev?receiver)
     ).

:- pce_end_class.
```

Als weitere Besonderheit wurde die *timer*-Klasse von XPCE benutzt. Im Konstruktor übergibt man das Intervall. Dies kann auch mit der Methode *intervall* realisiert werden. Nach Erreichen der Zeit wird ein Event ausgelöst, der mit einem *message*-Objekt auswertbar ist.

Der Zeitgeber wurde verwendet, um die Framezeiten zu bestimmen und nach einer fest definierten Zeit den Zustand der Objekte zu erhalten und bei Bedarf an die Wissensbasis weiterzuleiten.

Die Daten der Kreise werden mit *assert* und *retract* als Fakten aufgenommen. Diese werden beim Aufrufen von *send_circles* aufbereitet und an die Wissensbasis geschickt. Dabei werden die Kreise zuerst nach der Tiefe sortiert und im Anschluss werden komplett verdeckte Kreise entfernt. Diese Kreise werden in der geordneten Reihenfolge mit den drei *bv_pl.dll*-Funktionen *bv_frame_begin*, *bv_frame_end* und *bv_frame_addobj* als Frame in die Wissensbasis aufgenommen.

5.3.4 Wissensbasis

Die Wissensbasis wurde vollständig in Prolog implementiert, ohne Verwendung von XPCE. Das hat den Vorteil, dass sie leicht auf andere Prolog-Systeme portiert werden kann, bzw. auch mit SWI-Prolog-Versionen benutzt werden kann, wo kein XPCE vorhanden ist. Die Datei *wiba.pl* ist das Kernmodul der Wissensbasis. Eine sehr detaillierte Beschreibung des Quellcodes ist leider nicht möglich, da er viele hundert Zeilen Quellcode umfasst. Es werden daher nur die wichtigsten Regeln und Fakten diskutiert.

Die Wissensbasis speichert die Daten in Datenbanken. Jede Datenbank ist eigenständig und damit existieren keine globalen Daten, die mehrere Datenbanken gleichzeitig nutzen.

Der Vorteil von mehreren Datenbanken ist, dass eine Bildverarbeitung, bzw. Anwendung zwischen diesen Datenbanken wechseln kann. Eine Szene könnte beispielsweise in mehrere Abschnitte aufgeteilt werden und diese werden dann in jeweils einer Datenbank analysiert. Der zusätzliche Aufwand ist nicht sehr hoch, denn es muss zusätzlich nur der Datenbankname in den Fakten gespeichert werden.

Folgende Datenstrukturen sind von Bedeutung:

1. db(Datenbank)

   Eine Datenbank wird hiermit registriert.

2. config(Datenbank, Parametername, Wert)

   Die Parameter der entsprechenden Datenbank werden hiermit festgelegt.

   Derzeit existieren folgenden Parameter:

   unit_size: ist der Umrechnungsfaktor, wieviel Meter einem Pixel entsprechen. Der Standardwert ist $0.001m/Pixel$.

   max_speed: ist die Maximalgeschwindigkeit, mit der sich die Objekte bewegen dürfen (in m/s).
   Der Standardwert ist $0.5m/s$.

   max_acceleration: ist die Maximalbeschleunigung der Objekte (in ms^{-2}). (wird derzeit noch nicht ausgewertet)

   history_size: ist die Größe der History. (wird derzeit noch nicht ausgewertet)

3. obj(Datenbank, ObjektID, [Typ, Klasse, Name, Objektdaten])

   Beschreibt ein Objekt.

4. loc(Datenbank, LocationID, ElternlocationID, Position, Zeit)

   Beschreibt eine Location.

Datenbank: ist der Name der Datenbank, zu dem diese Datenstruktur zugeordnet ist.

LocationID: ist eine fortlaufend vergebene Nummer, die die Location eindeutig macht.

ElternlocationID: ist die LocationID der übergeordneten Location. Genauer gesagt, unter welcher Location sich diese Location befindet (vollständig verdeckt wird). Ist die Location sichtbar, ist die ID gleich 0. Allerdings sollte *rootid(Root)* die bessere Wahl sein, um das System flexibel zu halten.

```
loc(default,1,_,0,_,_)
```

ist somit identisch mit

```
rootid(Root),
loc(default,1,_,Root,_,_)
```

Letzteres ist nicht nur flexibler, sondern auch übersichtlicher.

Position: ist die [X,Y] Position. Entweder ist es die Position, an der die Location sichtbar ist, also der Schwerpunkt des Kreisobjektes, oder wenn die Location nicht sichtbar ist, ist diese Position die Eintrittsposition, an der die Location unter der anderen Location verschwunden ist.

Zeit: ist der Zeitstempel, an der die Location zuletzt sichtbar war.

5. obj2loc(Datenbank, ObjectID, Location)

   Beschreibt die Zuordnung von einem Objekt zu einer Location, also die Verknüpfung zwischen Objekten und Locations.

6. rel(Datenbank, Typ, LID1, LID2, Relation)

   Beschreibt die Relationen zwischen zwei Locations.

   Datenbank: ist der Name der Datenbank, zu dem diese Datenstruktur zugeordnet ist.

   Typ: ist der Typ der Relation. Es wurde *rcc5* für die RCC5-Relation und *depth* für die Tiefenrelation verwendet.

   LID1: ist die erste Location.

   LID2: ist die zweite Location.

