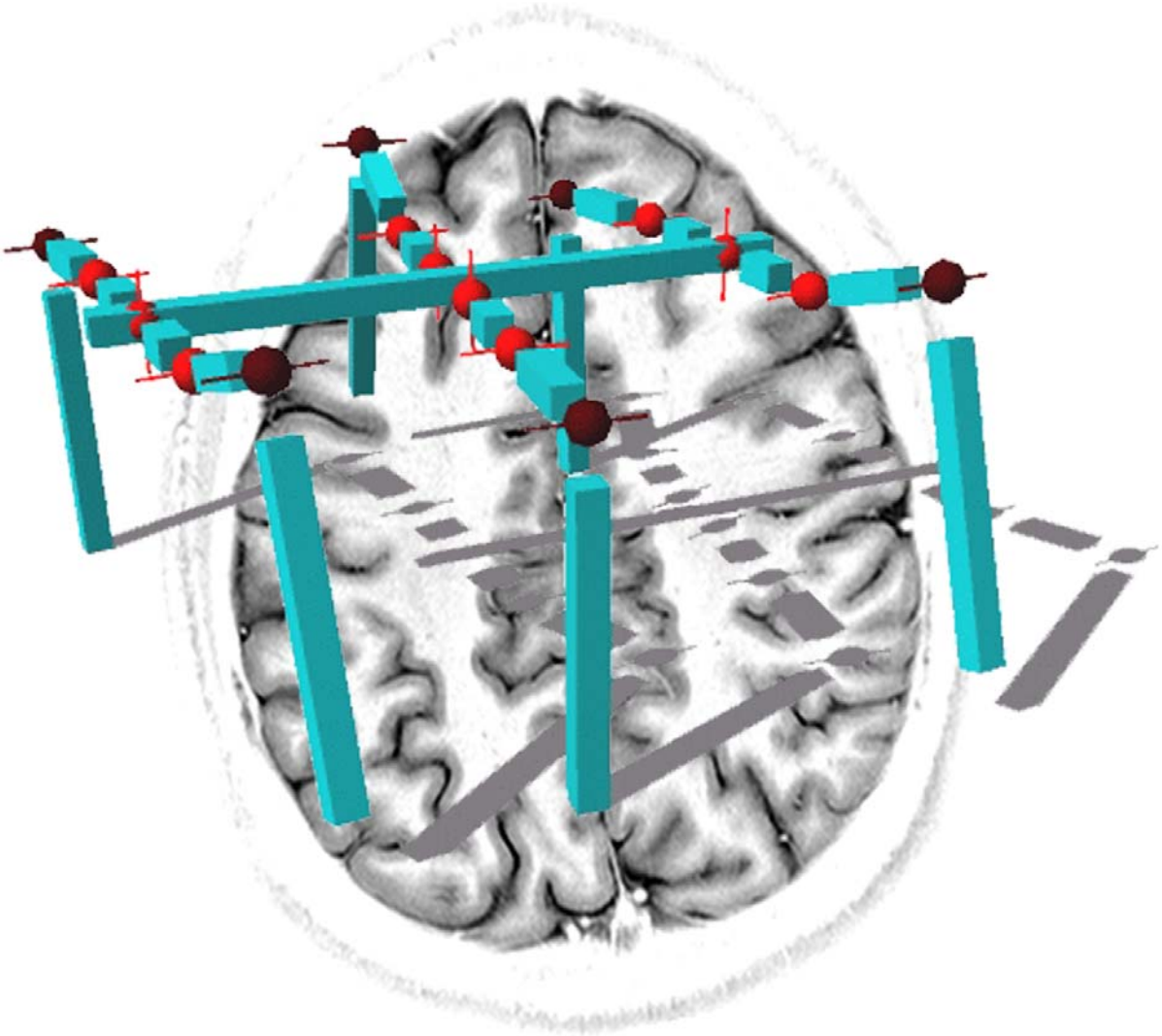
# CreatureBrain

Verhalten und Kognition einfacher Kreaturen



Diplomarbeit
**Ruedi Arnold**
Winter 2001/02

Computer Graphics Lab
Departement Informatik
**ETH Zürich**

Prof. Dr. Markus Gross
Christoph Niederberger

# Zusammenfassung

Diese Diplomarbeit behandelt die Entwicklung von CreatureBrain, einem System zur Simulation von Verhalten und Kognition einfacher Kreaturen. Dieses Modul ist Teil des CreatureZoo Projektes, welches die Simulation von autonomen, rationalen Kreaturen in einer virtuellen Umgebung zum Ziel hat.

Die zentrale Aufgabe des CreatureBrain liegt in der Abbildung von Wahrnehmung auf Verhalten. Die Spezifikation und Umsetzung von gewünschtem Verhalten erweist sich als vielschichtiges Problem. Durch den theoretischen Rahmen der Künstlichen Intelligenz und die Analyse von verwandten Arbeiten können jedoch konkrete Anforderungen an dieses Modul gestellt werden. Für die Verhaltenssteuerung wird ein auf drei Modi aufbauendes Modell präsentiert. Diese sind reflektorisches, zielbasiertes und reaktives Verhalten.

In diesem Bericht wird eine modulare Architektur mit vier Subsystemen für Wahrnehmung, Wahl von Aktionen, Ausführung von Aktionen sowie Rückkopplung und Benutzereingabe beschrieben. Deren Implementierung, die Integration in eine virtuelle, von einer Game Engine modellierten Umgebung, das Aufsetzen auf ein Bewegungssystem sowie die implementierte Logik zur Verhaltenssteuerung einer Beispielkreatur bilden die weiteren Schwerpunkte.

Das vorliegende System ermöglicht Verhaltenssimulation von einfachen Kreaturen. Es wurde wenig verhaltensspezifische Logik integriert. Das Modul ist als Grundgerüst und Prototyp für weitere Arbeiten im Rahmen des Projektes CreatureZoo zu sehen. In einer abschliessenden Diskussion werden Probleme sowie diverse Erweiterungsvorschläge aufgezeigt.

# Abstract

This diploma thesis covers the development of CreatureBrain, a system to simulate the behavior and congnition of simple creatures. It is part of the CreatureZoo project, which aims to simulate autonomous, rational creatures in a virtual environment.

The main task of the CreatureBrain is to map percepts to behavior. The specification and implementation of desired behavior proves to be a complex issue. Using the theoretical framework of artificial intelligence and through an analysis of related work, concrete requirements of this module can be formulated. A model based on three modes is presented for behavior control. These modes are reflex-based, goal-oriented, and reactive.

We describe a modular architecture with separate subsystems for perception, action selection, action execution as well as feedback and user input. Further aspects of this report include our implementation, integration into a virtual environment which is modeled by a game engine, the cooperation with the motorsystem as well as the implementation of the logic to control the behavior of a sample creature.

The presented system is capable of simulating the behavior of simple creatures. There was little behavior-specific logic integrated. This module should be seen as a framework and prototype for further work in the context of the CreatureZoo project. A final discussion exhibits some problems and many proposals for enhancement.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl computer graphics lab

**Diplomarbeit für Herrn Ruedi Arnold**

# Entwicklung eines Systems zur Simulation von Verhalten und Kognition einfacher Kreaturen

## Einleitung

Im Rahmen eines Forschungsprojekts soll ein System erstellt werden, welches zur Simulation und Animation von intelligenten Kreaturen dienen soll. Dieses ist in drei Module aufgeteilt: Rendering / Dynamik, Motion / Control und Verhalten / AI. Diese drei Teile zusammen sollen die Simulation von Kreaturen ermöglichen, welche Ihr Verhalten optimieren und der Umgebung anpassen können.

## Ziel

Im Rahmen dieser Diplomarbeit soll die Komponente zur *Simulation von Verhalten und Künstlicher Intelligenz* implementiert werden. Hierzu sollen verschiedene wissenschaftliche Arbeiten analysiert und kombiniert werden. Das Ziel ist eine selbständige Kreatur, welche auf Grund von Sensoren und Wissen dem Kontext angepasste Aktionen durchführt.

## Aufgabenstellung

- *Einarbeitung* in das Gebiet der Künstlichen Intelligenz und *Design* eines Systems zur Modellierung von Verhalten mit nachfolgenden Eigenschaften.
- Implementierung eines Systems zur *Generierung von Bewegungsabläufen*, welche auf Grund von Intention, Wissen und einem oder mehreren Zielen erzeugt werden.
- Grundlegende Fertigkeiten sollen implementiert werden, um der Kreatur die Möglichkeit zu geben, unter diesen eine geeignete auszuwählen.
- Das System soll die Möglichkeit bieten, grundlegende *Verhaltensmuster und Charaktereigenschaften* zu definieren und skalieren, welche einen Einfluss auf das Verhalten haben (z.B. Aggresivität, Soziales Verhalten, etc.). Diese und ihren Einfluss auf das Verhalten müssen zuerst definiert werden.
- Weiterhin soll das System ein *evolutionäres Wachstum* ermöglichen, welches im Sinne von Lernfähigkeit verstanden werden soll. Dabei soll zwischen grundlegendem Verhalten (“Kleinhirn”, Instinkt) und erlerntem und angepasstem Verhalten (“Grosshirn”, Wissen) unterschieden werden

## Bemerkungen

Ein schriftlicher Bericht und eine mündliche Präsentation schliessen die Arbeit ab. Die Diplomarbeit steht unter der Obhut von Prof. Markus Gross und wird von Christoph Niederberger, Institut für Wissenschaftliches Rechnen, betreut.

Ausgabe: 19. November 2001
Abgabe: 18. März 2002

# Dank

An erster Stelle möchte ich meinen Eltern danken, die mich während meiner Ausbildung immer voll unterstützten. Ohne Euch wäre dies nicht möglich gewesen!

Herzlichen Dank an Nadine für die unzähligen und aufstellenden SMS sowie die schönen Wochenenden.

Ein besonderes Dankeschön geht an meine *sich gut integrierende* WG in der Schweighofstrasse, speziell für die Motivationsversuche und das Tolerieren meiner Vernachlässigung diverser Pflichten... Besonderer Dank gilt Ingo sowie den beiden Hausfreunden Andreas und Tilman für das Korrekturlesen dieses Berichtes.

Bei Nadia Degonda möchte ich mich für das Bild meines Kopfquerschnittes auf der Titelseite bedanken, welches sie mir als MRI-Proband zukommen liess.

Bei Bruno und Adam, meinen beiden Mitstreitern, möchte ich mich für die interessanten und amüsanten Stunden bedanken sowie dafür, dass sie mir gezeigt haben, was gute (Graphik-) Programmierer sind.

Meinen abschliessenden Dank richte ich an Christoph, die andere Hälfte der *wir*-Entscheide in diesem Projekt, der trotz dreifacher Belastung den Humor nicht verlor und bei Bedarf hilfsbereit und kompetent zur Seite stand. Danke für die anregenden Diskussionen, das stundenlange Herumschlagen mit Torque sowie speziell für die Hilfe bei der Erstellung dieses Berichtes. Viel Spass, Glück und Erfolg beim Doktorieren mit dem CreatureZoo!

Zürich im März 2002

rarnold@wherever.ch

# Inhaltsverzeichnis

## Anhang

# 1

# Einführung

In der heutigen Zeit versucht man oft, die Wirklichkeit mitsamt ihren Bewohnern in möglichst realistischen virtuellen Welten zu simulieren. Beispiele in der Forschung gibt es dafür viele, aber auch kommerzielle Applikationen wie *Virtual Reality* Systeme oder Computerspiele können hier aufgezählt werden. Zu einer fortgeschrittenen Nachbildung gehören natürlich auch darin lebende, *autonome* Kreaturen, welche sich sinnvoll und intelligent verhalten. Doch die Simulation solcher Kreaturen, wie auch derer Umgebung, erweist sich als vielschichtiges und komplexes Problem.

## 1.1 Projekt CreatureZoo

Im Rahmen eines Forschungsprojektes mit Namen *CreatureZoo* soll ein System entwickelt werden, welches der Simulation von autonomen, intelligenten Kreaturen dient. Dazu wurde das frisch aufgesetzte Projekt in drei Komponenten aufgeteilt, welche jeweils den Inhalt einer Diplomarbeit bestimmen. Diese drei Teile beinhalten: *Simulation* (CreatureWorld), *Bewegung* (CreatureControl) und *Intelligenz* (CreatureBrain).

### 1.1.1 CreatureWorld

Virtuelle Kreaturen benötigen eine Umgebung, in welcher sie leben, sich fortbewegen und mit der sie interagieren können. Die Grundlage einer realitätsnahen Simulation von autonomen Kreaturen ist eine virtuelle Welt, welche der realen (oder zumindest unserer Wahrnehmung davon) visuell und physikalisch möglichst ähnlich ist. Eine Welt mit diesen Eigenschaften wurde im Rahmen des Projektes CreatureZoo durch das Modul *CreatureWorld* [12] entwickelt. Dessen Implementierung baut auf verfügbaren Systemen auf und kümmert sich nicht nur um die Simulation der künstlichen Umgebung sondern auch um deren Darstellung.

### 1.1.2 CreatureControl

In einer wirklichkeitsnahen Simulation müssen sowohl die physikalische Struktur als auch die Motorik einer Kreatur entsprechend realistisch modelliert werden. Das ist die Aufgabe des Moduls *CreatureControl* [8], welches die Erzeugung und Steuerung der Kreatur durch Befehle auf einer hohen Abstraktionsebene ermöglicht. Die primäre Aufgabe dieses Moduls ist die Umsetzung von Anweisungen des CreatureBrain in Bewegungen, wodurch eine Kreatur in einer virtuellen Welt ihr Verhalten erst manifestieren kann.

### 1.1.3 CreatureBrain

Der entstehende CreatureZoo soll von autonomen Kreaturen bevölkert sein, welche sich selbständig und möglichst natürlich verhalten. Geeignete Systeme zur Steuerung von solchen Agenten werden im Bereich der Künstlichen Intelligenz untersucht. Dieser Bericht über das *CreatureBrain* beschreibt einen möglichen Ansatz dazu.

## 1.2 Inhalt und Positionierung der Diplomarbeit

Die vorliegende Diplomarbeit befasst sich mit der Entwicklung des CreatureBrain und dessen Integration in das Projekt CreatureZoo.

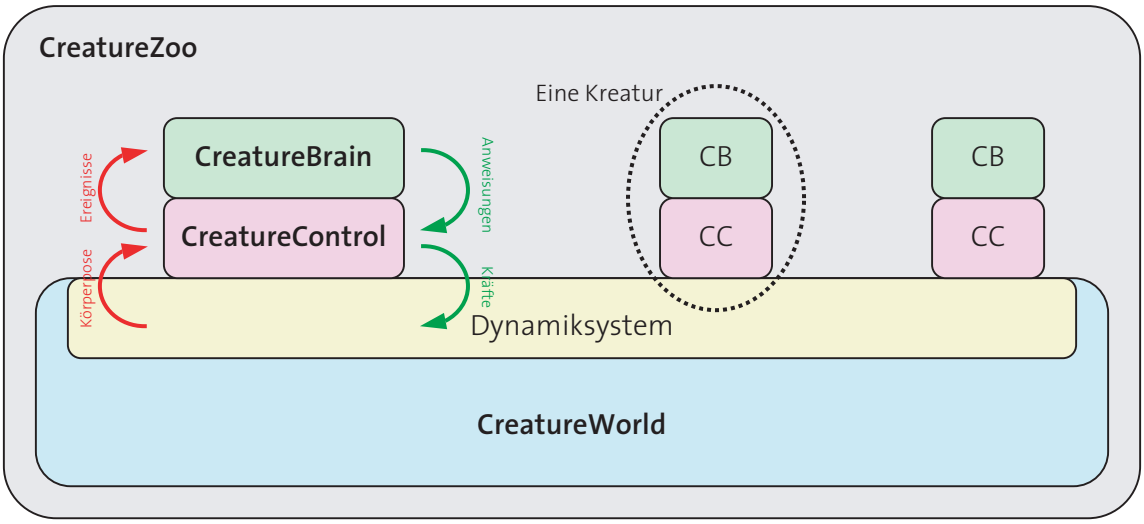



**Abbildung 1.1:** Das Modell des CreatureZoo.

Jede Kreatur im CreatureZoo besteht aus einer Instanz des CreatureBrain und einer zugehörigen Instanz der CreatureControl. Das Verhalten wird durch das CreatureBrain gesteuert. Die CreatureControl führt die vom CreatureBrain übergebenen Bewegungsanweisungen aus. Die tatsächliche Simulation dieser Bewegungen übernimmt das Dynamiksystem der CreatureWorld (Abb. 1.1).

Die Simulation von Kreaturen im CreatureZoo kann durch einen Zyklus dargestellt werden, wie Abb. 1.2 zeigt. Das CreatureBrain entscheidet, beeinflusst durch seine Wahrnehmung der CreatureWorld, über das Verhalten der Kreatur. Dazu nötige Bewegungen werden von der CreatureControl umgesetzt und verändern den Zustand der Welt.

## 1.3 Ablauf der Diplomarbeit

Wie in Abb. 1.3 dargestellt, wurde diese Diplomarbeit in mehrere Phasen unterteilt:

- **Design & Evaluation**. In der ersten Phase wurden durch Erarbeitung eines theoretischen Hintergrunds, Analyse von verwandten Arbeiten sowie durch lange Diskussionen die Anforderungen an das zu erstellende System gefunden und formuliert.
- **Implementierung CreatureBrain**. Das Modul CreatureBrain wurde dann unabhängig von CreatureControl und CreatureWorld implementiert. Dies ermöglichte eine eigenständige Entwicklung und war auch dadurch motiviert, dass die Schnittstellen mit den beiden anderen Modulen noch nicht vollständig definiert werden konnten.

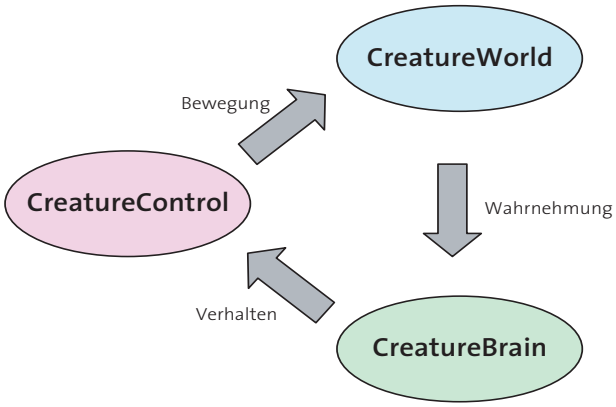



**Abbildung 1.2:** Die Simulation einer Kreatur im CreatureZoo.

- **Integration**. In einer nächsten Phase wurde das erstellte System mit den beiden anderen Komponenten zusammengeführt, wodurch das CreatureBrain eine Umgebung erhielt.
- **Anwendung**. In einem letzten Implementierungsschritt wurde dann das Verhalten, die "Intelligenz" der Kreatur, als Anwendung dieser Architektur verbessert.
- **Bericht/Dokumentation**. Zum Schluss galt es noch, den vorliegenden Bericht sowie weitere Dokumente zur Beschreibung der geleisteten Arbeit zu erstellen.

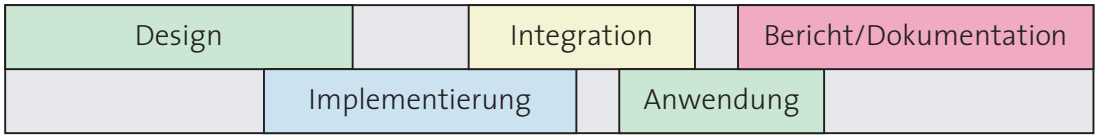



**Abbildung 1.3:** Die Zeitachse der Diplomarbeit.

## 1.4 Aufbau dieses Berichts

Dieser Bericht ist analog zu dem im vorangegangen Abschnitt beschriebenen Ablauf der Diplomarbeit aufgebaut.

Durch eine Einführung in das Gebiet der Künstlichen Intelligenz und die Definition von Begriffen wird im zweiten Kapitel ein theoretischer Rahmen geschaffen. Im dritten Kapitel werden durch Betrachtung von verwandten Arbeiten und Formulierung unserer Vorstellungen die Anforderungen an das Verhalten unserer Kreatur diskutiert und erarbeitet. Die Umsetzung der Anforderung in eine konkrete Architektur wird im vierten Kapitel behandelt. Das Thema des fünften Kapitels stellen einige Facetten unserer Implementierung der Architektur in C++ dar. Auf Aspekte der Integration des CreatureBrain in die CreatureWorld und des Zusammenspiels mit der CreatureControl geht das sechste Kapitel ein. Als Anwendung des CreatureBrain wird im siebten Kapitel die Implementierung von speziellem Wissen und Steuerungslogik zur Verhaltenssteuerung aufgezeigt. Das abschliessende achte Kapitel fasst zusammen, präsentiert eine Diskussion der Resultate, Probleme und mögliche Erweiterungen des CreatureBrain und beendet diesen Bericht.

# 2

# Konzepte & Begriffe

In diesem Kapitel werden Konzepte und Begriffe aus dem Gebiet der Künstlichen Intelligenz eingeführt und erklärt. Dieses Kapitel ist zu einem grossen Teil zusammenfassend aus [14] übernommen.

## 2.1 Warum Künstliche Intelligenz?

Das ultimative Ziel des Feldes *Künstliche Intelligenz* (KI) ist es, intelligente Entitäten zu verstehen und nachzubauen. Ein Grund, warum sich Menschen damit beschäftigen, ist, dass sie dadurch mehr über sich selbst lernen können. Wie kann ein langsames, kleines Hirn eine Welt, die viel grösser und komplexer ist als es selbst, wahrnehmen, verstehen, vorhersagen und manipulieren? Das ist von einer philosophischen Seite betrachtet die wohl grösste Motivation, sich mit KI zu befassen.

KI als Forschungsfeld hat sich in viele Teilbereiche aufgeteilt. Diese erstrecken sich von allgemeinen Bereichen wie Wahrnehmung und logischem Schlussfolgern hin zu spezifischen Aufgaben wie Schach spielen, Beweise mathematischer Theorien, medizinischen Diagnosesystemen oder Textverständnis.

### 2.1.1 Der Begriff "Intelligenz"

Für den Begriff *Intelligenz* gibt es keine einheitliche, allgemein anerkannte Definition. Es existieren jedoch Definitionen aus verschiedenen Disziplinen, wie zum Beispiel Biologie, Kognition, Kontext oder System [11].

Wir verzichten deshalb an dieser Stelle auf eine Definition von Intelligenz, und verwenden in der Folge diesen Begriff nicht mehr, ausser im Namen Künstliche Intelligenz. Der Leser muss also mit seinem eigenen, intuitiven Begriff von Intelligenz auskommen.

### 2.1.2 Verschiedene Definitionen von Künstlicher Intelligenz

Gemäss [14] lassen sich Definitionen von KI in vier Kategorien unterteilen. Dies geschieht durch ein Unterscheiden in den zwei Dimensionen *Denken* (Schlussfolgern) und *Verhalten*. Dabei wird unterschieden, ob der Erfolg von KI an menschlicher Leistung oder an Vernunft gemessen wird. Wird der Mensch als Massstab verwendet, dann versucht KI mehr oder weniger

empirisch, den Menschen nachzuahmen. Ist Vernunft der Massstab, dann müssen alle Entscheidungen rational begründet sein. Zusammengefasst sieht dies wie folgt aus:

| | |
|---|---|
| Systeme, die denken wie Menschen | Systeme, die rational denken |
| Systeme, die handeln wie Menschen | Systeme, die rational handeln |

**Tabelle 2.1:** Vier Definitionen von Künstlicher Intelligenz.

**Handeln wie Menschen: Turing Test**

Der berühmte Test von Alan Turing beabsichtigt, eine Definition von intelligentem Handeln zu geben. Die Idee des Turing Tests ist, dass ein Computer den Test besteht, falls er durch einen Fernschreiber eine Testperson so täuschen kann, dass diese nicht festzustellen vermag, ob sie mit einem Menschen oder mit einer Maschine kommuniziert. Das Ziel von KI ist hier, das Verhalten von Menschen zu simulieren.

**Denken wie Menschen: Erkenntnis-Modellierung**

Die interdisziplinäre Kognitionswissenschaft bringt Computer-Modelle aus der KI mit experimentellen Techniken aus der Psychologie zusammen und versucht, daraus präzise Theorien zu entwickeln, wie menschliche Denk- und Erkenntnisvorgänge funktionieren. Auf diesen Theorien aufbauende Modelle haben somit eine Kognition wie Menschen. Hier versucht KI, den menschlichen Verstand zu imitieren und benutzt ihn als Vorlage.

**Rational Denken: Die Gesetze der Gedanken**

Der griechische Philosoph Aristoteles versuchte als einer der ersten “richtig Denken” als unwiderlegbaren Schlussfolgerungsprozess zu formalisieren. Daraus entstand das, was wir heute Logik nennen. Bei dieser Annäherungsweise an KI geht es darum, Denken und die dazu nötigen Prozesse zu formalisieren.

**Rational Handeln: Rationale Agenten**

Rational Handeln heisst, so zu handeln, dass basierend auf gegebener Überzeugung Ziele erreicht werden. Dazu wird das Konzept des rationalen Agenten verwendet, einer Entität die wahrnimmt und rational handelt. Es ist wichtig zu unterscheiden, dass rationales Handeln nicht gleichzusetzen ist mit rationalem Denken. Rationales Denken kann wohl ein Teil eines rationalen Agenten sein. Es gibt jedoch nicht immer eine korrekte Schlussfolgerung oder eine beweisbar korrekte Handlung, aber der Agent muss trotzdem etwas tun. Andererseits gibt es auch rationale Handlungen, von denen nicht gesagt werden kann, dass sie mit einer logischen Schlussfolgerung verbunden sind. So ist zum Beispiel das rasche, reflexartige Wegziehen der Hand vom heissen Ofen erfolgreicher als eine langsamere Aktion nach sorgfältiger Überlegung.

Diese verschiedenen Ansätze sind in [14] ausführlicher beschrieben, wo der Schwerpunkt auf rationalen Agenten liegt. Dies gilt auch für diese Arbeit. Darum werden diese Ansätze hier erst definiert.

## 2.2 Das Konzept “Agent”

Als *Agent* wird etwas bezeichnet, das Dinge wahrnimmt und aufgrund dieser Wahrnehmung handelt. Diese Wahrnehmung der Umgebung geschieht mittels Sensoren, Handlungen (oder Aktionen) werden mittels Aktoren ausgeführt.

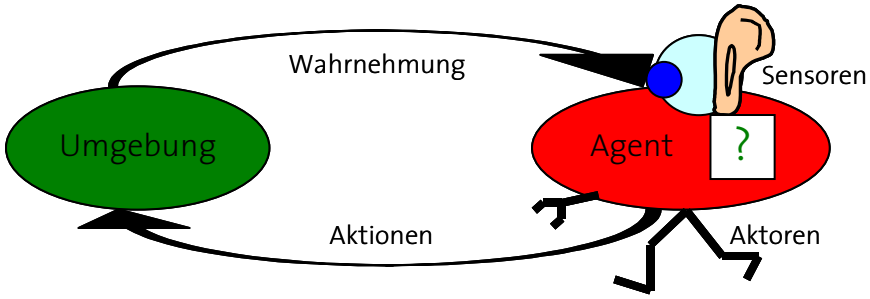



**Abbildung 2.1:** Interaktion des Agenten mit der Umwelt durch Sensoren und Aktoren.

Ein Beispiel zur Illustration: Als Agent betrachtet hat ein Mensch Augen und Ohren und andere Organe als Sensoren; Hände, Beine, der Mund und andere Körperteile gelten dabei als Aktoren.

