Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Carsten Canow

Simulation menschlicher Spieler in einem Massively Multiplayer Online Game durch Methoden der Künstlichen Intelligenz

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Carsten Canow

Simulation menschlicher Spieler in einem Massively Multiplayer Online Game durch Methoden der Künstlichen Intelligenz

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Thomas Thiel-Clemen

Abgegeben am 01. Februar 2010

Carsten Canow

Thema der Bachelorarbeit
Simulation menschlicher Spieler in einem Massively Multiplayer Online Game durch Methoden der Künstlichen Intelligenz
Stichworte
Jadex, BDI-Agenten, Potentialfelder, Spiele, Steuerung
Kurzzusammenfassung
Diese Arbeit beschreibt die Entwicklung eines Prototyps zur Steuerung von Spielern in einer Mehrspieler-Umgebung. Dazu werden zunächst einige Methoden der Künstlichen Intelligenz analysiert und auf ihre Anwendbarkeit hin geprüft.
Der Prototyp ist ein Schritt auf dem Weg hin zu einer generischen KI für den Einsatz in einer großen Spielwelt mit sehr vielen Spielern. Ziel ist es, den menschlichen Spielern sowohl realistische Gegner als auch Mitspieler zu bieten. Nicht zuletzt sollen die zahlreichen Charaktere der Welt, wie z.B. Tiere im Wald, die für den menschlichen Spieler von geringem Interesse sind, gesteuert werden.
Zu diesem Zweck wird eine Kombination aus BDI-Agenten mit Hilfe des Jadex-Frameworks und dem noch recht unerforschten Konzept der Potentialfelder implementiert und beschrieben.

Carsten Canow

Title of the paper
Simulating human-like players in a massively multiplayer online game by methods of artificial intelligence
Keywords
Jadex, BDI-Agents, Potential Fields, Games, Controlling
Abstract
This thesis describes the development of a prototype for the control of players in a multiplayer environment. Therefor some methods of artificial intelligence are analyzed and tested for their applicability.
The prototype is one step towards a generic AI for use in a large game world with a lot of players. The aim is to provide realistic opponents and teammates for the human players. Moreover, the numerous characters of the world, such as animals in the forest, which are of little interest for human players, are to be controlled.
For this purpose a combination of BDI agents using the Jadex framework, and the still relatively unexplored concept of potential fields is implemented and described.

Inhaltsverzeichnis

Kapitel 1

Einführung

Diese Arbeit befasst sich mit der Entwicklung eines Frameworks für die Steuerung von Spielern in einem Massively Multiplayer Online Game (MMOG). Neben dem Ansatz der Agentensteuerung wird versucht, ein bisher noch selten genutztes Konzept zu implementieren. Die Rede ist von so genannten Potentialfeldern (Potential Fields).

Dieses Konzept wurde bisher oft vernachlässigt, weil die Verwaltung als zu kostenintensiv angesehen wird. Daher soll hier gezeigt werden, dass eine performante Umsetzung möglich ist, die darüber hinaus gegenüber den weit verbreiteten Verfahren interessante Vorteile bietet.

1.1 Motivation

Den Rahmen für diese Arbeit liefert ein Projektmodul an der Hochschule für Angewandte Wissenschaften Hamburg. Weitere Informationen dazu finden sich im Anhang A.

Für diese Arbeit wurde das in der Agentensteuerung sehr bekannte „Räuber-Beute" (Hunter-Prey)-Szenario gewählt, in dem zwei oder mehr rivalisierende Lebensformen in einer geschlossenen Welt interagieren.

1.1.1 Wator

Ein sehr bekanntes Beispiel für das Szenario mit dem Namen „Wator"[1] spielt in einer Welt, die vollständig mit Wasser bedeckt ist. In ihr leben Fische und Haie. Die Wasseroberfläche ist mit Plankton überzogen. Die Fische ernähren sich vom Plankton und vermehren sich dadurch. Je mehr Fische es gibt, desto mehr Haie finden ausreichend Nahrung, um sich fortzupflanzen. Sterben die Haie, so wird aus ihnen wieder Plankton.

[1] von Alexander K. Dewdney und David Wiseman

Im Idealfall hält sich die Population die Waage. Bevor die Haie alle Fische auffressen können, sind zu wenige Fische übrig, so dass ein Teil der Hai-Population sterben muss und zu Plankton wird. Das neue Plankton nährt Fische, was die Fischpopulation wieder ansteigen lässt. Gibt es viele Fische und kaum noch Plankton, fressen die verbleibenden Haie die Fische und der Zyklus beginnt von vorn. Es kann jedoch auch passieren, dass die Haie alle Fische auffressen, was ein Verhungern der Haie nach sich ziehen würde. Ein weiterer möglicher Ausgang wäre, dass die Haie aussterben, was zur Folge hätte, dass die gesamte Wasserwelt mit Fischen überschwemmt würde, bis kein Plankton mehr existiert. In diesem Fall würden alle Lebewesen der Wasserwelt aussterben.

1.1.2 HuntersAndDeers

Analog zur Vorlage von „Wator" entstand das Szenario für diese Arbeit. Es handelt sich dabei um einen Wald in dem Rehe und Jäger leben. Die Simulation trägt den Namen „Hunters and Deers" (HaD).

Aufgabe der Jäger ist es, Rehe zu erlegen, während diese wiederum Gras fressen müssen, um zu überleben. Das im Wator-Beispiel beschriebene Verhalten der Welt ist nicht Gegenstand dieser Arbeit.[2]

1.2 Aufbau der Arbeit

In Kapitel 2 wird zunächst das System beschrieben, in dem diese Arbeit durchgeführt wird. Dazu gehören ein Überblick über die Architektur, sowie Besonderheiten und Einschränkungen, die diese Arbeit betreffen.

In den Kapiteln 3 und 4 wird auf das Konzept der Agenten eingegangen und ein Agentensystem ausgewählt. Während in Kapitel 3 die Grundlagen beschrieben werden, geht es in Kapitel 4 um die konkrete Umsetzung.

Kapitel 5 stellt das zweite wichtige Konzept vor, die Potentialfelder. Nach einer kurzen Einführung mit Beispielen werden potenzielle Problemfälle und die Ansätze zu deren Lösung besprochen. Abschließend werden Optimierungsmöglichkeiten aufgezeigt.

Das Kapitel 6 zeigt dann eine erste Implementation der Potentialfelder in einer in der Größe beschränkten Welt. Bei dieser Implementation wird eine statische Lösung angewandt, die für den Einsatz in einer dynamischen MMO-Welt ungeeignet ist. Aus diesem Grunde wird in Kapitel 7 die Erweiterung der Agentenbasis aus Kapitel 4 beschrieben.

Anschließend fasst Kapitel 8 noch einmal alle Erkenntnisse zusammen und gibt einen Ausblick über die in dieser Arbeit beschriebenen Konzepte hinaus.

Die Anhänge A und B enthalten Informationen über die in dieser Arbeit verwendeten Spiele-Frameworks.

[2] Sollte sich jedoch im Idealfall ebenfalls zeigen.

Kapitel 2

Systembeschreibung

In diesem Kapitel werden die Architektur und die besonderen Eigenschaften des Systems beschrieben.

Zunächst wird die Architektur erläutert, um einen Überblick zu schaffen. Danach wird beschrieben, wie die Welt aufgebaut ist und wie sie verwaltet wird. Abschließend werden besondere Merkmale der Umwelt und der (Spieler-)Objekte erklärt.

2.1 Systemarchitektur

Der Aufbau des Systems folgt einer Client-Server-Architektur. Sowohl der Server als auch alle bisherigen Clients sind in der Programmiersprache Java entwickelt worden. Da der Sourcecode öffentlich ist, wird es dem Spieler theoretisch (und auch gewollt) ermöglicht, einen eigenen Client zu schreiben. Dies wiederum kann nur unter einem souveränen Server funktionieren, der sämtliche Aktionen überwacht. In Abbildung 2.1 ist der schematische Aufbau zu sehen, welcher im Folgenden erläutert wird.

2.1.1 Gameserver

Wie bereits erwähnt, übernimmt der Server die Rolle eines souveränen Beobachters. Als solcher ist er für den konsistenten Zustand der Welt verantwortlich. Er überprüft und korrigiert gegebenenfalls die übermittelten Aktionen der Clients, um Fehler aber auch Betrugsversuche zu beheben bzw. zu verhindern.

Zum Zeitpunkt der Erstellung dieser Arbeit liegt lediglich ein Prototyp des Gameservers vor. Daher ist dieser nicht vollkommen souverän und gewisse Eigenschaften müssen in der Kommunikationsschnittstelle des Clients abgebildet werden. Dazu gehört unter anderem die Sichtbarkeit von Objekten. In der Beschreibung der Implementierung der Agenten in Kapitel 4 wird auf diesen und weitere Umstände näher eingegangen.

In einem späteren Stadium wird es mehr als einen Server in einem Clusterverbund geben. Da zum jetzigen Zeitpunkt jedoch nur ein Prototyp vorliegt, wird von „dem Gameserver" im Singular gesprochen. Die Kommunikationsplattform der Anwendung setzt auf dem „Projekt Darkstar"-Framework [DAR] auf.[3] Dieses Framework ermöglicht die Skalierbarkeit des Servers. Außerdem regelt es den Transport von Nachrichten zwischen den Clients und dem Gameserver.[4]

2.1.2 Game-Clients

Mit ein Grund für die Offenlegung der Quellen ist die Hoffnung, dass sich eine große Entwicklergemeinde bildet, welche verschiedenste Client-Arten entwickelt. Das „Project Darkstar"-Framework bietet zu diesem Zweck nicht nur Schnittstellen für die Programmiersprache Java. Die Implementation in Java ermöglicht es jedoch (zumindest theoretisch), einen von der Plattform unabhängigen Client zu entwickeln, so dass auch Apple- und Linux-User an dem Projekt teilnehmen können.

Neben einem 3D-Client und einem 2D Observer[5] ist ebenfalls eine Schnittstelle für Browsergame-Clients angedacht, so dass der Spieler sich auch von unterwegs oder an fremden PCs ins Spiel einloggen kann.

Auch die in dieser Arbeit entwickelte Anwendung ist ein Game-Client. Erläuterungen zur Funktionsweise der Kommunikation finden sich in Kapitel 4.1.

2.1.3 Game-Verwaltung

Der Gameserver wird mittels einer WebApplikation konfiguriert und verwaltet. Die Game-Verwaltung ist für diese Arbeit nicht relevant und wird nur der Vollständigkeit halber erwähnt.

[3] Project Darkstar ist die Open Source Veröffentlichung des SGS (Sun Game Server)

[4] Weiterführende Informationen zum Prototyp des Servers, siehe Anhang A

[5] Der Observer ist ein Werkzeug zum Überwachen der Welt. Er erlaubt es, Teile der Welt zu beobachten, ohne aktiv am Spiel teilzunehmen.

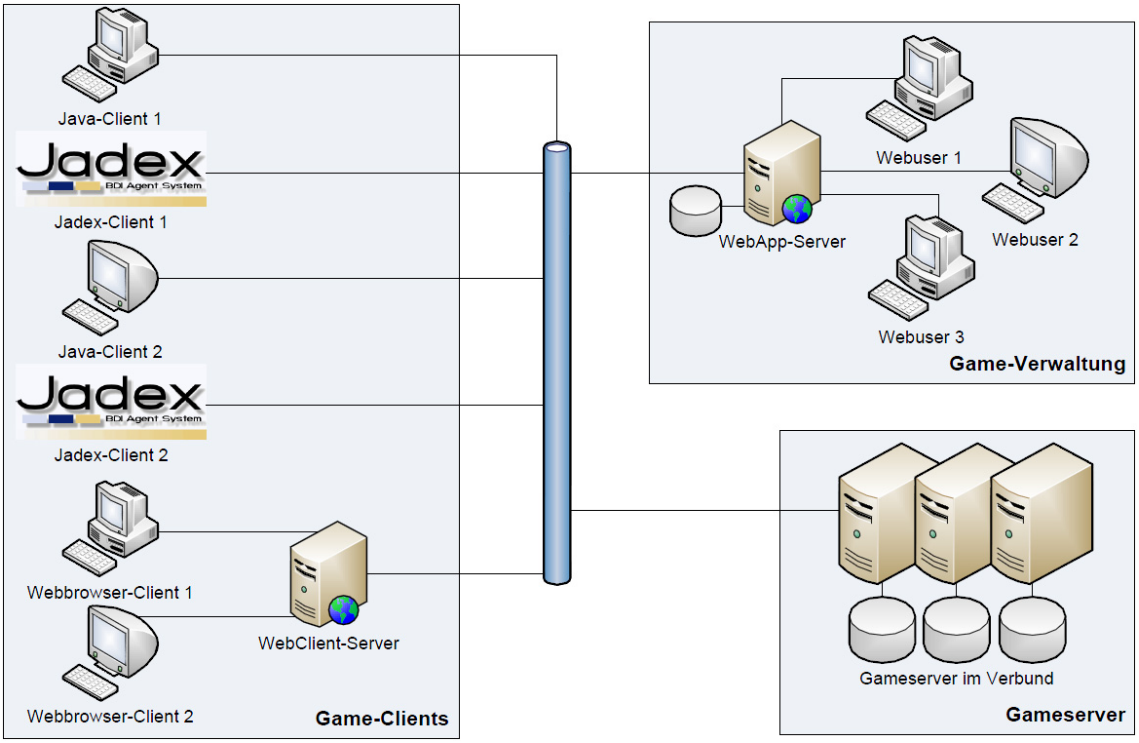

Abbildung 2.1: Architektur-Darstellung - Timadorus

2.2 Die Welt

Die Spielwelt hat den Anspruch, die reale Welt mit möglichst wenigen Einschränkungen widerzuspiegeln. Dazu gehört unter anderem auch, dass der Spieler (zumindest theoretisch) mit allem und jedem in der Welt interagieren können soll, während die Welt möglichst realistisch darauf reagiert.