Relation: ist eine (Basis-)Relation zwischen den beiden Locations.

Diese Fakten werden mit *assert* und *retract* verwaltet. Ein leichter Zugriff auf die Fakten ist somit möglich.

Datenbankverwaltung Um die Datenbanken zu verwalten, stehen einige Regeln zur Verfügung. Diese lauten *db_create/1*, *db_clear/1*, *db_delete/1*, *db_exists/1* und *set_config/3*. Die Namen der Regeln sind selbsterklärend.

Einfügen neuer Daten in die Datenbank Nach welchem Schema die Daten in die Wissensbasis aufgenommen werden, ist im Kapitel Konzeption ausführlich erläutert. Wie bei dem Simulator schon erwähnt, existieren drei Funktionen, um Daten der Wissensbasis zuzuführen:

frame_begin/3: ist die Regel, die zuerst aufgerufen wird, wenn ein neuer Frame hinzugefügt wird.

frame_addobj/4: fügt ein Objekt zu dem Frame hinzu.

frame_end/1: schließt einen Frame ab und startet die interne Berechnung und das Übernehmen der Daten in die Wissensbasis.

Die Regel frame_end ist die mit Abstand umfangreichste Regel, wo viele Berechnungen durchgeführt werden. Hier findet genau der Ablauf statt, der im Kapitel Konzeption besprochen wurde. Mögliche Bewegungen werden berechnet, Objekte zugeordnet, sowie Relationen zwischen den Objekten gebildet.

Berechnung der möglichen Bewegungen Die Regel *calc_possible_movements/1* berechnet aus den aktuell in der Datenbank vorhandenen Daten die möglichen Bewegungen der Objekte, wobei die neuen Daten erst einmal ignoriert werden. Falls neue Locations entstehen (ein Objekt kann sich unter ein anderes bewegen), werden diese in die Datenbank integriert. Erst nachdem die Objekte zu bekannten Locations zugeordnet wurden, werden die Locations optimiert und inkonsistente Locations entfernt. Inkonsistente Locations sind Locations, auf denen keine Objekte mehr verweisen.

Objektzuweisung Die Zuweisung der Objekte zu bekannten Locations wird mit der Regel *assign_objects/3* vorgenommen. Außerdem wird bei *optimize_assigns/2* die Zuweisung optimiert, wie es bei der Konzeption dargelegt wurde.

Konsistenz Die Datenbank wird nach jedem Hinzufügen eines Frames auf Konsistenz überprüft. Bei inkonsistenten Daten wird versucht diese zu reparieren und zusätzlich wird eine Fehlermeldung ausgegeben. Diese Inkonsistenzen können nur dann auftreten, wenn einer der definierten Randbedingungen verletzt wurde.

Ein Beispiel einer Inkonsistenz ist, wenn ein Objekt existiert, welches auf keine Location mehr verweist. Das bedeutet, die Wissensbasis kann nicht mehr feststellen, an welcher Position sich das Objekt befindet.

Benachrichtigung der Anwendung Wenn alle Berechnungen erfolgreich abgeschlossen wurden, wird die Anwendung benachrichtigt, dass neue Daten in der Wissensbasis vorhanden sind. Dies erfolgt mit *wiba_frame_callback(DB,Time)*, welches die pl_wiba.dll zur Verfügung stellt und damit den Client benachrichtigt.

pl_bv.dll und pl_wiba.dll

Diese beiden DLL's werden von der Wissensbasis (Prolog) eingeladen und stellen die Server zur Verfügung. Eine DLL verwaltet die Kommunikation zur Bildverarbeitung von der pl_bv.dll zur bv.dll. Die andere DLL verwaltet die Kommunikation zur Anwendung von der pl_wiba.dll zur wiba.dll.

Beide basieren auf der Klasse cServer, die im Abschnitt 5.3.2 schon erläutert wurde. Beim Empfangen eines Kommandos von der Bildverarbeitung oder der Anwendung leitet sie diese an Prolog weiter und übermittelt das Ergebnis an den jeweiligen Client.

Die wichtigsten Methoden sind:

void run()

: Dies ist eine überladene Funktion von cThread, welches auf Datenpakete des Clients wartet, das Kommando extrahiert und an *handle_command* weiterleitet.

void handle_command(int client, int command, int len, char *data)

: Wertet das empfangene Datenpaket (len, data) aus und arbeitet das entsprechende Kommando ab.

void send_succ(int client, int succ, const char *data, int size)

: Übermittelt das Ergebnis an den Client, bestehend aus dem Wahrheitswert, ob die Funktion fehlgeschlagen ist oder nicht, und einem Datenpaket, wo Rückgabewerte gespeichert werden, die für den Client wichtig sind.

5.3.5 Anwendung

Es wurden verschiedene Beispielanwendungen implementiert. Zwei sehr einfache Anwendungen, einmal in C++ und einmal in Prolog, um die Grundfunktionalität zu zeigen. Eine komplexere Anwendung wurde mit Prolog und XPCE realisiert, die die Daten der Wissensbasis auswertet und visuell als Graph darstellt.

Alle Anwendungen basieren auf den DLL's *wiba.dll* bzw. *wiba_pl.dll*, die die Kommunikation zur Wissensbasis ermöglichen. Diese werden im Anschluss genauer erläutert und dann kurz erklärt, wie die Beispielanwendungen umgesetzt wurden.