Wie ein Agent agiert, das ist die interessante Frage in diesem Zusammenhang. Wie wird das Fragezeichen in Abb. 2.1 gefüllt? Verschiedene Arten von Agenten und Möglichkeiten zur Auswahl von Handlungen sind zum Beispiel in [14] detailiert beschrieben, hier werden später in Abschnitt 2.2.6 und Abschnitt 2.2.7 zwei Möglichkeiten eingeführt.

### 2.2.1 Rationale Agenten

Ein *rationaler Agent* ist ein Agent, welcher das tut, was ihm möglichst viel Erfolg bringt. Der Erfolg von Handlungen wird anhand von bestimmten Kriterien gemessen. Diese Kriterien gelten nicht universell, sondern können je nach Agent und Umgebung unterschiedlich aussehen. Wichtig ist, dass diese Kriterien von einer dem Agenten externen Autorität festgelegt werden, damit der Agent die für ihn festgelegten Ziele nicht selber manipuliert. Zudem sollte festgelegt werden, wann der Erfolg eines Agenten gemessen werden soll.

Ein Beispiel: Für einen Staubsauger-Agenten könnte ein einfaches Kriterium die Menge des aufgenommenen Staubs sein. Ein verfeinertes Kriterium könnte auch noch den Stromverbrauch und den Lärm berücksichtigen. Könnte ein Agent seine Erfolgskriterien manipulieren, dann könnte unserer Staubsauger-Agent plötzlich entscheiden, dass die aufgenommene Menge Staub gar nicht mehr sein Erfolgskriterium ist, sondern bloss, den Stromverbrauch tief zu halten, und in der Folge gar nichts mehr tun. Die Bestimmung des Zeitpunkts der Messung des Erfolgs ist notwendig, weil wir zum Beispiel keinen Staubsauger-Agenten wollen, welcher nach einer Stunde viel Staub aufgenommen hat, aber dann nichts mehr tut. Deshalb wollen wir den Erfolg meistens über längere Zeitperioden oder gar die ganze Lebenszeit messen.

Zu unterscheiden gilt es zwischen Vernunft und Allwissenheit. Ein rationaler Agent kann nicht alles wissen. Wenn zum Beispiel ein Mensch über die Strasse läuft und ihm die Türe eines Flugzeugs auf den Kopf fällt und ihn tötet, würden wir trotzdem nicht sagen, dass das Überqueren der Strasse nicht rational war.

Was zu einem bestimmten Zeitpunkt rational ist, hängt also von vier Dingen ab:

- Dem **Erfolgskriterium**, welches den Erfolg der Agenten festlegt,
- allen **Wahrnehmungen**, welche der Agent bisher gemacht hat (**Wahrnehmungs-Sequenz**),
- dem **Wissen** des Agenten über die Umgebung und
- den **Handlungen** (Aktionen), welche dem Agenten zur Verfügung stehen.

Das führt zur Definition des idealen rationalen Agenten.

### 2.2.2 Ideale rationale Agenten

Ein *idealer rationaler Agent* führt für jede mögliche Wahrnehmungs-Sequenz die Handlung aus, welche aufgrund seiner Wahrnehmungs-Sequenz und jeglichem eingebauten Wissen den Erwartungswert der Erfolgs-Messung maximiert.

Diese Definition scheint Handlungen zuzulassen, welche nicht rational sind. Zum Beispiel könnte man annehmen, dass ein Agent, der eine Strasse überquert, ohne auf beide Seiten geschaut zu haben, auch rational handelt. Die Argumentation dazu könnte sein, dass der Agent durch seine Wahrnehmungs-Sequenz nicht wusste, dass von rechts ein Auto kam und deshalb rational gehandelt hat. Diese Interpretation ist aber falsch. Ein Kreuzen der Strasse wäre nicht rational, da das Risiko, dies zu tun ohne dabei auf beiden Seiten zu schauen, zu gross ist. Ein idealer rationaler Agent hätte die Aktion "schauen" ausgeführt, bevor er auf die Strasse getreten wäre, da dies geholfen hätte, seinen erwarteten Erfolg zu maximieren.

### 2.2.3 Autonomität

Die *Autonomität*[1] eines Agenten drückt aus, bis zu welchem Grad das Verhalten eines Agenten durch seine Erfahrung bestimmt werden kann. Autonomität beschreibt also, wie stark sich das Verhalten der Umgebung anpassen kann.

Agenten haben im Allgemeinen eingebautes Wissen. Ist ein Agent nur vom eingebauten Wissen abhängig, dann besitzt er keine Autonomität. Eine Uhr kann zum Beispiel als Agent ohne Autonomität angeschaut werden. Sie braucht keine Wahrnehmung zu berücksichtigen, sie wird ihr Verhalten nicht ändern.

Das Verhalten eines Agenten ist vom eingebauten Wissen und seiner Erfahrung in der aktuellen Umgebung abhängig. Ein Agent ohne eingebautes Wissen ist wohl nicht wünschenswert, da dieser ganz zufälliges Verhalten an den Tag legen würde. Wünschenswert ist also eine Mischung zwischen eingebautem Wissen und Lernfähigkeit, beziehungsweise der Möglichkeit, sich der aktuellen Umgebung anzupassen.

### 2.2.4 Die ideale Abbildung von Wahrnehumgs-Sequenzen auf Aktionen

Da das Verhalten eines Agenten nur von seiner Wahrnehmungs-Sequenz abhängt, kann grundsätzlich jeder Agent durch eine Tabelle beschrieben werden, welche für jede mögliche Wahrnehmungs-Sequenz die passende Aktion enthält. Eine komplette, ideale Abbildungstabelle würde also einen idealen rationalen Agenten beschreiben. Leider lässt sich eine solche Lösung aus folgenden Gründen praktisch nicht umsetzen:

- Die Tabelle würde enorm gross. Für einen schachspielenden Agenten wären bereits rund $35^{100}$ Einträge notwendig.
- Die Erstellung der Tabelle würde viel Zeit beanspruchen.
- Der Agent besitzt keine Autonomität, die Wahl seiner Aktionen ist komplett und fix eingebaut. Wenn sich die Umgebung in unerwarteter Weise ändert, ist der Agent verloren.
- Auch wenn der Agent einen Lernmechanismus hätte, würde es enorm viel Zeit brauchen, bis für jeden Tabelleneintrag der richtige Wert erlernt worden wäre.

1. Autonomität (Autonomy) wird in [14] in diesem starken Sinn verwendet. Der Begriff ist nicht identisch mit dem Begriff von selbstständigem Verhalten, welcher in Abschnitt 3.4.1 eingeführt wird.

### 2.2.5 Die Struktur von Agenten (PAGE)

Bisher wurde nur das Verhalten von Agenten beschrieben, sprich, was für eine Aktion ein Agent als Antwort auf eine bestimmte Wahrnehmungs-Sequenz ausführt. Nun betrachten wir kurz den Aufbau von Agenten.

Die Aufgabe, das Verhalten zu bestimmen, liegt beim Agenten-Programm, einer Funktion welche die Abbildung von Wahrnehmung auf Aktionen vornimmt. Dieses Programm läuft auf irgend einer Art von Ausführungseinheit, dies kann sowohl ein herkömmlicher Computer als auch spezielle Hardware sein. Zusammenfassend lässt sich folgende Formel aufstellen:

Agent = Ausführungseinheit + Programm

Wenn ein Agent implementiert werden soll, ist es wichtig, sich Gedanken über mögliche Wahrnehmungen und Aktionen, über Ziele oder Erfolgs-Kriterien des Agenten und die Umwelt, in der ein Agent agieren wird, zu machen. Diese vier Bereiche werden in [14] mit **PAGE** abgekürzt, für Percepts (Wahrnehmung), Actions (Aktionen), Goals (Ziele) und Environment (Umgebung).

Betrachten wir als Beispiel einen Taxifahrer-Agenten unter diesen PAGE Kriterien:

| Wahrnehmung | Aktionen | Ziele | Umgebung |
|---|---|---|---|
| Kameras, Tachometer, GPS, Mikrophone | steuern, beschleunigen, bremsen, mit dem Passagier sprechen | sichere, schnelle, legale, komfortable Fahrt, hoher Profit | Strassen, Verkehr, Fussgänger, Kunden |

**Tabelle 2.2:** PAGE für einen Taxifahrer-Agenten.

Diese vier Kriterien haben einen sehr starken Einfluss auf die Anforderungen eines Agenten und prägen deshalb massgebend dessen Aufbau. Sie sind deswegen ein geeignetes Mittel um Agenten zu spezifizieren.

**Wahrnehmungen** bestimmen die aktuelle Sicht der Umgebung eines Agenten. Wahrnehmung kann zum Beispiel *diskret*, *kontinuierlich* sowie mit *Unsicherheit* behaftet sein. Dies hängt stark von der aktuellen Umgebung ab, und ein Agent sollte passend damit umgehen können. Die Sensoren müssen die benötigten Informationen beschaffen können, der Agent muss eine Schnittstelle zu benötigten Informationsquellen haben und von dort erhaltene Informationen auch interpretieren können.

**Aktionen** geben die möglichen Handlungen eines Agenten vor. Die Aktoren eines Agenten müssen in der Lage sein, diese Aktionen gemäss ihrer Spezifikation auszuführen. Aktionen werden aufgrund von Wahrnehmungen und Zielen ausgewählt.

**Ziele** bestimmen das Verhalten eines Agenten. Die zu erreichenden Ziele geben vor, welche Aktionen für einen Agenten in der aktuellen Umgebung angebracht sind. Die vorgegebenen Ziele bestimmen letztendlich, ob das Verhalten eines Agenten rational ist oder nicht. Ein idealer rationaler Agent wählt dabei immer die beste Aktion aus, der Erfolg wird an seinen Zielen gemessen.

Die **Umgebung** eines Agenten kann verschiedene Eigenschaften haben und dadurch einen Agenten prägen. Wenn die Sensoren Zugriff auf den gesamten Zustand der Umgebung haben, beschreibt man diese als *zugänglich*. Zugänglichkeit ist praktisch, denn dadurch muss der Agent zum Beispiel nicht intern den Zustand der Welt festhalten. Wenn der nächste Zustand der Umgebung nur vom aktuellen Zustand und der gewählten Aktionen abhängt, dann wird die

Umgebung *deterministisch* genannt. Wenn sich die Umgebung während der Nachdenkzeit des Agenten verändert, dann sagt man, die Umgebung sei *dynamisch*. In einer nicht-deterministischen oder dynamischen Umgebung sollte ein Agent Annahmen über die Effekte seiner Aktionen und die Veränderung der Umgebung treffen.

Um nun das Konzept des Agenten-Programms ein bisschen klarer zu machen, werden zwei Beispiele kurz eingeführt: Der einfache Reflex-Agent und der zielbasierte Agent.

### 2.2.6 Reflex-Agenten

Eine erste und einfache Implementierung eines Agenten-Programms ist ein Reflex-Agent. Dieser entscheidet über die auszuführende Aktion aufgrund von Bedingungs-Aktions-Regeln nach folgendem Schema:

**Wenn** Bedingung X erfüllt ist, **dann** führe Aktion Y aus.

Beim Taxifahrer-Agent könnte so eine Regel wie folgt aussehen: Wenn die Bremslichter des voranfahrenden Autos aufleuchten, dann bremse ebenfalls. Dieser einfache Reflex-Agent funktioniert nur, wenn über die auszuführende Aktion aufgrund der aktuellen Wahrnehmung entschieden werden kann. Als erste Erweiterung kann der Reflex-Agent einen internen Zustand halten und sich Dinge merken. So könnte der Taxifahrer-Agent zum Beispiel die Bremse nur schwach betätigen, wenn er davor durch einen Blick in den Rückspiegel festgestellt hat, dass dicht hinter ihm ein weiteres Fahrzeug folgt.

Ein Reflex-Agent entscheidet also immer bloss aufgrund des Zustands der Umgebung über sein Verhalten. Natürlich funktioniert ein Taxifahrer-Agent nicht bloss durch solch einfache Regeln, aber sie können sehr wohl einen Teil seines Verhaltens modellieren.

### 2.2.7 Zielbasierter Agent

Wissen über den Zustand der Umgebung genügt nicht immer, um über die auszuführende Aktion zu entscheiden. Unser Taxi kann an einer Kreuzung nach rechts, links oder geradeaus fahren. Die richtige Entscheidung ist davon abhängig, wohin das Taxi will. Also brauchen wir zusätzlich zur Information über die Umgebung noch eine Zielangabe. Bei der Abbildung von Wahrnehmungen auf Aktionen werden dann bei einem zielbasierten Agenten auch noch allfällig vorhandene Ziele berücksichtigt. Was genau ein Ziel ist und wie es erreicht werden kann, wird im Abschnitt 2.3 erklärt.

### 2.2.8 Wissensrepräsentation

Damit ein Agent Wissen über sich, die Welt und seine Aufgabe halten kann, muss eine geeignete Form zur Wissenrepräsentierung gefunden werden. Eine solche Sprache zur Beschreibung von Wissen sollte ausdrucksstark, präzise, kompakt, eindeutig, kontext-unabhängig und effektiv sein. Eine Wissensbasis sollte klar und korrekt sein. Wichtige Beziehungen sollen definiert sein, irrelevante Details sollen unerwähnt bleiben. Um für einen Agenten eine Wissensbasis zu erstellen, müssen immer Kompromisse eingegangen werden. Um Wissen zu beschreiben wird in der KI oft Logik erster Ordnung verwendet, auch Prädikatenkalkül genannt. Wir wollen hier nicht genauer darauf eingehen, für eine Einführung sei wiederum auf [14] verwiesen. Der Aufbau einer passenden Wissenbasis für einen Agenten ist kein einfaches Unterfangen und geschieht meist in einem iterativen Prozess.

## 2.3 Ziel und Plan

Bisher wurde der Begriff Ziel in diesem Kapitel mehrmals benutzt. An dieser Stelle wird nun eine Definition für diesen und einen weiteren Begriff gegeben:

- Ein *Ziel* beschreibt einen zu erreichenden Zustand.
- Ein *Plan* ist eine Sequenz von Aktionen durch deren Ausführung ein Ziel-Zustand erreicht werden soll.

Nehmen wir zum Beispiel an, unser Ziel ist es, Milch und Bananen zu haben. Ein möglicher Plan dazu könnte sein, in den Supermarkt zu gehen und dort Milch und Bananen zu kaufen.

Um Ziele und Pläne formal beschreiben zu können, ist eine formale Notation der Zustände und Aktionen notwendig. Die in der KI klassisch für diesen Zweck verwendete Sprache ist STRIPS [5] oder eine Erweiterung davon. Das Situationenkalkül (Situation Calculus) ist eine Notation, welche Veränderung in Logik erster Ordnung beschreibt. STRIPS basiert darauf und behält viel von dessen Ausdruckskraft, ermöglicht aber trotzdem effiziente Planungsalgorithmen. Für eine formale Einführung wendet man sich zum Beispiel an [14].

In STRIPS würde der Zustand zum obigen Beispiel etwa durch folgende Literale[1] festgehalten:

$$sein(zuhause) \wedge haben(Geld) \wedge \neg haben(Milch) \wedge \neg haben(Bananen) \wedge \ldots$$

Ein Ziel wird ebenfalls durch Literale ausgedrückt:

$$sein(zuhause) \wedge haben(Milch) \wedge haben(Bananen)$$

Aktionen werden in STRIPS durch Operatoren repräsentiert. Ein STRIPS Operator besteht aus drei Komponenten:

- Einer *Aktionsbeschreibung*, die angibt, was der Agent an die Umgebung liefert,
- einer *Vorbedingung*, welche angibt, was zutreffen muss, damit der Operator angewandt werden kann und
- einem *Effekt*, der beschreibt, wie sich der Zustand verändert, wenn der Operator angewandt wird.

Ein Operator wird in [14] als Diagramm wie folgt dargestellt:

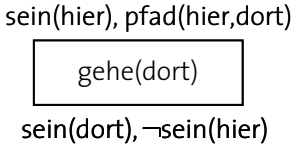



**Abbildung 2.2:** Schematische Darstellung des gehe-Operators. Die Vorbedingungen erscheinen oberhalb, die Effekte unterhalb.

Durch einen Planungsvorgang wird dann aus diesen Angaben ein Plan, eine Sequenz von Operatoren ermittelt, welcher den vorgegebenen Ziel-Zustand erreicht. Dieser Planungsvorgang stellt hier die eigentliche Herausforderung dar.

1. Für eine Definition von Literalen verwende man die angegebene Literatur. Dieses Beispiel soll einen kurzen, nicht-formalen Einblick in STRIPS vermitteln.

Planen entspricht grundsätzlich einer Suche, es sind also verschiedene Strategien (z.B. Tiefensuche, Breitensuche, iterative Tiefensuche) möglich. In komplexen Umgebungen ist es jedoch praktisch nicht möglich, den Raum aller Zustände in geeigneter Zeit zu durchsuchen. Es kann jedoch der Raum der Pläne durchsucht werden, wobei mit einem minimalen Plan gestartet und dieser dann nach und nach zu einer Lösung ausgebaut wird. In der KI ist zu diesem Thema ein breites Repertoire an Wissen erarbeitet worden. Für die STRIPS Sprache ist zum Beispiel POP (Partial-Order Planning) ein korrekter und kompletter Planungsalgorithmus.

## 2.4 Lernfähigkeit

Die Idee von Lernen ist, dass Wahrnehmung nicht bloss zum Handeln benutzt wird, sondern auch dafür, die Fähigkeit zu Handeln in der Zukunft zu verbessern. Lernen findet als Resultat eines Zusammenspiels von Agent und Welt und von Beobachtungen des Agenten über seinen eigenen Entscheidungsprozess statt. Lernen kann eine einfache Memorierung von Erfahrungen sein oder ein Agent kann zum Beispiel sein gesamtes Verhalten erlernen. Lernen ist ein vielseitiger Begriff.

Ein Agent mit Lernfähigkeit besitzt Autonomität gemäss der Definition in Abschnitt 2.2.3. In der KI fällt Lernen unter das Teilgebiet Maschinen Lernen (machine learning).

Bei der Beschreibung von Lernen ist es wichtig, sich über einige Punkte im Klaren zu sein:

- **Welche Komponente eines Agenten soll etwas lernen?**
  Ein Agent vollbringt durch verschiedene Komponenten verschiedene Aufgaben, wie zum Beispiel: eine direkte Abbildung vom aktuellen Zustand auf Aktionen, Schlussfolgerungen ziehen über die Umgebung, Wissen haben über die Entwicklung der Welt oder die Auswirkungen seiner Aktionen. All diese Bereiche könnten durch Lernen beitragen, dem Agenten in Zukunft besseres Handeln zu ermöglichen.

- **Was für Rückmeldungen sind verfügbar?**
  Wenn die Eingaben und Ausgaben einer Komponente verfügbar sind, dann spricht man von *überwachtem Lernen* (supervised Learning). Einer Komponente für die Vorhersage von Effekten einer Aktion sagt die Rückmeldung über ausgeführte Aktionen, was die richtigen Effekte einer Aktion sind. Wenn zum Beispiel ein Agent für eine bestimmte Aktion (“bremsen”) einen bestimmten Effekt (“halten nach 7 Metern”) vorhersagt, und die Wahrnehmung sogleich den richtigen Effekt (“halten nach 12 Metern”) mitteilt, ist dadurch für diesen Agenten überwachtes Lernen möglich.
  Wenn jedoch Rückmeldung über eine Aktion kommt (“saftige Busse für einen Auffahrunfall”), aber die richtige Aktion (“früher und behutsamer bremsen”) nicht mitgeteilt wird, dann spricht man von *Reinforcement Learning*. Dadurch können Bedingungen zum Ausführen von Aktionen gelernt werden.
  Lernen ohne Hinweis über die Effekte nennt man *unüberwachtes Lernen* (unsupervised Learning).

- **Soll Vorwissen mit einbezogen werden?**
  Die meisten Forschungsarbeiten in KI und Psychologie verfolgen den Fall, dass Agenten ohne Vorwissen aufgrund von Erfahrungen bei Null zu Lernen beginnen. Dies ist ein wichtiger Spezialfall, aber sicher nicht der allgemeine Fall. Viel menschliches Lernen zum Beispiel findet im Kontext von Hintergrundwissen statt. Wie dies beim Menschen genau vor sich geht, ist noch nicht bekannt, aber sicher ist, dass Vorwissen beim Lernen enorm hilfreich sein kann.

### 2.4.1 Lernansätze in der Künstlichen Intelligenz

In der KI werden verschiedene Ansätze verfolgt, die das Lernen ermöglichen sollen. *Induktives Lernen* ist überwachtes Lernen, wobei für eine zu lernende Funktion zu gewissen Eingaben die korrekte Ausgabe bekannt ist. Aufgrund dieser Rückmeldungen kann dann die Repräsentation der Funktion angepasst und dadurch laufend verbessert werden.

*Neuronale Netze* sind der Funktionsweise des Gehirns nachempfunden. Sie bestehen aus vielen einzelnen Knoten, welche untereinander verbunden sind, parallel arbeiten und keine zentrale Kontrolleinheit haben. Einige Knoten sind mit der Aussenwelt verbunden und werden als Eingabe, respektive Ausgabe, angesehen. Lernen geschieht dann typischerweise durch eine Anpassung der Gewichte der Verbindungen aufgrund von Übungsdaten.

*Genetische Algorithmen* erreichen Verstärkungslernen indem sie die Verstärkung dazu benutzen, den Anteil von erfolgreichen Funktionen in einer Population von Programmen zu erhöhen. Bei diesem der biologischen Evolution nachempfundenen Prozess werden schrittweise aufgrund einer Tauglichkeitsfunktion als erfolgreich eingestufte Funktionen untereinander durch Rekombination und Mutationen zu neuen Funktionen gemischt. Dies wird generationenweise solange durchgeführt, bis eine Funktion bestimmten Anforderungen entspricht.

## 2.5 Zusammenfassung

In diesem Kapitel wurde das Gebiet, in welchem wir uns vom Standpunkt der KI aus gesehen bewegen, dargestellt. Es wurden verschiedene Konzepte und Ansätze der KI kurz eingeführt. Für eine weiterreichende Einführung wendet man sich zum Beispiel an [14], wo ein verständlicher, breiter, umfassender und übersichtlicher Einblick in das Gebiet der Künstlichen Intelligenz geboten wird.

# 3

# Verwandte Arbeiten und Diskussion

Dieses Kapitel zeigt auf, wie sich das CreatureBrain entwickelt hat. Anhand unserer Ansprüche und der Analyse von verwandten Arbeiten werden die Anforderungen an dieses System erarbeitet, und das gewünschte Verhalten unserer Kreatur allgemein, sowie in konkreten Szenarien beschrieben.

## 3.1 Einführung

Verhalten und Kognition sind, wie das ganze Gebiet der Künstlichen Intelligenz, sehr weitläufige Themen. Wie in der Einleitung beschrieben, steht CreatureZoo am Anfang eines neuen Projektes und deshalb hatten wir keine Erfahrung mit der Simulation von Verhalten und Kognition einfacher Kreaturen. Aus diesem Grund mussten zuerst die Möglichkeiten, Anforderungen, Wünsche und Szenarien gefunden und spezifiziert werden, denen unser System gerecht werden sollte. Um dies adäquat behandeln zu können, braucht es Grundlagen und Wissen, um abschätzen zu können, was möglich ist und um herauszufinden, wie die aktuelle Forschung an dieses Thema herangeht. Aus diesem Grund werden in diesem Kapitel nach der Besprechung des in Abschnitt 2.2.5 beschriebenen PAGE für unseren Agenten zuerst einige für uns wichtige Merkmale aus verschiedenen geleisteten Arbeiten diskutiert. In einem letzten Teil werden dann unsere Ideen und Szenarien dargestellt, welche sich aus unserer Beschäftigung mit verwandten Arbeiten entwickelten und welche die Anforderungen an unser System definieren werden.

### 3.1.1 Rahmenbedingungen

Das System soll auf ein Modul, welches die Geometrie und die Bewegungen einer virtuellen Kreatur in einer dreidimensionalen Welt kontrolliert, aufgesetzt werden. Damit ist klar, dass das CreatureBrain System auf irgendeine Art mit einem Motoriksystem kommunizieren muss. Die Bewegungen, welche von der Motorik aufgrund von Anweisungen des CreatureBrain ausgeführt werden, visualisieren das Verhalten unserer Kreatur und sollen sinnvoll erscheinen. Dazu wird Information über die Umgebung, in welcher die Kreatur lebt, notwendig sein, so dass die Kreatur dies bei der Wahl ihres Verhaltens mit einbeziehen kann. Die Kreatur wird also auch mit ihrer Welt in Verbindung stehen, um diese in ihren Entscheidungen über das Verhalten berücksichtigen zu können.

Vom Standpunkt der KI aus wird das zu erstellende System als rationaler Agent betrachtet, was für unser System eine Rahmenstruktur vorgibt. Agenten werden in Abschnitt 2.2 beschrieben. Doch das Verhalten eines Agenten[1] wird erst durch dessen Programm festgelegt. Die Wahl des Programmes für einen Agenten ist eine nicht-triviale Aufgabe, wenn dieser ein interessantes und vielseitiges Verhalten zeigen soll. Denn dieses muss zuerst einmal spezifiziert werden. Und wie man dabei vorgeht und dies in ein Agentenprogramm umsetzt ist a priori nicht klar ersichtlich.

Vage stellten wir uns zu Beginn eine virtuelle Welt vor, in welcher unsere Kreatur herumlaufen kann und bestimmtes, wenn möglich rationales Verhalten zeigt. Aber wie könnte solch rationales Verhalten aussehen? Was kennt die Kreatur für Bewegungsabläufe? Wie selbständig ist die Kreatur? Kann die Kreatur gesteuert werden? Wie? Können wir die Kreatur steuern? Was kann oder soll die Kreatur über die Welt wissen? Ist die Kreatur lernfähig? Kann sich die Kreatur ihrer Umgebung anpassen?

Um zu diesen und ähnlichen Fragen Antworten zu finden oder zumindest ihren Umfang und ihre Komplexität abschätzen zu können, haben wir uns mit verschiedenen verwandten Arbeiten beschäftigt, um einen Einblick zu erhalten, wie man die Steuerung von Verhalten autonomer Kreaturen angehen kann. Zuerst schauen wir jetzt jedoch unseren Agenten unter den vier Gesichtspunkten von PAGE an.

## 3.2 PAGE für unseren Agenten

Um unseren gewünschten Agenten genauer zu spezifizieren, wird an dieser Stelle auf die vier Dimensionen des in Abschnitt 2.2.5 beschriebenen PAGE eingegangen:

### 3.2.1 Wahrnehmung (Percepts)

**Wahrnehmung grundsätzlich**

Gemäss [1] sind für einen autonomen animierten Agenten mindestens drei Quellen von Empfindungen möglich:

- **reale Welt**
  Mit Mikrophonen, Kameras usw. kann mit Unsicherheit und Rauschen behaftete Information aus der realen Welt aufgenommen werden.
- **virtuelle Welt**
  Durch Abfrage anderer Objekte (Agenten) der virtuellen Welt kann direkt exakte Information beschafft werden.
- **synthetisches Sehen**
  Der Agent benutzt Bilderkennungstechniken um von seiner Position aus berechnete Bilder der virtuellen Welt zu verarbeiten.