Die Idee dazu stammt aus einem Projektmodul der HAW Hamburg [TIMa] mit dem Namen „Der Weg nach Timadorus" [TIMb]. Hauptziel des Projekts ist der Lernerfolg, der sich bei der Planung und Umsetzung ergibt. Dennoch wird versucht ein lauffähiges Spiel zu erstellen.

2.3 Eigenschaften

Wie bereits angedeutet, besitzt alles in der Welt ein gewisses Eigenleben. Damit der Spieler z.B. mit einem Baum interagieren kann, muss dieser die entsprechenden Reaktionen wiedergeben können. Einige dieser Eigenschaften werden im Folgenden näher beschrieben.

2.3.1 Allgemein

Grundsätzlich soll der Gameserver keine Logiksteuerung übernehmen. Dazu gehört insbesondere das Steuern so genannter „Non Player Charakters" (NPCs).[6] Dennoch muss der Server gewisse Regeln überwachen. Dazu gehört die Verwaltung folgender Attribute:

- Lebensenergie:
  - sinkt die Lebensenergie auf 0, „stirbt" das Objekt
- Ausdauer
  - bewegt sich ein Objekt durch die Welt, verbraucht es Energie
  - der Gameserver muss die Geschwindigkeit des Objekts entsprechend anpassen und verwalten
  - steht ein Objekt still, regeneriert es seine Ausdauer automatisch
- Hunger
  - jedes „lebende" Objekt in der Welt braucht Nahrung
  - sinkt der Wert für die Sättigung auf 0, „leidet" das Objekt, was wiederum Auswirkungen auf die Lebensenergie hat

2.3.2 Spieler

Der Spieler[7] hat zusätzliche Attribute, die der Gameserver überwachen muss. Für die hier beschriebene Anwendung sind davon jedoch nur folgende relevant:

- Sichtbereich
  - Ausschnitt der Welt, die der Spieler wahrnimmt
- Angriffs-Reichweite
  - die maximale Entfernung, in der ein Angriff des Spielers erfolgreich sein kann

[6] Die Steuerung nicht-menschlicher Spieler ist der Grund für diese Arbeit.
[7] An dieser Stelle sind ebenfalls die NPCs gemeint.

2.3.3 Umwelt

Neben den Spielern gibt es auch nicht spielbare, aber dennoch „lebendige" Objekte in der Welt. Dazu gehören im Prototyp das Gras und die Bäume.

Der Gameserver verwaltet für diese Objekte ebenfalls Eigenschaften wie z.B. Lebensenergie beim Gras. Wenn nun ein Reh zu fressen beginnt, sinkt die Lebensenergie des Gras-Objektes. Wird es jedoch in Ruhe gelassen, also nicht gefressen, steigt die Lebensenergie bis auf ein Maximum an. Dieser Wert stellt die Menge des verfügbaren Grases dar und kann grafisch durch Höhe und Dichte des Grases repräsentiert werden.

Ursprünglich war geplant, dass Objekte mit maximaler Lebensenergie sich „fortpflanzen" können. Realistischer ist es jedoch, wenn der Server regelmäßig (per Zufall) irgendwo in der Welt Objekte dieser Klassen (Gras, Bäume, etc.) erzeugt. Natürlich nur sofern der zufällige Ort geeignet ist. Des Weiteren kann über das Attribut „Ausdauer" die Wachstumsgeschwindigkeit reguliert werden. So könnte z.B. in einer trockenen Gegend weniger Gras wachsen als in einer feuchten Umgebung.

2.4 Einschränkungen

Um die Komplexität überschaubar zu halten und da der Prototyp des Gameservers zu Beginn dieser Arbeit die meisten Funktionen noch gar nicht unterstützt hat, werden nur einige der beschriebenen Eigenschaften umgesetzt:

- Lebensenergie
  - wichtig für das Wachstum von Gras
  - wichtig, damit ein Jäger ein Reh erlegen kann
- Ausdauer
  - unterschiedliche Geschwindigkeiten der Spieler
- Sichtbereich
  - der Einfachheit halber handelt es sich um einen kreisrunden Sichtbereich, so dass auch von Sichtweite gesprochen werden kann[8]
- Angriffs-Reichweite
  - wichtig, damit ein Jäger ein Reh angreifen kann

[8] Ein realistischerer Sichtbereich wäre insbesondere für ein „menschlicheres" Verhalten wünschenswert, wird jedoch vom Prototyp des Gameservers nicht unterstützt. Anregungen zum Entwurf eines besseren Sichtbereichs finden sich u.a. hier [RD08]

Kapitel 3

Agentensystem

Dieses Kapitel bietet eine kurze Einführung in die Agentensysteme. Dazu werden zunächst die Grundlagen erläutert. Im Anschluss werden zwei Frameworks vorgestellt, von denen schließlich eines für die Implementation im nächsten Kapitel ausgewählt wird.

3.1 Grundlagen

Eine selbstständig handelnde Anwendung, die die Umgebung, in der sie sich befindet, wahrnimmt und sich an mögliche Änderungen anpasst, wird als Agent bezeichnet. (vgl. [RN04] S. 21)

3.1.1 Wahrnehmung

Damit ein Agent dies leisten kann, benötigt er Zugang zu seiner Umgebung. Dieser Zugang wird über die Sensoren und Aktuatoren ermöglicht.

Sensoren können z.B. Kameras oder Temperaturmesser sein. Sie sind vergleichbar mit den Augen und Ohren menschlicher Agenten. Während Sensoren die Eingabemittel darstellen, sind Aktuatoren die Mittel zur Ausgabe und zum Handeln. Typische Beispiele für Aktuatoren bei menschlichen Agenten sind Hände, Füße und Mund. Über seine Sensoren kann ein Agent bestimmte Fakten wahrnehmen, jedoch nicht deren Wirkung.

Ein häufig betrachtetes Szenario für einen einfachen Agenten ist die Staubsauger-Welt. Dabei handelt es sich um eine meist sehr einfache Welt, in diesem Fall um zwei benachbarte Felder. Der Agent steht entweder auf dem einen oder dem anderen Feld. Über seine Sensoren (hier: Kamera) kann er das aktuelle Feld scannen und feststellen ob es sauber ist oder gereinigt werden muss. Seine möglichen Aktionen sind „Rechts", zur Bewegung auf das rechte Feld und „Links" zur Bewegung auf das linke Feld. Diese Aktionen können nur aus den entsprechenden Positionen aufgerufen werden. Der Staubsauger-Agent besitzt noch eine dritte Aktion, „Saugen", mit deren Hilfe er das aktuelle Feld reinigen kann. Er betrachtet

also sein aktuelles Feld, reinigt es bei Bedarf, wechselt auf das andere Feld und wiederholt dieselben Schritte immer wieder. (vgl.[RN04] S. 54)

3.1.2 Intelligente Agenten

Unter einem intelligenten Agenten wird im Allgemeinen ein rationaler Agent verstanden. Dies „ist ein Agent, der sich so verhält, dass er das beste Ergebnis erzielt". ([RN04] S. 21)

In [Woo02] (S. 23) werden Agenten dann als rational betrachtet, wenn sie reaktiv, proaktiv und sozialfähig sind.

- Reaktiv bedeutet, dass ein Agent auf Veränderungen in seiner Umwelt zeitnah reagiert.
- Proaktiv ist ein Agent, wenn er ein Ziel über längere Zeit bzw. mehrere Einzelschritte hinweg zu erreichen versucht.
- Unter sozialfähig wird die Kommunikation mit anderen Agenten verstanden. Insbesondere wenn Agenten sich austauschen, um gemeinsam ein Ziel zu erreichen.

In [LP04] (S. 3) wird diese Liste noch um vier weitere Begriffe erweitert: situiert, autonom, flexibel und robust.

- Situiert bedeutet, dass der Agent einen Bezug zu seiner Umgebung herstellt.
- Autonom bedeutet, dass der Agent unabhängig ist und sich selbst kontrolliert.
- Der Agent sollte flexibel in Bezug auf den Weg zur Erreichung eines Ziels sein. Schlägt ein Versuch fehl, sollte ein anderer Weg gefunden und genutzt werden.
- Ein robuster Agent ist in der Lage Fehler zu erkennen und angemessen auf sie zu reagieren.

3.1.3 Eigenschaften einer Umgebung

Prinzipiell kann ein Agent in jeder Umgebung eingesetzt werden. Daher ist es notwendig, die möglichen Umgebungen zu klassifizieren. Eine sehr umfangreiche und oft zitierte Beschreibung findet sich in [RN04] auf den Seiten 66 und 67:

- Vollständig und teilweise beobachtbar
  In einer vollständig beobachtbaren Umgebung hat der Agent über seine Sensoren jederzeit Zugriff auf alle Informationen, die er zum Treffen einer Entscheidung benötigt. Ist der Sichtbereich des Agenten hingegen eingeschränkt oder gestört, ist die Rede von einer nur teilweise beobachtbaren Umgebung.

- Deterministisch und stochastisch
  In einer deterministischen Umgebung ist der nächste Zustand vollständig vom gegenwärtigen Zustand und den vom Agenten ausgeführten Aktionen festgelegt. Ist das nicht der Fall, ist die Umgebung stochastisch. Ein Sonderfall liegt dann vor, wenn die Umgebung von den Aktionen anderer Agenten abgesehen, deterministisch ist. Eine solche Umgebung wird als strategisch bezeichnet.

- Episodisch und sequenziell
  Wenn ein Agent in sich geschlossene Aktionsfolgen durchläuft, die danach abgeschlossen sind und die Zukunft nicht beeinflussen, ist seine Umgebung episodisch. Ein Druck-Agent würde jeden Druckauftrag unabhängig von der Zukunft und der Vergangenheit verarbeiten. Ein solcher Agent ist in der Regel frei von der Planung seiner Arbeitsschritte. Wird das Handeln des Agenten jedoch von Aktionen in der Vergangenheit beeinflusst, bzw. beeinflusst seinerseits die Zukunft, handelt es sich um eine sequenzielle Umgebung.

- Statisch und dynamisch
  Wenn die Umgebung des Agenten sich verändern kann, während er seine Aktionen ausführen möchte, ist seine Welt dynamisch. Liefert jede Abfrage des Zustandes der Welt den gleichen Wert, solange der Agent keine Aktion ausgeführt hat, ist sie statisch. In einer statischen Umgebung braucht der Agent nur ein einziges Mal den Zustand abzufragen und kann die neuen Zustände über seine ausgeführten Aktionen verfolgen.

- Diskret und stetig
  Eine Umgebung ist diskret, wenn sie eine abzählbare Menge unterschiedlicher Zustände aufweisen kann. Andernfalls ist sie stetig.

- Einzelagent und Multiagent
  Der Unterschied zwischen Einzel- und Multiagenten-Umgebungen scheint einfach zu sein. Befinden sich in einer Welt mehrere Agenten, ist es eine Multiagenten-Umgebung. Schwierig wird es, wenn nicht klar ist, ob ein anderes Objekt der Welt ein Agent ist oder nicht. Daher muss jeweils betrachtet werden, ob das Verhalten eines Objektes von dem Verhalten eines anderen Objektes abhängt.

„Wie zu erwarten, ist der schwierigste Fall teilweise beobachtbar, sequenziell, dynamisch, stetig und Multiagent." ([RN04] S. 68) Diese Klassifizierung trifft auf die Umgebung, in der sich die Agenten dieser Arbeit befinden, genau zu.

3.1.4 Agentenarten

Zur weiteren Klassifizierung der Agenten werden in diesem Abschnitt die verschiedenen Agentenarten betrachtet. Die folgende Liste ist angelehnt an [RN04] (S. 70ff).

- Einfache Reflex-Agenten
  Ein solcher Agent wählt die jeweils als nächstes auszuführende Aktion in Abhängigkeit von seinem aktuellen Zustand aus. Er reagiert unmittelbar auf die Veränderung seiner Umgebung. Der Agent im Abschnitt 3.1.1 ist somit ein Reflex-Agent. Die einfachste Implementation erfolgt über so genannte WENN-DANN-Regeln[9].