wiba.dll

Diese Bibliothek stellt die Schnittstelle von einer Anwendung zur Wissensbasis dar. Hierüber können Anfragen an die Wissensbasis gestellt und die Datenbanken verwaltet werden. Die Umsetzung ist ähnlich wie bei der bv.dll. Die DLL verbindet sich unter Verwendung der Client/Server-Klassen mit der pl_wiba.dll.

Sie stellt folgende DLL-Funktionen zur Verfügung:

void wiba_free(char *obj)

Diese Funktion gibt ein Objekt wieder frei, welches die DLL angelegt hat. Dies ist nach dem Aufruf der Funktionen *wiba_get_all_objects*, *wiba_get_all_locations*, *wiba_get_obj* und *wiba_get_loc* notwendig.

bool wiba_connect(const char *type)

Verbindet sich mit der Wissensbasis über das Kommunikationsprotokoll *type*, wo derzeit nur *sharedmem* möglich ist.

Rückgabewert ist der Erfolg.

bool wiba_disconnect()

Schließt eine geöffnete Verbindung zur pl_wiba.dll.

Rückgabewert ist der Erfolg.

bool wiba_select_database(const char *name)

Selektiert eine Datenbank, als aktuelle Datenbank.

Rückgabewert ist der Erfolg.

bool wiba_create_database(const char *name)

Erstellt eine neue Datenbank. Wenn diese schon existiert, wird *false* zurückgegeben.

Rückgabewert ist der Erfolg.

bool wiba_delete_database(const char *name)

Entfernt eine existierende Datenbank. Wenn diese nicht existiert, wird *false* zurückgegeben.

Rückgabewert ist der Erfolg.

bool wiba_clear_database(const char *name)

Entfernt alle Daten aus einer existierenden Datenbank. Die Datenbank und deren Einstellungen werden nicht entfernt.

Rückgabewert ist der Erfolg.

bool wiba_set_config(const char *name, const char *value)

Ändert einen Parameter der aktuellen Datenbank.

name: ist der Name des Parameters, der geändert werden soll.

value: ist der Wert, den der Parameter annehmen soll. Der Wert wird als Zeichenkette übergeben.

Rückgabewert ist der Erfolg.

bool wiba_get_all_objects(int **objlist, int *count)

Anfrage nach allen bekannten ObjektID's.

objlist: ist der Zeiger auf das Array von ID's, das von der Funktion erzeugt wird.

Es ist zu beachten, das dieses mit *wiba_free* wieder freigegeben wird.

count: ist die Anzahl der Elemente im Array.

Rückgabewert ist der Erfolg.

bool wiba_get_all_locations(int **loclist, int *count)

Anfrage nach allen bekannten LocationID's.

loclist: ist der Zeiger auf das Array von ID's, das von der Funktion erzeugt wird.

Es ist zu beachten, das dieses mit *wiba_free* wieder freigegeben wird.

count: ist die Anzahl der Elemente im Array.

Rückgabewert ist der Erfolg.

bool wiba_get_obj(int obj, char **dat)

Anfrage nach den Daten eines Objektes.

obj: ist eine gültige ObjektID, die z.B. vorher mit *wiba_get_all_objects* angefordert wurde.

dat: ist ein Zeiger auf die Zeichenkette, welches die Objektdaten enthält, die die Funktion zurückgibt. Um die einzelnen Daten des Objektes zu erhalten, muss diese Zeichenkette analysiert werden. Der Aufbau entspricht einer Prolog-Liste und wird weiter hinten bei der XPCE-Anwendung erläutert.

Es ist zu beachten, dass diese mit *wiba_free* wieder freigegeben wird.

Rückgabewert ist der Erfolg.

bool wiba_get_loc(int loc, char **dat)

Anfrage nach den Daten einer Location.

loc: ist eine gültige LocationID, die z.B. vorher mit *wiba_get_all_locations* angefordert wurde.

dat: ist ein Zeiger auf die Zeichenkette, welches die Location-Daten enthält, die die Funktion zurückgibt. Um die einzelnen Daten der Location zu erhalten, muss diese Zeichenkette analysiert werden. Der Aufbau entspricht einer Prolog-Liste und wird weiter hinten bei der XPCE-Anwendung erläutert.

Es ist zu beachten, dass diese mit *wiba_free* wieder freigegeben wird.

Rückgabewert ist der Erfolg.

void wiba_set_framecallback(FRAME_CALLBACK callback)

Bestimmt eine Funktion, die von der wiba.dll aufgerufen wird, wenn ein neuer Frame von der Wissensbasis ausgewertet wurde.

Die übergebene Funktion hat folgenden Aufbau:

```
void FAR WINAPI foo(char *dbname, double time);
```

Es wird der Name der Datenbank und der aktuelle Zeitstempel übergeben.

wiba_pl.dll

Diese DLL ist aufgebaut, wie die wiba.dll, nur dass Funktionen zur Verfügung gestellt werden, die von Prolog aus benutzt werden können. Daher wird hier auf eine erneute Auflistung der Funktionen verzichtet.

C++ Anwendung

Es wurde eine Beispielanwendung mit C++ erstellt, welches in der Konsole arbeitet und auf Eingaben vom Benutzer wartet. Diese werden ausgewertet und die entsprechenden Funktionen in der Wissensbasis ausgeführt.