Unsere Kreatur existiert in einer virtuellen Welt. Also ist es naheliegend, dass sie zumindest einen Teil davon wahrnehmen können sollte. Für unsere Kreatur stehen primär Empfindungen der virtuellen Welt im Vordergrund. Interaktion mit der realen Welt ist für unser System nicht vorgesehen. Ebensowenig das in [1] als Hilfe zur Navigation beschriebene synthetische Sehen. Diese sind aber beide als Erweiterungen denkbar.

1. Oder einer Kreatur. Die Begriffe Agent und Kreatur werden von nun an nebeneinander verwendet und als äquivalent betrachtet.

**Interpretation von Wahrnehmung**

In [9] wird klar zwischen Abtasten (sensing) und Wahrnehmen (perceiving) von Information unterschieden. Zuerst muss ein Stimulus aus der Welt aufgezeichnet werden, dann erst kann er wahrgenommen werden. [9] beschreibt einen Wahrnehmungsbaum, welcher die Abbildung von gemessenen Daten auf interpretierte Daten vornimmt. Diese Unterscheidung floss vereinfacht umgesetzt in unser Modell ein.

**Wahrnehmung unseres Agenten**

In der Anfangsphase war noch nicht klar, welche Wahrnehmungen unserem Agenten zur Verfügung stehen werden. Wir gingen davon aus, dass unsere Kreatur die virtuelle Welt um sich wahrnehmen kann. Konkret stellten wir uns vor, dass von anderen Objekten in der Welt, zumindest in einer näheren Umgebung der Kreatur, die genaue Lage und Attribute wie Farbe, Form oder Gewicht zugänglich sind. Ebenfalls denkbar waren für uns auch Eigenschaften wie "bewegbar" oder "verbunden mit", um dadurch mehr und interessantere Interaktionen der Kreatur mit ihrer Umgebung zu ermöglichen.

Als weitere Wahrnehmung soll unser Agent Anweisungen vom einem Benutzer aufnehmen können, damit wir die Kreatur zur Laufzeit steuern können. Unsere Kreatur wird primär durch Ziele vom Benutzer gesteuert. Siehe Abschnitt 3.5.2.

### 3.2.2 Aktionen unseres Agenten (Actions)

Um ein interessantes Verhalten zu ermöglichen, muss unsere Kreatur Aktionen kennen, mit denen sie sich in der Ebene bewegen kann, um Objekte zu greifen, Objekte niederzulegen sowie Objekte zu stossen. Für solche Aktionen ist eine Abbildung auf Motorikanweisungen naheliegend. Die Abtrennung von der Bewegungssteuerung war durch den Rahmen des Projekts klar vorgegeben und dadurch auch die Notwendigkeit zur Spezifizierung der Schnittstelle zwischen dem CreatureBrain und der CreatureControl auf geeignetem Abstraktionsniveau.

### 3.2.3 Ziele unseres Agenten (Goals)

In der Aufgabenstellung sind die Ziele unseres Agenten auf hoher Abstraktionsstufe dargestellt. Für die Realisierung eines Agenten ist es wichtig, Ziele auf hoher Abstraktionsstufe in konkretere Teilziele umzuwandeln. Für unseren Agenten haben wir diese, geordnet nach Priorität, wie folgt festgelegt:

1. **Selbsterhaltung**: Die Kreatur befriedigt überlebenswichtige Bedürfnisse. Sie erkennt lebensbedrohende Gefahren und versucht, diesen zu entkommen. Siehe Abschnitt 3.5.1.
2. **Steuerbarkeit**: Die Kreatur kann vom Benutzer vorgegebene Anweisungen in Form von Zielen interpretieren und umsetzen. Siehe Abschnitt 3.5.2.
3. **Sinnvolles autonomes Verhalten**: Falls die Kreatur nicht mit Selbsterhaltung oder der Ausführung von Anweisungen beschäftigt ist, zeigt die Kreatur anderweitig sinnvolles Verhalten. Wie dieses aussehen kann, ist in Abschnitt 3.5.3 beschrieben.

### 3.2.4 Die Umgebung unseres Agenten (Environment)

Die Kreatur lebt in einer dreidimensionalen virtuellen Umgebung. Diese Welt ist nicht-deterministisch, hochgradig dynamisch und kontinuierlich. Sie kann demnach als komplex eingestuft werden. Zusätzlich müssen wir davon ausgehen, dass die Welt nicht vollständig zugänglich ist. Dies ist sogar wünschenswert, denn sonst könnte es zum Beispiel Kreaturen geben, welche durch Objekte hindurch oder enorm präzise in die Ferne sehen könnten, was nicht sehr realistisch erscheinen würde.

### 3.2.5 Zusammenfassung

Das System CreatureBrain steuert also eine Kreatur, welche in einer komplexen und unvollständig zugänglichen Umgebung existiert. Die Kreatur kann ihre virtuelle Welt wahrnehmen und befolgt die drei Ziele Selbsterhaltung, Steuerbarkeit durch den Benutzer und sinnvolles autonomes Verhalten. Dazu stehen ihr einfache Aktionen wie Verschieben oder Ergreifen von Objekten zur Verfügung.

Im nächsten Kapitel wird nun aufgezeigt, wie ähnliche Anforderungen andernorts umgesetzt wurden.

## 3.3 Verwandte Arbeiten

Hier wird eine Übersicht über verschiedene Arbeiten auf dem Gebiet von autonomen virtuellen Kreaturen gegeben.

### 3.3.1 Evolution von Kreatur und Verhalten

Sims beschreibt in [15] ein System zur Generierung von Kreaturen in einer virtuellen, physikalisch simulierten Welt. Durch einen der biologischen Evolution angelehnten Mechanismus werden Gestalt und Verhalten der Kreaturen generationenweise mutiert und selektiert und mit Hilfe von Bewertungsfunktionen zu optimieren versucht.

Die Morphologie der Kreaturen wird dabei durch einen gerichteten Graphen repräsentiert. Die Knoten stehen für starre Körper und die Kanten für die Gelenke zwischen Körperteilen, zum Beispiel fest, rotierend oder biegend. Ein Nervengraph, der die Kreatur steuert, wendet auf diese Gelenke Kräfte an. Durch Rekombination und zufällige Mutationen werden nun aus diesen so beschriebenen Kreaturen immer neue Generationen von Kreaturen erzeugt. Dazu werden aufgrund einer Zielfunktion periodisch die Kreaturen mit dem besten Verhalten ausgewählt und weiterverwendet.

Die Resultate dieser Arbeit sind trotz des einfachen Ansatzes beeindruckend. Es konnten durch verschiedene Zielfunktionen schwimmende, laufende, springende und einer Lichtquelle folgende Kreaturen erzeugt werden. Es wurde eine neue Möglichkeit aufgezeigt, wie realistische Kreaturen erzeugt werden können, die zusätzlich noch steuerbar sind. Die Problematik zwischen Komplexität und Kontrollierbarkeit bei der Erzeugung von Kreaturen und Bewegungen wird dadurch gelöst, dass sich diese selbständig entwickeln.

Diese Arbeit zeigt, wie sich Kreaturen bestimmtes Verhalten aufgrund einer Zielfunktion aneignen und über Generationen verbessern. Dieser Ansatz scheint interessant, um ein bestimmtes Verhalten beherrschen zu können oder zu verbessern. Unsere Kreatur sollte aber verschiedenartiges Verhalten zeigen können und sollte sich in einer einfachen Version ihr Verhalten nicht selber aneigenen oder es erlernen müssen. Zudem sind bei diesem Ansatz die Gestalt und die Verhaltenssteuerung der Kreatur nahezu untrennbar miteinander verbunden. Für unsere Kreatur war jedoch eine Trennung der Steuerung der Motorik und des Verhaltens schon vorgegeben.

### 3.3.2 Verhaltenssteuerung durch Absichten

In [19] wird ein System präsentiert, in dem das Verhalten von künstlichen Fischen auf ihren Absichten basiert. Die aktuelle Absicht wird aufgrund der Wahrnehmung der Umwelt (Sicht, Temperatur), ihrer Eigenschaften (z.B. mag warm oder ist weiblich) und ihres eigenen mentalen Zustandes bestimmt. Dieser wird in drei Dimensionen Hunger, Libido und Angst festgehalten. Die Wahl der Absicht geschieht in jedem Zeitschritt durch einen einfachen Zustandsautomaten.

Dabei steht eine relativ kleine Anzahl von Absichten zur Auswahl, zum Beispiel fliehen oder essen. Nachdem eine Absicht bestimmt worden ist, wird die Kontrolle an ein Verhaltensmuster übertragen, welches diese Absicht umsetzten soll, beispielsweise: Nahrungsaufnahme, einem Objekt ausweichen, einem Fisch ausweichen.

Diese Arbeit zeigt wie verschiedene Fischtypen (Jäger, Beutetiere, Pazifisten) in dieser Architektur einfach spezifiziert werden können. Durch dieses einfache Modell lässt sich realistisches Verhalten von Fischen simulieren, wobei die einzelnen Kreaturen komplexes und vielfältiges Verhalten aufzeigen. Trotzdem scheint uns dieses System zu unflexibel und damit zu wenig offen zu sein. Die ganze Steuerung des Verhaltens ist in Form von Zustandsautomaten fest vorgegeben, welche auf eine fixe Menge von Verhaltensmustern verweisen. Dies schien uns zu restriktiv, denn unser System sollte sich neues Verhalten auch während der Betriebsphase aneignen können. Das scheint hier nicht einfach integrierbar zu sein.

Zudem sind die hier gezeigten Kreaturen komplett selbständig, das heisst, sie bestimmen ihr Verhalten selber. Es ist keine Möglichkeit vorgesehen, dass ein Benutzer die Steuerung übernehmen kann. Für unsere Kreatur jedoch soll die Möglichkeit der Steuerbarkeit durch einen Benutzer explizit gegeben sein.

### 3.3.3 Virtueller Durchschnittshund

In [3] wird als Motivation der Arbeit die Frage aufgeworfen, ob es möglich ist, eine Künstliche Intelligenz zu bauen, welche so schlau, anpassungsfähig und so bezaubernd ist wie ein durchschnittlicher Hund.

Folgende Anforderungen werden an diese Kreatur gestellt:

- **Robustheit**, um aufgrund von unvollständigem Wissen Entscheide treffen zu können.
- Passende **Reaktionen** als Antwort zu plötzlichen Veränderungen in der Umgebung.
- **Anpassungsfähigkeit**, um von eigenen Erfahrungen in der Welt zu lernen.
- **Ehrlichkeit**, um durch genügend Integrität bei der Wahrnehmung überrascht zu sein, wenn Dinge anders verlaufen als geplant.
- **Ausdrucksstärke**, die Kreatur hat Persönlichkeit und kann zum Beispiel Überraschung ausdrücken oder ihrem Wesen treu bleiben.
- **Vernunft**, um unabhängig von ihrer Persönlichkeit so etwas wie gesunden Menschenverstand zu zeigen.
- **Skalierbarkeit**, um eine ganze Gruppe von solchen Kreaturen zu ermöglichen.

Diese Ziele sind alle abstrakt formuliert und es wird nicht klar ausgedrückt, was damit genau gemeint ist und wie es umgesetzt werden soll. Teilweise ist dies auch gar nicht möglich, weil verwendete Begriffe, wie zum Beispiel Persönlichkeit, Ausdrucksstärke oder gesunder Menschenverstand gar nicht exakt definierbar sind. Andererseits sind andere Kriterien wie passende Reaktionen oder Anpassungsfähigkeit sehr stark von der jeweiligen Domäne und jeweiligen spezifischeren Anforderungen abhängig. Nichtsdestotrotz decken sich diese Anforderungen grundsätzlich mit denen an unsere Kreatur.

In [3] wird eine Architektur beschrieben, welche versucht, diesen Anforderungen gerecht zu werden. In dieser wird das “Gehirn” der Kreatur in mehrere separate Subsysteme aufgeteilt, welche durch ein internes Blackboard miteinander kommunizieren. Dieser Aufbau ist in Abb. 3.1 dargestellt.

Hier werden nun kurz die Aufgaben der einzelnen Subsysteme erklärt:

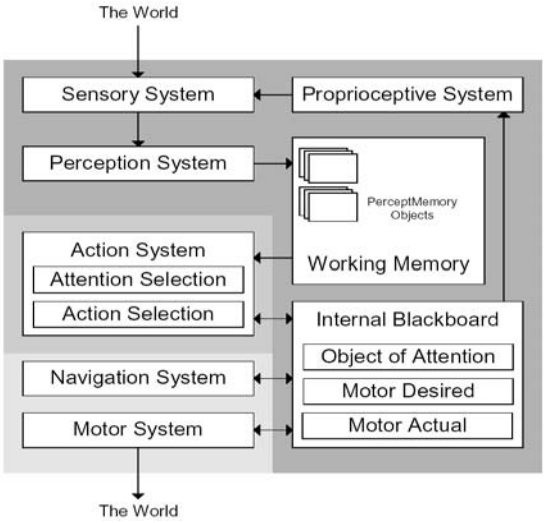



**Abbildung 3.1:** Die in [3] vorgeschlagene Architektur mit den verschiedenen Subsystemen.
(c) Synthetic Characters Group, MIT Media Lab.

Das **Sensory System** stellt den einzigen Eingangspunkt zum Gehirn einer Kreatur dar. Die wichtigste Aufgabe dieses Systems ist es, die Ehrlichkeit der Sensorik sicherzustellen, so dass alle Information, die in das System eingebracht wird, der Sichtweise und den Fähigkeiten der Kreatur angepasst ist.

Beim **Perception System** wird der Stimulus aus der Umgebung erst richtig wahrgenommen. Es wird Wert darauf gelegt, zwischen Abtasten und Wahrnehmen zu unterscheiden. Wahrnehmung entsteht erst durch Interpretation des Stimulus, und dies geschieht hier durch eine hierarchische Klassifikation der gemessenen Daten.

Reale Tiere haben Sensoren, welche die eigene Muskelaktivität und die Position ihrer Glieder feststellen. Diese Art von Selbstwahrnehmung ermöglicht das **Proprioceptive System**, dessen Namen dem Fachbegriff für diesen Vorgang entspricht. Dieses System ermöglicht durch die Wahrnehmung des eigenen Zustandes eine Rückkopplung und damit Selbstbeeinflussung. Rückkopplung auf diesem Weg ist notwendig da das Sensory System als einziger Informationseingangspunkt festgelegt wurde.

Das **Working Memory** ist einem psychologisch motivierten Konzept von Arbeitsspeicher angelehnt und hat den Zweck, den Teil des Zustandes der Umgebung festzuhalten und zu verfolgen, welcher für die aktuelle Aufgabe relevant ist. Das Working Memory stellt damit den aktuellen Kontext der Kreatur dar.

Das **Action System** ist zuständig für die Wahl der nächsten auszuführenden Aktionen. Für die Repräsentation von Aktionen müssen gemäss [3] die vier Fragen “Wann dies tun?”, “Was tun?” und “Wie, auf was angewendet dies tun?” sowie “Wie lange dies tun?” beantwortet werden. Entsprechend werden Aktionen dann in Tupeln mit den vier Dimensionen auslösender Kontext, Aktion, Objekt-Kontext und TueBis-Kontext repräsentiert. Bei der **Attention Selection** wird festgelegt, auf was die Kreatur ihre Aufmerksamkeit richtet. Als auszuführende Aktionen können grundsätzlich mehrere Aktionen ausgewählt werden (**Action Selection**). Dazu stehen verschiedene Gruppen von Aktionen zur Auswahl, welche wiederum in Listen unterteilt sind. Konkret wurden zwei Aktionsgruppen, eine primäre und eine Aufmerksamkeit-Aktionsgruppe verwendet mit je zwei Listen, einer mit gewöhnlichen (default) und eine mit speziellen (startle) Aktionen. Von den Listen der speziellen Aktionen wird jeweils diese mit dem

höchsten Wert einer Evaluation anhand ihres Aktionstupels im aktuellen Kontext ausgeführt. Die Aktionen der gewöhnlichen Liste werden zufällig ausgewählt. Im weiteren wird noch erklärt, wie Reinforcement Learning in dieser Architektur angewendet werden kann.

Das **Navigation System** und das **Motor System** sind verantwortlich für die Generierung von Bewegungen. Das Navigation System ist zuständig für räumliche Bewegungen, es führt Ausrichtungen und Verschiebungen der Kreatur sowie allfällige Ausweichmanöver bei Hindernissen durch. Das Motor System ist verantwortlich für die eigentliche Umsetzung von Bewegungen. Hier wird eine Lösung durch Pose-Graphen als Erweiterung von Verb-Graphen präsentiert. In Verb-Graphen beschreiben die Knoten Verben und die Kanten zeigen mögliche Übergänge zwischen Verben an. Pose-Graphen erreichen eine bessere Aufteilung von Bewegungen, indem sie nicht mögliche Übergänge zwischen Handlungen, sondern zwischen einzelnen Posen beschreiben.

Das **Internal Blackboard** schliesslich dient der Kommunikation zwischen dem Action System, dem Navigation System und dem Motor System. Hier wird relevante Information über den gewünschten und aktuellen Zustand der Motorik (**Motor Desired** und **Motor Actual**) sowie über das Objekt, welchem zur Zeit die Aufmerksamkeit gilt (**Object of Attention**), gehalten.

Dieser Aufbau wurde von den Autoren in zwei signifikanten Projekten verwendet. Im ersten, einer interaktiven Installation, spielt der Benutzer einen Schäfer, welcher durch akkustische Kommandos mit einem Schäferhund interagieren kann. Dieser Hund kümmert sich um eine Schafherde. Hund und Schafe werden durch oben beschriebene Architektur gesteuert. Dadurch werden einige der reaktiven, räumlichen und die Wahrnehmung betreffenden Fähigkeiten demonstriert.

Beim zweiten Projekt trainiert der Benutzer einen Hund mit der Clicker-Trainingsmethode, welche bei realen Hunden angewandt wird. Mit diesem System wird die Lernfähigkeit von Kreaturen mit dieser Architektur demonstriert.

Die beschriebene Interaktion mit dem Benutzer war ein Merkmal, welches wir in unserer Umgebung ebenfalls verlangten. Eine Architektur mit separaten Systemen ermöglicht eine klare Aufgabentrennung und lässt sich relativ einfach durch neue Systeme erweitern. Zusätzlich lässt sie sich durch Ersetzen mit fortschrittlicheren oder ausgereifteren Systemen verbessern. Dies ist für unser System wünschenswert. Als sinnvoll stuften wir eine Auftrennung in Sensor- und Wahrnehmungssystem ein.

Insgesamt schien uns dies eine gute Basis für unser neues System zu sein, und hat entsprechende Ähnlichkeiten zum gewählten Ansatz. Die Wahl der auszuführenden Aktionen sahen wir jedoch als zu unintuitiv und zu komplex modelliert an. Aktionen werden nur durch einen auslösenden Zustand aktiviert und dann durch eine Bewertungsfunktion ausgewählt. Die Umsetzung der Steuerung durch einen Benutzer schien uns darin zu umständlich umgesetzt.

Die Wahl der auszuführenden Aktion für unser System soll auf verschiedene Verhaltensmodi aufbauen, wie in Abschnitt 3.5 festgehalten wird. Und vor allem soll die Benutzersteuerung über Ziele erfolgen. Explizite Formulierung und Repräsentation von Zielen scheint uns im beschriebenen System nicht möglich zu sein.

## 3.4 Wahl des Verhaltens

Das Verhalten eines Agenten ist vorgegeben durch seine Aktionen. Die Auswahl von Aktionen ist abhängig von den Wahrnehmungen und dem Wissen einer Kreatur und davon, wie sie diese interpretiert. Bei der Steuerung des Verhaltens einer Kreatur ist diese Selektion der aus-

zuführenden Aktion, wie in Abschnitt 2.2 festgehalten, der entscheidende Knackpunkt - das "?" in Abb. 2.1.

### 3.4.1 Selbständiges und gesteuertes Verhalten

Um Verhalten von Agenten zu beschreiben, wollen wir die beiden Begriffe Selbständigkeit und Steuerbarkeit in unserem Sinne definieren:

- Ein Agent verhält sich **selbständig** (autonom[1]), wenn er von sich aus in seiner Umgebung handelt.
- Ein Agent ist **steuerbar**, wenn eine externe Entität Einfluss auf sein Verhalten nehmen kann.

Pures autonomes Verhalten ist wohl im Allgemeinen nicht das Ziel. Nehmen wir an, wir haben einen vollständig autonomen Agenten, der das Verhalten eines Hundes nachahmt. Dieser Agent verhält sich nun ausgezeichnet wie ein Hund, aber er spielt nicht mit Kindern. Und wenn die Kinder sich auf etwas anderes konzentrieren sollen, lenkt er sie ab. In beiden Fällen wäre die Möglichkeit der Steuerung wünschenswert; einmal um dem Agenten eine höhere Motivation zum Spielen zu geben, und im zweiten Fall, um ihn dazu zu bringen, eine Pause einzulegen. Dieses Beispiel ist sinngemäss aus [1] übernommen und soll aufzeigen, wieso eine Kreatur steuerbar sein sollte.

Steuerbarkeit kann als das Instrument des Benutzers von Agenten gesehen werden, um die Kontrolle über deren Verhalten zu behalten.

### 3.4.2 Verhaltenssteuerung auf verschiedenen Stufen

Die Steuerung eines autonomen Agenten kann gemäss [1] auf verschiedenen Stufen geschehen. Es wird ein Modell präsentiert, welches für die Steuerung des Verhaltens drei Stufen vorschlägt (Tabelle 3.1).

| **Stufe** | **Beschreibung** | **Beispiel** |
|---|---|---|
| Motivation | "Tue das richtige" | "Du bist hungrig" |
| Aufgabe | "Tue *dies* richtig" | "Iss dieses Stück Fleisch" |
| Direkt | "Tue was ich dir sage" | "Öffne den Mund" |

**Tabelle 3.1:** Drei Stufen der Verhaltenssteuerung gemäss [1].

Motivation ist dabei die höchste Stufe. Hier wird für das Verhalten des Agenten am meisten offen gelassen, respektive dieser muss sich am stärksten mit Konkretisierung von Steuerungsanweisungen beschäftigen. Auf der Stufe Aufgabe ist das Verhalten schon genauer vorgegeben. Es liegt nämlich ein klares Ziel vor, welches angestrebt wird. Auf einer direkten Stufe ist das Verhalten des Agenten klar vorgegeben. Die Anweisung entspricht direkt einer Aktion.

### 3.4.3 Verhaltenssteuerung nur durch Ziele

Unsere ersten Entwürfe einer Architektur waren noch stark an die von [3] vorgeschlagene "cognitive Architecture" angelehnt. In dieser Architektur liegt die Steuerung des Verhaltens in der Wahl der auszuführenden Aktionen zu einem gegebenen Zeitpunkt. Das heisst, es stehen

1. Der Begriff Autonomität wird hier nicht mit genau gleichem Sinn wie in Abschnitt 2.2.3 verwendet.

eine Menge von Aktionen zur Auswahl, diese werden evaluiert und es werden diejenigen ausgeführt, welche am Besten zur aktuellen Situation passen.

Für unser System wollten wir jedoch eine explizite Notation von Zielen erlauben, weil sich so das Verhalten auf einer hohen Abstraktionsstufe steuern lässt. Speziell die Validierung von Plänen und ein allfälliges Wiederplanen schienen uns sehr wünschenswert.

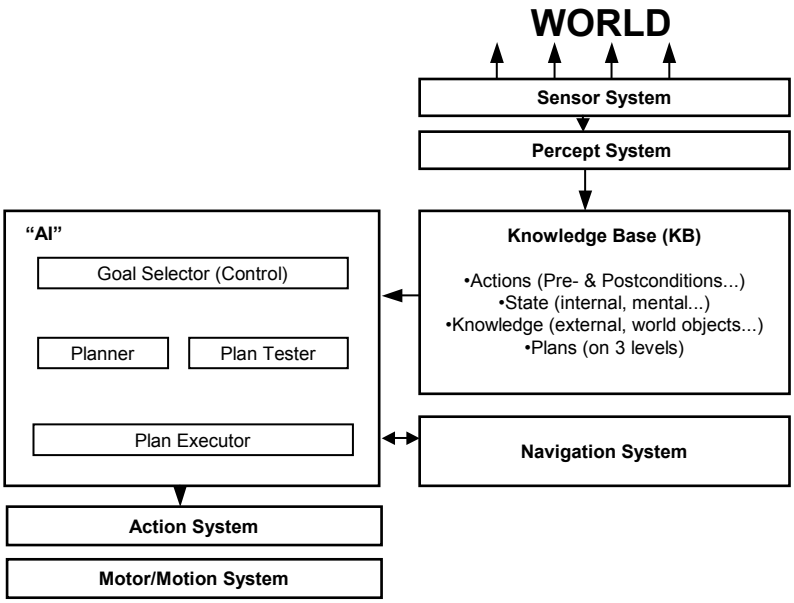



**Abbildung 3.2:** Entwurfsskizze einer Architektur zur zielbasierten Verhaltenssteuerung.

Die Entwurfsskizze in Abb. 3.2 aus der Designphase war sehr stark auf zielbasiertes Verhalten ausgerichtet. In dieser Phase überlegten wir uns, ob jegliches Verhalten als Ziel formuliert werden kann. Dies wäre möglich, denn eine reflektorische Handlung lässt sich als Ziel formulieren, genauso wie selbständiges Verhalten. Gerade für reflektorisches Verhalten schien uns aber die Notation von Zielen nicht sinnvoll, da ein Reflex schnell und bedingungslos, ohne Überlegung ausgeführt werden soll. Es geht wie in Abschnitt 2.2.6 beschrieben, um einfaches wenn-dann Verhalten.

Für einfaches autonomes Handeln schien uns die Definition eines Zieles ebenfalls nicht notwendig. Sonst müsste die Kreatur in der Lage sein, sich zu jedem Zeitpunkt ein Ziel vorzugeben. Wir entschieden uns für autonomes Handeln durch einfache Auswahl einer dem aktuellen Kontext am besten angebrachten Aktion. Des Weiteren sahen wir vor, dass die Kreatur selbst von reaktivem Verhalten in zielbasiertes wechseln kann, sprich die Kreatur kann sich auch selber Ziele geben. Ziele müssen somit nicht in jedem Fall vom Benutzer mitgeteilt sein.

## 3.5 Anforderungen an die Kreatur

Unsere Kreatur soll Verhalten auf drei verschiedenen Stufen mit unterschiedlichen Prioritäten zeigen:

1. **Reflektorisches Verhalten**
2. **Zielgesteuertes Verhalten**
3. **Reaktives Verhalten**

Durch diese Dreiteilung erreichen wir eine Mischung von autonomem und steuerbarem Verhalten. Einerseits ist rein autonomes Verhalten kein wünschenswertes Verhaltensmerkmal einer Kreatur, andererseits erwarten wir von der Kreatur auch autonomes Verhalten. Ihre Aktionen sollen nicht bloss von externen Anweisungen abhängen. Ihr Verhalten soll sinnvoll erscheinen, sie soll eine Art vorgegebenes Grundverhalten besitzen. Dieses Modell scheint uns angebracht, weil es offen ist und viel Spielraum lässt. Wir stellten fest, dass wir typisches Verhalten von Tieren mit diesen drei Stufen einfach modellieren können. Dies soll in den nächsten Abschnitten erläutert werden.