- Modellbasierte Reflex-Agenten
  Diese Art Agenten besitzen den gleichen Aufbau wie die einfachen Reflex-Agenten. Allerdings können sie den Umstand einer nur teilweise beobachtbaren Umgebung ausgleichen. Zu diesem Zweck werden die vom Agenten über die Zeit gesammelten Informationen zu seiner Umgebung intern verwaltet. Diese Informationen werden dann zu einem virtuellen Modell[10] der Welt zusammengeführt.

- Zielbasierte Agenten
  Für manche Aufgabenstellungen ist der aktuelle Zustand (ob tatsächlich oder modellbasiert) nicht ausreichend. Die zielbasierten Agenten treffen ihre Entscheidung für die nächste Aktion, in dem sie die zur Verfügung stehenden Aktionen simulieren. Der jeweilige Ausgang einer Simulation wird mit dem Zielzustand verglichen. Der Agent sucht so lange die Menge seiner Aktionen ab, bis er diejenige gefunden hat, die ihn näher an sein Ziel bringt. Diese Art von Agenten benötigt also Verfahren zum Suchen und Planen seiner nächsten Schritte.

- Nutzenbasierte Agenten
  Auch die zielbasierten Agenten sind nicht für alle Umgebungen ausreichend. Daher besitzen nutzenbasierte Agenten eine Nutzenfunktion. Diese vergleicht die verschiedenen Lösungswege z.B. bezüglich Aufwand, Kosten oder Gefahren und entscheidet sich dann für die für den Agenten beste Aktionsfolge. Nutzenbasierte Agenten werden so implementiert, dass sie versuchen ihren Nutzen zu maximieren.

3.1.5 BDI Agenten

Zum Abschluss der im letzten Unterabschnitt genannten Agentenarten, soll hier noch eine konkrete Architektur genannt werden. Die BDI-Architektur beschreibt ein Vorgehen zur Implementation von Agenten. BDI steht für „Belief Desire Intention". Ein „belief" ist eine Information, die der Agent über seine Umwelt besitzt. Ein „desire" entspricht einem Ziel des Agenten. Ohne ein Ziel würde der Agent keine Aktionen ausführen. Durch die Definition eines „desires" ist der Agent außerdem in der Lage, einen alternativen Lösungsweg zu suchen, falls eine Aktion fehlschlägt. „intentions" sind die Absichten oder auch Pläne des Agenten.

[9] Beispiel: WENN zustand = müde DANN schlafen
[10] Daher auch der Name „modellbasiert".

Der Agent hat eine Sammlung von Plänen, die zu unterschiedlichen Zielen führen. Er sucht nach allen Plänen, die ihn zu seinem aktuellen Ziel führen würden und entscheidet über Priorität oder per Zufall, welchen er als nächstes durchführt.

3.2 Auswahl eines Agentensystems

Um das Konzept der Agenten sinnvoll einsetzen zu können, bietet es sich an, ein bestehendes Framework zu verwenden, statt das ganze System selbst zu implementieren. Aus diesem Grund werden nun zwei etablierte Systeme vorgestellt. Im Anschluss wird eines von beiden begründet ausgewählt.

3.2.1 Jason

Anand Rao [Rao96] entwickelte 1996 eine Logik-basierende Sprache für BDI-Agenten mit dem Namen AgentSpeak(L).

Jason ist eine Entwicklungsumgebung, die auf einer Erweiterung dieser Sprache aufsetzt und sie interpretiert. Der Sourcecode ist Open Source und unter der GNU LPGL[11] veröffentlicht worden und bringt viele anwenderspezifische Erweiterungen mit sich. Weiterführende Informationen sowie Beispiele befinden sich auf der Webseite zu Jason [HB] oder in dem 2007 erschienenen Buch [BHW05], an dem Michael Wooldridge [Woo02] mitgewirkt hat.

3.2.2 Jadex

Jadex ist ebenfalls ein Framework zum Bau von zielorientierten BDI-Agenten. Es wurde am Department Informatik der Universität Hamburg von den Studenten Lars Braubach und Alexander Pokahr entwickelt. Das Framework kann auf verschiedenen Middleware-Plattformen wie z.B. JADE [Jad] aufsetzen.

Die Entwicklung erfolgt in der Programmiersprache Java, die Konfiguration mit Hilfe von XML. Der Sourcecode ist ebenfalls Open Source und unter der GNU LPGL veröffentlicht worden. Weitere Informationen finden sich auf der Webseite zu Jadex [BP].

3.2.3 Auswahl und Begründung

Nachdem für die Umsetzung der Arbeit die beiden oben genannten Frameworks zur Verfügung standen, fiel die Entscheidung aus verschiedenen Gründen auf das Jadex-Framework:

1. Es bietet durch die Trennung von Implementation und Konfiguration eine sehr gute Grundlage für eine generische Anwendung.

[11] GNU Lesser General Public License

2. Es wird in der gleichen Programmiersprache entwickelt, die auch für die Entwicklung dieser Arbeit gewählt wurde.

3. Das Ergebnis dieser Arbeit soll von anderen Teilnehmern am Projekt „Timadorus" später leicht nachvollzogen und weitergeführt werden können.[12]

4. Es wurde dem Autor von mehreren Personen empfohlen, die bereits positive Erfahrungen damit gesammelt haben.

Zum Zeitpunkt dieser Arbeit ist die aktuelle Version die 0.96 vom 15.06.2007. Die Jadex-Entwickler arbeiten gegenwärtig unter Hochdruck an einer neuen Version, die bereits als Beta zur Verfügung steht. Der Autor hat sich jedoch für die ältere Version entschieden, da zum einen die Portierung der Agenten aus der Version 0.96 in die neue Version laut den Jadex-Entwicklern sehr einfach sein soll und zum anderen keine der Verbesserungen in der Beta für die Umsetzung der Arbeit benötigt wird.

3.3 Beschreibung des Agentensystems

Nachdem ein Agentensystem ausgewählt worden ist, werden zunächst noch die einzelnen Komponenten in Bezug auf ihre Implementation in Jadex betrachtet. Die Informationen zu diesem Abschnitt stammen von [AP07].

3.3.1 BDI Struktur

Die Beliefs werden in Jadex durch einen objektorientierten Ansatz dargestellt. Es gibt sowohl einzelne Fakten (Belief) als auch Mengen von Fakten (Beliefset). Ändert sich z.B. der Wert eines Beliefs oder die Liste eines Beliefsets, werden Events ausgelöst. Diese können wiederum Seiteneffekte auslösen, wie die Erzeugung neuer Goals. Da die Menge von Fakten in einem Beliefset sehr groß werden kann, kann mit Hilfe von Expressions[13] – ähnlich wie in Datenbanken – der Suchraum verarbeitet bzw. eingeschränkt werden.

In Jadex entsprechen die Ziele (Goals) den Desires. Üblicherweise startet der Agent mit geeigneten Initialgoals, die dann wiederum bei Bedarf neue Goals erzeugen können. Wie schon gesagt, können auch Events neue Goals erzeugen, so dass der Agent entscheiden muss, welches Ziel zuerst verfolgt wird. Außerdem muss er sich bei entsprechenden Zielen auch vom aktuellen Plan lösen können. Das Verfahren dafür lautet in Jadex „Easy Deliberation Strategy".

Anhand von Plänen erhält der Agent die Möglichkeit, seine Ziele zu erreichen. Sie sind grob in zwei Bereiche unterteilt. Der erste Teil ist der Header, in dem die Bedingungen für den

[12] Die vorherrschende Sprache im Projekt ist Java.

[13] Die Syntax basiert auf der Object Query Language und ist Java Expressions sehr ähnlich.

Plan stehen. Ist ein Plan zur Erreichung eines bestimmten Ziels geeignet und stimmen die Vorbedingungen mit dem aktuellen Zustand überein, kann ein Plan zur Ausführung gelangen. Dazu wird der zweite Teil benötigt. Pläne werden durch Java-Klassen realisiert. Die Klassen erben von der Superklasse „Plan"[14] und müssen eine Methode „body" implementieren. In dieser Methode sind die Aktionen festgehalten, die der Agent durchzuführen hat.

Es können auch Ziele existieren, für die es aktuell keinen gültigen Plan gibt. In einem solchen Fall muss der Agent entweder zunächst ein anderes Ziel verfolgen oder versuchen ein Unterziel (Subgoal) zu erreichen. Beispielsweise könnte ein Agent in einer Spielwelt das Ziel haben, zu einer bestimmten Position zu gelangen. Ist sein Weg dorthin blockiert, müsste er zunächst eine Lösung für dieses Problem finden. Dazu könnte er einen Plan für eine Teillösung ermitteln. Im Anschluss kann er dann sein eigentliches Ziel erreichen.

Bei dieser Art der Implementation gibt es noch einen weiteren Vorteil. Der Agent ist in der Lage, Ziele zu wiederholen, falls z.B. ein Plan fehlschlägt, weil sich die Bedingungen für den Plan bzw. das Ziel geändert haben.

Jadex unterscheidet verschiedene Goaltypen:

- Perform Goal
  Diese Ziele haben keine Zielbedingung. Sie werden ohne Rücksicht auf das Ergebnis ausgeführt.

- Achieve Goal
  Diese Ziele bringen den Agenten dazu, einen bestimmten Zustand der Welt zu erreichen. Unter Umständen kann der Agent unendlich oft versuchen, dieses Ziel zu erreichen, falls alle Pläne dafür immer wieder fehlschlagen.

- Query Goal
  Hierbei geht es darum, bestimmte Informationen zu sammeln. Dazu werden die Fakten des Agenten üblicherweise durch Expressions durchsucht. Sind die benötigten Informationen nicht vorhanden, muss der Agent versuchen, diese von der Welt zu erhalten. Dazu müssen andere Pläne ausgewählt werden. Der Staubsauger-Agent aus Abschnitt 3.1.1 könnte in einer größeren Welt ein solches Ziel haben, um die nächste verschmutzte Fläche zu finden. Wird in der Wissensbasis kein Eintrag gefunden, würde er einen Plan wählen, der ihn die Welt erkunden lässt, bis er Schmutz gefunden hat.

- Maintain Goal
  Ein Agent hat üblicherweise die Absicht, einen bestimmten Zustand zu vermeiden. Der Staubsauger-Agent könnte eine Ladestation in seiner Welt besitzen. Das Beseitigen von Schmutz kostet Energie. Ein Maintain Goal könnte sein, nicht unterhalb eines

[14] jadex.runtime.Plan

bestimmten Energielevels zu geraten. Tritt dieser Fall ein, wird das Ziel gestartet und üblicherweise mit hoher Priorität abgearbeitet.

- Meta Goal
  Meta Goals werden genutzt, wenn zu einem Ziel mehrere Pläne möglich sind. Dann kann über ein Meta Goal entschieden werden, welcher der Pläne gewählt wird („meta-level resoning").

3.3.2 Capabilities

Durch Capabilities können in Jadex bestimmt Fähigkeiten gruppiert werden. Spezialisierte Agenten können dann die Fähigkeiten wiederverwenden. Somit kann der Entwickler die Agenten anhand von Modulen erstellen.

Capabilities werden in eigenen XML-Dateien festgehalten. Sie besitzen einen eigenen Namensraum (Scope). Die Komponenten sind standardmäßig nicht sichtbar und müssen zur Verwendung exportiert werden (Kapselung). Es ist ebenfalls möglich Elemente abstrakt zu definieren. Dann kann ein Agent bestimmte Teile importieren und muss die referenzierten Teile zunächst selbst implementieren.[15]

3.3.3 Jadex Control Center

Das Framework bringt eine Entwicklungsumgebung für Agenten mit sich. Über das Control Center können Agenten gestartet, pausiert und beendet werden. Weiterhin bietet es die Möglichkeit, sich die aktuellen Beliefs, Goals und Plans einzelner Agenten ausgeben zu lassen. Diese Funktion ist äußerst hilfreich und für die Entwicklung komplexer Agenten unerlässlich.

[15] Das ist vergleichbar mit abstrakten Klassen in einer Objekt-orientierten Programmiersprache.

Kapitel 4

Implementation der Agenten

Nachdem das letzte Kapitel allgemeine Eigenschaften von Agentensystemen erläutert hat, geht es in diesem Kapitel um die Implementation der Agenten in der Timadorus-Welt.

4.1 Kommunikation zwischen Agenten und Gameserver

Zunächst gilt es, die Verbindung zwischen dem Agenten und dem Gameserver aufzubauen. Dazu lädt der Agent die im Common-Paket enthaltenen Klassen und Methoden. Sowohl Server als auch Client nutzen dieselben Klassen zur Erzeugung von Nachrichten.