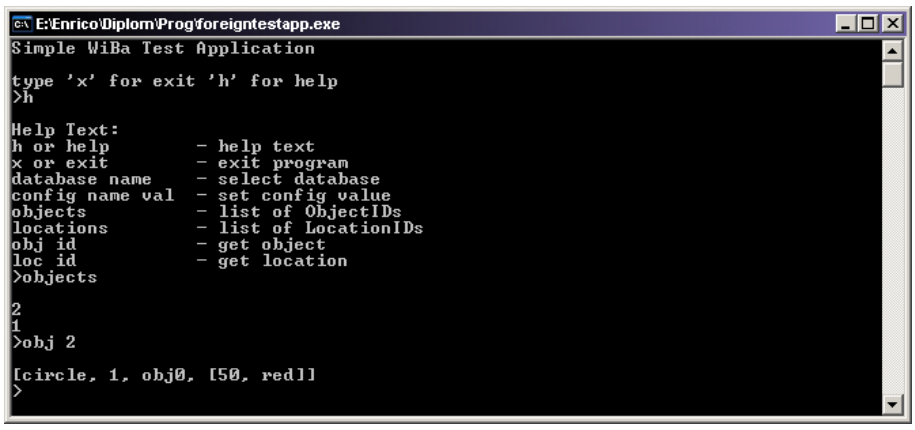


Abbildung 5.6: Beispiel C++ Anwendung

Mit dieser Anwendung kann die Datenbank selektiert, Konfigurationen vorgenommen und Anfragen an die Wissensbasis gestellt werden. Der Quellcode der Beispielanwendung ist sehr kompakt, sodass er leicht nachvollzogen werden kann.

Es wird einfach die *wiba.dll* benutzt, um über die entsprechenden Funktionen auf die Wissensbasis zuzugreifen.

Sehr einfache Prolog-Anwendung

In der Datei *simple_test_app.pl* ist eine minimale Beispielanwendung in Prolog implementiert, um die wesentlichen Teile zur Kommunikation mit der Wissensbasis zu verdeutlichen.

```
% die wiba_pl.dll laden, die die Kommunikation
% zur Wiba herstellt
:-load_foreign_library(wiba_pl),
    % sich mit der Wiba verbinden
    wiba_connect('sharedmem'),
    % Datenbank default auswählen
    wiba_select_database(default),
```

```
    % Als Beispiel, ein Konfigurationsparameter ändern
    wiba_set_config(size,0.002),
    % Liste der Objekte anfordern
    wiba_get_all_objects(A),
    % Resultat ausgeben
    writeln(A).
% wiba_pl.dll wieder entfernen
:-unload_foreign_library(wiba_pl).
```

Prolog-Anwendung mit XPCE-Oberfläche

Eine vollständige Anwendung wurde in der Datei *test_app.pl* implementiert. Dabei wurde Prolog und XPCE benutzt. Die Benutzeroberfläche wird in Abbildung 5.3.5 gezeigt. Auf der linken Seite befinden sich zwei Listen mit den Daten, wie sie von der Wissensbasis kommen. Auf der rechten Seite sieht man die visuelle Darstellung dieser Daten als Graphen mit der Auswahl der aktuellen Darstellung, also welcher Graph gerade angezeigt werden soll. Unten befinden sich diverse Datenbank-Einstellmöglichkeiten.

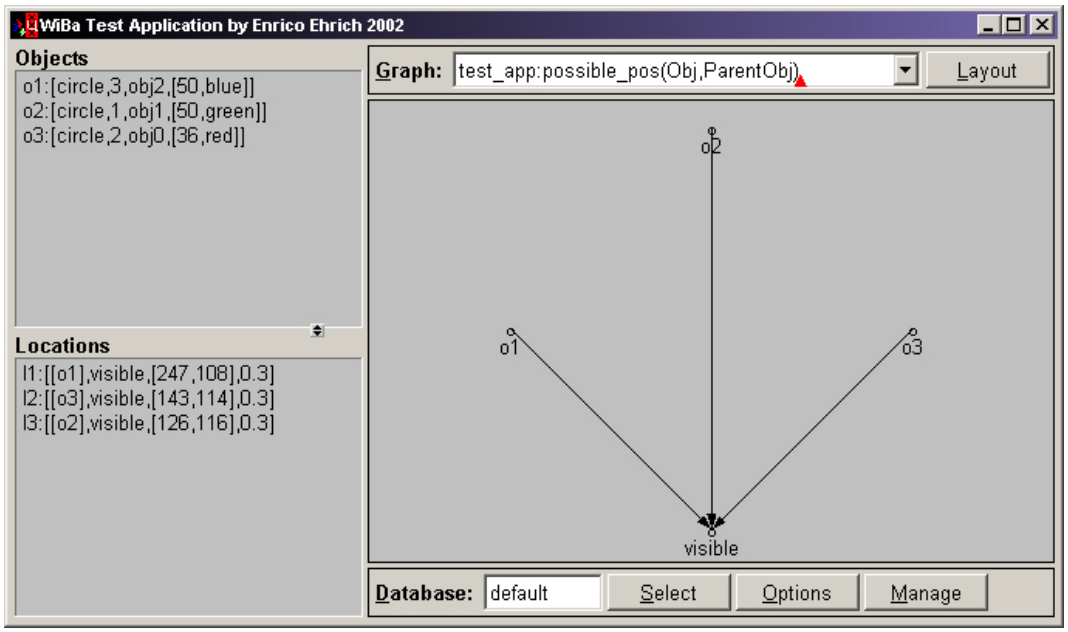


Abbildung 5.7: Prolog Anwendung mit XPCE-Oberfläche

Objects

Beinhaltet die Liste der bekannten Objekte und deren Eigenschaften. Der Aufbau ist dabei folgender:

ObjektID:[Typ, Klasse, Objektname, Objektdaten]

ObjektID wurde in der Anzeige mit dem Präfix o erweitert, damit zu erkennen ist, dass es sich um eine ObjektID handelt.