### 3.5.1 Reflektorisches Verhalten - Überleben

Reflektorisches Verhalten wird autonom ausgeführt, sobald eine bestimmte Wahrnehmung vorliegt. Durch Reflexe wird die Selbsterhaltung der Kreatur sichergestellt. Sie verhindern zum Beispiel, dass die Kreatur nicht einfach ins Verderben gesteuert werden kann. Reflektorisches Verhalten hat deshalb die höchste Priorität, denn Überleben ist für unsere Kreatur das höchste Ziel. Wenn mehrere Reflexreize vorliegen, regelt eine Priorisierung die Dringlichkeit. Ein Reflex wird solange ausgeführt, wie die entsprechende Wahrnehmung vorliegt und höchste Priorität unter anderen Reflexreizen hat.

**Szenarien:**

- **Rote Ampel:** Die Kreatur kreuzt eine Strasse nicht, solange die Ampel rot anzeigt.
  -> Die Kreatur kann externe Gefahrensignale wie rote Ampeln wahrnehmen.
- **Feind-Flucht:** Wenn die Kreatur zu nahe an ein feindliches oder gefährliches Objekt (z.B. Raubtier, Feuer, Gift) kommt, dann weicht sie diesem aus oder flüchtet bis auf eine bestimmte Distanz.
  -> Die Kreatur nimmt wahr, wann sie einem Gefahrenobjekt zu nahe kommt und wieder genügend weit entfernt ist. Sie kennt eine Aktion zum Ausweichen oder Flüchten.
- **Verhungern:** Wenn die Kreatur kurz vor dem Verhungern steht, dann beschafft sie sich Nahrung.
  -> Die Kreatur kann ihren eigenen Zustand wahrnehmen und hat die Möglichkeit, diesen zu beeinflussen.

Trotz dieser Reflexe ist natürlich das Überleben der Kreatur nicht garantiert. Erstens ist es in einer komplexen Umgebung praktisch unmöglich, dass zu jeder Lebensbedrohung ein rettender Reflex vorhanden ist. Dies ist auch in der realen Welt nicht gegeben. Zweitens kann das reflektorische Verhalten auch zu spät eintreten oder zu wenig wirksam sein; die ausgehungerte Kreatur kann zum Beispiel vor dem Zusammenbrechen nichts Essbares finden, oder eine feindliche Kreatur bewegt sich schneller auf die Kreatur zu, als diese flüchten kann. Drittens kann wegen der Priorisierung von Reflexen nicht garantiert werden, dass auf jeden Reflexreiz reagiert wird. Die Kreatur kann also beispielsweise auf der Flucht verhungern.

### 3.5.2 Steuerbares Verhalten - Zielverfolgung

Die Kreatur hat ein Ziel und arbeitet darauf hin. Ziele werden durch einen Plan erreicht, wie in Abschnitt 2.3 beschrieben. Der Benutzer hat die Möglichkeit, der Kreatur Ziele vorzugeben. Die Kreatur kann sich Ziele auch selber vorgeben.

**Szenarien:**

- **Erreiche Ort X:** Der Kreatur kann ein örtliches Ziel vorgegeben werden.
  -> Um sich gezielt an eine bestimmte Stelle in der Umgebung zu bewegen, müssen bei der Kreatur unter anderem entsprechende Aktionen sowie bestimmtes Wissen über die Umgebung vorhanden sein. Der Kreatur muss zum Beispiel bekannt sein, wo sich das Ziel befindet und wie man dorthin kommt, oder sie muss dies herausfinden können.
- **Greife Objekt Y:** Die Kreatur nimmt ein bestimmtes in der Umgebung vorhandenes Objekt auf.
  -> Dazu muss die Kreatur das Objekt kennen, es finden und Aktionen zur Verfügung haben, mit denen ein Objekt aufgegriffen werden kann.
- **Sammle alle roten Kugeln:** Die Kreatur kann ein Ziel der Form “Greife alle Objekte mit Eigenschaft E” verfolgen.
  -> Dazu muss die Kreatur mögliche Zielobjekte und die entsprechenden Eigenschaften wahrnehmen können und ebenfalls Objekte greifen können.
- **Verfolgung anderer Kreaturen:** Die Kreatur kann einer vorgegeben Kreatur oder einem Objekt folgen.
  -> Die Kreatur kann andere Kreaturen wahrnehmen und identifizieren. Sie weiss, wie die Verfolgung einer anderen Kreatur in Aktionen umgesetzt werden kann.

Im Folgenden werden die Argumente beschrieben, weshalb sich in unserem Modell Verhaltenssteuerung durch Ziele und nicht direkt oder durch Motivation abspielt.

Die Steuerung von Verhalten durch ein Ziel ist deshalb vorteilhaft für den Benutzer, weil ein Ziel eine Anweisung auf hoher Stufe darstellt. Dem Benutzer müssen die einzelnen Aktionen und allfällig darunterliegende Motorikanweisungen nicht bekannt sein. Eine direkte Steuerung mutete uns zu unflexibel an, denn dabei muss der Benutzer mehr Anweisungen geben und diese genau dem aktuellen Zustand der Welt anpassen. Bei einem Ziel ist das anders. Wenn sich die Situation ändert, muss dieses nicht geändert werden, denn im Allgemeinen reicht es, wenn die Kreatur ihren Plan validiert und unter Berücksichtigung der aktuellen Situation neu erstellt.

Der Erfolg eines Agenten kann aufgrund der Erfüllung eines Zieles einfach und binär bewertet werden. Dies ist bei der in [1] vorgeschlagenen Verhaltensteuerung durch Motivation schon schwieriger, denn dabei wird einer Kreatur nicht direkt eine überprüfbare Aufgabe vorgegeben, sondern Motivation zu bestimmtem Verhalten. Die Umsetzung dieser Motivation in Taten lässt sich jedoch im Allgemeinen nicht so einfach überprüfen. Nehmen wir zum Beispiel eine Kreatur, die das Ziel hat, etwas zu essen. Sie wird etwas Essbares in ihrer Umgebung suchen, dies zu sich nehmen und so ihr Ziel erreichen. Wenn einer anderen Kreatur nun die Motivation vorgegeben wird, hungrig zu sein, dann wird sie vielleicht auch etwas zu essen suchen, vielleicht aber auch nicht, weil ihre Motivation dazu zu gering ist. Oder sie sucht plötzlich alles Essbare in ihrer Umgebung und nimmt es auf. Wird das Verhalten einer Kreatur durch Motivation gesteuert, dann hat die Kreatur viel mehr Möglichkeiten in der Interpretation. Dadurch ist auch das erwartete Verhalten nicht klar vorgegeben und kann nicht direkt überprüft und bewertet werden.

### 3.5.3 Reaktives Verhalten

Im reaktiven Modus befindet sich die Kreatur, wenn keine Steuerungsanweisungen vorliegen und keine Reflex-Aktionen erfordert sind. Die Kreatur verhält sich möglichst rational und reagiert soweit als möglich sinnvoll auf ihre Wahrnehmung. Hier ist die Kreatur autonom und kann aufgrund ihrer Wahrnehmung und des internen Zustandes (z.B. ihrer Motivation) über angebrachte Aktionen entscheiden.

**Szenarien:**

- **Erkunden der Umgebung:** Die Kreatur bewegt sich in ihrer Umgebung und kann durch ihre Wahrnehmung neues Wissen über die Welt aquirieren. Sie kann zum Beispiel entfernte oder verdeckte Objekte entdecken.
  -> Die Kreatur kann sich bewegen und für sie relevante Dinge der Welt wahrnehmen. Die Kreatur braucht eine Strategie zur Erkundung der Umgebung, diese kann auch auf Zufall basieren.
- **Hunger:** Die Kreatur hat Bedürfnisse, welche sie befriedigen will. Diese sind beispielsweise durch interne Zustände festgehalten. Solche Bedürfnisse werden befriedigt, wenn ein Schwellwert dazu überschritten wird; vorher nicht unbedingt, da sie für die Kreatur nicht lebensnotwendig sind. Es ist aber denkbar, dass für eine Kreatur, die ein Bedürfnis (z.B. Hunger) lange vor sich hin schiebt, dies plötzlich zur Überlebensfrage wird und somit ein reflektorisches Verhalten (z.B. Verhungern) provoziert.
  -> Die Kreatur nimmt ihre Bedürfnisse wahr, welche irgendwie modelliert werden müssen. Die Kreatur weiss, wie (durch welche Aktionen) sie Bedürfnisse befriedigen kann. Um Bedürfnisse zu befriedigen ist es auch denkbar, dass sich eine Kreatur selber ein Ziel vorgeben kann (z.B. Essen finden).
- **Emotionales und soziales Verhalten:** Die Kreatur kann zufällig oder aufgrund von internen emotionalen Zuständen verschiedenes Sozialverhalten zeigen und zum Beispiel die Nähe von andern Kreaturen oder Paarungsverhalten zeigen oder aber auch die Einsamkeit suchen oder trauern.
  -> Dazu müssen interne Zustände modelliert werden und die Kreatur kennt Aktionen, um Gruppen-, Paarungs- und ähnliches Sozialverhalten zu zeigen.

### 3.5.4 Zusammenspiel der drei Stufen

Wichtig ist zu sehen, wie dieses Verhaltensmodell mit drei Prioritäten funktioniert. Reflexe werden durch einen Reiz ausgelöst. Sobald ein solcher vorliegt, wird darauf reagiert. Wenn mehrere Reflexreize vorliegen, regelt eine Priorisierung deren Dringlichkeit. Diese Reflexreaktion unterbricht andere Aktionen, welche zuvor in Ausführung waren. Diese Reaktion hält an, solange ein Reflexreiz vorliegt. Sobald dies nicht mehr der Fall ist, kommt wieder eine Aktion der unterliegenden Verhaltensebene zum Zug. Wird die Ausführung von gesteuertem Verhalten durch einen Reflex unterbrochen, wird nach Beendigung der Reflexreaktion weiter der Plan ausgeführt, falls er noch aktuell ist. Wenn der Plan annulliert wurde, keine weiteren Ziele vorliegen oder schon vor dem Reflex keine Ziele vorlagen, dann kommt eine Aktion des selbständigen, reaktiven Verhaltens zum Zug.

### 3.5.5 Ein abschliessendes Beispielszenario

Angenommen, unsere Kreatur lebt in einer ihr nicht vollständig bekannten Welt. Es liegen zur Zeit keine Ziele und keine Reflexreize vor. Also ist die Kreatur im reaktiven Modus und wählt die beste Aktion aus. Die könnte hier sein, herumzulaufen und die unbekannte Welt auszukundschaften. Während die Kreatur also in der Welt herumläuft und versucht, neues Wissen zu aquirieren, wird ihr ein Ziel mitgeteilt: “Essen haben”. Nun wendet die Kreatur ihren Planungsmechanismus an, welcher ihr zu diesem Ziel einen Plan erstellt. Dieser Plan könnte zum Beispiel[1] so aussehen: 1. Gehe zum Essen. 2. Nimm Essen auf. 3. Geh nach Hause. Nun beginnt die Kreatur mit der Abarbeitung dieses Plans und macht sich auf den Weg zu einem ihr bekann-

1. Je nach vorhandenem grundlegenden Verhalten kann ein solcher Plan auch ganz anders aussehen. Dies ist abhängig von der für Aktionen gewählten Abstraktionsstufe.

ten und als essbar wahrgenommenen Objekt. Auf dem Weg dorthin trifft sie nun auf eine feindliche Kreatur, was bei unserer Kreatur sofort einen Fluchtreflex auslöst, welcher die Kreatur zwei hundert Meter wegrennen lässt. Nachdem dieser Reflex vorbei ist, fällt die Kreatur wieder in den zielbasierten Modus zurück, denn das Ziel “Essen haben” ist immer noch vorhanden. Nun wird der Plan revalidiert; neues Planen aufgrund der veränderten Situation ist nötig. Nun wird dieser neue Plan ausgeführt und wenn dies erfolgreich abgeschlossen ist, wenn also alle Nachbedingungen der einzelnen Aktionen des Plans erfüllt sind, dann fällt die Kreatur wieder in den reaktiven Modus zurück und führt die Aktion aus, die im aktuellen Kontext am sinnvollsten scheint. Dies könnte “Essen” sein, da der Hungerzustand der Kreatur inzwischen über einen gewissen Schwellwert gewachsen ist und die Kreatur im Besitz von etwas Essbarem ist.

## 3.6 Schlussfolgerung

Mit diesem umfangreichen Kapitel erhoffen wir, einen Eindruck vermittelt zu haben, wie unser System entstanden ist. Verhaltensteuerung ist ein vielschichtiges Problem. Gewünschtes Verhalten soll je nach Anforderungen und Umgebung ganz verschieden aussehen und entsprechend unterscheiden sich auch die Modelle und Systeme. Zudem gibt es für Verhalten keine allgemeingültigen Notationen oder Modelle. Wir haben in diesem Kapitel einige Anforderungen an unseren Agenten festgehalten, indem wir PAGE angewandt haben. Zudem haben wir verschiedene Ansätze aus der Forschung dargestellt und kritisch analysiert. Zum Schluss haben wir uns auf ein System festgelegt, welches durch zahlreiche Szenarien beschrieben wurde. Im folgenden Kapitel wird nun dargestellt, wie wir unser System aufgebaut haben.

# 4 Architektur

In diesem Kapitel wird der Aufbau des CreatureBrain vorgestellt. In einem ersten Teil werden Rahmenbedingungen, Anforderungen und Entscheide aufgezeigt. Dann wird eine Übersicht über das gesamte System gegeben und anschliessend werden die einzelnen Subsysteme vorgestellt. Zum Abschluss folgt eine kurze Zusammenfassung.

## 4.1 Rahmenbedingungen

Gemäss Aufgabenstellung war das Ziel dieser Diplomarbeit die Implementierung eines Systems zur Simulation von Verhalten und Künstlicher Intelligenz. Damit unsere Kreatur, welche konzeptionell einem in Abschnitt 2.2 beschriebenen rationalen Agenten entspricht, in einer Welt handeln kann, muss sie zuerst in einem System modelliert werden und dieses passend in eine Umgebung eingebunden sein. In einem zweiten Schritt kann dann das Verhalten dieses Agenten durch geeignete Agenten-Programme gesteuert werden.

Durch den Rahmen des CreatureZoo Projekts waren die zwei grundlegenden Schnittstellen des CreatureBrain vorgegeben. Einerseits lebt die Kreatur in einer virtuellen, dreidimensionalen, physikalisch basierten Welt. Diese Umgebung muss die Kreatur in irgendeiner Form wahrnehmen können. Ohne dies lässt sich kaum sinnvolles Verhalten simulieren - das Verhalten eines Agenten ohne Sensoren würde bloss vom internen Wissen abhängen. Andererseits setzt das CreatureBrain auf eine Geometrie auf, welche die Kreatur als Bewohner einer virtuellen Welt erst visualisiert. Diese Geometrie kann durch das CreatureBrain gesteuert werden und so die Kreatur in der virtuellen Welt bewegt werden. So manifestiert die Kreatur ihr Verhalten in ihrer Umgebung. Das Zusammenspiel des CreatureBrain mit der unterliegenden Motorik und der umschliessenden künstlichen Welt ist im Kapitel 6 beschrieben.

## 4.2 Anforderungen

**Konkretisierung**

Die Anforderungen an das CreatureBrain sind durch die Aufgabenstellung sehr offen gelassen. Im Kapitel 2 wurde diese in die Welt der Künstlichen Intelligenz eingeordnet, um ihr einen theoretischen Rahmen zu geben. Aber dies alleine reicht noch nicht, um ein System zu implementieren. In Kapitel 3 wurden, anhand von verwandten Arbeiten und eigenen Ideen, die Anforderungen an das CreatureBrain etwas konkreter gefasst, bis hin zu konkreten Szenarien

mit vorgegebenen Erwartungen an das Verhalten der Kreatur. All die im Kapitel 3 beschriebenen Anforderungen beeinflussten die hier aufgezeigte Architektur.

**Verhaltenssteuerung von verschiedenen Kreaturtypen**

Den CreatureZoo werden dereinst hoffentlich verschiedenartige Kreaturen bevölkern. Das CreatureBrain wurde zur Verhaltenssteuerung von verschiedenartigen Kreaturen entworfen. Seine Architektur soll grundsätzlich auf beliebigen Kreaturen mit verschiedenen Geometrien und Bewegungsmöglichkeiten aufsetzen können, welche sich in unterschiedlichsten Umgebungen aufhalten.

**Modulares Grundsystem**

Das CreatureBrain als funktionierendes System zur Verhaltenssimulation stellt einen Rahmen für Erweiterungen des Verhaltens und Experimente in der Verhaltenssteuerung von virtuellen Kreaturen dar. Deswegen soll das CreatureBrain ein modulares System mit vielen Möglichkeiten zur Erweiterung sein. So können einfach einzelne Subsysteme ersetzt oder neue hinzugefügt werden. Eine Modularisierung erleichtert zudem die Implementierung und macht das System verständlicher.

**Eingabe - Verarbeitung - Ausgabe**

Die drei Hauptaufgaben des CreatureBrain, die Welt wahrzunehmen, aufgrund dieser Information und internen Wissens über das Verhalten zu entscheiden und dann dieses Verhalten auszuführen, bieten drei naheliegende Grundmodule an. Deshalb soll die Grundstruktur des CreatureBrain diese in der elektronischen Datenverarbeitung oft angewandte Struktur auf drei Untersysteme aufzeigen. Diese nennen wie hier Input-, Intelligence- und ActionSystem. Sie können weiter unterteilt und gegebenenfalls durch weitere Module ergänzt werden.

**Entscheidung**

Zusammenfassend wurden für die Architektur des CreatureBrain folgende auf den Anforderungen basierenden Punkte festgehalten:

- In einer ersten Phase wird ein System entworfen, welches unabhängig von der Anbindung an eine Umgebung und ein Bewegungssystem funktioniert. Dieses System soll einfach die Grundmechanismen testen und als Versuchsmodul dienen.
- In einer zweiten Phase wird dieses Grundsystem erstellt, ohne es mit viel Logik zu füllen. Dies kann später geschehen.
- Das Zusammenführen mit den andern Modulen findet in einer dritten Phase statt.
- Die Implementierung von Steuerungslogik für unsere Kreatur erfolgt je nach verbleibender Zeit in einer letzten Phase.
- Das System besteht aus vier Subsystemen, nämlich den drei oben beschriebenen für Eingabe, Verarbeitung und Ausgabe und noch einem weiteren Subsystem zur Kommunikation durch Austausch von Nachrichten.
- Zu jeder Zeit wird vom System eine grundlegende Fertigkeit, Aktion genannt, ausgeführt.
- Das Verhalten der Kreatur spielt sich in den drei in Abschnitt 3.5 beschriebenen Modi ab. Dies ist in der Architektur berücksichtigt.
- Es besteht die Möglichkeit, dem System Ziele mitzuteilen. Zu diesen werden Pläne erstellt und ausgeführt.

## 4.3 Übersicht Grundsystem

Nach mehreren Designzyklen einigten wir uns auf das in Abb. 4.1 dargestellte Design. Darin sind die vier Subsysteme InputSystem, IntelligenceSystem, ActionSystem und Blackboard gut erkennbar.

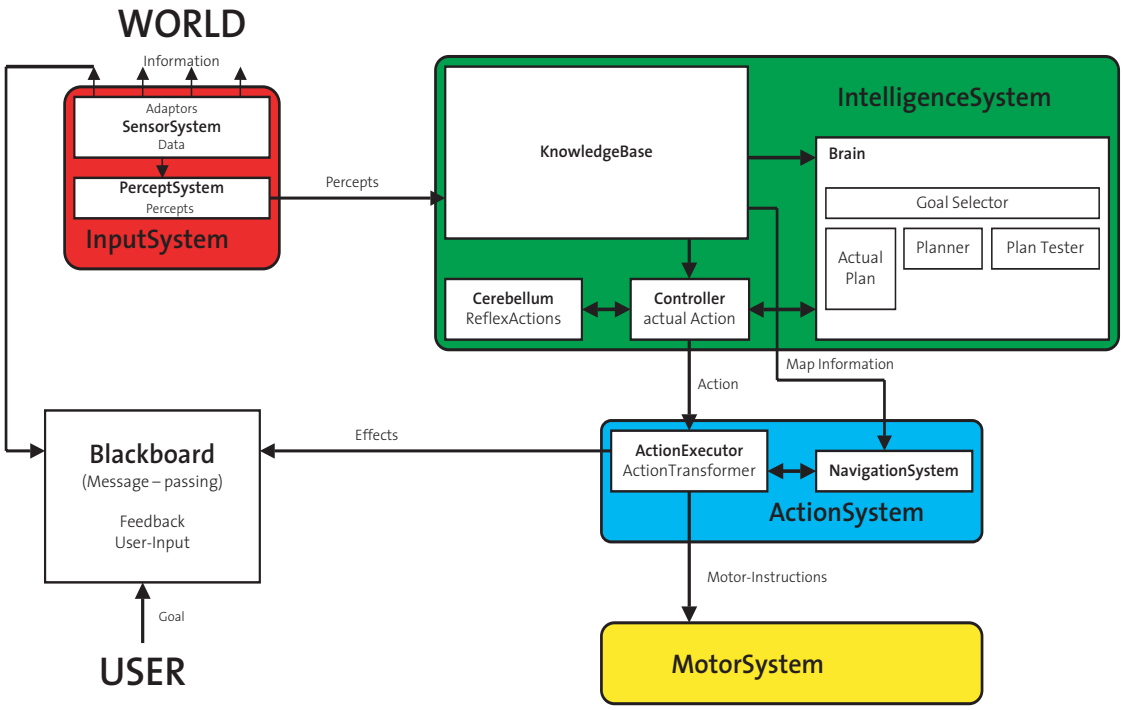



**Abbildung 4.1:** Übersicht der CreatureBrain Architektur.

Das InputSystem ist zuständig für Informationsbeschaffung, es ist die einzige Quelle für Informationen. Das IntelligenceSystem verwendet die vom InputSystem gelieferten Informationen und entscheidet aufgrund derer und mittels intern vorhandenem Wissen über das Verhalten. Im ActionSystem schliesslich wird das gewählte Verhalten umgesetzt und ausgeführt. Das Blackboard dient als Rückkopplungskanal und Verbindung zum Benutzer. Eine ausführlichere Beschreibung der einzelnen Subsysteme folgt in den nächsten Abschnitten.

## 4.4 InputSystem

Das InputSystem ist zuständig für die Beschaffung und Bereitstellung von sachgerechter und brauchbar aufbereiteter Information. Es ist die einzige Quelle für Informationen; alle Sensoreingaben gehen durch dieses System, wie der Ablauf in Abb. 4.2 darstellt.

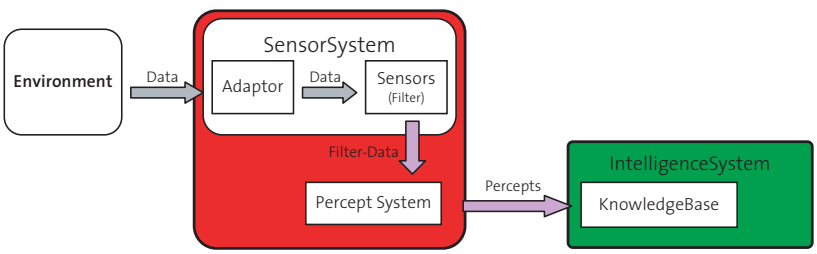



**Abbildung 4.2:** Ablauf im InputSystem.

### 4.4.1 Ablauf

Die Eingabe in das InputSystem besteht aus einer beliebigen Information. Diese kann grundsätzlich von verschiedenen Orten kommen. Das Inputsystem interagiert also mit verschiedenen anderen Systemen um benötigte Informationen zu beschaffen. Es kann diese in irgendeiner Weise erfassen oder abtasten.

Das InputSystem bringt diese dann in eine einheitliche Form und interpretiert sie.

Die Ausgabe dieses Systems besteht aus interpretierten Aufzeichnungen, eigentlichen Wahrnehmungen, welche in eine einheitliche Form gebracht wurden. Diese werden dann an das IntelligenceSystem weitergereicht.

Das InputSystem besteht aus SensorSystem und PerceptSystem:

### 4.4.2 SensorSystem

Das SensorSystem als Subsystem des InputSystems ist zuständig für das Abtasten von verschiedenen Informationsquellen. Es besteht aus Adaptoren und Sensoren:

**Adaptoren**

Adaptoren sind verantwortlich für die Beschaffung von Daten. Ein Adaptor ist jeweils seiner Datenquelle angepasst und bringt die Daten dieser Quelle in ein einheitliches Format, bevor diese dann seinem Sensor oder seinen zugehörigen Sensoren weitergereicht werden. Jeder Adaptor hat mindestens einen zugehörigen Sensor.

Im Allgemeinen hat jede Datenquelle einen speziellen Adaptor. Ein Adaptor ist auf seine Datenquelle abgestimmt und weiss, wie er dort Daten beschaffen kann.

Ein Beispiel: Ein Adaptor könnte als Datenquelle ein Thermometer haben. Bei Aktivierung beschafft sich der Adaptor die aktuellen Daten, das heisst die aktuelle Temperatur. Dann stellt er diese in einem einheitlichen Format dar, zum Beispiel als 32 Bit Gleitkommazahl, und reicht sie an seine Sensoren weiter.

**Sensoren**

Sensoren funktionieren als Filter, jeder Sensor ist für bestimmte Daten vorgesehen. Sensoren bekommen als Eingabe Daten. Sie sind dafür verantwortlich zu entscheiden, welche Information weiterverwendet wird und welche nicht. Sensoren filtern aus den eingehenden Daten die gewünschten aus. Jeder Sensor ist dafür ausgelegt, einen bestimmten Typ von Daten herauszufiltern. Diese werden dann auf eine von zwei Arten weiterverwendet:

1. Der Sensor generiert eine Wahrnehmung. Jeder Sensor ist ausgelegt, genau eine Art von Wahrnehmungen zu generieren. Jedesmal, wenn ein Sensor aus den Daten ein Datenelement ausfiltert, wird daraus eine Wahrnehmung eines bestimmten Typs generiert. Mehr über Wahrnehmungen folgt im nächsten Abschnitt.
2. Der Sensor leitet die ausgefilterten Daten weiter an seine Untersensoren, falls solche vorhanden sind.

Diese Vorgänge sind unabhängig voneinander. Das heisst ein Sensor generiert eine Wahrnehmung oder nicht, und falls Untersensoren vorhanden sind, werden die ausgefilterten Daten an diese weitergereicht. Sensoren können also hierarchisch kombiniert werden, wie Abb. 4.3 illustriert.

Dieser hierarchische Aufbau von Filtern ermöglicht eine systematische Filterung und dadurch Klassifizierung der Daten.