4.1.1 Die Klasse „Client"

Das Erzeugen eines neuen Agenten erzeugt gleichzeitig eine neue Instanz der Klasse „Client". Diese Klasse ermittelt die Verbindungsdaten und baut eine Verbindung zum Gameserver auf. Sind die Verbindungsdaten fehlerhaft bzw. der Server nicht erreichbar, wird der Verbindungsaufbau abgebrochen. In diesem Fall wird die Erzeugung des Agenten ebenfalls unterbrochen und eine entsprechende Nachricht in der Systemkonsole ausgegeben.

4.1.2 Die Klasse „ClientMessageListener"

Diese Klasse stellt das Tor zum Gameserver dar. Jeder Client wird auf dem Server separat geführt und bei einen so genannten Channel registriert. Über diesen Channel werden Nachrichten an alle Objekte geschickt. Dabei handelt es sich um allgemeine Nachrichten, wie die Bewegung eines Objektes von einem Ort zum nächsten.

Um das Nachrichtenaufkommen gering zu halten, wird für alle Agenten einer Plattform nur eine einzige Verbindung zum Kanal aufgebaut. Die eingehenden Nachrichten werden in die lokale Repräsentation der Welt überführt. Die Agenten richten ihre Anfragen an die Welt nicht direkt an den Server, sondern an die lokale Instanz.

4.1.3 Die Klasse „ClientMessageProcessor”

Diese Klasse nimmt die eingehenden Nachrichten entgegen und leitet sie an den jeweiligen Agenten weiter. Zu diesem Zweck hat jeder Agent seinen eigenen Nachrichten-Prozessor.

4.1.4 Die Klasse „Environment”

Wie bereits erwähnt, werden allgemeine Nachrichten von einer lokalen Instanz verarbeitet. Diese lokale Instanz ist durch das Singleton-Pattern realisiert worden. Ein Agent, der Wissen über die Welt benötigt, kann hier nachfragen.

Ebenso verwaltet diese Klasse für alle Agenten eine einzige Instanz der Umgebung. Somit wird zusätzlich Speicherplatz gespart, wodurch mehr Agenten auf einem einzigen Computer verwaltet werden können.

4.2 Aufbau

Die gemeinsamen Fähigkeiten der Agenten sind in einer einzigen Capability-Datei festgehalten worden.[16]

4.2.1 Beliefs

Abbildung 4.1 zeigt den Ausschnitt der Beliefs in der Object.capability.xml.

Erläuterung: [<name>][<typ>][<wert>]

- [me][Agent][abstract]
  Dieses Belief ist die eigene Repräsentation des Agenten. Er erhält darüber Informationen zu seiner gegenwärtigen Situation (Position, Sichtbereich) und kann darüber hinaus Aktionen (bewegen, fressen, etc) ausführen.

- [enemies][WorldObject][](set)
  Dies ist eine Liste der bekannten Feinde. Befindet sich der Feind aktuell im Sichtbereich, kann der Agent direkt auf ihn reagieren. Die Liste kann auch Feinde enthalten, die sich nicht mehr im Sichtbereich befinden. Dann handelt es sich um Kopien der originalen Objekte, die nicht mehr verändert werden. Somit bleibt die letzte bekannte Position eines Feindes erhalten. Sollte sich der Feind mittlerweile entfernt haben, bleibt die Position trotzdem solange erhalten, bis der Agent die Position erneut besucht und überprüft hat.
  Dieses Belief wird in der Implementation eigentlich nur von den Rehen genutzt, da für

[16] Sie trägt den Namen „Object.capability.xml”.

die Jäger keine Feinde in der Welt existieren. Jedoch sind sowohl Jäger als auch Rehe so konfiguriert, dass sie unbekannte Objekte als Feinde wahrnehmen.

- [friends][WorldObject][](set)
  Ähnlich wie das Beliefset „enemies", entspricht dies der Liste mit den Freunden. Der Einfachheit halber sind Objekte gleicher Klasse als Freunde definiert (also Rehe und Jäger jeweils untereinander).

- [prey][WorldObject][](set) & [food][WorldObject][](set)
  Der Agent unterscheidet seine Beute je nach Entfernung oder Zustand. Für Rehe ist Gras solange „prey", bis sie auf Fressreichweite herangekommen sind. Dann wird aus „prey" automatisch „food". Für Jäger gelten lebendige Rehe als „prey". Diese werden erst zu „food", wenn sie erlegt wurden.

- [markers][Marker][](set)
  Dieses Belief wird für die im Abschnitt 5.5.4 eingeführten Marker zur Beseitigung des Problems der lokalen Maxima benötigt. Weitere Informationen dazu finden sich im Abschnitt 7.4.4.

- [vision][Vision][new Vision()]
  Der Agent erhält eine Instanz der Klasse Vision (siehe Abschnitt 7.4.3). Diese Klasse verwaltet für ihn den Sichtbereich und liefert auf Anfrage eine Liste aller Objekte im Sichtbereich. Außerdem aktualisiert sie bei jeder Anfrage die Beliefs „enemies", „friends", „prey" und „food".

- [gui][boolean][false]
  Über dieses Belief wird festgelegt, ob ein Debugfenster für die Sicht des Agenten angezeigt werden soll. Der Agent selbst verwendet dieses Belief nicht weiter.

4.2.2 Goals

Alle Ziele sind als „unique" definiert und werden daher immer nur einmal zur Zeit erzeugt. Die Agenten besitzen zwei gemeinsame Ziele. Abbildung 4.2 zeigt den Ausschnitt der Goals.
Erläuterung: [<name>][<typ>]...[<bedingung>:<wert>]

- [achieve_consumeFood][achieve][creationcondition:food.length > 0][contextcondition:me.isAlive][targetcondition:food.length == 0]
  Dieses Ziel wird sowohl von Rehen als auch Jägern unterstützt. Es wird ausgelöst, sobald Nahrung in der Nähe ist.

```
<beliefs>
    <beliefref name="me" class="Agent" exported="true">
        <abstract />
    </beliefref>
    <beliefset name="enemies" class="WorldObject" exported="true" />
    <beliefset name="friends" class="WorldObject" exported="true" />
    <beliefset name="prey" class="WorldObject" exported="true" />
    <beliefset name="food" class="WorldObject" exported="true" />
    <beliefset name="markers" class="Marker" exported="true" />
    <belief name="vision" class="Vision" exported="true">
        <fact>new Vision()</fact>
    </belief>
    <belief name="gui" class="boolean" exported="true">
        <fact>false</fact>
    </belief>
</beliefs>
```

Abbildung 4.1: Die Beliefs in der Object.capability.xml

- [perform_walk][perform][contextcondition:me.isAlive]
  Auch dieses Ziel wird von beiden Agententypen unterstützt. Es wird gleich zu Anfang erzeugt und sorgt für die Auslösung der anderen Ziele. Daher darf dieses Ziel auch niemals ausgeschlossen werden (exclude=”never”).

Zusätzlich zu den gemeinsamen Zielen, haben sowohl Jäger als auch Rehe jeweils noch ein weiteres Ziel. Auf die grafische Darstellung wird verzichtet, sie entspricht dem Achieve Goal aus Abbildung 4.2.

- [achieve_attack][achieve][creationcondition:prey.length > 0][targetcondition:prey.length == 0]
  Dieses Ziel ist derzeit nur für die Jäger relevant, da Rehe ihre Beute (Gras) nicht angreifen müssen. Das Ziel wird solange erzeugt, wie Beute in der Nähe ist.

- [achieve_escape][achieve][creationcondition:enemies.length > 0][targetcondition:enemies.length == 0]
  Dieses Ziel ist derzeit nur für die Rehe relevant, da Jäger in dieser Welt keine Feinde haben. Das Ziel wird solange erzeugt, wie Feinde in der Nähe sind.

Die Kontextbedingung „me.isAlive” sorgt dafür, dass die Ziele erst verfolgt werden, wenn der Agent vollständig am Gameserver angemeldet und nur solange der Agent nicht gestorben ist.

```
<goals>
    <achievegoal name="achieve_consumeFood" exclude="never"
        exported="true">
        <unique />
        <creationcondition>
            $beliefbase.food.length &gt; 0
        </creationcondition>
        <contextcondition>
            $beliefbase.me.isAlive()
        </contextcondition>
        <targetcondition>
            $beliefbase.food.length == 0
        </targetcondition>
    </achievegoal>
    <performgoal name="perform_walk" retry="true" exclude="never"
        retrydelay="1000" exported="true">
        <unique />
        <contextcondition>
            $beliefbase.me.isAlive()
        </contextcondition>
    </performgoal>
</goals>
```

Abbildung 4.2: Die Goals in der Object.capability.xml

4.2.3 Plans

Abbildung 4.3 zeigt die gemeinsam genutzten Pläne. Es existiert für jedes Ziel jeweils nur genau ein Plan. Daher wird auf die gesonderte Auflistung an dieser Stelle verzichtet.

Die Implementation des Plans „ConsumeFood" prüft lediglich, ob sich der Agent tatsächlich innerhalb der entsprechenden Reichweite zum Konsumieren befindet und löst ggf. die entsprechende Aktion aus.

Entsprechend der oben genannten zusätzlichen Ziele der Jäger und Rehe, gibt es noch die Pläne „Attack" und „Escape". Auf die grafische Darstellung wurde hier ebenfalls verzichtet, da sie im Aufbau dem „ConsumeFood"-Plan ensprechen. Die Implementation des „Attack"-Plans prüft lediglich, ob sich das Ziel innerhalb der Angriffs-Reichweite befindet. Der „Escape"-Plan greift auf das bereits erwähnte Belief „markers" zu. Die Idee dahinter wird in Abschnitt 5.5.4 erläutert, für den Moment reicht es zu wissen, dass der Liste ein weiterer Marker hinzugefügt wird.

Der „Walking"-Plan ist etwas komplexer. Jedoch fehlen zum Verständnis an dieser Stelle noch die Informationen über Potentialfelder aus Kapitel 5. Daher wird der Inhalt dieses Plans in Abschnitt 7.5 erläutert.

```
<plans>
    <plan name="consumeFood" exported="true">
        <body class="ConsumeFoodPlan" />
        <trigger>
            <goal ref="achieve_consumeFood" />
        </trigger>
    </plan>
    <plan name="walk" exported="true">
        <body class="WalkingPlan" />
        <trigger>
            <goal ref="perform_walk" />
        </trigger>
    </plan>
</plans>
```

Abbildung 4.3: Die Plans in der Object.capability.xml

4.2.4 Configurations

Es existieren zwei Konfigurationen für jeden Agenten. Sie unterscheiden sich lediglich darin, ob ein Debug-Fenster angezeigt werden soll. Die Konfigurationen heißen „default” und „showMap”. Sie erzeugen beide das Initialziel „perform_walk”.

4.3 Konkrete Implementation

Wie bereits erwähnt, werden nicht alle Fähigkeiten von beiden Agententypen genutzt.

4.3.1 Rehe

Die Rehe nutzen alle in Abschnitt 4.2.1 erläuterten Beliefs. Das abstrakte Belief „me” wird mit einer Instanz der Klasse „Deer” belegt. Die Klasse erbt von der Klasse „Agent” und setzt typspezifische Werte (z.B. Sichtweite). Außerdem überschreibt die Klasse die folgenden Methoden „isEnemy”, „isFriend”, „isPrey” und „isFood”. Durch diese Methoden kann ermittelt werden, welche Bedeutung ein bestimmtes Objekt für ein Reh hat.

Des Weiteren importieren die Reh-Agenten die beiden Ziele „perform_walk” und „achieve_consumeFood”. Wie bereits erwähnt implementieren sie zusätzlich das Ziel „achieve_escape”.

Natürlich werden die beiden Konfigurationen ebenfalls importiert.

4.3.2 Jäger

Die Jäger nutzen ebenfalls alle in Abschnitt 4.2.1 erläuterten Beliefs. Sie belegen das Belief „me" jedoch mit einer Instanz der Klasse „Hunter". Die Klasse dient dem selben Zweck wie die Klasse „Deer" im vorherigen Unterabschnitt.

Auch von diesen Agenten werden die beiden Ziele „perform_walk" und „achieve_consumeFood" und die Konfigurationen importiert. Zusätzlich wird auch noch das Ziel „achieve_attack" implementiert. Es wird ausgelöst, sobald sich Beute in Sichtweite befindet.

4.4 Zusammenfassung

Damit wäre die Implementation der Agenten in Jadex bereits abgeschlossen. Es fehlt lediglich die Klasse „WalkingPlan", die wie gesagt in Kapitel 7.5 erläutert wird.

Kapitel 5

Theorie der Potentialfelder

In diesem Kapitel wird das Konzept der Potentialfelder vorgestellt. Dadurch sind die Agenten in der Lage, sich in der Welt zu orientieren. Die Implementation erfolgt in den kommenden beiden Kapiteln.