Locations

Beinhaltet die Liste der bekannten Locations. Der Aufbau ist dabei folgender:

LocationID:[Objektliste, Elternlocation, Position, Zeitstempel]

LocationID: ist die eindeutige ID der Location und wurde in der Anzeige mit dem Präfix l erweitert, damit zu erkennen ist, dass es sich um eine LocationID handelt.

Objektliste: ist eine Liste der Objekte, die sich an dieser Location befinden könnten.

Elternlocation: ist die Location, unter dem sich diese Location befindet. Steht dort *visible*, ist die Location sichtbar und wird nicht von einer anderen vollständig verdeckt.

Position: ist die [X,Y] Position, wo sich die Location befindet. Befindet sich die Location unter einer anderen Location, so ist die Position die (mögliche) Eintrittspositon, an der das Objekt unter dem anderen verschwunden ist. Eine genaue Positionsangabe, wo sich das Objekt befinden kann, ist nicht möglich, da alle Positionen möglich sind, die das Objekt verdecken.

Zeitstempel: ist der Zeitstempel, an dem die Location unter der anderen verschwunden ist. Ist die Location sichtbar, so steht dort der aktuelle Zeitstempel.

Mit Hilfe des Zeitstempels und der Eintrittsposition kann also ermittelt werden, wann und wo ein Objekt unter dem anderen verschwunden ist bzw. an welcher Position sich ein gerade sichtbares Objekt befindet. Mit der Objektliste kann bestimmt werden, welche Objekte sich an einer Location befinden können (Stichwort Verwechslung).

Graph

Ist die Funktion, die den Graph darstellt. Dabei werden zwei Typen unterschieden: Erstens die Funktionen, die die übertragenen Daten der Wissensbasis

visuell darstellen und zweitens Debug-Funktionen, die direkt auf die Wissensbasis zugreifen, was aber nur dann funktioniert, wenn die Wissensbasis und die Anwendung im selben Prolog-Interpreter gestartet wurden. Letzteres dient nur zur Fehlersuche und zur Darstellung interner Datenstrukturen der Wissensbasis.

Layout

Versucht die Knoten des Graphen optimal anzuordnen. Man kann auch die Knoten mit der Maus verschieben, um eine individuelle Darstellung des Graphen zu erhalten.

Select

Aktuelle Datenbank auswählen.

Options

Hier öffnet sich ein Dialog-Fenster, wo diverse Einstellungen getroffen werden können (Abbildung 5.3.5). Einmal kann der Typ der Kommunikation gewählt werden, wobei derzeit nur *sharedmem* möglich ist. Desweiteren können die Parameter der aktuellen Datenbank verändert werden.

Speed: ist die Maximalgeschwindigkeit in m/s.

Acceleration: Maximalbeschleunigung in ms^{-2}. Hat derzeit keine Auswirkung auf die Berechnung.

Historysize: hat derzeit keine Auswirkung auf die Wissensbasis.

Unitsize: ist der Umrechnungsfaktor, wieviel Meter ein Pixel entspricht ($m/Pixel$).

Manage

Es öffnet sich ein Dialog-Fenster mit Kommandos, die die aktuelle Datenbank manipulieren (Abbildung 5.3.5).

Update data from database: es werden die Daten neu von der Wissensbasis angefordert (Objektdaten, Locationdaten).

Clear database: es werden alle Daten aus der Datenbank entfernt.

Create database: es wird eine neue Datenbank erzeugt. Der Name wird aus dem Textfeld entnommen, wo der Datenbankname angegeben wird (im Hauptfenster).

Delete database: die aktuelle Datenbank wird vollständig gelöscht.

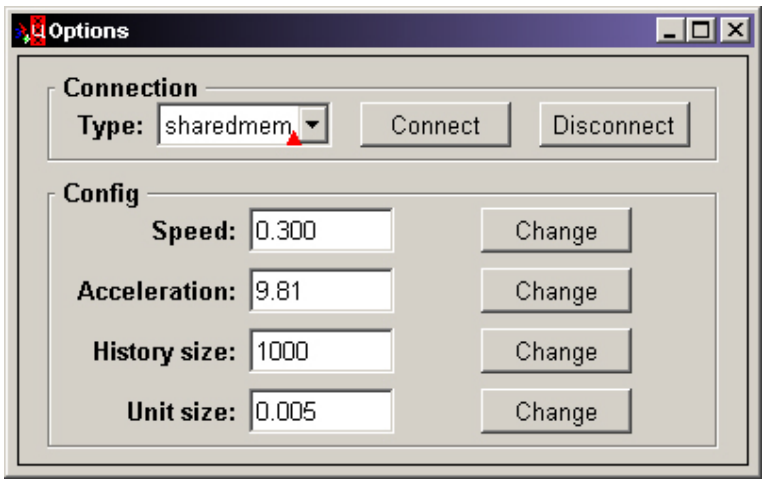


Abbildung 5.8: Die Einstellungsmöglichkeiten der Datenbank und der Kommunikationsverbindung.