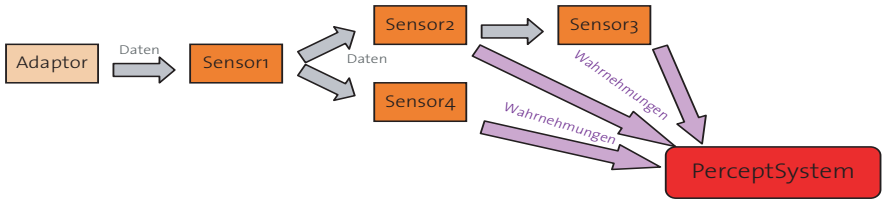



**Abbildung 4.3:** Hierarchisch kombinierte Sensoren.

### 4.4.3 PerceptSystem

Das PerceptSystem ist zuständig für die Verwaltung von Wahrnehmungen. Hier werden alle Wahrnehmungen gehalten, bevor sie dann dem IntelligenceSystem weitergereicht werden.

**Wahrnehmungen**

Wahrnehmungen sind die kleinsten Informationseinheiten. Wahrnehmungen geben Daten erst einen Sinn. Mit einer Wahrnehmung ist eine Interpretation verbunden. Die Sensoren fühlen und filtern Stimuli aus der Welt, aber wahrgenommen wird dadurch noch nichts. Erst durch eine Klassifizierung können die Daten wahrgenommen werden. Diese Unterscheidung zwischen Abtasten und Wahrnehmen ist wichtig. Wahrnehmungen haben eine Bedeutung, blosse Daten haben noch keine.

Zwei Beispiele dazu: Die Daten über ein Objekt in der Welt müssen zuerst interpretiert werden. Ohne diese Interpretation hat das Objekt keine Bedeutung. Es ist a priori nicht klar, was damit angefangen werden soll, ob das Objekt eine Gefahr darstellt oder ein Ziel. Wenn nun diese Information aber in eine Wahrnehmung verpackt ist, dann gibt diese darüber Auskunft, den Daten einen Sinn. Eine Wahrnehmung kann Objektdaten zur Position einer Gefahr oder zu Zieldaten machen. Ein anderes Beispiel ist eine Nachricht in Textform. Sie kann verschiedenes bedeuten. Sie kann die Notation eines Zieles sein, den Zustand der Welt oder den Zustand der Kreatur beschreiben. Was diese Daten bedeuten, entscheidet sich durch die Klassifizierung in Wahrnehmungen. Erst durch diese Verpackung werden so Textdaten beispielsweise zu einem Ziel. Das ist die Aufgabe von Wahrnehmungen - sie klassifizieren Daten.

## 4.5 IntelligenceSystem

Das IntelligenceSystem stellt den eigentlichen Kern der Steuerung des Verhaltens unserer Kreatur dar. Hier befindet sich, wie in Abb. 4.4 dargestellt, einerseits das Wissen der Kreatur (KnowledgeBase), das Steuerungszentrum (Controller), eine einfache Reflex Logik (Cerebellum) sowie kompliziertere Logik zur Verhaltenssteuerung (Brain). Das IntelligenceSystem steuert das Verhalten der Kreatur durch die Wahl von geeigneten Aktionen.

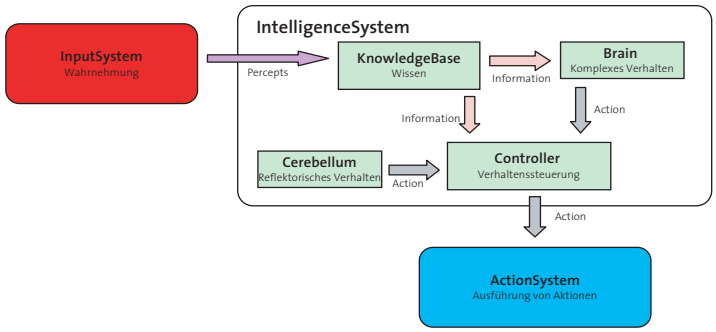



**Abbildung 4.4:** Funktionsweise des IntelligenceSystem.

### 4.5.1 Ablauf

Das IntelligenceSystem bekommt als Eingabe vom InputSystem Wahrnehmungen mitgeteilt. Diese werden hier verarbeitet und aufgrund derer und dem internen Zustand des IntelligenceSystem wird dann die Aktion ausgewählt, welche auszuführen ist. Hier geschieht die eigentliche Verhaltenssteuerung unserer Kreatur. Die Ausgabe des IntelligenceSystem ist die Beschreibung einer Aktion, welche dann vom ActionSystem umgesetzt werden soll.

Das IntelligenceSystem setzt sich aus folgenden Untersystemen zusammen:

### 4.5.2 KnowledgeBase

Die KnowledgeBase hält das aktuelle Wissen der Kreatur. Hier werden die vom InputSystem mitgeteilten Wahrnehmungen angenommen und, falls sie für die Kreatur von Interesse sind und von der KnowledgeBase verstanden werden, gespeichert.

Die KnowledgeBase enthält grundsätzlich Wissen über folgende Dinge:

- **Externer Zustand (bekanntes Wissen über die Welt)**. Hier wird zum Beispiel festgehalten, welche Objekte der Welt bekannt sind und welche Attribute diese Objekte haben, inklusive deren Position. Im Weiteren wird eine Weltkarte über die bekannte Topographie und Verteilung von Objekten und Hindernissen in der Welt gehalten, sowie die aktuelle Position der Kreatur.
- **Interner (mentaler) Zustand**. Hier wird festgehalten, in welchem Zustand sich die Kreatur befindet. Zum Beispiel wie stark Gefühle wie Hunger, Durst, Aggression, Libido oder Müdigkeit sind.
- **Mögliche Aktionen**. Die Menge von möglichen Aktionen mit Vorbedingungen und Effekten, welche angeben, wann eine Aktion angewandt werden kann, respektive was sie bewirkt.
- **Ziele**. Vom Benutzer mitgeteilte Ziele werden hier gehalten.
- **Reflexreize**.

Information über diese Punkte wird von der KnowledgeBase gehalten und für Abfragen bereitgestellt. Dieses Wissen wird von verschiedenen Systemen abgefragt, sicher vom Brain und vom Controller, um über aktuelle Ziele und den aktuellen Zustand der Welt und der Kreatur informiert zu werden. Ebenso stellt das NavigationSystem Anfragen, um an die aktuelle Weltkarte und an aktuelle Positionen von Objekten zu gelangen.

### 4.5.3 Cerebellum

Das Cerebellum (Kleinhirn) ist, angelehnt an die Neurobiologie, zuständig für grundlegendes Verhalten, also für einfaches reflektorisches Verhalten. Das Cerebellum erhält vom Controller eine Reflexwahrnehmung, entscheidet dann gemäss einer wenn-dann Nachschlagetabelle, mit welcher Aktion darauf geantwortet wird, und liefert diese Aktion als Ausgabe dem Controller zurück.

Wenn zum Beispiel ein Reflexreiz im CreatureBrain auftaucht, der ausgelöst wurde, weil sich der Kreatur ein gefährliches Objekt nähert, dann antwortet das Cerebellum darauf direkt mit einer Flucht-Aktion.

Dieser Teil entspricht dem Reflex-Agenten von Abschnitt 2.2.6. Hier wird das in Abschnitt 3.5.1 geforderte reflektorisches Verhalten umgesetzt.

### 4.5.4 Brain

Für Aktionen, welche über simples reflektorisches Verhalten hinausgehen, ist das Brain Subsystem zuständig. Dieses Subsystem soll in der Lage sein, dem Controller auf Anfrage zu jeder Zeit eine Aktion zurückzuliefern[1]. Hier wird die aktuelle Situation überdacht und die am besten zum aktuellen Kontext passende Aktion bestimmt.

Das Brain bezieht seine Eingabe aus der KnowledgeBase. Es kann sich dort jegliche Information holen, die es benötigt, um zu einem Schluss zu kommen.

Der Entscheid über die auszuführende Aktion hängt dabei vom Modus ab, in welchem sich die Kreatur befindet. Der Reflexmodus wird, wie in Abschnitt 4.5.3 beschrieben, vom Cerebellum behandelt. Zielgesteuertes und reaktives Verhalten, die beiden alternativen Modi von Abschnitt 3.5, werden hier im Brain behandelt:

**Zielgesteuertes Verhalten**

Wenn mehrere Ziele vorhanden sind, entscheidet das Brain, welches als nächstes verfolgt wird. Ein Ziel wird, wie in Abschnitt 2.3 beschrieben, durch einen Planungsvorgang in einen Plan umgewandelt. Dieser Planungsvorgang ist das Kernstück der Steuerungslogik im Brain. Zu diesem Zweck steht dem Brain das gesamte Wissen der KnowledgeBase zur Verfügung. Ein Plan ist eine Sequenz von Aktionen, welche sequentiell abgearbeitet wird. Dies geschieht, indem jeweils eine Aktion dem Controller überreicht wird. Sobald dieser meldet, dass diese Aktion ausgeführt ist, wird die nächste überreicht und so weiter, bis der Plan abgearbeitet ist.

Wenn ein Plan abgearbeitet worden ist, wird ein nächstes Ziel ausgewählt, wiederum ein Plan dazu erstellt und ausgeführt. Falls keine weiteren Ziele vorhanden sind, fällt das System in den reaktiven Modus zurück.

Beim Ausführen von Zielen wird das verfolgte Ziel immer wieder validiert, um sicherzustellen, dass die Verfolgung des Ziels im aktuellen Kontext noch Sinn macht. Gegebenenfalls werden Pläne abgebrochen und Pläne neu erstellt.

Dem Benutzer soll die Möglichkeit gegeben werden, Ziele mitzuteilen, den aktuellen Plan abzubrechen oder vorgegebene Ziele zu streichen.

**Reaktives Verhalten**

Falls keine Ziele vorhanden sind, entscheidet das Brain, welche Aktion im aktuellen Kontext ausgeführt werden soll. Dazu kann das gesamte Wissen aus der KnowledgeBase verwendet werden. Mögliche Aktionen können beispielsweise evaluiert und mit einem Wert aufgrund des aktuellen Kontextes versehen werden; die Aktion mit dem höchsten Wert wird dann ausgeführt. Wie ein solches Verhalten aussehen kann, ist in Abschnitt 3.5.3 beschrieben.

### 4.5.5 Controller

Das Controller Subsystem kontrolliert das aktuelle Verhalten der Kreatur und hält zu jeder Zeit die vom IntelligenceSystem gewünschte Aktion, welche am besten auf die aktuelle Situation passt. Dies ist die Aktion, welche vom ActionSystem ausgeführt werden soll.

Als Eingabe erhält der Controller Information aus der KnowledgeBase und Aktionen von Cerebellum und Brain sowie Rückmeldungen vom ActionSystem. Er fragt in jedem Schritt die KnowledgeBase nach erhaltenen Reflexwahrnehmungen. Falls eine solche eingegangen ist, wird darauf reagiert. Der Controller benutzt dazu das Cerebellum, welches eine passende

1. Dieser Mechanismus ist in der Literatur als *anytime algorithm* bekannt. [4]

Aktion zurückliefert. Falls kein Reflexreiz vorliegt, informiert sich der Controller beim Brain, welches die zum aktuellen Kontext passende Aktion ist.

Die aktuelle Aktion kann sich ändern, wenn:

- Die aktuelle Aktion vorbei ist. Dies wird vom ActionSystem mitgeteilt. Der Controller prüft nun, ob die Nachbedingung dieser Aktion erreicht ist. Wenn dies der Fall ist, wird eine neue Aktion gestartet, sonst wird diese Aktion nochmals ausgeführt.
- Ein neuer Reflexreiz vorliegt. Darauf muss reagiert werden.
- Das Brain eine neue Aktion ausgeführt haben will. Dies ist zum Beispiel möglich, wenn ein neues Ziel vorliegt oder sich die Situation verändert hat.

Wenn sich die aktuelle Aktion ändert, dann überprüft der Controller zuerst, ob die Vorbedingungen für die Ausführung dieser neuen Aktion überhaupt erfüllt sind. Falls dies nicht der Fall ist, wird diese Aktion nicht ausgeführt sondern eine festgelegte Aktion, deren Vorbedingungen immer erfüllt sind.

## 4.6 ActionSystem

Das ActionSystem ist für die Ausführung von Aktionen zuständig. Hier werden die vom IntelligenceSystem gewählten Aktionen ausgeführt.

### 4.6.1 Ablauf

Das ActionSystem bekommt vom IntelligenceSystem Aktionen zur Ausführung und Information über die Position von Objekten und die Topographie der Umgebung. Als weitere Eingabe erhält es Rückmeldungen vom Bewegungssystem.

Die Aufgabe des ActionSystem besteht darin, Aktionen auszuführen. Diese erhält es vom IntelligenceSystem und wandelt sie dann in Bewegungen und sonstige Effekte wie Zustandsänderungen um. Um Bewegungsaktionen auszuführen, werden Daten über die Umgebung benötigt. Diese können von der KnowledgeBase bezogen werden. Das Bewegungssystem meldet, wenn eine Bewegung ausgeführt wurde, damit dieses für eine nächste instruiert werden kann.

Die Ausgabe des ActionSystem besteht aus Anweisungen an das Bewegungssystem, aus Meldungen über Effekte von Aktionen und aus den Rückmeldungen von beendeten Aktionen an das IntelligenceSystem.

Das ActionSystem wird in folgende Untersysteme unterteilt:

### 4.6.2 ActionExecutor

Der ActionExecutor kontrolliert das ActionSystem und ist für die eigentliche Ausführung von Aktionen verantwortlich.

Der ActionExecutor bekommt bei Bedarf vom IntelligenceSystem die aktuelle Aktion. Vom ActionTransformer kann er auf Anfrage die zur Ausführung einer Aktion notwendigen Motorikanweisungen für einzelne Bewegungen erhalten. Vom MotorSystem empfängt er eine Meldung, sobald eine Bewegung ausgeführt wurde.

Der ActionExecutor führt Aktionen aus, indem er sie einerseits durch den ActionTransformer in eine für die unterliege Motorik verständliche Form umwandeln lässt und diese dann an das Bewegungssystem schickt. Andererseits setzt er zusätzliche Effekte von Aktionen, wie beispielsweise Zustandsänderungen, um.

Ein Beispiel: Es gehe darum, die Aktion “Greife Objekt X” auszuführen. Diese Aktion wird vom ActionTransformer in kleine Bewegungen der Art “Bewege den Arm” und “Schliesse Greifer” umgewandelt. Damit die Kreatur aber nach erfolgreicher Handlung auch weiss, dass sich Objekt X jetzt in ihrem Besitz befindet (z.B.: *haben(X)*), wird der Effekt dieser Aktion als entsprechende Meldung an das Blackboard geschickt, wodurch dieser dann wahrgenommen werden kann. Dieser Rückkopplungsmechanismus wird in Abschnitt 4.7.3 beschrieben.

Während der Ausführung einer Aktion wird periodisch getestet, ob diese im akutellen Kontext noch gültig ist und weiter ausgeführt werden soll. Aktionen welche diese Validitätsprüfung nicht bestehen, werden abgebrochen oder der Situation angepasst.

Der ActionExecutor testet periodisch, ob die aktuell ausgeführte Aktion noch der gewünschten vom IntelligenceSystem entspricht. Falls dies nicht der Fall ist, wird die Aktion, welche sich in Ausführung befindet, abgebrochen und mit der Ausführung der neuen, vom IntelligenceSystem mitgeteilten Aktion begonnen.

Wenn alle für eine Aktion notwendigen Bewegungen und Effekte vom Bewegungssystem ausgeführt worden sind, wird diese Aktion als ausgeführt betrachtet und dem IntelligenceSystem mitgeteilt.

Die Ausgabe des ActionExecutor besteht zusammenfassend aus Anweisungen an das Bewegungssystem, aus Anfragen an den ActionTransformer, aus Nachrichten an das IntelligenceSystem und aus Effektmeldungen an das Blackboard.

### 4.6.3 ActionTransformer

Aktionen beschreiben quasi atomares Verhalten auf hoher Abstraktionsstufe. Die Aufgabe des ActionTransformer ist es, Aktionen in Anweisungen umzuwandeln, welche vom MotorSystem, dem unterliegenden Bewegungssystem, verstanden werden können.

Dieses System erhält vom ActionExecutor Aktionen und wandelt diese in Motorikanweisungen um. Um dies zu tun, kann es für Bewegungsaktionen das NavigationSystem benutzen. Als Antwort liefert es dann dem ActionExecutor eine Sequenz von Motorikanweisungen zurück.

### 4.6.4 NavigationSystem

Dieses System ist zuständig für Bewegungen der Kreatur. Hier werden Pfade berechnet und in entsprechende Motorikanweisungen umgewandelt. Es bietet der Kreatur die Möglichkeit, sich in der Welt zurechtzufinden und zu bewegen.

Das NavigationSystem erhält vom ActionTransformer Aktionen und vom IntelligenceSystem Informationen über bekannte Objekte und die Topographie der Umgebung. Die erhaltenen Aktionen werden dann in Motorikanweisungen, in Anweisungen für das Bewegungssystem, umgewandelt. Um an die benötigte Information über die Welt zu gelangen, kann das NavigationSystem auf das Wissen des CreatureBrain im IntelligenceSystem zugreifen. Als Ausgabe liefert es eine Sequenz von Motorikanweisungen an den ActionTransformer zurück.

Ein Beispiel: Nehmen wir an, wir haben eine Aktion, die besagt, die Kreatur solle sich zu einem Objekt X bewegen. Das NavigationSystem berechnet aufgrund der von der Aktion mitgelieferten Information und Daten über die Welt aus der KnowledgeBase einen Pfad von der aktuellen Position der Kreatur zur Position von Objekt X. Dieser Pfad wird dann in eine Sequenz von Motorikanweisungen umgewandelt. Eine solche sieht zum Beispiel wie folgt aus: drehe 30° nach rechts, gehe 3 Einheiten vorwärts, drehe 12° nach links, und so weiter. Diese Sequenz wird dem ActionExecutor via ActionTransformer zurückgeliefert.

## 4.7 Blackboard System

Die vierte Komponente des CreatureBrain ist das Blackboard[1]. Es hat die Aufgabe, den Nachrichtenaustausch zu ermöglichen. Ein solcher wird für Rückkopplung und Interaktion mit dem Benutzer benötigt.

### 4.7.1 Ablauf

Die Eingabe des Blackboard besteht aus Nachrichten welche von Sendern deponiert werden können und in diesem System gehalten werden. Die Ausgabe besteht aus der Übergabe dieser Nachrichten an ihre Empfänger.

Da alle Information über und von der Welt via InputSystem in das CreatureBrain kommen muss, ist als Empfänger von Nachrichten primär das InputSystem interessant. Das InputSystem braucht zu diesem Zweck natürlich einen speziellen Blackboard Adaptor, wie in Abschnitt 4.4.2 motiviert wurde.

Der Mechanismus zur Nachrichtenübergabe des Blackboards wird in der gewählten Architektur für zwei grundsätzlich unterschiedliche Vorgänge benutzt:

### 4.7.2 Benutzereingabe

Das Blackboard System wird benutzt, um Kommunikation mit dem Benutzer, welcher ausserhalb von Kreatur und CreatureBrain existiert, zu ermöglichen. Vom Benutzer mitgeteilte Anweisungen gelangen via Blackboard in das CreatureBrain.

Durch das Blackboard System können also beispielsweise Ziele mitgeteilt werden. Benutzereingaben werden hier deponiert und dann via InputSystem, welches das Blackboard abfragt und allfällige Nachrichten vom Blackboard ausliest, in das CreatureBrain gebracht.

### 4.7.3 Rückkopplung

Der zweite Zweck, welchen das Blackboard erfüllt, ist die Rückkopplung von Effekten der Aktionen. Diese Möglichkeit ist nötig, da das InputSystem die einzige Informationsquelle für das CreatureBrain darstellt. Aktionen müssen nicht nur in Motorikanweisungen resultieren, sie können auch andere Effekte haben. Diese Effekte müssen vom ActionSystem bei der Ausführung einer Aktion auch in das IntelligenceSystem propagiert werden. Dafür ist dieser Rückkopplungsmechanismus vom ActionSystem zum InputSystem vorgesehen.

Zur Illustration: Wenn eine Aktion “Essen” ausgeführt wird, dann wird dazu wohl die Motorikanweisungen im Stil von “Mund öffnen” verwendet. Aber wenn die Aktion erfolgreich beendet wird, dann sollte sich auch der interne Zustand der Kreatur verändern, zum Beispiel wird ein Prädikat “nicht hungrig” gesetzt oder der Zähler für das Hungergefühl um eine Einheit verkleinert. Diese Information muss aber zuerst an das CreatureBrain System gelangen, und dies geschieht eben mit diesem Mechanismus.

## 4.8 Zusammenfassung

In diesem Kapitel werden die vier Komponenten der Architektur des CreatureBrain beschrieben. Das InputSystem ist für die Wahrnehmung von Information über die Welt zuständig. Das

1. Ein Blackboard wird hier anders verwendet als bei der in Abschnitt 3.3.3 vorgestellten Architektur, wo dieses ausschliesslich der internen Kommunikation dient. Hier wird ein Blackboard zur Rückkopplung und als Schnittstelle zum externen Benutzer verwendet.

IntelligenceSystem steuert aufgrund der Wahrnehmung und basierend auf internem Wissen und integrierter Steuerungslogik das Verhalten. Das ActionSystem ist für die Umsetzung des gewünschten Verhaltens zuständig, es wandelt abstrakte Aktionen in Bewegungen und andere Effekte um. Ein Blackboard ermöglicht Rückkopplung und Kommunikation mit einem Benutzer.

Auf die Implementierung dieser Architektur wird im folgenden Kapitel eingegangen. Die Integration des CreatureBrain in die Umgebung und das Aufsetzten auf das Bewegungssystem wird in Kapitel 6 dokumentiert. Kapitel 7 zeigt auf, was für konkrete Elemente zur Verhaltenssteuerung implementiert wurden.

# 5 Implementierung

In diesem Kapitel wird beschrieben, wie die in Kapitel 4 aufgezeigte Architektur umgesetzt wurde.

## 5.1 Allgemeines

Die Implementierung des Systems CreatureBrain erfolgte in C++ [2] [17] [18] unter Benutzung des Microsoft Visual Studios.

Um die Verwendung von Datenstrukturen zu vereinfachen, wurde vielerorts die C++ Standard Template Library (STL) [13] eingesetzt.

Eine komplette Dokumentation aller Klassen und deren Funktionen kann mit Hilfe von Doxygen [20] automatisch aus den Kommentaren im Quelltext erzeugt werden.

Die in Kapitel 4 beschriebene Architektur wurde direkt auf ein Programm abgebildet, jedes System oder Subsystem entspricht einer Klasse in C++. Dies gilt auch für Daten, Wahrnehmungen und Aktionen. Bei diesen wurde von Spezialisierung durch Vererbung Gebrauch gemacht. Die Funktionalität der einzelnen Systeme wurde im Kapitel 4 beschrieben, hier wird nur noch auf spezielle Aspekte der Implementierung eingegangen.

Der Begriff *Torque* taucht in diesem Kapitel mehrmals auf. Dies ist der Name der für die Simulation der Welt verwendeten Game Engine. Die Details des Zusammenspieles mit der Torque-Umgebung werden im nächsten Kapitel behandelt.

## 5.2 Übersicht

Das CreatureBrain wird durch eine gleichnamige Klasse repräsentiert. Diese wird, wie in Abschnitt 6.2.2 genauer beschrieben wird, durch die Klasse CreatureBrainThread instanziiert. Durch die Klasse CreatureBrain werden dann die vier Subkomponenten InputSystem, IntelligenceSystem, ActionSystem und Blackboard generiert:

```
//create the Brain Components
inputSys = new InputSystem();
intellSys = new IntelligenceSystem();
actionSys = new ActionSystem();
bbSys = new BBSystem();
```

Das Klassendiagramm in Abb. 5.1 stellt diese vier Klassen dar. Diese Subkomponenten instanziieren jeweils rekursiv ihre weiteren Subsysteme.

Das CreatureBrain funktioniert sequentiell in dem Sinn, dass aus der Klasse CreatureBrain periodisch nacheinander `InputSystem::run()`, `IntelligenceSystem::run()` und `ActionSystem::run()` aufgerufen und damit aktiviert werden.

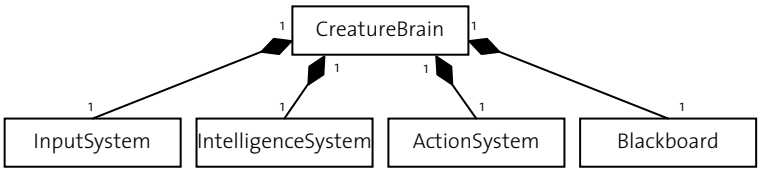



**Abbildung 5.1:** Klassendiagramm CreatureBrain.

## 5.3 InputSystem

Abb. 5.2 zeigt, wie das InputSystem auf Klassen abgebildet wurde. In der Methode `InputSystem::run()` werden durch `SensorSystem::senseAll()` und `PerceptSystem::processPercepts()` die beiden Subsysteme aktiviert.

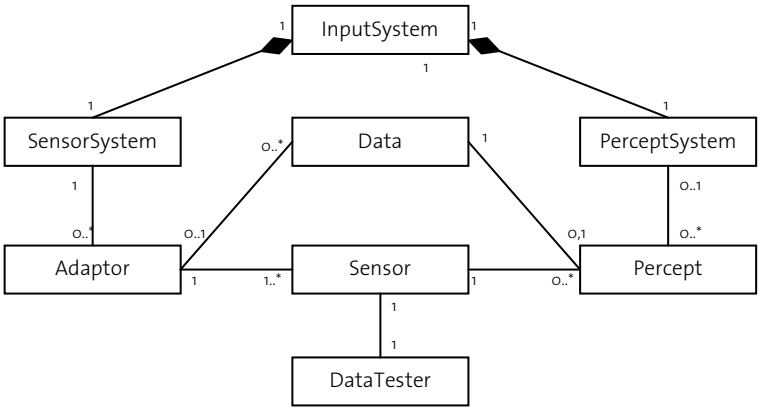



**Abbildung 5.2:** Klassendiagramm InputSystem.

**Sensor**

Die Klasse Sensor als Filter hat einen DataTester assoziiert. Dieser stellt die eigentliche Filterfunktion dar, mit welcher einkommende Data-Objekte beurteilt werden. Daten werden dem Sensor in Form eines Vektors von Data-Objekten mit der Methode `Sensor::sense()` übergeben.

**Adaptor**

Die Klasse Adaptor ist für die Beschaffung von Informationen verantwortlich. Sie tut dies mit der Methode `Adaptor::getData()`. Diese Methode wird in jedem Zyklus aufgerufen und packt die Daten in Data-Objekte, welche dann dem oder den Sensoren übergeben werden. Folgende Spezialisierungen der Klasse Adaptor wurden implementiert:

- Der **TorqueAdaptor** holt von der Torque-Umgebung die benötigten Daten. Wie dieser Mechanismus funktioniert, wird in Abschnitt 6.4 beschrieben.
- Der **BBAdaptor** bezieht Daten vom Blackboard. Er holt dort Nachrichten ab, welche für das InputSystem bestimmt sind.