5.1 Grundlagen

Die Idee der Potentialfelder stammt aus der Robotersteuerung. Roboter, die sich in einer unbekannten Umgebung fortbewegen, benötigen Sensoren, um Hindernissen ausweichen zu können. Dazu kann die Umgebung beobachtet werden, um Objekte zu erkennen und mit Hilfe des Abstandes zu einem Objekt eine Kollision zu vermeiden. Objekterkennung ist jedoch nicht trivial und kostet viel Rechenleistung.

Für die reine Navigation braucht der Roboter nicht zu wissen, was das Objekt ist, es würde reichen, wenn z.B. im entsprechenden Abstand zum Objekt eine Linie auf dem Boden gezogen würde, die er mittels einer Kamera erkennen und deren Überschreitung verhindern könnte. Es könnte jedoch auch ein Magnetband (oder ein stromdurchflossener Leiter) angebracht werden. Im entsprechenden Abstand könnte der Roboter das Magnetfeld erkennen (messen) und wüsste, dass der Bereich unpassierbar ist.

In diesem Beispiel wurde die abstoßende Kraft von Magnetfeldern genutzt. Übertragen auf eine virtuelle Welt, kann ein Agent nun jedem Objekt ein eigenes Feld zuordnen, welches anhand der Polung (positiv, negativ) Auswirkungen auf ihn hat. So könnte er z.B. einem Objekt der Begierde oder einem Reiseziel ein positives Feld zuordnen, welches ihn anziehen würde.[17]

[17] In der Literatur wird die Polung (je nach Quelle) unterschiedlich ausgelegt. Wird der Agent als positives Objekt betrachtet, würde er von positiven Feldern abgestoßen und von negativen angezogen. Hier passt eher das Bild des negativen Teilchens (Neutron) in einem Magnetfeld, daher werden hier positive Felder als anziehend betrachtet.

Anders als bei den Robotern, werden die Felder jedoch nicht vorgegeben indem an den entsprechenden Stellen Magnetbänder angebracht werden. Die Felder sind Bestandteil der Sicht eines Agenten. Die von ihm als relevant identifizierten Objekte werden intern mit entsprechenden Feldern belegt. Die Identifikation von Objekten in virtuellen Welten ist deutlich leichter als die Interpretation von Kamerabildern. Die Welt liefert in der Regel Informationen zum Typ eines Objektes. Welche Ausprägung ein Feld hat, wird ebenfalls vom Agenten selbst bestimmt.

In Abbildung 5.1 ist ein Ausschnitt einer Beispielwelt zu sehen. In ihr befinden sich drei Objekte (Agent, Gegner, Gegenstand). Außerdem gibt es dort Berge und ein weiteres Konzept aus der Spieleprogrammierung, den „Nebel des Krieges" (Fog of War, FoW)[FOW]. Somit gibt es fünf unterschiedliche Objektarten und auch Felder.

Der Einfachheit halber wurde die Welt rasterisiert. Die Auflösung des Rasters beeinflusst die Genauigkeit, aber auch die Berechnungsdauer. In den folgenden Kapiteln wird gezeigt, dass die Welt nicht zwingend rasterisiert werden muss.

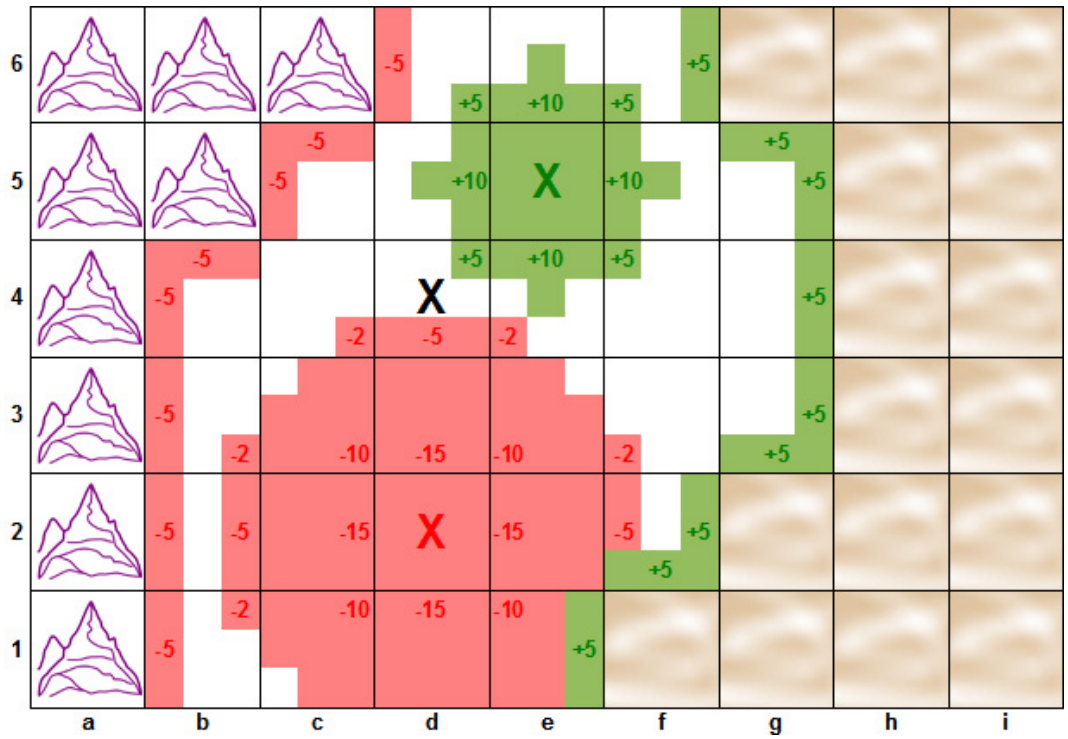

Abbildung 5.1: Beispielwelt mit einem Agenten (schwarz), einem Gegner (rot) und einem Gegenstand (grün).

Der Gegner und die Berge erhalten jeweils negative Felder. Der Agent soll sich vom Gegner entfernen und von den Bergen fern halten. Der Gegenstand (z.B. ein Schatz) und der FoW erhalten jeweils positive Felder. Der Agent soll den Schatz einsammeln. Das posi-

tive Feld des FoW würde den Agenten dazu bewegen, die Welt zu erkunden, sofern keine stärkeren Felder existieren (Schatz und Gegner).

In Abbildung 5.2 wurden die Feldstärken überlagert (aufsummiert). Das Resultat ist eine Karte der Umgebung mit Werten, die für den Agenten die Beliebtheit darstellen. Er kann sich immer nur waagerecht oder senkrecht ein Feld weit fortbewegen. Betrachtet er nun sein Umfeld, erkennt er als nächsten Schritt eine Bewegung nach oben (auf d5), da dort der höchste Wert zu finden ist. Im nächsten Schritt würde er den Gegenstand einsammeln.

Nach Einsammeln des Gegenstandes verschwindet dessen Feld. Unter der Annahme, dass sein Gegner dieselben Schritte gemacht hat (also hoch und dann links), stünde der Gegner jetzt in Feld e3 und der Agent auf d5. Da sich das Feld des Gegners mit verschoben hat, entspricht dessen Einfluss auf den Agenten den Werten in Abbildung 5.1. Der Agent würde sich für eine Bewegung auf das Feld e6 (kein Feld, Wert = 0) entscheiden, da dort der beste Wert zu finden ist.

Wäre der Agent allein in der Welt (nur der Agent, die Berge und der FoW), hätten alle vier Felder den Wert 0. Um den Agenten in Bewegung zu bringen, könnte eine Umkreissuche implementiert werden, die solange den Suchradius erweitert bis ein Wert größer 0 gefunden wurde.[18] Dieses Vorgehen ist vergleichbar mit einer Breitensuche ([RN04] S. 105) bzw. einem optimistischen Bergsteigen ([RN04] S. 151), bei der schrittweise die umliegenden Bereiche durchsucht werden. Alternativ könnte eine abgewandelte A*- oder vergleichbare Suche ([RN04] S. 134 und [DeL08]) den höchsten Wert im Sichtbereich ermitteln und einen Pfad zu diesem liefern.[19] Dazu müsste jedoch der gesamte Sichtbereich durchsucht werden.

Im weiteren Verlauf dieser Arbeit wird gezeigt, dass bei entsprechender Nutzung der Felder überhaupt keine Suchalgorithmen benötigt werden. Dazu werden in Anlehnung an die Quellen in Abschnitt 5.6 Anwendungsfälle und mögliche Probleme aufgezeigt.

5.2 Navigation

Wie schon im Beispiel mit dem Roboter, dienen die Felder hauptsächlich der Navigation. Befindet sich der Agent im Einflussbereich eines Objektes, also innerhalb des Feldes, wird er unmittelbar durch das Feld beeinflusst.

In Abbildung 5.3 ist eine weitere Beispielwelt mit einem Agenten, drei Hindernissen und einem Zielort zu sehen. Der Agent möchte zum eingetragenen Ziel gelangen. Daher wird das Ziel mit einem positiven Feld belegt, welches über die Entfernung abnimmt. Eine typische Formel für die Berechnung der Feldstärke findet sich in Algorithmus 5.1.

[18] Hierbei ist zu beachten, dass der jeweilige Umkreis komplett durchsucht werden sollte. Falls z.B. im Abstand von drei Feldern um den Agenten zwei Werte existieren, kann nicht garantiert werden, dass der erste gefundene Wert auch der beste ist.

[19] Abgewandelt in dem Sinne, dass nicht ein Ziel, sondern das beste Ziel gesucht würde.

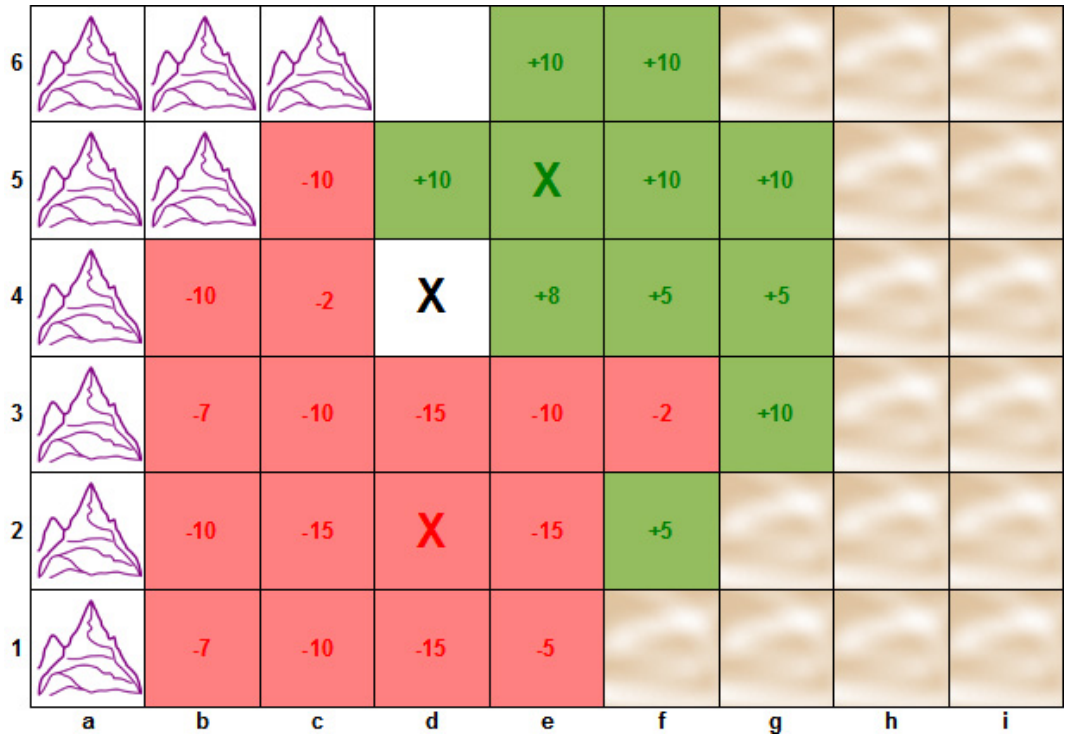

Abbildung 5.2: Die resultierende Karte aus Abbildung 5.1.

Algorithm 5.1 Erzeugung eines typischen Feldes.

```
IF dist < range
        field = force / dist
ELSE
        0
```

Der Agent prüft nun die umliegenden acht Felder (Himmelsrichtungen) und ermittelt das positivste[20]. Dies ist sein nächstes Ziel. Dort angekommen, prüft er wieder die umliegenden Felder und so weiter. Dieser Vorgang wird wiederholt, bis der Agent sein endgültiges Ziel erreicht hat[21].

Damit der Agent nicht unmittelbar an den Hindernissen entlang läuft, sondern einen Sicherheitsabstand einhält (ein Hindernis könnte ja auch ein Abhang sein), wird jedes Kästchen, das teilweise oder vollständig blockiert ist, mit einem negativen (abstoßenden) Feld

[20] Der Wert kann auch kleiner 0 sein, wenn alle Felder negativ sind.

[21] Der Agent kann auch vorzeitig zum Stillstand kommen, falls er kein Feld mit positiverem Wert mehr findet, siehe Abschnitt 5.4.