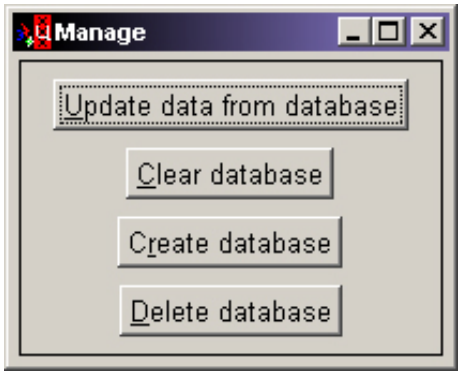


Abbildung 5.9: Kommandos, die die Datenbank manipulieren.

5.4 Zusammenfassung der Umsetzung

In diesem Kapitel wurde detailliert erklärt, wie der Prototyp umgesetzt wurde. Dabei wurde zuerst eine Einführung in die API von SWI-Prolog gegeben, sowie die Oberflächenbibliothek XPCE vorgestellt. Es folgte die Beschreibung der selbst entwickelten Tools-Bibliothek, welches hauptsächlich die Kommunikation zwischen den einzelnen Modulen realisiert. Die Funktion und der Aufbau der Module des Prototypen wurden erläutert. Die einzelnen Module sind: der Simulator, die Wissensbasis und die Anwendung, welche über DLL's miteinander kommunizieren.

Im nächsten Abschnitt wird die gesamte Arbeit nochmals zusammengefasst und Ausblicke auf die weitere Entwicklung gegeben.

Kapitel 6

Fazit und Ausblick

6.1 Zusammenfassung der Arbeit

In dieser Arbeit wurde ein System theoretisch ausgearbeitet und implementiert, welches eine dynamische Szene beobachtet und dabei Objekte verfolgt, die sich verschieben. Diese Objekte können sich gegenseitig überlappen oder ganz verdecken.

Eingangs wurde das Problem analysiert und Einschränkungen getroffen (Randbedingungen). Wegen der Komplexität der Aufgabe wurde im Rahmen dieser Arbeit nur mit Kreisen als Objektformen gearbeitet.

Das System wurde in drei große Module eingeteilt: Bildverarbeitung, Wissensbasis und Anwendung.

In den theoretischen Grundlagen wurden einige schon bestehende Modelle des zeitlichen und räumlichen Schließens vorgestellt und festgelegt, welche für dieses Projekt zur Lösung des Problems zum Einsatz kommen. Dabei wurde auf verschiedene Arten von Beziehungen zwischen Punkten, Intervallen und Regionen eingegangen. So wurde erstens auf die in der Informatik allgemein bekannten Allen-Zeitrelationen und die verschiedenen Arten von RCC-Modellen, die die Beziehungen zwischen Regionen in Bezug auf die Überlappung (Überschneidung) topologisch beschreiben, eingegangen. Desweiteren wurden Modelle erläutert, die anhand von Relationen eine Beschreibung der Orientierung von Punkten zueinander gestattet. Und zum Schluss wurde die Datenstruktur *Szenegraph* vorgestellt, mit dem der Ablauf einer Szene kompakt und sinnvoll gespeichert werden kann. Es wurde das RCC5-Modell verwendet, um Objektbeziehungen zu repräsentieren. Es hat sich aber herausgestellt, dass die vorhanden Modelle nicht ausreichen, um das Problem ausreichend zu lösen. Mehrdeutigkeiten müssen ebenso in das Modell einfließen wie die Reihenfolge der Objekte in der Tiefe.

Dazu wurde eine neues Modell entwickelt, die Tiefenrelation, um Objekte zuein-

ander in der Tiefe zu beschreiben. Außerdem wurde ein Datenstruktur entwickelt, um Mehrdeutigkeiten abzubilden, sowie Algorithmen, um unbekannte Objekte schon bekannten Objekten zuzuweisen und mögliche verdeckte Objektbewegungen zu ermitteln. Es mussten Probleme gelöst werden wie beispielsweise das Gruppieren von Objekten und die Ermittlung der möglichen Objektbewegungen im verdeckten Zustand.

Es wurde ein Prototyp entwickelt. Hauptteile des Systems wurden in Prolog und XPCE entwickelt. Die Kommunikation der einzelnen Module untereinander erfolgt über ein Server/Client-System, welche in DLL's verankert sind. Die Beispielimplementierung benutzt das Protokoll *sharedmem* für die Kommunikation.

Die Bildverarbeitung wurde als Simulator realisiert, welcher eine Benutzeroberfläche besitzt, mit der der Benutzer verschiedene Szenen modellieren kann, sowie vordefinierte Ablaufskripte abspielen kann.

Es wurden verschiedene Beispielanwendungen implementiert. Zwei dieser Anwendungen sind sehr einfach gehalten, um die Grundfunktionalität zu zeigen. Eine weitere Anwensung sit recht komplex. Diese besitzt eine XPCE-Oberfläche und stellt die Daten aus der Wissensbasis als Graphen dar. Diese Anwendung bietet auch die Möglichkeit der Konfiguration der Datenbanken.

Das System, inklusive Quellcode, befindet sich auf der beigefügten CD.