**Data**

Die Klasse Data steht für eine Informationseinheit im CreatureBrain. Data bildet die Grundklasse für die in Abb. 5.3 dargestellten Spezialisierungen.

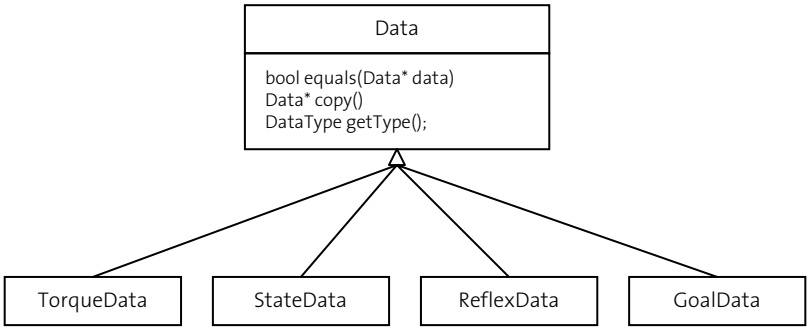



**Abbildung 5.3:** Die Klasse Data und ihre Ableitungen.

- **TorqueData** stellt Torque-Objekte dar. Es werden Torque-ID, Position, Orientierung, Farbe und Form festgehalten.
- **StateData** repräsentiert ein Element des internen Zustandes. Es wird beispielsweise die Anzahl der ausgeführten Aktionen gezählt.
- **ReflexData** hält Daten, welche einen Reflex darstellen.
- **GoalData** steht für ein Ziel.

**DataTester**

Die Klasse DataTester implementiert die Filterfunktion der Sensoren. Dabei ist die Methode `bool DataTester::test(Data* data)` interessant, welche Daten auf die jeweiligen Bedingungen testet. Für Erweiterungen der Klasse Data wurde ein entsprechender DataTester implementiert. Von DataTester existieren die folgenden Spezialisierungen:

- **TorqueDataTester** testet, ob die Daten Torque-Daten sind und unterscheidet, ob es sich bei den Daten um Kreaturdaten oder andere handelt.
- **TorqueReflexTester** ist eine spezielle Testfunktion, welche testet, ob die Kreatur zu nahe an ein als gefährlich eingestuftes Torque-Objekt geraten ist.
- **BBDataTester** erkennt in Nachrichten vom Blackboard zusätzlich mitgelieferte Daten.

**Percept**

Die Klasse Percept repräsentiert Wahrnehmungen des CreatureBrain. Die Klasse Percept hat ein Data-Objekt assoziiert, welches die Daten repräsentiert, aufgrund welcher eine Wahrnehmung gemacht wurde. Auf diese Daten kann mit `Percept::getDataRef()` zugegriffen werden. Percepts werden in der KnowledgeBase durch die Methode `Percept::process()` verarbeitet. Eine Auflistung aller Erweiterungen der Grundklasse Percept ist in Tabelle 5.1 gegeben. Diese stellen mögliche Wahrnehmungen dar.

**Ablauf**

Adaptorobjekte beziehen von ihren Datenquellen Informationen, welche sie in Data-Objekte umwandeln und den ihnen assoziierten Sensoren weiterreichen. Diese filtern mit Hilfe ihrer DataTester die gewünschten Daten aus und generieren gegebenenfalls Percept-Objekte. Diese

werden von den Sensoren an das PerceptSystem geleitet, und von dort dann weiter an das IntelligenceSystem.

| Name | Entspricht Wahrnehmungen von |
|---|---|
| TorquePercept | Objekten in Torque |
| StatePercept | dem internen Zustand |
| GoalPercept | Zielen |
| PositionPercept | der Position der Kreatur in der Welt |
| ReflexPercept | allgemeinen Reflexreizen |
| EscapeReflexPercept | einem Reiz der zu Flucht veranlasst |
| StopReflexPercept | einem Reiz der zu Halten veranlasst |

**Tabelle 5.1:** Erweiterungen der Klasse Percept.

## 5.4 IntelligenceSystem

Das IntelligenceSystem wurde wie in Abb. 5.4 dargestellt auf Klassen abgebildet. Bei Aktivierung durch die Methode `IntelligenceSystem::run()` wird nacheinander `Brain::think()` und `Controller::run()` aufgerufen und die Kontrolle so diesen Subsystemen übertragen.

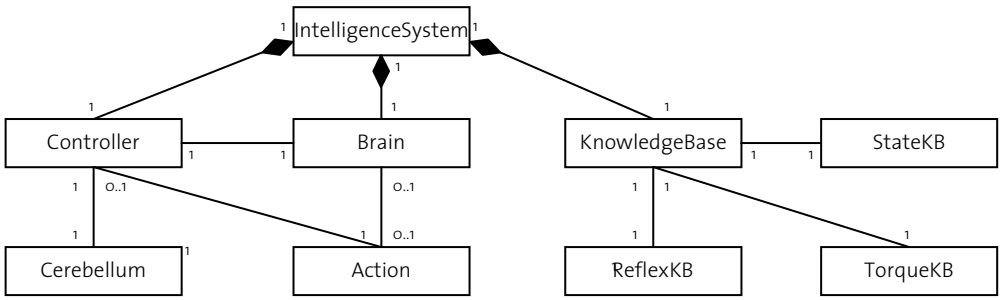



**Abbildung 5.4:** Klassendiagramm IntelligenceSystem.

**Die Struktur der KnowledgeBase**

Die KnowledgeBase hat mit StateKB, ReflexKB und TorqueKB drei Untersysteme, welche die jeweiligen Daten halten und nimmt Percept-Objekte durch die Methoden addGoalPercept(), addTorquePercept(), addReflexPercept(), addStatePercept() und addPositionPercept() auf. Für die Abfrage nach entsprechenden Wahrnehmungen existieren analoge ask-Methoden.

**Controller**

Der Controller testet in seinem run-loop durch den Aufruf von `KnowledgeBase::askReflex()` zuerst, ob ein neuer Reflex vorhanden ist. Falls dem so ist, liefert das Kleinhirn die entsprechende Aktion mittels `Cerebellum::getReflexAction()`. Falls kein Reflex vorliegt, werden Aktionen vom `Brain::getAction()` bezogen und dem ActionSystem zur Ausführung übergeben. In `Controller::setCurrentAction()` wird die aktuelle Aktion gesetzt. Dabei wird zuerst die Vorbedingung der Aktion geprüft (`Action::preCondition()`). Wenn diese nicht erfüllt ist, dann wird eine FreezeAction

gesetzt, deren Vorbedingung immer zutrifft. Wenn vom ActionSystem eine Aktion als erledigt gemeldet wird, dann wird analog auf die Erreichung der Nachbedingung geprüft (`Action::postCondition()`). Bei negativem Befund wird die aktuelle Aktion beibehalten.

**Action**

Die wichtigsten Methoden der Klasse Action sind in Abb. 5.5 dargestellt, `Action::feedback()` ist für Effekte von Aktionen zuständig und `Action::getDataRef()` liefert das assoziierte Data-Objekt. Die Methode `Action::evaluateValue()` ist gedacht als Evaluationsfunktion zur Bestimmung des Wertes einer Aktion in einem bestimmten Kontext und wird hier nicht weiter verwendet. Die Methoden zur Überprüfung von Vor- und Nachbedingungen sind per Default auf wahr gesetzt. Eine Anwendung von `Action::valid()`, wo die Gültigkeit einer Aktion im aktuellen Kontext überprüft wird, ist in Abschnitt 7.5 beschrieben.

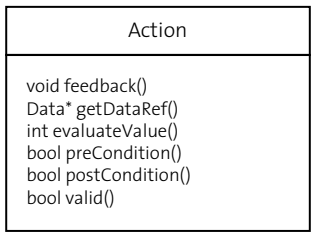



**Abbildung 5.5:** Die Klasse Action.

Action stellt für folgende Klassen die Oberklasse dar:

- **GoAction** für Bewegungen (Herumlaufen).
- **EscapeAction** für Fluchtbewegungen.
- **FreezeAction** hält die Kreatur an.
- **GrabAction** um Objekte zu greifen. (*Nicht verwendet.)*
- **PushAction** um Objekte zu drücken. *(Nicht verwendet.)*

**Brain**

Das Brain hält sich eine Liste von auszuführenden Aktionen (b_actionList) und eine Liste mit Zielen (b_goalDataList). Falls ein Ziel verfolgt wird, entspricht die Liste von Aktionen dem aktuellen Plan. Wenn kein Ziel verfolgt wird, wird in dieser Liste die aktuell gewählte Aktion festgehalten. Die Implementierung der Umwandlung eines bestimmten Ziels in Aktionen wird in Abschnitt 7.4 genauer betrachtet.

## 5.5 ActionSystem

Die Abbildung des ActionSystem auf Klassen ist in Abb. 5.6 dargestellt.

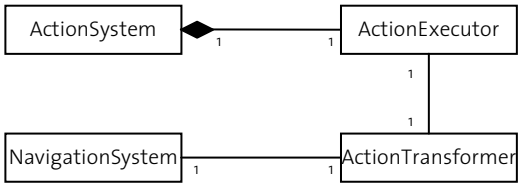



**Abbildung 5.6:** Klassendiagramm ActionSystem.

Das ActionSystem (Abb. 5.6) übergibt im Aufruf von `ActionSystem::run()` durch `ActionExecutor::run()` die Kontrolle direkt an den ActionExecutor, welcher zuerst prüft, ob seine aktuell ausgeführte Aktion (b_currentAction) noch der vom Controller gewünschten entspricht. Falls dies nicht zutrifft, setzt er seine aktuelle Aktion mittels `Controller::getCurrentAction()` neu. Aktionen werden durch `ActionTransformer::getMAVector()` in einen Vektor von Motorikanweisungen umgewandelt, welcher dann von der Methode `ActionExecutor::executeAction()` sequentiell abgearbeitet wird.

## 5.6 Blackboard

Die Klasse BBSystem (Abb. 5.7) stellt das eigentliche Blackboard dar. Sie ermöglicht den Nachrichtenaustausch zwischen den verschiedenen Systemen.

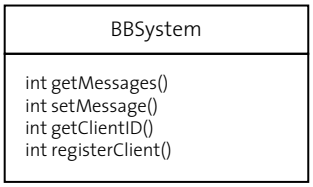



**Abbildung 5.7:** Die Klasse BBSystem.

Über die Klasse BBSystem werden Nachrichten ausgetauscht. Diese werden durch BBMessage und ihre Spezialisierungen (Abb. 5.8) für Ziele, allgemeine Informationen, Reflexe und Zustände modelliert.

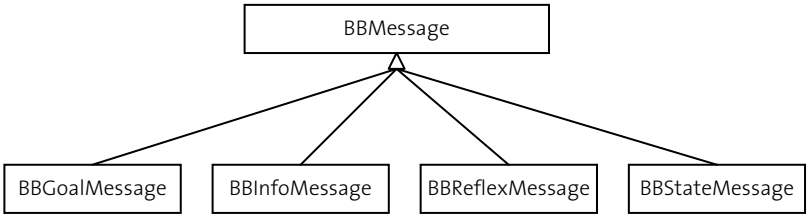



**Abbildung 5.8:** Die Klasse BBSystem und ihre Erweiterungen.

## 5.7 Zusammenfassung

Dieses Kapitel vermittelt einen Einblick in die Implementierung. Die Anforderungen und die Funktionalität des Systems CreatureBrain ist im Kapitel 4 beschrieben. Hier wurden einige Details der Implementierung erläutert und das Zusammenspiel der verwendeten Klassen in Klassendiagrammen dargestellt. Implementierungsaspekte der Integration in die Torque-Umgebung und die Schnittstelle mit dem Bewegungssystem *CreatureControl* werden im Kapitel 6 behandelt. Die Implementierung von Fähigkeiten, welche interessantes Verhalten unserer Kreatur ermöglichen, wird in Kapitel 7 beschrieben.

# 6

# Integration

Wie bereits in der Einleitung beschrieben, bestehen die zum Projekt CreatureZoo geleisteten Arbeiten aus drei Teilen. In diesem Kapitel wird nun die Integration des CreatureBrain in die beiden Systeme “Welt” und “Motorik” beschrieben. Zuerst wird die Einbindung des CreatureBrain in die Game Engine Torque [12] beschrieben, dann die Interaktion mit der CreatureControl [8].

## 6.1 Torque Game Engine

Die für dieses Projekt verwendete Torque Game Engine (“Torque”) der Firma GarageGames.com [7] stellt den Rahmen für die Simulation des CreatureZoos dar. Torque ist eine komplette Game Engine für die Darstellung von dreidimensionalen, virtuellen Spielwelten und basiert auf einem Client/Server System, so dass mehrere Clients in derselben Welt miteinander interagieren können. Auf dem Server geschieht die Simulation der Welt und auf den Clients wird jeweils ein Ausschnitt der Welt dargestellt. Torque ist plattformunabhängig und mit einer eigenen Skriptsprache und Editoren für Missionen, GUI’s, Terrains usw. ausgestattet.

**Abbildung 6.1:** Screenshot aus Tribes2, das auf Torque aufbaut.
(c) Sierra Entertainment, Inc.

Die zentrale Datenstruktur der Torque Game Engine ist der Szenengraph, in welchem alle Objekte der simulierten Welt als Knoten abgespeichert sind. Der Server besitzt immer den kompletten Szenengraph, die Clients jeweils die Objekte in der näheren Umgebung des Spielers. Diese Daten werden laufend zwischen dem Server und den Clients synchronisiert, um allen Clients eine konsistente Sicht der Welt zu bieten. Da nicht alle Daten permanent aktualisiert werden können, wird auf Seite der Clients eine Inter- und Extrapolation durchgeführt, um störende Artefakte beim Rendering zu vermeiden. Diese werden ebenfalls auf dem Server durchgeführt, um den Zeitpunkt einer Aktualisierung zu bestimmen. Die Daten des Servers beschreiben immer die verbindliche Sicht der Welt, die Clients arbeiten mit unverbindlichen Kopien. Mehr und genauere Informationen über für diese Arbeit relevante Aspekte von Torque sind in [12] zu finden.

## 6.2 Torque - CreatureBrain

Das CreatureBrain wurde entwickelt, um als Steuerzentrale oder als “Gehirn” für Kreaturen im CreatureZoo eingesetzt zu werden. Hier wird nun ausgeführt, wie diese Steuerungskomponente in die virtuelle Umgebung einer Spielwelt von Torque eingebunden wurde, wovon das Resultat in Abb. 6.2 zu sehen ist.

### 6.2.1 Grundsätze

- Das CreatureBrain existiert clientseitig. Torque wurde so angepasst, dass pro Client nicht ein benutzergesteuerter Spieler, sondern eine autonome Kreatur beim Server angemeldetet wird.
- Das CreatureBrain läuft in einem eigenen Thread, abgelöst von Torque. Damit soll erreicht werden, dass allfällig benötigte Rechenleistung für das CreatureBrain nicht den ganzen Torque-Client lahmlegen kann und so zum Beispiel das Rendering ausgebremst wird.
- Jegliche Kommunikation zwischen Torque und dem CreatureBrain geschieht mittels einer Verbindungsklasse namens CreatureBrainThread.

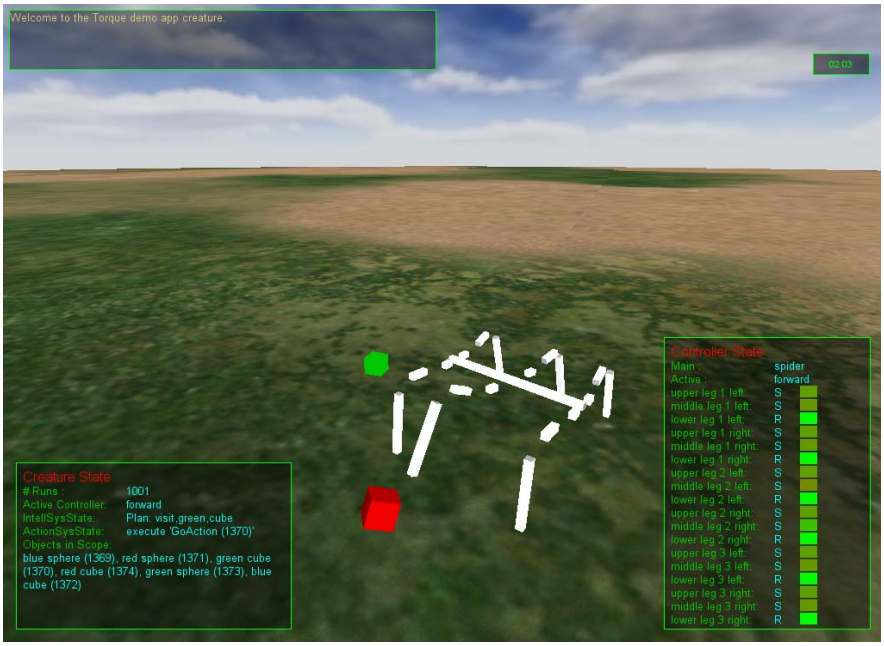



**Abbildung 6.2:** Screenshot der CreatureZoo Beispielumgebung.

### 6.2.2 Ein eigener Thread

Das CreatureBrain läuft neben Torque in einem eigenen Thread. Die Kommunikation zwischen diesen beiden Systemen, welche je in eigenen Threads existieren, spielt sich durch die Klassen *CreatureBrainThread* und *CreatureBrain* ab.

Im zeitlichen Ablauf sieht dies wie folgt aus:

In der **Initialisierungsphase** wird die Klasse CreatureBrainThread durch den *CreatureNode*, welcher in Torque unsere Kreatur repräsentiert, instanziiert. Dies geschieht nur clientseitig, da das CreatureBrain nur clientseitig existiert. In der Methode `CreatureNode::onAdd()` wird also die Klasse CreatureBrainThread instanziiert und durch den Aufruf von `CreatureBrainThread::startThread()` der eigentliche Thread auch gestartet. Es ist wichtig zu unterscheiden, dass die Klasse CreatureBrainThread nicht einen eigenen Thread darstellt, sondern einen solchen startet und kontrolliert. Beim Start des Threads wird durch folgende Zeile ein CreatureBrain instanziiert:

```
CreatureBrain *cb = new CreatureBrain();
```

Dieses CreatureBrainobjekt wird dann unverzüglich beim CreatureBrainThread registriert:

```
cbt->registerBrain(cb)
```

Dadurch ist durch die Klasse CreatureBrainThread, welche im gleichen Thread wie Torque läuft, eine Verbindung mit der Klasse CreatureBrain, welche in einem eigenen Thread lebt, hergestellt. Die Startphase ist damit vorbei und das System bereit für den Betrieb.

Während der **Betriebsphase** wird von der Klasse CreatureBrainThread periodisch die Methode `CreatureBrain::runLoop()` aufgerufen und damit das CreatureBrain aktiviert. Wie die Wahrnehmung der Torque-Welt durch das CreatureBrain funktioniert, wird im Abschnitt 6.4 erklärt.

In der **Endphase** wird durch `CreatureNode::onRemove()` die Methode `CreatureBrainThread::stopThread()` aufgerufen. Diese beendet den Thread und löscht das CreatureBrain.

## 6.3 Statusinformation

Um dem Benutzer Information über den aktuellen Status der Kreatur zu geben, wurde ein Status-Fenster für unsere Kreatur hinzugefügt (Abb. 6.3).


**Abbildung 6.3:** Das CreatureBrain Status-Fenster.

Durch Auflistung von zur Zeit aktiven Zielen und Aktionen sowie einer Liste von Objekten kann sich der Benutzer ein Bild vom aktuellen Zustand der Kreatur machen. Damit lässt sich ihr Verhalten besser nachvollziehen.

Die dort dargestellte Information wird durch die Klasse CreatureBrain via CreatureBrainThread in der Klasse *CreatureState* gesetzt. Für die Darstellung des Zustandes ist die Klasse *GuiCreatureCtrl* zuständig.

## 6.4 Wahrnehmung des CreatureBrain

**Grundsätzliches**

Unsere Kreatur soll von der Welt, in der sie lebt, etwas mitbekommen. Sie soll ihre Umgebung, oder mindestens einen Teil davon, wahrnehmen können. Im Gegensatz zur Robotik, wo zum Beispiel visuelle und akustische Wahrnehmung meist durch Bild- oder Spracherkennung funktioniert, haben wir hier den entscheidenden Vorteil, dass unsere Welt virtuell und klar definiert ist. Alle Objekte in unserer Welt sind grundsätzlich bekannt, denn Torque definiert sie alle. Um die Wahrnehmung durch das CreatureBrain einfach zu machen, brauchen wir nur Torque als Informationsquelle anzuzapfen.

**Implementierungsdetails**

Bei der Erweiterung von Torque durch die Open Dynamics Engine (ODE) [16] durch Moravanszky [12] wurde die Klasse Item, welche selber eine Generalisierung von diversen Torque-Objekten ist, durch die Klasse *DynamicsItem* erweitert. Objekte dieser Klasse werden in die Dynamiksimulation von ODE miteinbezogen.

Für unsere Simulation der Wahrnehmung wurde DynamicsItem noch einmal um die Klassen *CreatureNode* und *CreatureItem* erweitert. Die Klasse CreatureNode repräsentiert in Torque die Kreatur an sich und existiert demnach einmal pro Client. Der CreatureNode ist der Hauptknoten der Kreatur, seiner Position entspricht die Position der Kreatur. Die restlichen Glieder der Skelettarchitektur der Kreatur werden in Torque durch Objekte der Klasse DynamicsItem dargestellt. CreatureItems stellen allgemeine Objekte in der Welt dar, welche von der Kreatur wahrgenommen werden sollen und mit welchen die Kreatur interagieren kann.

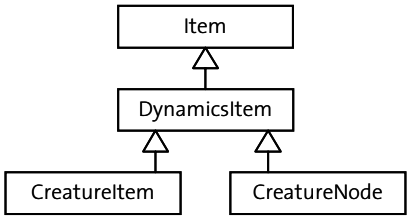



**Abbildung 6.4:** Ableitungen der Torque-Klasse Item.

Unsere Kreatur nimmt alle CreatureItem-Objekte in ihrer Umgebung wahr. Dies geschieht mit unserer eigenen Verbindungsklasse CreatureConnection. Über diese Klasse werden die Objekte in der Methode `CreatureConnection::addTraversalObject()` via die Methoden `CreatureNode::setObjectInScope()` und `CreatureBrainThread::setScopeObject()` in der Klasse CreatureBrainThread gesammelt. Von dort werden diese dann vom CreatureBrain gelesen.

Diese von Torque und dem CreatureBrain gemeinsam und asynchron genutzten Daten werden an der Verbindungsstelle durch Sicherstellung von exklusivem Zugriff vor Korrumpierung geschützt.

**Missionfiles**

Neue CreatureItems in der Umgebung lassen sich auf einfache Weise erzeugen. Für CreatureItems können auch neue Attribute hinzugefügt werden. Wir haben als Exempel je ein Attribut für Farbe und Form hinzugefügt (Methode `CreatureItem::initPersistFields()`). CreatureItems mit allen vordefinierten und eigenen Attributen lassen sich im Missionfile einfach einfügen. Diese Missionen werden dann serverseitig eingelesen und der jeweils relevante Ausschnitt davon an die Clients übertragen. Bei selbst definierten Attributen muss dabei sichergestellt werden, dass diese auch zu den Clients übertragen werden. Damit unsere Attribute für Form und Farbe auch clientseitig bekannt sind, haben wir die Methoden `CreatureItem::pakkUpdate()` und `CreatureItem::unpackUpdate()` entsprechend angepasst.

### 6.4.1 Konsolenkommando BrainCommand

Um dem Benutzer die Möglichkeit zu geben, mit dem CreatureBrain zu kommunizieren, wurde die in Torque vorhandene Konsole um den Befehl "BrainCommand" erweitert (siehe `CreatureNode::consoleInit()`). Dieses Kommando übernimmt als Argument eine Zeichenkette, welche dann der Klasse CreatureBrainThread übergeben wird (siehe `CreatureNode::brainCommand()`).

Diese Konsole ist in Torque als Standard für zeilenbasierte Steuerung vorgesehen. Wir haben sie hier benutzt, um die Kreatur von extern als Benutzer steuern zu können. Auf diesem Weg können dem CreatureBrain Ziele (Abschnitt 7.4) oder sonstige Anweisungen (Abschnitt 7.1 oder Abschnitt 7.6) mitgeteilt werden. Diese Kommandos werden via die Klasse CreatureBrainThread als Nachrichten im Blackboard deponiert, von wo sie durch einen speziellen Blackboard-Adapter vom CreatureBrain aufgenommen werden. Alle verfügbaren BrainCommand sind im Anhang A aufgelistet.

## 6.5 Motorik: Dynamik Simulation

Die Torque Game Engine wurde im Rahmen dieses Projektes durch die Open Dynamics Engine (ODE) [16], ein Open Source Dynamiksystem, erweitert [12], um in Torque eine physikalische Simulation von Kreaturen zu ermöglichen, welche von Heidelberger implementiert wurde [8]. Das heisst, unsere Kreaturen bewegen sich basierend auf einer physikalischen Simulation, Objekte bewegen sich aufgrund von einwirkenden Kräften. Das Modul, welches für die Bewegungen der Kreatur zuständig ist, nennen wir CreatureControl.

### 6.5.1 Bewegungsanweisungen als Grundbausteine des Verhaltens

Für das Verhalten einer Kreatur spielt die Art der Implementierung des Bewegungssystems keine Rolle. Wichtig ist, dass die Schnittstelle zwischen CreatureControl und CreatureBrain bekannt ist. Vereinbarte Bewegungsanweisungen dienen dann als Grundbausteine des Verhaltens der Kreatur. Wir haben uns entschieden, Bewegungen auf folgender Stufe zu abstrahieren: vorwärts und rückwärts bewegen, drehen, greifen. Das heisst, das Modul CreatureControl versteht Anweisungen auf diesem, an Turtle-Graphics angelehnten, Level. Dies schien uns sinnvoll, da eine Abstraktion auf tieferer Stufe (z.B.: "mache einen Schritt" oder "hebe das vordere linke Bein") dem CreatureBrain zuviel Wissen über die Geometrie der Kreatur abverlangt und es Aufwand betreiben muss, um geeignete Anweisungsfolgen zu berechnen. Eine Abstraktion auf höherer Stufe (z.B.: "Gehe zum Punkt X" oder "Greif Objekt X") schien uns ebenfalls unpassend, da damit zuviel Logik beim CreatureControl Modul liegen würde, denn die Navigation soll Aufgabe des CreatureBrain sein. Das CreatureBrain legt die Bewegung fest und das

CreatureControl Modul führt sie aus. Durch diese einfache Trennung soll auch ein Aufsetzen auf andersartige Bewegungssysteme einfach möglich sein.

### 6.5.2 Nachrichten an die Motorik

Anweisungen vom CreatureBrain an die CreatureControl und die Antworten in umgekehrter Richtung werden in Form von Nachrichten gesandt. Dafür wird die Klasse `CreatureControl::Message` verwendet. Nachrichten vom CreatureBrain an die CreatureControl werden Anweisungen (`Instructions`) genannt. Die CreatureControl antwortet auf solche durch Nachrichten an das CreatureBrain, welche wir Ereignisse (`Events`) nennen. Der Austausch dieser Nachrichten erfolgt ebenfalls über die Klasse CreatureBrainThread, wo für Anweisungen und Ereignisse je eine has-, get- und set-Methode implementiert ist. Ereignisse beziehen sich durch Angabe einer Identifikationsnummer auf die Anweisung mit derselben Nummer.