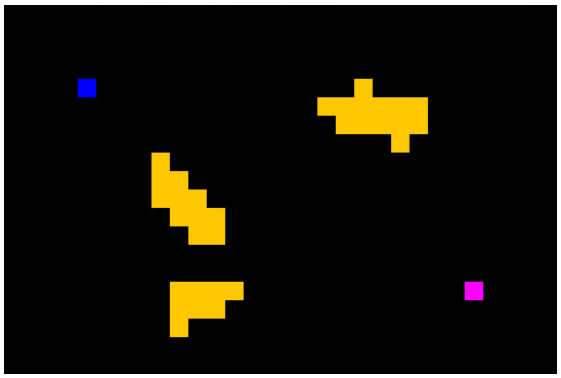

Abbildung 5.3: Ausschnitt einer Welt mit drei Hindernissen (orange) einem Agenten (blau) und einem Ziel (pink).

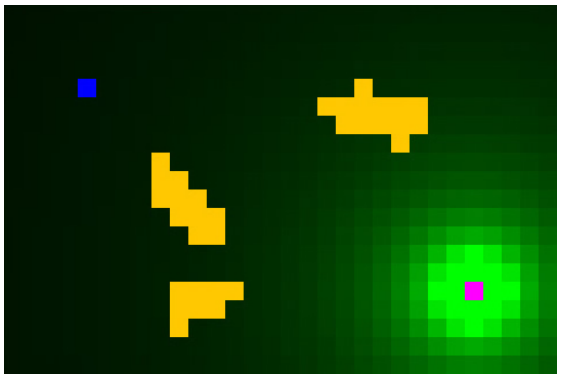

Abbildung 5.4: Das Ziel (pink) erzeugt ein positives Feld.

belegt (Abbildung 5.5). Der zurück zu legende Weg des Agenten ist in Abbildung 5.6 zu sehen.

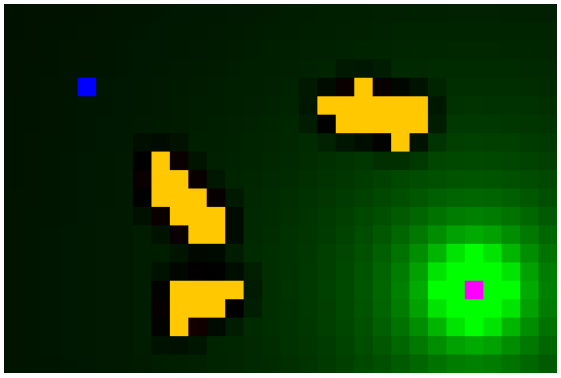

Abbildung 5.5: Ausschnitt der Welt mit negativen Feldern um die Hindernisse.

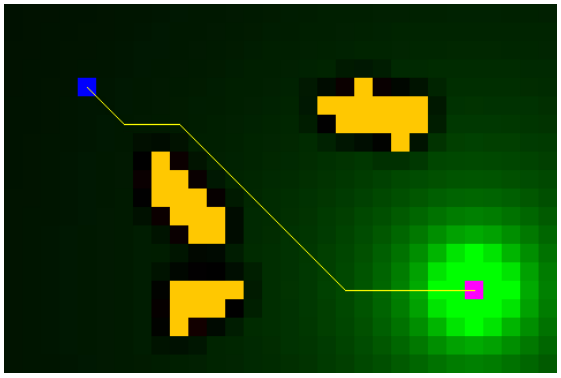

Abbildung 5.6: Der Pfad auf dem sich der Agent zum Ziel bewegt.

Durch die Nutzung der Potentialfelder werden Wegfinde-Algorithmen (zumindest theoretisch) überflüssig. Das Problem der lokalen Maxima in Abschnitt 5.4 könnte mit Hilfe eines Wegfinde-Algorithmus gelöst werden. Ein Vorteil der Nutzung von Potentialfeldern ist jedoch der Verzicht auf solche Algorithmen und es existieren Lösungen die dies tun. Die Ressourcennutzung und der Rechenaufwand werden in den kommenden Kapiteln genauer betrachtet.

5.3 Weitere Anwendungen

Wie bereits gesehen, können positive Felder einen Agenten zu einem bestimmten Ort führen, während negative Felder dafür sorgen, dass der Agent sich von einem Ort fern hält. Es gibt aber auch noch andere Effekte, die sich mit entsprechenden Feldern erzielen lassen.

5.3.1 Kollisionsvermeidung

Da zwei Objekte nicht zur selben Zeit am selben Ort sein dürfen, kann mittels der Potentialfelder eine Kollisionserkennung erfolgen. Dazu wird jedem Objekt in der Welt ein stark negatives Feld zugeordnet, welches jedoch nur über seine Ausmaße hinweg strahlt. Ein Fass mit einem halben Meter Radius entspräche dann einem Feld mit maximaler Stärke (unpassierbar) und einer Reichweite von 0,5. Da ein Fass normalerweise Rund ist, funktioniert das sehr gut. Wird hingegen eine Kiste (die in der Regel eckig ist) betrachtet, wird ein Problem offensichtlich. Die Kollisionserkennung funktioniert nur grob, da Felder kreisrund abstrahlen. Bei einer rasterisierten Welt können zusätzliche Fehler auftreten, wenn eine Wand der Kiste nicht direkt auf einer Rasterlinie liegt. Entweder wird ein Feld fälschlicherweise als blockiert erkannt (teilweise Überlappung) oder als nicht blockiert, sodass Objekte doch wieder in einander stehen könnten.

Potentialfelder eignen sich jedoch als günstiges Mittel zur Kollisionsvermeidung (Abstand halten), wie schon in Abschnitt 5.1 mit den Hindernissen gesehen. Eine Kiste mit zwei Metern Kantenlänge erhält dann ein Feld mit maximaler Stärke und einer Reichweite von 1,5 (1,414 bzw. „Wurzel aus 2” würden bereits ausreichen), so dass der Agent sicher um das Objekt herum navigieren würde.

5.3.2 Angriff mit Fernwaffen

In einem Szenario mit zwei einander feindlich gesinnten Agenten könnten Felder dazu genutzt werden, die Agenten in Angriffsreichweite zu bringen.

Ein Agent mit einer Hieb-/Stichwaffe (z.B. ein Schwert) müsste auf kurze Distanz an seinen Gegner heran, während ein Agent mit einer Fernwaffe (z.B. Pfeil und Bogen) lieber Abstand halten würde. Für den Agenten mit der Fernwaffe wäre daher eine Kombination zweier Felder sinnvoll. Das eine Feld zieht den Bogenschützen an, während ein zweites, negatives Feld den Schützen in sicherer Distanz zum Schwertkämpfer hält. In der Praxis würden einem Objekt nicht zwei Felder zugeordnet, sondern ein heterogenes, siehe Formel 5.2.

Algorithm 5.2 Erzeugung eines heterogenen Feldes, als Überlagerung zweier Felder.

```
IF dist < range
        IF dist <= attackRange
                field = neg(force / dist)
        ELSE
                field = pos(force / dist)
ELSE
        field = 0
```

5.3.3 Flucht

Ein weiterer Aspekt der abstoßenden Felder ist die Nutzung als Flucht-Signal. Begegnet ein Agent einem ihm feindlich gesinnten Agenten, der noch dazu kräftiger ist als er selbst, sollte er sich für einen Rückzug entscheiden. Zu diesem Zweck würde ein stark negatives Feld, welches alle anderen Felder überdeckt, den Agenten von seinem Gegner wegtreiben.

Ähnliches gilt für ein im vorigen Abschnitt beschriebenes Szenario. Begegnen sich z.B. zwei Panzer, die zwischen zwei Schüssen eine Cooldown-Phase[22] haben, sollte der Agent nicht in Schussreichweite verharren, sondern sich möglichst schnell entfernen.

5.4 Probleme

Das Konzept der Potentialfelder ist trotz der dargestellten Stärken nicht perfekt, nicht zuletzt da es oftmals als zu rechenintensiv angesehen wird. Die folgenden Kapitel versuchen das Gegenteil zu zeigen. Hier jedoch zunächst ein paar bekannte Probleme und mögliche Lösungen:

5.4.1 Lokale Maxima

Eines der Hauptprobleme bei der Verwendung von Potentialfeldern liegt in den so genannten „lokalen Maxima". Das sind Positionen, von denen aus der Agent im direkten Umfeld (acht angrenzende Felder) kein besseres Feld findet. So eine Position kann der Agent in drei Fällen erreichen.

Der erste Fall ist trivial: Der Agent hat ein Ziel erreicht[23].

[22] Nachladen und Abkühlen des Rohres

[23] Durch das Erreichen eines Ziels würde jedoch dessen Feld entfernt und die Karte würde sich verändern.

Zweiter Fall: Versperrter Weg

Zwischen dem Agenten und seinem Ziel befindet sich ein Hindernis, sodass der Agent eine Position erreicht, die keinen Potentialunterschied besitzt. Solch ein Szenario beschreibt Abbildung 5.7.

Aus dieser Position gibt es verschiedene Auswege. Zum einen kann wie in Abschnitt 5.1 bereits erwähnt, eine Umkreissuche durchgeführt werden. Jedoch soll der Einsatz von Suchverfahren vermieden werden. Daher wäre zum anderen der Einsatz einer Pheromon-Spur wie in [CT04] beschrieben möglich. Der Agent belegt die letzten n Felder seines Weges (einschließlich seiner eigenen Position) mit einem negativen Wert. Somit werden bereits besuchte Felder bzw. das eigene für den Agenten weniger attraktiv und es finden sich attraktivere Felder im Umkreis, auf die der Agent dann ausweichen kann.

Abbildung 5.8 zeigt den endgültigen Pfad, den der Agent beschreitet. Die rote Linie zeigt Bewegungen, die der Agent nur durch die Nutzung der Pheromon-Spur getätigt hat. Die erste Bewegung (nach oben oder nach unten) wurde per Zufall entschieden, da beide Felder den selben Wert besaßen.

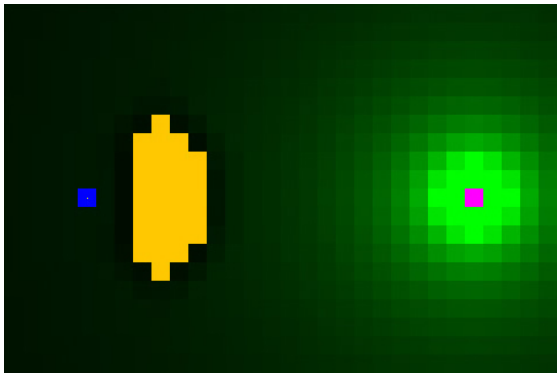

Abbildung 5.7: Zwischen dem Agenten und seinem Ziel liegt ein Hindernis.

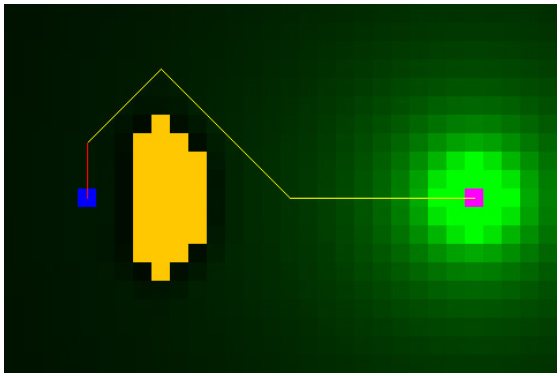

Abbildung 5.8: Der Pfad auf dem sich der Agent zum Ziel bewegt.

Dritter Fall: Sackgasse

Ein weitaus schwierigeres Problem sind Sackgassen in der Landschaft. Mit Hilfe der Pheromon-Spur aus dem letzten Unterabschnitt könnte ein Agent zwar wieder heraus finden, jedoch müsste für eine große Sackgasse (langer Weg) die Anzahl n der gemerkten Felder entsprechend hoch gewählt werden.

Eine Unterstützung für die Lösung mit der Pheromon-Spur bietet das konvexe Füllen von Lücken, wie in [HJ08c] („Assigning charges”) beschrieben. Dabei wird die Landschaft Kästchen für Kästchen überprüft und mit den jeweils umliegenden Kästchen verglichen. Sind

ausreichend viele umliegende Kästchen blockiert, wird auch das aktuell betrachtete Kästchen als blockiert markiert. Abbildung 5.9 zeigt solch einen Fall und Abbildung 5.10 zeigt die Landschaft nach dem Füllen der Lücken. Die Ausgangssituation ist nun die gleiche wie in Abbildung 5.8 und somit auch der resultierende Pfad.

Nachteil dieser Lösung ist, dass der Agent doch wieder mehr als nur die ihn umgebenden Felder betrachten muss, um Lücken füllen zu können. Ein Vorschlag für eine alternative Lösung, bei der dies nicht der Fall ist, findet sich im Abschnitt 5.5.