6.2 Ungenauigkeiten und Fehler

Es existieren derzeit noch 2 bekannte Fehler in dem System:

1. In sehr seltenen Fällen geht das System beim Beenden der Kommunikations-DLL's in ein Deadlock-Zustand über. Dies hat zur Folge, dass der Prologinterpreter noch im System verbleibt. Wenn ein weiteres Mal versucht wird, das System zu starten, wird eine Fehlermeldung ausgegeben, die besagt, dass der Server schon existiert. Um ein Weiterarbeiten zu ermöglichen, muss zuvor der Prozess des Interpreters aus dem System entfernt werden.

   Der Fehler konnte noch nicht lokalisiert werden. Das Auftreten dieses Fehlers liegt im Allgemeinen bei unter einem Prozent.

2. Bei vielen Objekten verursacht der Constraint-Propagation-Algorithmus der Tiefenrelation einen Überlauf des Stacks beim Prologinterpreter. Es ist noch nicht genau geklärt, ob dies daran liegt, dass der Algorithmus exponentiell und hoch rekursiv ist oder ob er in speziellen Situationen nicht terminiert. Ersteres ist am wahrscheinlichsten.

Die maximale Anzahl der Objekte liegt bei ca. 12-20 Objekten.

Um das Problem zu lösen, ist entweder der interne Stack zu erhöhen oder ein anderer Algorithmus zu implementieren.

Alle Ungenauigkeiten, die auftreten, sind zugunsten des Systems. D.h. es werden u.U. mehr Positionen vermutet, als eigentlich möglich sind:

1. Durch das Gruppieren von Objekten kann, wegen der vereinfachten Berechnung, ein Übergang zwischen Objekten entstehen, für den angenommen wird, dass ein verdecktes Objekt diesen benutzt, um verdeckt zu einem anderen Objekt zu gelangen (Stichwort Lochproblem).

2. Die Berechnung des kürzesten Pfades erfolgt durch vereinfachte Algorithmen. Dadurch kann der mögliche Pfad kürzer werden, als er sein darf, und somit wird angenommen, dass ein Objekt sich schneller zu einem anderen bewegen kann, als die Randbedingungen es zulassen. Dadurch entstehen u.U. zusätzliche Locations.

6.3 Ausblick

Das System sollte in weiterer Arbeiten verbessert und erweitert werden. Hierfür werden folgende Punkte vorgeschlagen:

- Der BV-Simulator sollte durch eine echte Bildverarbeitung ersetzt werden, welche reale Bilder auswertet und daraus seine Szenebeschreibung erstellt. Der schwierige Teil bei der Umsetzung wird die Rekonstruktion der nicht sichtbaren Objektteile sein und damit verbunden, die zuverlässige Klassifizierung der Objekte. Ein Fehler darf derzeit bei der Klassifizierung nicht auftreten, da die Wissensbasis dieses noch nicht vorsieht.

- Das Einbeziehen der maximalen Beschleunigung der Objekte, um die möglichen Positionen einzuschränken.

- Durch das Zulassen von mehr Objektformen wird das System wesentlich flexibler, allerdings werden damit auch die Algorithmen viel komplexer. Dazu gehört auch die Umsetzung der Schnittstelle zur Übertragung von Objektformen wie es im Abschnitt Konzeption diskutiert wurde.

- Einbinden eines Netzwerk-Protokolls zur Kommunikation der einzelnen Module über mehrere Computer-Systeme.

- Umsetzung (einer abgewandelten Form) des Szenegraphen, um den Ablauf der Szene zu repräsentieren und nicht nur die aktuelle Situation der Szene.

- Das Arbeiten mit Wahrscheinlichkeiten, damit einer Anwendung ermöglicht werden kann, die beste mögliche Position auszuwählen, um eine bestimmte Aktion auszulösen (z.B. ein Roboter bewegt sich zu diesem Punkt).
- Die Voraussage der Objektbewegung. Wohin wird sich das Objekt mit aller Wahrscheinlichkeit bewegen ?

Anhang

Literaturverzeichnis

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Ben94] B. Bennett. Spatial reasoning with propositional logics. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 51–62. Morgan Kaufmann, San Francisco, California, 1994.

[Bit01] T. Bittner. Approximate Qualitative Temporal Reasoning, 2001. `http://citeseer.nj.nec.com/bittner01approximate.html`.

[BK91] P. Bothner and W. M. Kähler. *Programmieren in Prolog*. Fied. Vieweg & Sohn Verlagsgesellschaft mbH, 1991.

[Bre] Brenner. Diplomarbeit. `http://ki.cs.tu-berlin.de/diplomarbeiten/brenner/`.

[Coh96] A. G. Cohn. Calculi for Qualitative Spatial Reasoning. In Jacques Calmet, J. Campbell, and J. Pfalzgraf, editors, *Artificial Intelligence and Symbolic Mathematical Computation*, pages 124–143. Springer-Verlag, Berlin, 1996.

[Cov93] M. A. Covington. ISO Prolog, A Summary of the Draft Proposed Standard, 1993. `ftp://ai.uga.edu/pub/prolog.standard/isoprolog.ps.Z`.

[Fra91] A. U. Frank. Qualitative spatial reasoning with cardinal directions. In *Proceedings of the Austrian Conference on Artificial Intelligence*, pages 157–167, 1991.

[Fre92] C. Freksa. Using orientation information for qualitative spatial reasoning. In A. U. Frank, I. Campari, and U. Formentini, editors, *Proceedings of International Conference on Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*. Springer-Verlag, 1992.