Das CreatureControl Modul kann die folgenden Anweisungen interpretieren: `forward` (vorwärts laufen), `backward` (rückwärts laufen), `rotate left` (drehen nach links), `rotate right` (drehen nach rechts) und `relax` (Ruhestellung einnehmen). Diese beinhalten eine Zeitangabe (`duration`), welche angibt, wie lange die gewünschte Bewegung ausgeführt werden soll.

Zur Steuerung der Kommunikation haben wir folgende Konvention für die Namen von Ereignissen festgelegt:

- **success.** Die Anweisung wurde verstanden und kann interpretiert und ausgeführt werden.
- **done.** Die Anweisung wurde für die gewünschte Zeitdauer erfolgreich ausgeführt.
- **fail.** Die Anweisung konnte nicht ausgeführt werden.
- **error.** Die Anweisung wurde nicht verstanden. Die gewünschte Anweisung ist zum Beispiel nicht vorhanden oder es liegt ein Syntaxfehler vor.

Im Falle eines fail- oder error-Ereignisses wird eine entsprechende Fehlermeldung ausgegeben. Abb. 6.5 illustriert den zeitlichen Ablauf einer erfolgreichen Nachrichtenfolge.

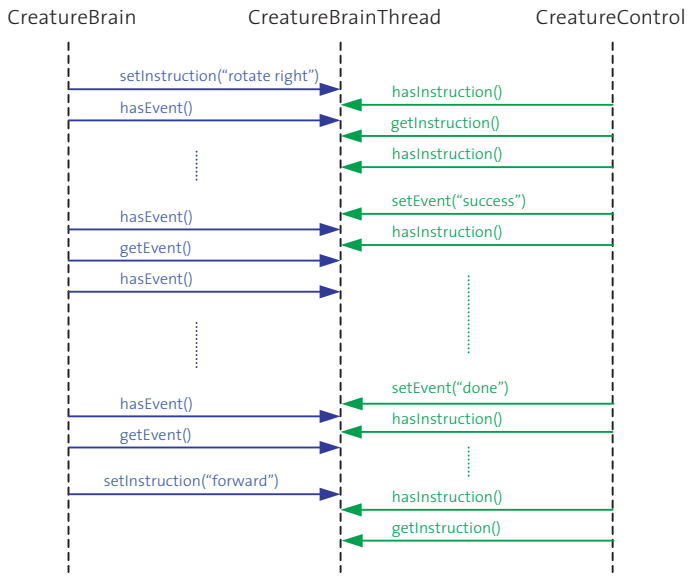



**Abbildung 6.5:** Ablauf der Kommunikation zwischen CreatureBrain und CreatureControl.

## 6.6 Resultate

Das CreatureBrain konnte erfolgreich in Torque eingebunden werden. Es existiert clientseitig neben Torque in einem eigenen Thread. Die Kommunikation zwischen diesen beiden Systemen wird durch die Klasse CreatureBrainThread ermöglicht. Um das physikalische Modell der Kreatur in Torque zu steuern, wurde das CreatureBrain des Weiteren erfolgreich auf das CreatureControl Modul aufgesetzt. Durch einfache Bewegungskommandos des CreatureBrain an das Bewegungssystem kann so das Verhalten der Kreatur gesteuert werden.

Durch diese Integration kann unsere Kreatur nun zusammenfassend ihre virtuelle Umgebung wahrnehmen und sich darin bewegen. Die beiden in Abschnitt 4.1 beschriebenen Schnittstellen des CreatureBrain konnten erfolgreich umgesetzt werden.

# 7 Anwendung des CreatureBrain

Dieses Kapitel beschreibt, welche Art von Steuerlogik und welches Wissen in unserem CreatureBrain implementiert wurde. Dies ist als Anwendung der in Kapitel 4 beschriebenen Architektur zu verstehen. Es wird hier gezeigt, wie sich in der beschriebenen Architektur Wissen zur Steuerung von Verhalten einbauen lässt. Zuerst wird gezeigt, wie zwei verschiedene Arten von Reflexreizen in der Architektur umgesetzt wurden und so das Verhalten gesteuert werden kann. Im Weiteren wird dann ausgeführt, mit welcher Funktionalität das NavigationSystem ausgestattet wurde und wie vom Brain ein einfaches Ziel in einen Plan umgewandelt wird. Den Abschluss dieses Kapitels bildet ein Abschnitt über die Kalibrierung des CreatureControl Moduls durch das CreatureBrain.

## 7.1 Stop'n'Go-Reflex

Wie im Kapitel 4 beschrieben, hat Verhalten auf der Stufe Reflex die höchste Priorität. Wenn ein Reflexreiz vorliegt, wird automatisch darauf reagiert, unabhängig davon was sonst gerade verfolgt wird. Dieser Mechanismus kann nun natürlich auch vom Benutzer zur Steuerung der Kreatur verwendetet werden. Um dies zu illustrieren und auch um den gesamten Reflex-Mechanismus des Systems zu testen, wurde ein vom Benutzer generierter Reflexreiz implementiert. Dieser "stop" Reflex ist so ausgelegt, dass er die höchste Priorität unter den verschiedenen Reflexen hat.

Um die Benutzereingabe in das CreatureBrain System zu propagieren, wurde wiederum der in Abschnitt 6.4.1 beschriebene Mechanismus verwendet. Der Benutzer kann mit dem Brain-Command "stop" einen Reflexreiz auslösen, und mit "go" wieder aufheben. Im InputSystem reagiert ein am BBAdaptor angeschlossener Sensor auf diese Nachricht und generiert darauf ein StopReflexPercept, welche dann vom PerceptSystem an die KnowledgeBase weitergeleitet wird. Dort wird dieses vom Controller des IntelligenceSystem erkannt und das Cerebellum zu Hilfe beigezogen. Die Antwort des Cerebellum auf diesen Reflex ist eine FreezeAction. Diese wird dann im ActionSystem vom ActionTransformer in eine "relax" Motorikanweisung umgewandelt. Abb. 7.1 illustriert diesen Sachverhalt.

Ein kleines Problem ergab sich bei der Implementierung dieses Reflexes. Auf einen Reflexreiz wird nur so lange reagiert, wie dieser da ist. Nun war es aber weder möglich noch wün-

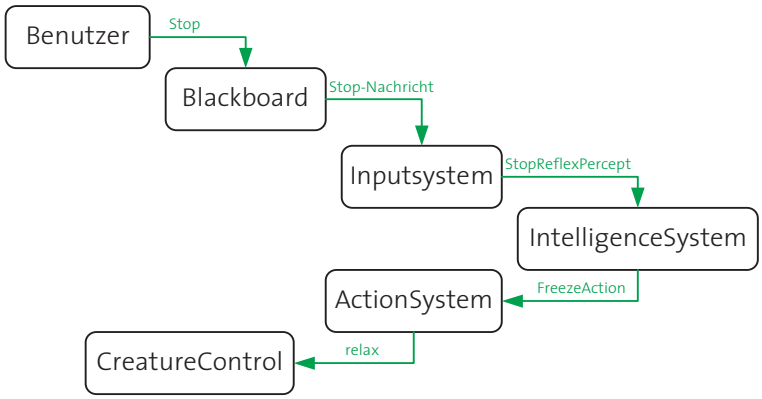



**Abbildung 7.1:** Schematischer Ablauf “Stop-Reflex”.

schenswert, dass der Benutzer in jedem Zeitschritt ein “stop” mitteilt. Deshalb musste der entsprechende Reiz irgendwo gehalten werden: der BBAdaptor übernimmt dies und fügt in jedem Zyklus in der Liste der Nachrichten die entsprechende StopReflexnachricht wieder ein, falls eine vorhanden war:

```
if (stopReflex && !goReflex)
{
      msgs->addMessage(new BBReflexMessage("stop"));
}
```

Somit nimmt das InputSystem immer wieder neu einen Stop-Reflexreiz wahr, solange kein “go” Reflexreiz vom Benutzer mitgeteilt wird. Durch diesen Stop and Go Mechanismus lässt sich also die Kreatur jederzeit in ihrer aktuellen Aktion unterbrechen und fährt dann wieder mit dieser fort. Durch diesen vom Benutzer ausgelösten Reflexreiz kann die Kreatur direkt gesteuert werden. Die Steuerung des Verhaltens unserer Kreatur erfolgt grundsätzlich durch Ziele. Die hier aufgezeigte Möglichkeit kann als Spielerei in unserem System angeschaut werden, indem gewissermassen ein bestimmter Reflexreiz vom Benutzer zur unmittelbaren Verhaltenssteuerung missbraucht wird.

## 7.2 Escape-Reflex

Der oben beschriebene Stop-Reflex illustriert als Beispiel die Funktionsweise von Reflexen. Jedoch war die Reaktion auf diesen Reiz bloss ein Stillstehen der Kreatur. Um einen Reflex mit einer Bewegungsaktion zu erhalten, wurde ein Flucht-Reflex implementiert. Die Idee hierbei ist, dass die Kreatur bestimmten Objekten ausweichen soll.

Wenn also ein solcher Reiz ausgelöst wird, dann antwortet das Cerebellum darauf mit einer EscapeAction, welche dann vom ActionSystem in eine Ausweichbewegung (Abschnitt 7.3.2) umgewandelt wird.

Zu diesem Zweck wurde eine spezielle Filterfunktion für Sensoren, nämlich die Klasse TorqueReflexTester, geschrieben. Diese filtert Torque-Objekte heraus, welche bestimmte Bedingungen erfüllen und sich innerhalb eines bestimmten Radius zur Kreatur befinden. Der Reflexreiz wird solange aufrecht erhalten, bis sich die Kreatur in einem grösseren Abstand von diesem Objekt als dessen äusseren Gefahrenradius befindet. Die Verwendung von zwei Radien um dieses Gefahrenobjekt ist sinnvoll, um ein Oszillieren der Kreatur zu vermeiden.

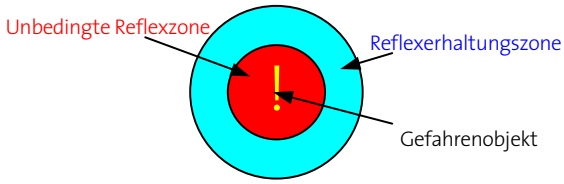



**Abbildung 7.2:** Die beiden Radien um ein Gefahrenobjekt.

Abb. 7.2 zeigt das Konzept von den zwei Radien um das Gefahrenobjekt. Sobald die Kreatur in den inneren, kleineren Kreis (rot) tritt, wird ein Reflexreiz ausgelöst. Dieser wird solange aufrechterhalten, bis die Kreatur den äusseren, grösseren Kreis (blau) verlassen hat.

Da auf Reflexreize nur solange reagiert wird, wie sie wahrgenommen werden, muss deren Aufrechterhaltung sichergestellt sein. Dies ist notwendig, da erstens Torque und das CreatureBrain asynchron zueinander funktionieren und zweitens die Aktualisierung der Torque-Objekte unregelmässig vor sich geht. Somit werden im allgemeinen solche gefährlichen Torque-Objekte nicht in jedem Zeitschritt vom CreatureBrain wahrgenommen.

Diese Aufrechterhaltung wurde hier anders umgesetzt als beim Stop-Reflex. Hier wird von der Möglichkeit von Aktionen, andere Effekte als Motorikanweisungen zu haben, Gebrauch gemacht. Wir betrachten die Aufrechterhaltung eines Escape-Reflexes als Effekt der EscapeAction, welche die Reaktion auf einen Escape-Reflex darstellt. Somit wird sichergestellt, dass durch den implementierten Rückkopplungs-Mechanismus via Blackboard immer wieder neue Escape-Reflexreize generiert werden. Die Methode `EscapeAction::feedback()` setzt eine BBReflexMessage, falls der äussere Radius um das Gefahrenobjekt nicht verlassen wurde:

```
BBReflexMessage *msg = new BBReflexMessage("escape");
msg->setTorqueData((TorqueData*)this->getDataRef()->copy());
msg->setReceiver(REC_CLIENT, "InputSystem");
bbSystem->setMessage(bbInputSystemID, msg);
```

Diese vom ActionSystem bei der Ausführung der EscapeAction im Blackboard gesetzte Nachricht wird dann im InputSystem von einem Sensor mit entsprechendem Filter (BBDataTester) erkannt und daraus wieder ein EscapeReflexPercept generiert. Somit wird durch diese Rückkopplung via Blackboard der Kreis geschlossen und der Escape-Reflex aufrecht erhalten.

## 7.3 NavigationSystem

Wie in Abschnitt 4.6.4 beschrieben, ist das NavigationSystem dafür verantwortlich, Bewegungsaktionen in Motorikanweisungen umzuwandeln. Dies ist notwendig, um der Kreatur ein sinnvolles Bewegen in der Welt zu ermöglichen.

Das NavigationSystem kann GeheZu- und Flucht-Aktionen verarbeiten. Konkret werden hier also GoActions und EscapeActions in Motorikanweisungen “forward”, “backward”, “rotate left” und “rotate right” umgewandelt, welche von der CreatureControl verstanden und ausgeführt werden können. Siehe Abschnitt 6.5.2 und [8].

### 7.3.1 GoAction

Die Klasse GoAction steht für einfache Bewegung der Kreatur. Durch die Ausführung einer GoAction soll sich die Kreatur vom aktuellen Standort zu einem vorgegebenen Ziel bewegen.

**Einfaches Drehen und Laufen**

In der vorliegenden Implementierung haben wir ein simples NavigationSystem umgesetzt. Die Idee ist, dass sich die Kreatur zuerst in Richtung des Ziels dreht und sich dann gerade darauf zu bewegt.

Es lag nicht im Rahmen dieser Diplomarbeit, ein komplexeres Navigationssystem zu implementieren, da die topographischen Anforderungen unserer flachen Welt ohne grössere Hindernisse dies auch nicht erfordert. Unser gewählter Ansatz ist deshalb einfach gehalten. Es werden weder Hindernisse umgangen, noch werden optimale Pfade gesucht, und auf die Topographie der Welt wird ebenfalls keine Rücksicht genommen.

**Implementierungsdetails**

Die Umwandlung von GoActions in Motorikanweisungen ist in folgender Methode der Klasse NavigationSystem umgesetzt:

```
MsgVector NavigationSystem::getGoMAVector(GoAction *goAction);
```

Diese Methode nimmt als Input einen Pointer auf ein Objekt der Klasse GoAction und liefert als Resultat einen Vektor mit Motorikanweisungen in Form von Nachrichten der Klasse `CreatureControl::Message` zurück.

Objekte der Klasse GoAction haben ihr Ziel assoziiert. Sie besitzen wie alle Action-Objekte eine Referenz auf ein Datenobjekt. Im Fall der GoAction ist dieses Datenobjekt ein Torque-Datenobjekt und kann wie folgt abgefragt werden:

```
TorqueData *targetData = (TorqueData*)goAction->getDataRef();
```

Analog dazu werden die Informationen über die Kreatur aus der KnowledgeBase bezogen:

```
TorqueData *creatureData = b_knowledgeBase->askCreatureData();
```

TorqueData ist die von uns verwendete Klasse für die interne Repräsentation von Torque-Objekten. Mehr darüber ist im Kapitel 5 zu finden.

Damit ist alle Information vorhanden. Jetzt kann von `creatureData` und `targetData` die spezifisch benötigte Information über die Weltkoordinaten der beiden Objekte abgefragt werden:

```
Point3F *targetPosition = targetData->getPosition();
Point3F *creaturePosition = creatureData->getPosition();
```

Point3F ist eine Torque-Klasse welche für Vektoren mit drei Dimensionen steht. Wir verwenden von dieser Klasse die Variablen x, y und z und die Methode len().

Die aktuelle Orientierung der Kreatur wird auch noch benötigt, da sich die Kreatur zuerst in die gewünschte Richtung drehen muss:

```
QuatF *creatureOrientation = creatureData->getOrientation();
```

QuatF stellt ein Quaternion [10] dar. Diese werden in Torque zur kompakten Darstellung von Rotationen und Translationen verwendet.

Mit diesen Daten lässt sich die Ausrichtung der Kreatur und der Verbindungsvektor Kreatur-Ziel bestimmen.

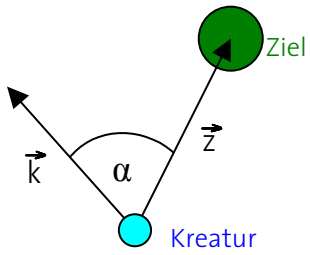



**Abbildung 7.3:** Der Richtungsvektor $\vec{k}$ der Kreatur und der Verbindungsvektor Kreatur-Ziel $\vec{z}$.

Daraus lässt sich nun mittels

$$\cos\alpha = \frac{\vec{k} \bullet \vec{z}}{|\vec{k}| \cdot |\vec{z}|} \tag{7.1}$$

der Winkel $\alpha$ zwischen den zwei Vektoren $\vec{k}$ und $\vec{z}$ bestimmen.

In der aktuellen Implementierung dreht sich die Kreatur immer um weniger als 90° und bewegt sich damit gegebenfalls rückwärts auf ein Ziel zu. Mit Hilfe des Vektorprodukts $\vec{k} \times \vec{z}$, welches zwischen dem Richtungsvektor $\vec{k}$ der Kreatur und dem Verbindungsvektor $\vec{z}$ zwischen Kreatur und Ziel berechnet wird, lässt sich die Drehrichtung feststellen. Die Grundebene unserer Welt liegt parallel zur xy Ebene und mit Hilfe der z-Komponente des Vektorprodukts lässt sich nun feststellen, ob es sich um eine Rechts- oder um eine Linksdrehung handelt. Es tritt also

| **Fall** | | **Bewegung** | **Drehung** |
|---|---|---|---|
| $z < 0$ | $\alpha < 90°$ | vorwärts | rechts |
| $z < 0$ | $\alpha > 90°$ | rückwärts | links |
| $z > 0$ | $\alpha > 90°$ | rückwärts | rechts |
| $z > 0$ | $\alpha < 90°$ | vorwärts | links |

**Tabelle 7.1:** Die vier möglichen Fälle für die Dreh- und Bewegungsrichtung.

einer der vier in Tabelle 7.1 aufgeführten Fälle auf. Entsprechende Nachrichten werden erstellt und, wie in Abschnitt 6.5 erklärt, an die CreatureControl gesandt.

### 7.3.2 EscapeAction

EscapeActions modellieren Flucht oder Ausweichen vor einem anderen Objekt. Im Gegensatz zu GoActions steht bei EscapeActions das assoziierte Datenobjekt nicht für das Ziel sondern genau für das Objekt, welches eine Gefahr darstellt, also den zu meidenden Ort.

**Implementierung**

Die Umwandlung von EscapeActions in Motorikanweisungen spielt sich analog zur Umwandlung von GoActions ab. Hier wird im Gegensatz zur GoAction unterschieden, ob die Kreatur auf ein zu meidendes Objekt zu läuft oder sich bereits davon entfernt. Die Unterscheidung zwischen dem Gebiet, in welchem die Kreatur auf ein Objekt zu läuft und sich davon entfernt, ist in Abb. 7.4 dargestellt. Diese Unterscheidung ist nötig, da die Wahrnehmungsfrequenz durch das asynchron laufende Torque stark variieren kann. Es kann durchaus vorkommen, dass sich die Kreatur innerhalb des Gefahrenradius bereits von einem Gefahrenobjekt entfernt, bis dieses als solches wahrgenommen wird.

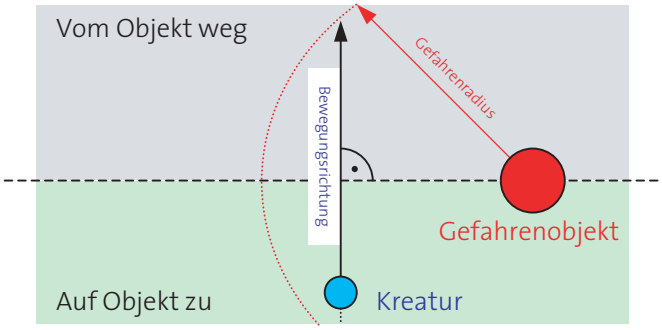



**Abbildung 7.4:** Auf ein Objekt zulaufen und sich davon wegbewegen.

Wenn sich die Kreatur bereits vom Gefahrenobjekt entfernt, dann behält sie ihren Kurs bei ohne zu drehen. Falls sie sich aber darauf zu bewegt, dann wird das Objekt gemieden, und die Kreatur wird auf ihrer Bewegungsbahn, wie in Abb. 7.5 aufgezeigt, tangential abgelenkt.

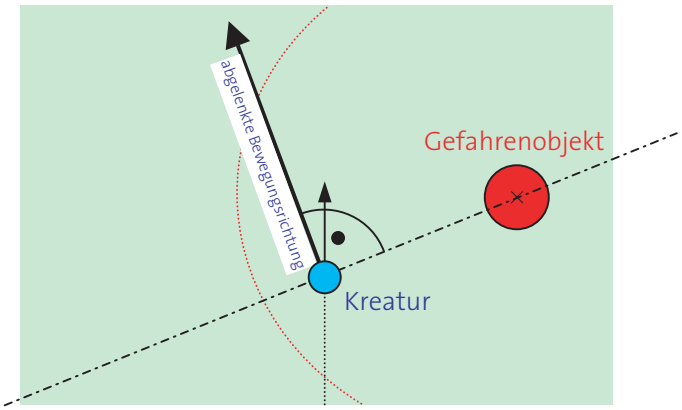



**Abbildung 7.5:** Bewegt sich die Kreatur auf das Gefahrenobjekt zu, wird sie tangential abgelenkt.

Das Fluchtverhalten wurde hier so gewählt, weil radiales Entfernen vom Gefahrenzentrum als einfache und naheliegende Alternative im Allgemeinen nicht sehr sinnvoll erscheint. Die Kreatur macht dadurch oftmals, wenn sie sich schon vom zu meidenden Objekte weg bewegt, viel zu grosse und unnatürlich anmutende Ausweichbewegungen, oder sie kann sich unter Umständen gar nicht um ein Gefahrenobjekt herumbewegen.

## 7.4 Ziel "visit"

Wie im Kapitel 4 beschrieben, wird die Kreatur mit Zielen gesteuert. Um die dazu nötigen Mechanismen auch ohne komplexes Planungs- und Planvalidationssystem zu testen, wurde das einfache Ziel "visit" umgesetzt.

### 7.4.1 Grundidee

In der von uns verwendeten Testumgebung haben wir farbige Objekte verschiedener Formen eingefügt. Diese Objekte zu besuchen erschien uns ein sinnvolles Ziel, um unser System zu testen. Um die Kreatur dabei etwas geschickter erscheinen zu lassen, sollen nur Objekte besucht werden, welche bestimmte Eigenschaften erfüllen, und diese Objekte werden in sinnvoller Reihenfolge besucht. Der Benutzer hat dadurch die Möglichkeit, dem CreatureBrain zur Laufzeit Ziele mitzuteilen, welche dann von diesem verfolgt werden.

### 7.4.2 Umsetzung

Zuerst braucht der Benutzer die Möglichkeit, dem CreatureBrain ein Ziel mitzuteilen. Für die Übergabe eines Zieles wird das in Abschnitt 6.4.1 beschriebene, von uns hinzugefügte Torque-Konsolenkommando "CreatureBrain" und der entsprechende Mechanismus via Blackboard verwendet. Auf diese Art finden Ziele ihren Weg vom Benutzer in die KnowledgeBase des CreatureBrain.

Umgewandelt und ausgeführt wird das Ziel dann durch die Klasse Brain. Diese Klasse prüft durch die in jedem Zyklus aufgerufene Methode `Brain::think()`, ob in der KnowledgeBase ein Ziel vorhanden ist. Falls es "visit" Ziel heisst, wird es in die Liste der von dem Brain zu verfolgenden Ziele aufgenommen.

Der Plan zur Erreichung dieses Zieles wird durch die Methode `Brain::planGoal()` erstellt. Die Umwandlung eines Ziels "visit" in eine Sequenz von Aktionen geschieht in den folgenden Schritten:

1. Alle in der Welt vorhandenen Torque-Objekte werden von der KnowledgeBase geholt. Dies sind die potentiellen Zielobjekte. (`KnowledgeBase::askTorqueData()`).
2. Von den potentiellen Zielobjekten werden Objekte mit unpassenden Attributen ausgefiltert. (`Brain::filterColor()` und `Brain::filterForm()`).
3. Die verbleibenden Objekte werden mittels der Nearest Neighbor Methode geordnet. (`Brain::orderTargets()`).
4. Für jedes Objekt wird eine GoAction in die Liste der auszuführenden Aktionen hinzugefügt. Diese Liste stellt den eigentlichen Plan zur Erfüllung dieses "visit" Zieles dar.

Diese geplanten Aktionen werden nun sequentiell abgearbeitet. Wenn ein Plan abgearbeitet worden ist, wird das nächste Ziel verfolgt. Während dieser Abarbeitung wird keine Validierung des Plans und somit auch kein Wiederplanen vorgenommen. Im Gegensatz dazu werden Aktionen jedoch, wie der folgenden Abschnitt aufzeigt, auf ihre Gültigkeit überprüft.

## 7.5 GoAction Validierung

Aktionen werden während ihrer Ausführung validiert. Es macht keinen Sinn, dass eine Aktion ausgeführt wird, welche im aktuellen Kontext unmöglich wird, da sich dieser verändert hat. In der vorliegenden Implementierung wird diese Möglichkeit anhand der GoAction aufgezeigt.

Während der Ausführung einer GoAction wird vom ActionExecutor periodisch deren Gültigkeit geprüft. Die Gültigkeit einer GoAction wird dabei mittels zwei Fragen eruiert:

1. **Hat sich das Ziel bewegt?**

Hier wird geprüft, ob sich die Koordinaten des Zielobjektes, auf welches sich die GoAction bezieht, verändert haben. Dies geschieht in der Methode `GoAction::valid()`. Das ist möglich, da die GoAction eine Referenz auf ihr Zielobjekt hat und somit auf dessen aktuelle Koordinaten zugreifen kann.

2. **Ist die Kreatur auf einem falschen Kurs?**

Um festzustellen, ob sich die Kreatur auf dem richtigen Kurs befindet, wird überprüft, ob der Winkel zwischen Kreatur und Ziel nicht zu gross ist. Dazu sind neben den Koordinaten des Zielobjekts auch jene der Kreatur und deren Ausrichtung notwendig. Dieser Test wurde in der Methode `NavigationSystem::goActionValid()` implementiert. Befindet sich die Kreatur weit weg von einem Ziel, dann sollte sie sich relativ genau auf das Ziel zu bewegen. Wenn die Kreatur aber nahe am Ziel ist, dann spielt ein relativ grosser Abweichungswinkel vom Ziel keine Rolle mehr. Deswegen hängt der maximal zulässige Abweichungswinkel vom Abstand Kreatur - Ziel ab, wie Abb. 7.6 illustriert. Wir haben die Abhängigkeit des Winkels von dieser Distanz als linear angenommen, im Abstand von 40 ist eine Abweichung von bis 10° tolerierbar, bei einem Abstand von 10 maximal 40°.