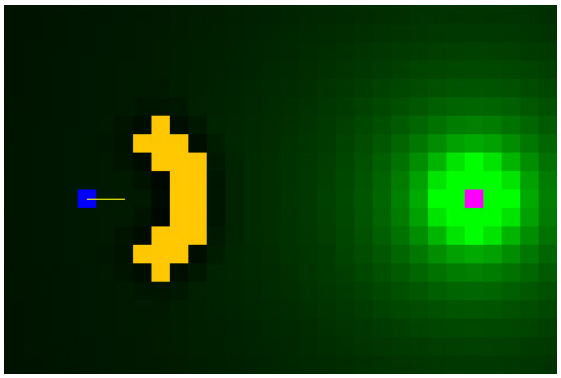

Abbildung 5.9: Der Agent läuft in eine Sackgasse.

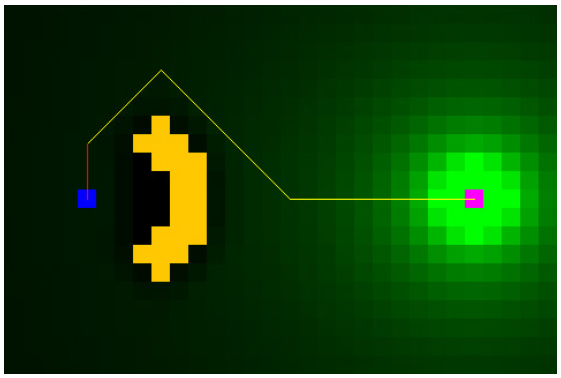

Abbildung 5.10: Der Pfad auf dem sich der Agent zum Ziel bewegt, wie in Abbildung 5.8.

5.4.2 Überlagerungen

Ein weiteres Problem liegt in der Verarbeitung sich überlagernder Felder. Zwei benachbarte Magneten wirken nach außen als ein einzelnes, stärkeres Magnetfeld. In einigen Situationen ist das auch wünschenswert. Beispielsweise wenn der Agent sich in einer Umgebung befindet, in der zu seiner Linken ein Objekt seiner Begierde (z.B. Geld) und zu seiner Rechten gleich zwei Objekte liegen und er nur eine begrenzte Zeit zum Einsammeln hat. Wenn sich die Felder addieren, würden die ggf. weiter entfernten Felder ihn anziehen. Handelt es sich hingegen beispielsweise um eine Aufgabenstellung ähnlich dem Rundreiseproblem in [UL08] (S. 141ff), bei dem es darum geht den kürzesten Pfad über mehrere Zwischenziele (die Objekte) zu finden, wäre dies nicht erwünscht. In Abbildung 5.11 ist die Überlagerung von Feldern zu sehen. Der resultierende Pfad ist jedoch der Pfad zu einem weit entfernten Objekt.

Um dieses Problem zu umgehen, dürfen die Felder nicht aufaddiert werden. Stattdessen

muss der maximalste Wert in einem Punkt ermittelt werden[24], siehe dazu Abbildung 5.12. Nun findet der Agent das dichteste Ziel zuerst.

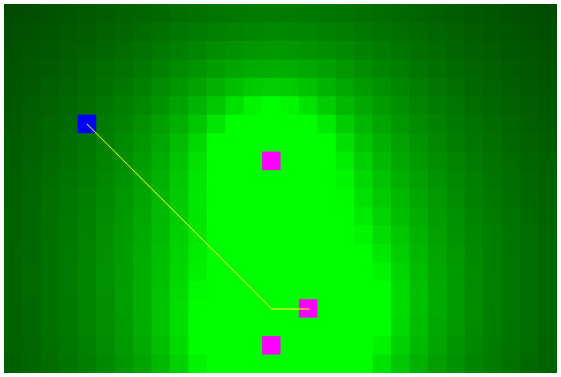

Abbildung 5.11: Durch die Addition von Felder, findet der Agent nicht das dichteste Ziel.

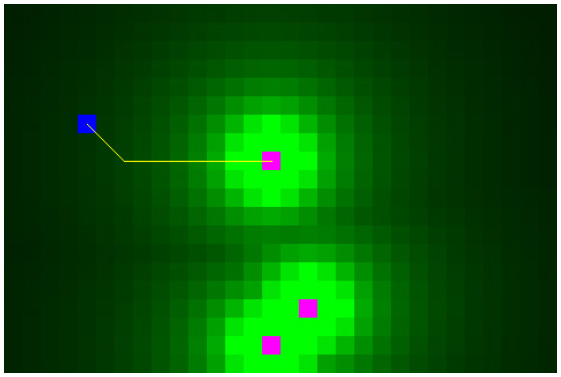

Abbildung 5.12: Durch Bestimmung der maximalen Feldstärke, findet der Agent das dichteste Ziel.

5.4.3 Feinabstimmung

Neben der Entscheidung, welche Felder additiv sind und welche maximiert werden müssen, existiert für die Implementierung eine weitere Schwierigkeit.

Nachdem bestimmt wurde, welche Felder es gibt und welche Ausprägungen sie haben (positiv oder negativ), gilt es zu bestimmen mit welcher Stärke sie den Agenten beeinflussen. Diese Entscheidung ist stark von der Situation und, wie schon das Problem im vorherigen Abschnitt, von der Aufgabenstellung abhängig. Daher muss sie für jede Anwendung erneut getroffen werden.

Hierbei bietet es sich an, einen selbstlernenden Ansatz zu wählen. Dazu erhält jedes Feld einen Multiplikator von initial 1. Durch Beobachtung seiner Situation kann der Agent dann selbstständig die Stärke des Feldes abstimmen. Dabei verstärkt ein Wert oberhalb von 1 den jeweiligen Wert des Feldes, während ein Multiplikator-Wert zwischen 0 und 1 die Feldstärke abschwächt. In [CT04] wird beschrieben, wie ein Agent eine komplett unbekannte Welt erlernt. Dazu werden weitaus komplexere Methoden als die gerade beschriebene angewandt.

Für diese Arbeit war der Aufwand eines selbstlernenden Ansatzes zu komplex. Da das Verfahren nicht unbedingt zum Verständnis der Potentialfelder beiträgt. Daher wurden bei der Implementierung die Feldstärken durch ausprobieren bestimmt.

[24] Dieses Verhalten widerspricht den magnetischen Gesetzen, nach denen Felder addiert werden.

5.5 Optimierung

Da die Verwaltung und Berechnung der Felder offensichtlich einen Aufwand bedeuten, ergeben sich potentielle Optimierungsmöglichkeiten, die im Folgenden angesprochen werden sollen.

5.5.1 Aufteilung nach Feldarten

In einer dynamischen Welt müssen bestimmte Felder immer angepasst werden. Es gibt jedoch auch statische Felder. In [Hag09b] werden drei Feldarten unterschieden:

1. Statische Felder: Die von der Landschaft bzw. den stationären Hindernissen ausgehenden Felder sind über die Zeit unveränderlich und könnten in einer Karte (z.B. zweidimensionales Array/Feld) gespeichert werden. Dann müsste zur Laufzeit keine Berechnung, sondern lediglich ein Lesezugriff erfolgen.

2. Semi-statische Felder: Das sind Felder von stationären Objekten, die unter bestimmten Umständen jedoch verändert werden können. Ein Beispiel dafür wäre eine feindliche Basis, die zerstört wird. Diese könnten in eine weitere Karte eingetragen werden. Verändert sich eines dieser Objekte, müsste lediglich der betroffene Teil der Karte aktualisiert werden.

3. Dynamische Felder: Bewegliche bzw. flüchtige Objekte (z.B. Gegenstände die nach Ablauf einer Zeit entfernt werden) erzeugen nahezu bei jeder Anfrage einen anderen Wert und müssen daher zur Laufzeit verarbeitet werden. Das muss jedoch nicht zwingend eine Berechnung sein, wie der folgende Unterabschnitt zeigt.

5.5.2 Berechnung der Feldstärke

Zur Berechnung der Feldstärke in einem Punkt wird üblicherweise der Abstand zwischen dem Zentrum des Feldes und dem angefragten Punkt benötigt. Neben unterschiedlichen Berechnungsvarianten für die Entfernung kann sich auch die Vorberechnung von Werten im Vergleich zur Laufzeit-Berechnung optimierend auswirken.

Entfernungen

Bei großen Entfernungen kann auf die Genauigkeit des euklidischen Abstandes verzichtet werden. Zur Berechnung dieses Abstandes müssen zunächst zwei Differenzen der Werte (x- und y-Koordinaten) jeweils quadriert und dann addiert werden. Das Ergebnis ist der Abstand im Quadrat (Satz des Pythagoras). Um den richtigen Abstand zu erhalten, wird eine

teure Wurzelberechnung nötig[25] (siehe Algorithmus 5.3). Alternativ bietet es sich an, den zu vergleichenden Abstand einmalig zu quadrieren und dessen Quadrat mit den Quadratdistanzen (ohne Wurzel) zu vergleichen, da üblicherweise sehr viele Vergleiche stattfinden.

Eine schnellere Berechnung für zumindest grobe Abschätzungen (z.B. weit außerhalb des Einflussgebietes), bietet die Manhattan-Distance oder auch Mannheimer-Entfernung[26]. Zur Berechnung dieser Entfernung müssen lediglich die Absolutwerte der Differenzen der Werte (x- und y-Koordinaten) addiert werden (siehe Algorithmus 5.4). Was im Vergleich zum euklidischen Abstand weniger Aufwand bedeutet.

Unter Umständen lassen sich die Formeln für die Felder auch derart gestalten, dass sie nicht auf dem euklidischen Abstand, sondern einem der anderen beiden Abstände basieren.

Algorithm 5.3 Euklidischer Abstand

```
// Differenzen bilden
deltaX = x1 - x2
deltaY = y1 - y2
// Quadrate addieren
distanceSquared = deltaX * deltaX + deltaY * deltaY
// Wurzel ziehen
distance = squareRoot(distanceSquared)
```

Algorithm 5.4 Manhattan-Distance

```
// Absolutwerte der Differenzen
absoluteX = absolute(x1 - x2)
absoluteY = absolute(y1 - y2)
// Quadrate addieren
manhattanDistance = absoluteX + absoluteY
```

Vorberechnung

Ein weiteres Mittel zur Optimierung, ist das Berechnen der Feldstärke beim Start der Anwendung (bzw. beim ersten Aufruf). Gleichartige Objekte besitzen die gleichen Felder und könnten sich ein statisches Feld teilen, aus dem die Werte bei Bedarf ausgelesen werden könnten. Das bringt jedoch den Nachteil der Ungenauigkeit mit sich. Normalerweise kann (je nach Implementierung) jeder Punkt abgefragt werden. Um die Datenmenge im Speicher

[25] Im späteren Verlauf der Arbeit wird gezeigt, dass die Programmiersprache Java in der aktuellen Version, ganz gut mit solchen Berechnungen umgehen kann.

[26] Sowohl Manhattan als auch Mannheim sehen (von oben betrachtet) wie eine rasterisierte Welt mit parallelen Straßen und quadratischen Häuserblöcken aus.

gering zu halten, müssten vorberechnete Felder aber in einer angemessenen Auflösung rasterisiert werden. Wie bereits erwähnt, zeigt die Implementierung in den kommenden Kapiteln, dass diese Form der Optimierung nicht wirklich einen Vorteil bringt.

Berechnung zur Laufzeit

Wie bereits erwähnt könnten die Formeln zur Berechnung auf weniger rechenintensive Abstandsverfahren aufsetzen. Diese Optimierung ist sehr sinnvoll.

Des Weiteren könnten angefragte Werte zwischengespeichert werden, falls sie in naher Zukunft erneut angefragt werden. Hierbei ist jedoch die Frage der Gültigkeit und Lebensdauer solcher Werte schwer zu beantworten. Noch dazu ist der Aufwand für die Verwaltung unter Umständen deutlich teurer als die Neuberechnung bei Anfrage.

Berechnung nur wenn nötig

Je nach Implementierung bzw. Optimierungsansatz (siehe Abschnitt 5.5.1 „statische Felder"), muss nicht immer der komplette Sichtbereich eines Agenten berechnet werden. Es genügt völlig die acht umliegenden Felder zu berechnen. Dieser Vorgang lässt sich noch weiter optimieren, wenn im nächsten Schritt dann nicht mehr acht Felder, sondern nur noch vier weitere Felder berechnet werden. Dazu merkt sich der Agent die Werte eines Durchgangs immer einen weiteren Durchgang lang. Die vier neuen Werte setzen sich aus dem Feld der letzten Position und den drei nicht an die letzte Position angrenzenden Felder zusammen[27].

5.5.3 Felder lernen

Wie bereits angesprochen, wäre es von Vorteil, wenn die Feldstärken nicht vom Programmierer „erraten" werden müssten. Sinnvollerweise lassen sich die richtigen Ausprägungen gut lernen. Das muss nicht zwingend wie oben angedacht ein selbstlernendes Verfahren (z.B. Reinforcement Learning in [RN04] Kapitel 21) sein, es kann auch durch überwachtes Lernen und Vorgabe von Richtwerten realisiert werden (vgl. [RN04] Kapitel 18).