[Fre96] C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1-2):199–227, 1996.

[HSA99] J. Heinsohn and R. Socher-Ambrosius. Zeitliches Wissen. In *Wissensverarbeitung - Eine Einführung*, pages 259–273. Spektum Akademischer Verlag GmbH, 1999.

[IHM] A. Isli, V. Haarslev, and R. Möller. Combining cardinal direction relations and relative orientation relations in Qualitative Spatial Reasoning. `http://kogs-www.informatik.uni-hamburg.de/~haarslev/publications/`.

[NB95] B. Nebel and H. J. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *Journal of the Association for Computing Machinery*, 42:43–66, 1995.

[NNS01] M. Nabil, A. H. H. Ngu, and J. Shepherd. Modeling and retrieval of moving objects. *Multimedia Tools and Applications*, 13(1):35–71, 2001.

[PE97] D. Papadias and M. J. Egenhofer. Algorithms for hierarchical spatial reasoning. *GeoInformatica*, 1(3):251–273, 1997.

[PMT01] D. Papadias, N. Mamoulis, and Y. Theodoridis. Constraint-based processing of multiway spatial joins. *Algorithmica*, 30(2):188–215, 2001.

[Sch93] C. Schlieder. Representing visible locations for qualitative navigation. In P. Carrete and M. Singh, editors, *Qualitative reasoning and decision technologies*, pages 523–532. CIMNE Barcelona, 1993. `http://www.iig.uni-freiburg.de/cognition/members/cs/cs-pub.html`.

[Str98] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 1998.

[Tre97] L. Trevisan. Approximating satisfiable satisfiability problems. *Lecture Notes in Computer Sience*, 1284:473–485, 1997.

Verzeichnisstruktur

Verzeichnisstruktur mit den wichtigsten Projektdateien:

scripts/	- Verzeichnis für die BV-Skripte
src/	- Verzeichnis mit den C++-Dateien
foreign_test_app/	- Die C++ - Applikation
ForeignTestApp.dsp	- VC++ Projekt der C++ TestApp
ForeignTestApp.cpp	- C++ Datei der Test-Applikation
includes/	- Diverse Includedateien
SWI-Prolog.h	- SWI-Prolog API
SWI-Stream.h	- SWI-Prolog Streams
SWI-cpp.h	- SWI-Prolog C++ API
SWI-cpp-ext.h	- Eigene Erweiterungen der C++ API
lib/	- Diverse Link-Bibliotheken
libpl.lib	- SWI-Prolog Link-Library
tools.lib	- Tools-Bibliothek (Release)
toolsD.lib	- Tools-Bibliothek (Debug)
!all.dsw	- VC++ Arbeitsbereich aller Projekte
pldll.h	- xxx_pl.dll/pl_xxx.dll PL-DLL Def.
tools.dsp	- tools.lib Projekt
thread.h	- tools.lib Threads
thread.cpp	- tools.lib die Windows-Version
clientserver.h	- tools.lib Klassen-Definition
clientserver.cpp	- tools.lib die Windows-Version
bv.dsp	- bv.dll Projekt
bv.h	- bv.dll Quellcode
bv.cpp	- bv.dll Quellcode
pl_bv.dsp	- pl_bv.dll Projekt
pl_bv.h	- pl_bv.dll Quellcode
pl_bv.cpp	- pl_bv.dll Quellcode
command_bv.h	- bv.dll/pl_bv.dll Kommandos
bv_pl.dsw	- bv_pl.dll Projekt
bv_pl.cpp	- bv_pl.dll Quellcode
geometric.h	- Geometrische Template-Klassen
depthbuffer.h	- Aussortieren von verdeckten Objekten
depthbuffer.cpp	- Aussortieren von verdeckten Objekten

wiba.dsp	- wiba.dll Projekt
wiba.h	- wiba.dll Quellcode
wiba.cpp	- wiba.dll Quellcode
pl_wiba.dsp	- pl_wiba.dll Projekt
pl_wiba.h	- pl_wiba.dll Quellcode
pl_wiba.cpp	- pl_wiba.dll Quellcode
command_wiba.h	- wiba.dll/pl_wiba.dll Kommandos
wiba_pl.dsw	- wiba_pl.dll Projekt
wiba_pl.cpp	- wiba_pl.dll Quellcode

bv.dll	- bv.dll C-DLL
pl_bv.dll	- pl_bv.dll PL-Master
bv_pl.dll	- bv_pl.dll PL-Slave
wiba.dll	- wiba.dll C-DLL
pl_wiba.dll	- pl_wiba.dll PL-Master
wiba_pl.dll	- wiba_pl.dll PL-Slave
foreigntestapp.exe	- C++ TestApp Exe

start.pl	- Startdatei des Systems
start.bat	- Batchdatei zum starten
util.pl	- Prolog Hilfsmittel
wiba_debug.pl	- Ausgabefenster der WiBa
rcc5.pl	- RCC5-Relationen
depth.pl	- Tiefenrelationen
wiba.pl	- Die Wissensbasis
simulator.pl	- Der BV-Simulator
simple_test_app.pl	- Einfache Prolog Anwendung
test_app.pl	- XPCE Prolog Anwendung

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit von mir selbst und ohne fremde Hilfe verfasst wurde. Alle benutzten Quellen sind im Literaturverzeichnis angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Brandenburg, den ...