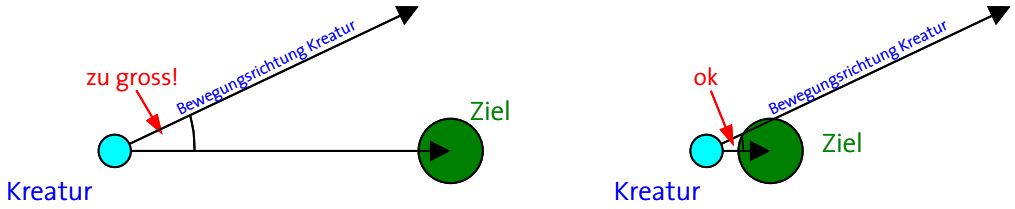



**Abbildung 7.6:** Der maximal zulässige Abweichungswinkel der Kreatur ist von der Distanz zum Ziel abhängig.

### 7.5.1 Revalidierung

Wenn nun eine dieser zwei Gültigkeitsfragen mit “ja” beantwortet wird, dann startet eine Revalidierung der GoAction, indem diese noch einmal vom NavigationSystem in Motorikanweisungen umgewandelt wird. Dabei werden natürlich sowohl die aktuellen Ziel- als auch Kreaturdaten verwendet.

Mit diesem Mechanismus wird einerseits sichergestellt, dass sich die Kreatur nicht zu einem Ort hin bewegt, wo sich kein Ziel-Objekt mehr befindet, sondern dieses eher verfolgt. Andererseits kann sich die Kreatur dadurch nicht in eine komplett falsche Richtung bewegen, falls sie beispielsweise durch das Terrain vom Kurs abgekommen ist.

## 7.6 Kalibrierung

### 7.6.1 Motivation

Die Schnittstelle des CreatureBrain Systems zur unterliegenden CreatureControl [8] ist absichtlich offen gehalten. Das CreatureBrain System soll grundsätzlich auf verschiedenen Kreaturen mit unterschiedlichem Aufbau und Bewegungsarten funktionieren. Es ist vorgesehen, dass das CreatureBrain in einer Initialisierungsphase die CreatureControl abfragt, was für Anweisungen zur Verfügung stehen. Die Effekte dieser theoretisch unbekannten Anweisungen könnten von der CreatureControl mitgeteilt oder vom CreatureBrain gelernt werden.

### 7.6.2 Vorgehen

In unserer Implementierung haben wir folgenden Ansatz gewählt:

- Die CreatureControl kennt die Effekte ihrer Anweisungen nicht. Diese müssen zuerst vom CreatureBrain gelernt werden.
- Folgende Anweisungen müssen vorhanden sein: vorwärts bewegen, rückwärts bewegen, drehen links, drehen rechts. Falls diese nicht vorhanden sind, kann sich das CreatureBrain nicht richtig initialisieren und in der Folge nicht korrekt funktionieren.
- Zusätzliche Anweisungen wie “greifen” oder “drücken” sind vorgesehen, werden aber bisher nicht verwendet, da sie von der CreatureControl nicht angeboten werden.

Das heisst, das CreatureBrain muss bei der Initialisierung die aktuelle CreatureControl kalibrieren, um die entsprechenden Anweisungen und dazugehörigen Parameter richtig setzen zu können.

Die gemäss Abschnitt 6.5.2 implementierten Motorikanweisungen forward, backward, rotate left und rotate right brauchen als Parameter je eine Zeitangabe, welche angibt, wie lange diese Bewegungen ausgeführt werden sollen. Das CreatureBrain, speziell das NavigationSystem, arbeitet aber nicht primär mit Zeitangaben, sondern mit Koordinaten, welche sich in Distanzen und Winkel umrechnen lassen.

In der Initialisierungsphase testet das CreatureBrain, ob die CreatureControl die vier oben genannten Anweisungen anbietet. Falls dies nicht zutrifft, ist das System in einem Fehlerzustand und funktioniert nicht korrekt. Dies wird durch einer entsprechende Fehlermeldung mitgeteilt. Andernfalls werden diese Anweisungen während einer bestimmten Zeit (`ActionExecutor::b_duration`) ausgeführt und danach der Effekt (die zurückgelegte Distanz beziehungsweise der Drehwinkel) ausgewertet. Daraus kann sich das CreatureBrain dann die Umrechnung von Distanzen und Winkel in eine Zeitangabe extrahieren:

$$\text{DistanzFaktor} = \frac{\text{Distanz}}{\text{Zeit}} \tag{7.2}$$

beziehungsweise

$$\text{WinkelFaktor} = \frac{\text{Winkel}}{\text{Zeit}} \tag{7.3}$$

Diese Kalibrierung wurde als endlicher Zustandsautomat in der Klasse ActionExecutor implementiert.

### 7.6.3 Kalibrierungsdatei

Um die oben beschriebene Kalibrierung der Anweisungsparameter nicht bei jedem Start des CreatureBrain vornehmen zu müssen, werden die entsprechenden Umrechnungsfaktoren in einer Datei abgelegt. Wenn nun also für eine bestimmte CreatureControl schon eine Kalibrierungsdatei vorhanden ist, wird diese verwendet. Deshalb wird zuerst die Identifikation der aktuellen CreatureControl abgefragt und dann geschaut, ob für diesen Controller eine Kalibrierungsdatei vorhanden ist. Trifft dies zu, werden die entsprechenden Werte von dieser verwendet. Ansonsten wird der oben beschriebene Vorgang durchgeführt.

### 7.6.4 BrainCommand("calib")

Es gibt ebenfalls die Möglichkeit, diese Kalibrierung manuell zu starten, und zwar mit dem Konsolenkommando "calib". Dies ist praktisch für Testzwecke, kann aber auch notwendig sein, wenn zum Beispiel die Kalibrierungsdatei korrumpierte Daten enthält.

Es ist aber auch möglich, dass falsche Faktoren extrahiert werden: Wenn zum Beispiel während dem Test eine Kollision mit einem anderen Objekt auftritt, dann wird das Resultat verfälscht. Da die aktuelle Implementierung keine Kollisionserkennung hat, kann dies auch nicht automatisch erkannt werden.

# 8

# Diskussion und Ausblick

Dieses letzte Kapitel beginnt mit einer Zusammenfassung des Projektes CreatureBrain. Danach werden die erzielten Resultate diskutiert, Probleme erörtert sowie mögliche Verbesserungen und Erweiterungen angegeben. Das Ganze wird durch ein abschliessendes Fazit abgerundet.

## 8.1 Zusammenfassung

In dieser Diplomarbeit wurde das Modul CreatureBrain entwickelt, welches gemäss der Aufgabenstellung der Simulation von Verhalten und Kognition einfacher Kreaturen dienen soll. Dies geschah im Rahmen des neuen Forschungsprojektes CreatureZoo. In einem Rückblick wird in diesem Abschnitt die geleistete Arbeit zusammenfassend dargestellt.

### 8.1.1 Einarbeitung

Im ersten Teil dieser Diplomarbeit galt es, die Aufgabenstellung zu konkretisieren und unsere Vorstellungen zu umschreiben, um sie fassbar zu machen. Dies geschah einerseits durch Einarbeitung in das Gebiet der Künstlichen Intelligenz, wobei das Konzept des rationalen Agenten sowie Ziele und Pläne klar im Zentrum des Interesses standen. Andererseits beschäftigten wir uns mit verwandten Arbeiten, um einen Überblick über den aktuellen Forschungsstand zu erhalten. Durch die Analyse dieser Arbeiten und der Formulierung von eigenen Ideen und Absichten sowie aufgrund des vorgegebenen theoretischen Rahmens konnten die Anforderungen an das Verhalten erstellt und anhand von Szenarien illustriert werden. In unserem Modell wird das Verhalten durch drei Verhaltensmodi mit verschiedenen Prioritäten bestimmt. Damit war die Grundlage für die Entwicklung unseres Systems gegeben.

### 8.1.2 Modul CreatureBrain

Aufgrund der erarbeiteten Anforderungen wurde die Architektur des CreatureBrain spezifiziert. Das modulare System besteht aus den vier Komponenten InputSystem (Wahrnehmung), IntelligenceSystem (Wahl von Aktionen), ActionSystem (Ausführung von Aktionen) und Blackboard (Rückkopplung und Benutzereingabe), welche jeweils klar getrennte und in Kapitel 4 genauer beschriebene Aufgaben erfüllen. Dieses Modul ermöglicht den Kreaturen reflektorisches, zielgesteuertes und reaktives Verhalten - die drei in den Anforderungen verlangten Modi.

### 8.1.3 Integration

Die Zusammenführung mit den beiden anderen Diplomarbeiten im CreatureZoo Projekt, CreatureControl und CreatureWorld, konnte erfolgreich vorgenommen werden, wie Abb. 8.1 illustriert. Zuerst wurde das CreatureBrain erfolgreich in die Umgebung der lizenzierten Game Engine Torque integriert. Hier war für das CreatureBrain vor allem die Wahrnehmung wichtig, denn davon ist das Verhalten der Kreatur abhängig. Damit diese ihre Umgebung wahrnehmen kann, wurde Torque entsprechend angepasst und erweitert. Die Schnittstelle mit der CreatureControl wurde durch Anweisungen auf einer geeigneten Abstraktionsebene festgelegt, nämlich auf der Stufe von einfachen Bewegungskommandos wie "vorwärts", "rückwärts", "drehen links" und "drehen rechts". Somit konnte das CreatureBrain problemlos auf das Modul CreatureControl aufgesetzt werden.

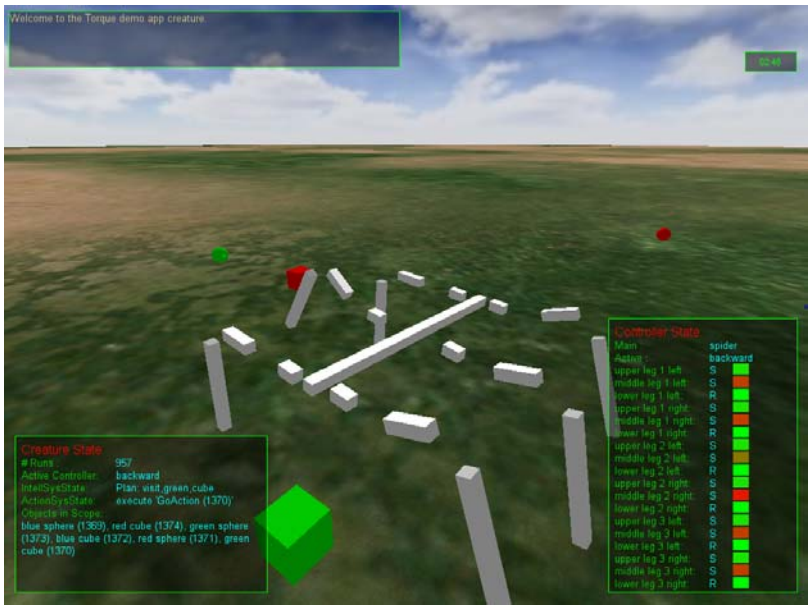



**Abbildung 8.1:** Screenshot der virtuellen Welt.

### 8.1.4 CreatureZoo

Durch das Zusammenspiel von CreatureWorld, CreatureControl und CreatureBrain war das Grundgerüst nun komplett und der Höhepunkt dieser Diplomarbeiten erreicht. Nun konnte die Kreatur ihr Verhalten unter Beweis stellen. Ihr reaktives Grundverhalten besteht darin, in einer Grundposition zu verharren. Der Benutzer kann die Kreatur durch Vorgabe eines Ziels dazu bringen, verschiedene Objekte in der Umgebung zu besuchen. Falls sie dabei einem Gefahrenobjekt zu nahe kommt, weicht sie diesem durch eine Reflexbewegung aus. Und falls sich ein Zielobjekt verschiebt, wird dieses anhand einer Gültigkeitsprüfung von Aktionen bemerkt und von der Kreatur neu angepeilt. Jegliche Aktion kann überdies jederzeit vom Benutzer durch einen Stop-Reflex unterbrochen werden. Der Benutzer hat auch die Möglichkeit, aktuelle Pläne abzubrechen und Ziele zu annullieren. Damit wird das Verhalten in den drei Modi demonstriert, sowie der Wechsel zwischen diesen. Das CreatureBrain konnte also erfolgreich umgesetzt werden und erfüllt die gestellten Anforderungen.

## 8.2 Diskussion

In diesem Abschnitt folgt nun eine kritische Diskussion der erreichten Resultate.

### 8.2.1 Verhaltenssimulation allgemein

Die Künstliche Intelligenz liefert uns mit dem Konzept vom idealen rationalen Agenten eine Definition für sinnvolles Verhalten. Doch ein solcher Agent ist vor allem theoretisch interessant, denn er kann nicht einfach auf eine Tabelle abgebildet werden. Andererseits ist es durchaus nicht trivial, beziehungsweise wohl unmöglich, einen Agenten in einer komplexen, dynamischen Umgebung mit Wissen, Regeln und anderen Simulationsvorschriften mit idealem rationalen Verhalten auszustatten. Darin liegt wohl eine Grundproblematik der Verhaltenssimulation.

Der Mensch benutzt bei der Verhaltenssimulation, wie bei vielen Bereichen der Künstlichen Intelligenz, oft sich selbst (oder Tiere) als Bewertungs- oder Vergleichskriterium für den Erfolg. Dies mag sehr naheliegend sein, ist aber reichlich hoch angesetzt.

Die Spezifikation von Verhalten ist keine einfach Aufgabe; es gibt keine universelle Notation. Aber für eine Modellierung müssen Verhalten und passende Bewertungskriterien auf geeigneter Abstraktionsstufe formuliert werden. Doch gerade menschlichem Verhalten scheint etwas sehr individuelles und nicht-deterministisches eigen zu sein. Vielleicht liegt hier der Grund, warum sich (menschliche) Intelligenz nicht in einer einfachen Definition fassen lässt.

### 8.2.2 CreatureBrain

Das implementierte System stellt im Rahmen des Projektes CreatureZoo den ersten Versuch eines Moduls zur Verhaltenssteuerung von autonomen Agenten dar. Das Projekt CreatureBrain verfolgte mehrere Ziele. Erstens zeigt es Probleme auf, welche beim Aufstellen der Anforderungen an ein System zur Verhaltenssteuerung auftreten können, da dies eine vielschichtige und wohl nicht universell lösbare Aufgabe ist. Zweitens legt es eine Basis für den CreatureZoo, einem System zur Simulation von autonomen, rationalen Kreaturen, indem Schnittstellen zu den beiden anderen Modulen CreatureWorld und CreatureControl definiert und erfolgreich eingesetzt wurden. Drittens schliesslich stellt es eine ausbaubare, erprobte und durchdachte Implementierung eines Systems zur Simulation von Verhalten einfacher Kreaturen dar.

Das gezeigte Verhalten der Kreatur mag nicht sehr vielseitig und ausgeklügelt erscheinen. Dies war aber auch nicht die Hauptaufgabe dieser Diplomarbeit. Diese bestand vielmehr darin, ein System zu erstellen, welches die Möglichkeit zu mannigfaltigem und komplexem Verhalten bietet. Die Idee war, die Architektur genügend offen zu halten, damit später viele Ideen und Ansätze integriert werden können.

Der Aufwand zur Implementierung von Logik zur Verhaltenssteuerung, welche vielseitiges und plausibles Verhalten ermöglicht, ist hoch. Das haben die Erfahrungen bei diesem Projekt klar gezeigt.

Nicht optimal scheint der Reflex-Mechanismus im vorliegenden Modell umgesetzt, da für reflektorisches Verhalten in jedem Simulationsschritt ein entsprechender Reflexreiz wahrgenommen werden muss. Dies erfordert teilweise eine künstliche Aufrechterhaltung von Reflexreizen durch Aktionen via das Blackboard System. Noch komplexere Situationen könnten bei einer Aufteilung der Subsysteme in einzelne Threads und somit durch einen asynchronen Zyklus entstehen.

## 8.3 Probleme und Unklarheiten

Während der Entwicklung des CreatureBrain traten einige Probleme und Unklarheiten auf, welche hier kurz dargestellt werden.

In der ersten Projekthälfte stellte die Unklarheit über das genaue Ziel dieser Arbeit ein Problem dar. Dies war wohl einerseits eine Auswirkung der sehr offenen Aufgabenstellung und andererseits sind auch viele Konzepte der Künstlichen Intelligenz allgemein gehalten und vor allem theoretisch interessant. Zudem lagen sehr viele Ideen und Vorstellungen in der Luft. All dies musste erst einmal festgehalten, aufgrund passender Kriterien und Entscheidungsgrundlagen geordnet und in konkrete Anforderungen umgewandelt werden.

Die Anbindung an Torque, unsere CreatureWorld, war sehr lange unklar und nicht spezifiziert. Die Aufgabenteilung war hier anfangs nicht klar geregelt. Zudem erforderte Torque als komplexe, umfangreiche Game Engine mit sehr vielen Möglichkeiten eine entsprechend lange Einarbeitungszeit.

Die Abgrenzung des CreatureBrain zur CreatureControl war eine weitere interessante Frage, die sich stellte. Die Aufgabenteilung liess mehrere Möglichkeiten offen, zum Beispiel könnten beide Module die Wegplanung übernehmen. Meist ging es darum, wie viel "Intelligenz" die CreatureControl erhalten darf, bevor sie selbst zu einem kleinen CreatureBrain wird. Schlussendlich wurde entschieden, dass die CreatureControl nur sehr grundlegende Bewegungen anbieten soll und alle Funktionalität, die mit Planen oder Lernen zu tun hat, im CreatureBrain untergebracht ist. In diesem Zusammenhang stellte sich die Frage nach den Motorikanweisungen. Es war lange offen, ob diese mit einer Zeitangabe, der Dauer, oder mit einer Distanz, respektive einem Winkel als Argument ausgestattet sein sollten. Ein letztes Problem sahen wir noch bei der genauen Spezifikation einer Greifbewegung, durch welche der Kreatur ein exaktes Greifen mit einem Arm ermöglicht werden sollte. Eine solche Bewegung wurde schliesslich nicht implementiert.

## 8.4 Mögliche Erweiterungen und Verbesserungen

Das CreatureBrain ist ein System, welches grundsätzlich für einen Ausbau vorgesehen ist und an diversen Stellen noch verbessert werden kann.

### 8.4.1 Ausbaubares System

Das CreatureBrain Modul wurde entwickelt, um die Simulation von verschiedenen Verhaltensweisen zu ermöglichen. Es ist deshalb klar als ausbaubares modulares System konzipiert. In der folgendenden Liste werden stichwortartig ein paar mögliche Ansatzpunkte aufgeführt:

- Implementierung von neuen und besseren Adaptoren für mehr Wahrnehmungsquellen (Temperatur, Helligkeit) oder neue Attribute (Gewicht, bewegbar)
- Interaktion mit der realen Welt, Steuerung durch gesprochene Kommandos oder Handzeichen
- Synthetischen Sehen zur Navigation
- Verfeinerung der Filter in den Sensoren, um Wahrnehmung genauer klassifizieren zu können
- Implementierung von neuen und feineren Wahrnehmungsklassen
- Modellierung von Unsicherheiten bei der Wahrnehmung
- Implementierung von neuen Aktionen mit komplexeren Vor-, Nach- und Gültigkeitsbedingungen
- Verfeinerter Reflex-Mechanismus, um differenzierter auf die Umgebung reagieren zu können

- Ausbau der KnowledgeBase, z.B. STRIPS-ähnliche Wissens-Repräsentation, evtl. Verwendung geeigneter Programmiersprachen (z.B. Prolog)
- Ausbau des Brain, Integration eines allgemeinen Planungssystems, Umsetzung von Planungsalgorithmen, Planvalidation
- Implementierung eines besseren Mechanismus zur Wahl von reaktivem Verhalten
- Möglichkeit des Mischens von Zielen. Konkurrenz und Priorisierung von Zielen
- Gruppenverhalten von Kreaturen im reaktiven Modus
- Controller kann gleichzeitig mehrere aktuelle Aktionen haben, ActionSystem kann Aktionen parallel ausführen
- Komplexeres Navigationssystem, Einbeziehung der Topographie, Kollisionsverhinderung
- Auftrennung der einzelnen Subsysteme in eigene Threads, Parallelisierung

### 8.4.2 Lernen auf verschiedenen Stufen

Lernverhalten ist, wie in Abschnitt 2.4 gesehen, ein weit gefasster Begriff. Lernen kann auf verschiedensten Stufen geschehen. Hier ist eine Liste von Ansatzpunkten, wo und wie dies im CreatureBrain möglich ist:

- Lernen als Erinnern. Es könnten bereits erstellte Pläne (oder Fragmente davon) verwendet werden, um schneller und effizienter zu planen.
- Lernen als Erweiterung. Pläne könnten als neue, komplexe Aktionen erlernt und so die grundlegenden Fähigkeiten erweitert werden.
- Lernen als Verstehen. Die Kreatur könnte ihre Umgebung auskundschaften und versuchen, kausale Zusammenhänge zu erkennen.
- Lernen als Verstehen. Das CreatureBrain könnte neue Bewegungen und ihre Effekte lernen. Unbekannte Bewegungen könnten von der CreatureControl durch eine klar definierte Schnittstelle abgefragt werden.

### 8.4.3 Kognitive Modellierung

In [6] wird mit CML, der Cognitive Modelling Language, eine Sprache und ein Modell vorgeschlagen, mit welcher sich das Verhalten von autonomen Kreaturen auf einer höheren Stufe als Verhaltensmodellierung kontrollieren lässt. Ein Konzept dieser Art könnte dem CreatureBrain aufgesetzt werden, um die Kreatur durch kognitive Modellierung steuern zu können.

## 8.5 Fazit

Abschliessend kann ein positives Fazit gezogen werden. Die Ziele der Diplomarbeit wurden erreicht, ein funktionierendes System entwickelt, implementiert und erfolgreich angewandt. Verhaltenssimulation wurde dabei als ein vielschichtiges und komplexes Problem erkannt, welches sich nicht universell und endgültig lösen lässt. Nichtsdestotrotz steht für den CreatureZoo nun ein Grundgerüst bereit, welches von zukünftigen Kreatur-Zoologen hoffentlich erfolgreich benutzt, verstärkt und ausgebaut wird!

# 9 Referenzen


[1] B. Blumberg und T. Galyean. “Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments.” *Computer Graphics*, 29(Annual Conference Series):47–54, 1995.

[2] B. Kernighan und D. Ritchie. *The C Programming Language, Second Edition, ANSI C*. Prentice-Hall International, 1988.

[3] R. Burke, D. Isla, M. Downie, Y. Ivanov, und B. Blumberg. “Creature Smarts: The Art and Architecture of a Virtual Brain.” In *Game Developers Conference Proceedings 2001*, pages 147–166, 2001.

[4] T. Dean und M. Boddy. “An Analysis of Time-Dependent Planning.” In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54, Saint Paul, Minnesota, USA, 1988. AAAI Press/MIT Press.

[5] R. Fikes und N. Nilsson. “Strips: A new approach to the application for theorem proving to problem solving.” In *Advance Papers of the Second International Joint Conference on Artificial Intelligence*, pages 608–620, Edinburgh, Scotland, 1971.

[6] J. Funge, X. Tu, und D. Terzopoulos. “Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters.” In A. Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 29–38, Los Angeles, 1999. Addison Wesley Longman.

[7] GarageGames.com. “Torque Game Engine.” http://www.garagegames.com, 2001.

[8] B. Heidelberger. “CreatureControl - Erzeugung und Steuerung physikalisch basierter Kreaturen.” Master’s thesis, Departement Informatik, ETH Zürich, 2002.

[9] D. Isla, R. Burke, M. Downie, und B. Blumberg. “A Layered Brain Architecture for Synthetic Characters.” In *International Joint Conference on Artificial Intelligence (IJCAI), Seattle, WA*, pages 1051–1058, 2001.

[10] M. Gross. *Graphische Datenverarbeitung*. Lecture Notes, Computer Science Departement, ETH Zürich, 2000.

[11] S. Mecklenbräuker. *Sprache und Denken.* Vorlesungsunterlagen, Fachschaft Psychologie, Universität Trier, 2000.

[12] A. Moravanszky. “A Graphical Environment for Virtual Creatures.” Master’s thesis, Computer Science Departement, ETH Zürich, 2002.

[13] P. Plauger, A. Stepanov, M. Lee, und D. Musser. *The C++ Standard Template Library*. Prentice Hall, 2001.

[14] S. Russel und P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice-Hall, Englewood Cliffs, 1995.

[15] K. Sims. “Evolving Virtual Creatures.” In *Computer Graphics (SIGGRAPH ’94 Proceedings)*, volume 28, pages 15–22, 1994.

[16] R. Smith. “ODE - Open Dynamics Engine.” http://www.q12.org/ode, 2001.

[17] S. Lippman and J. Lajoie. *C++ Primer, Third Edition*. Addison Wesley, 1998.

[18] B. Stroustrup. *The C++ Programming Language*. Addison Wesley Publishing Company, Special Edition, 2000.

[19] X. Tu und D. Terzopoulos. “Artificial Fishes: Physics, Locomotion, Perception, Behavior.” In *Computer Graphics (SIGGRAPH ’94 Proceedings)*, volume 28, pages 43–50, 1994.

[20] D. van Heesch. “Doxygen.” http://www.doxygen.org, 1997.

# A

# Konsolenkommando BrainCommand

## A.1 Konsolenkommandos

Die Konsole wird in Torque durch die Taste ‘~’ ein- und ausgeschaltet. Zur Illustration der Syntax von Kommandos sei hier ein Beispiel des BrainCommand “visit” gegeben:

```
BrainCommand(“visit,green,cube”);
```

## A.2 Verfügbare BrainCommands

| Kommando | Wirkung |
|---|---|
| **visit**[,*Farbe*][,*Form*] | Zielanweisung an die Kreatur, alle Objekte (CreatureItems) in der Welt zu besuchen. Durch Attribute Farbe und Form kann die Menge der zu besuchenden Objekte eingeschränkt werden. |
| **cancelPlan** | Bricht den aktuellen Plan ab und löscht ihn. |
| **cancelGoals** | Löscht allfällige Ziele. Der aktuelle Plan wird aber noch zu Ende ausgeführt, falls einer vorhanden ist. |
| **noPlan** | Löscht den aktuellen Plan und alle vorhandenen Ziele. Entspricht der Kombination von *cancelPlan* und *cancelGoals*. |
| **stop** | Hält die Kreatur an. |
| **go** | Lässt die Kreatur nach einem *stop* wieder handeln. |
| **calib** | Eine neue Kalibrierung der Bewegungsfaktoren wird gestartet. |

# B

# Kalibrierungsdatei

## B.1 Name und Pfad

Die Kalibrierungsdatei wird unter dem Namen des aktuellen Haupt-Controller mit Suffix ".instrDat" abgelegt, und zwar im Verzeichnis *~/zoo/data*.

## B.2 Format und typische Werte

Das Dateiformat ist einfach gehalten: Die vier gespeicherten Werte für die Faktoren zu *forward-, backward, rotate left* und *rotate right* sind als Fliesskommazahlen in aufeinanderfolgende Zeilen geschrieben.

Diese Werte sahen in unserer Testumgebung typischerweise so aus:

```
0.4086
0.408844
0.0709144
0.070853
```