In [CT04] wird detailliert ein geeignetes Lernverfahren in Bezug auf Felder behandelt. Außerdem interessant ist das Lernen so genannter Wegpunkte durch „Selbstorganisierende Karten" und „Neuronales Gas" (siehe [UL08] S. 261ff und S. 274ff).

[27] Es könnten auch nur drei neue Felder berechnet werden, jedoch müssten dazu im vorherigen Schritt nicht acht sondern neun Felder berechnet werden, sodass das keinen Unterschied macht.

5.5.4 Lokale Maxima vermeiden

Bei der Implementation für diese Arbeit hat sich herausgestellt, dass es für das Problem der lokalen Maxima noch keine „beste" Lösung gibt. Der Ansatz mit den Pheromon-Spuren ist auch nur bedingt optimal. Zudem ist ggf. die Verarbeitung des gesamten Sichtbereichs nötig, um, wie in Abschnitt 5.4.1, unerwünschte Lücken zu füllen.

Dadurch hat sich ein Konzept ergeben, welches so in der Literatur noch nicht gefunden werden konnte. Im Wesentlichen baut es ebenfalls auf der Idee mit den Ameisen-Pheromonen auf. Jedoch nicht als zu verwaltende Pheromon-Spur, sondern in Form eines neu erzeugten Objektes, einem „Pheromon-negativ-Marker".

Wenn der Agent in ein lokales Maximum geraten ist, platziert er an seiner aktuellen Position einen Marker, der ein zusätzliches Feld erzeugt. Dieses Feld macht die aktuelle Umgebung schlagartig unattraktiv, was den Agenten auf kürzestem Wege z.B. aus einer Lücke heraus treibt. Diese Reaktion ist eher vergleichbar mit der Abwehrreaktion eines Stinktiers, welches seinen Gegner mit einer Wolke aus Duftstoffen befeuert.

Ein Agent kann üblicherweise keine neuen Objekte in der Welt erzeugen. Daher muss er eine Liste der eigenen Marker führen. Da diese Liste unter Umständen sehr lang werden kann, könnte z.B. die Lebensdauer eines Markers beschränkt werden.

Je nach Anwendungsgebiet wären auch hier wieder unterschiedliche Nutzungsmöglichkeiten denkbar. Die Implementation in den folgenden Kapiteln zeigt zwei Anwendungen dieses Verfahrens, die zu gegebener Zeit näher erläutert werden.

Langfristiges Füllen von Lücken

Der Agent verzichtet aus Performancegründen auf die Verarbeitung des Sichtbereichs und nimmt Sackgassen in Kauf. Sobald er feststellt, dass er keinen nächsten Schritt findet und auch nicht an seinem Ziel angekommen ist, stellt er eine Sackgasse fest und setzt einen negativ-Marker. Je nach Stärke des Markers müssen unter Umständen mehrere gesetzt werden, bis er wieder frei ist und die Lücke geschlossen hat.

Diese Anwendung der Marker ist dazu bestimmt, die Lücke auf lange Sicht zu schließen, so dass er beim nächsten Mal nicht erneut in diese Lücke wandert. In einer dynamischen Welt, in der der Agent längere Zeit nicht in derselben Gegend gewesen ist, können diese Marker wieder verworfen werden. Somit kann der Speicher, den der Agent benötigt um die Marker zu verwalten, wieder freigegeben werden[28].

Ein Nachteil dieser Art von Markern ist, dass Bereiche oder ganze Regionen ggf. für die gesamte Spielzeit[29] fälschlicherweise gesperrt sind. Beispielsweise kann über einen un-

[28] In einem Strategiespiel mit gleichbleibendem Aktionsbereich und kommunizierenden Agenten wäre die Verwaltung dieser Marker durch einen Koordinator sinnvoll, damit die Agenten eines Teams nur ein einziges Mal in solch eine Lücke geraten müssen.

[29] In einer persistenten Spielwelt unter Umständen für immer.

passierbaren Fluss eine Brücke gebaut oder ein dicht gewachsener Wald abgeholzt worden sein.

Vorantreiben bei Inaktivität

Ein weiterer Einsatzgrund könnte Inaktivität sein. Im Kapitel 7 wird eine äußerst dynamische Welt beschrieben. In ihr leben Jäger, die Rehe jagen und Rehe, die Gras fressen. Gras wächst an zufälligen (also auch bereits besuchten) Orten in der Welt. Hat ein Reh das gesamte Gras in einem Bereich der Welt gefressen, bleibt es an der letzten Fundstelle stehen. Stellt das Reh nun fest, dass es längere Zeit nicht gefressen hat und auch kein Gras in Sicht ist, markiert es die Gegend mit einem ausreichend starken Feld, so dass es aus diesem Bereich heraus getrieben wird, in der Hoffnung neues Gras zu finden. Dieser Marker besitzt einen Timer, der das Feld im Laufe der Zeit so lange abschwächt, bis der Marker verworfen werden kann.

Gleiches gilt für den Jäger, der eine Zeit im Wald verharrt und auf Rehe wartet. Nach Ablauf dieser Zeit markiert er das Gebiet und treibt sich somit so lange zur nächsten Position, bis er ein Reh entdeckt hat.

5.6 Weiterführende Literatur

Die Konzepte und eigenen Ideen in diesem und den folgenden Kapiteln sind im großen Maße von den bereits erwähnten bzw. weiteren Quellen von Johan Hagelbäck und Stefan J. Johansson [HJ08a, HJ08b, HJ08c, HJ09, Hag09b] beeinflusst worden.

Zusätzlich empfiehlt es sich einen Blick auf die Arbeit von Johan Hagelbäck in [Hag09a] zu werfen. Dort werden sämtliche Konzepte erneut zusammengeführt und erläutert.

Kapitel 6

Statische Implementation

In diesem Kapitel soll das Konzept der Potentialfelder aus dem letzten Kapitel zunächst in einer statischen Welt umgesetzt werden. Im nächsten Kapitel wird dann der Unterschied zu einer Implementation für eine dynamische Welt, die sich häufig verändert, dargestellt.

6.1 Die Welt

Für die statische Implementation bietet sich ein kürzlich vom Autor entwickeltes Framework für die Entwicklung von NPCs[30] an. Das Framework trägt den Namen „CombatArena" und der erste veröffentlichte Prototyp heißt „TankArena" [Can]. Das Framework soll Studenten die KI-Spieler-Entwicklung näher bringen.

Das Szenario ist eine Welt mit einer quadratischen Grundfläche, Bergen und Wasser, in der sich bis zu vier Teams gleichzeitig bewegen können. Jedes Team hat eine Basis und fünf Panzer. Wird ein Panzer zerstört, wird er nach Ablauf einer Wartezeit in seiner Basis wieder zum Leben erweckt. Werden eine Basis oder Teile davon zerstört, können die Panzer nicht zurückkehren.

Ziel des Spiels ist es, die gegnerischen Basen zu finden[31] und zu zerstören, bevor die eigene Basis zerstört wird. Verlust eines Panzers (durch Wasser oder Zerstörung) wird bestraft. Ebenso das Zerstören der eigenen Basis bzw. eines eigenen Panzers. Hingegen wird die Zerstörung gegnerischer Panzer und Basen belohnt. Wird ein Panzer zerstört, erscheint an seiner Position eine Kiste mit Reparaturmaterial. Sammelt ein weiterer Panzer solch eine Kiste ein, werden seine Lebenspunkte wieder hergestellt. Das Team erhält ebenfalls eine Belohnung in Form von Punkten. Gewonnen hat, wer übrig bleibt.

[30] Non Player Characters

[31] Die Positionen der Basen sind vorgegeben und immer gleich. Die Teams werden jedoch per Zufall den Basen zugeteilt und haben jedes Mal einen neuen Ausgangspunkt. Bei weniger als vier Teams existieren Positionen ohne Basis.

Zu seiner Orientierung hat jeder Panzer Sensoren und Aktuatoren. Als Sensor dient eine Methode. Bei jedem Update der Animation Loop ([Dav05] S. 309) wird eine Liste aller im Sichtbereich befindlichen Objekte übergeben. Die roten Bereiche in Abbildung 6.1 bzw. die blaue Kugel in Abbildung 6.7 stellen den Sichtbereich eines Panzers dar. Die Update-Methode des Spielers wird ebenfalls in der Animation Loop aufgerufen. In dieser Methode kann er seine Entscheidungen treffen. Als Aktuatoren dienen ihm dabei unter anderem die Methoden „move(direction)" und „stop()". Da es hier lediglich um die Navigation und die Wahrnehmung der Welt geht, sind die restlichen Aktuatoren nicht weiter von Interesse. Weitere Informationen zu dem Framework finden sich in Anhang B.

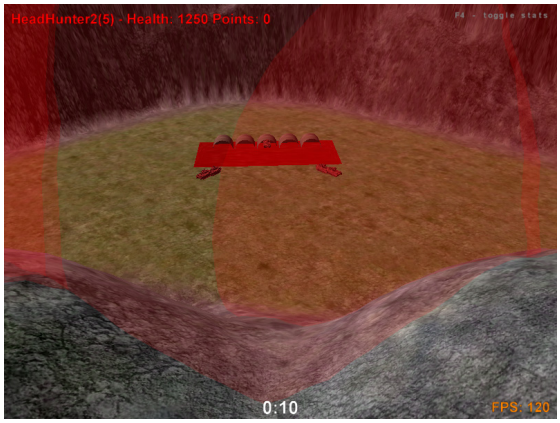

Abbildung 6.1: TankArena: Zu Beginn einer Runde. Vier Panzer machen sich auf den Weg, einer bewacht die Basis.

Abbildung 6.2: TankArena: Zwei Panzer nähern sich einem Gewässer.

6.2 Identifikation der Felder

Zunächst müssen die Felder samt ihrer Ausrichtungen identifiziert werden (vgl. [HJ08c] Kapitel 2).

1. Eigene Panzer: Der Einfachheit halber ignoriert das Framework die Kollisionen zwischen Objekten der Welt. Daher kann an dieser Stelle aus Performancegründen auf Felder bei eigenen Panzern verzichtet werden. Es sollte jedoch bedacht werden, dass zwei attackierte Panzer auf derselben Stelle auch beide Schaden nehmen. Daher wäre es taktisch klug, die Panzer mit einem starken, wenig strahlenden Feld auszustatten, so dass sie untereinander Abstand halten.

2. Generische Panzer: Die Panzer schießen in einer parabolförmigen Kurve mit Geschossen auf einander. Daraus ergeben sich optimale Schussweiten. Für Entfernungen größer der optimalen Schussweite, werden gegnerische Panzer mit einem positiven Feld belegt. Für die optimale Schussweite mit einem abnehmenden positiven Feld[32]. Falls er zu dicht an einen gegnerischen Panzer heran gerät, sorgt ein zunehmendes positives Feld dafür, dass er auf die Position des gegnerischen Panzers gezogen wird[33].

3. Materialkisten: Sie sind sehr wertvoll und erhalten daher ein positives Feld, damit ein Panzer die Kiste einsammelt. Befindet sich die Kiste im Wasser, wird sie ignoriert und erhält kein Feld. Die Stärke des Feldes ist geringer als die Felder eines gegnerischen Panzers, um die Priorität festzulegen.

4. Gegnerische Basis: Zu Beginn einer Runde ist weder bekannt, wie viele Teams existieren, noch wo sich die jeweilige Basis befindet. Da die Positionen auf jeder Karte dieselben sind, wird für jede Basis ein anziehendes Feld erzeugt, das über die gesamte Karte strahlt. Erreicht ein Panzer die Position und findet keine Basis vor, wird das Feld gelöscht. Wie in Punkt 2 dient auch hier ein weiteres Feld auf kurze Distanz dazu, die Panzer in optimale Schussweite zu bringen.

5. Eigene Basis: Die eigene Basis erhält kein Feld. Denkbar wäre es, einen der Panzer bei der Basis stehen zu lassen, sodass er gegnerische Panzer auf dem Weg zur Basis abfangen könnte. Dann wäre es möglich, über eine Finite State Machine (Endlicher Automat) den Zustand der Panzer in Reichweite auf „Verteidigung" zu stellen. Zu diesem Zweck würden die Felder der gegnerischen Basen deaktiviert und ein starkes anziehendes Feld für die eigene Basis generiert.

6. Landschaft: Die Panzer können sowohl an Bergen „hängen" bleiben, als auch im Wasser untergehen. Daher werden erkannte Berge und Wasser mit starken, wenig strahlenden Feldern ausgestattet, die die Panzer fern halten sollen.

6.3 Implementation der Felder

Nun, da die Felder identifiziert wurden, müssen sie implementiert werden.

Der Prototyp aus diesem Kapitel folgt den Ansätzen aus Kapitel 5. Die Werte der Felder werden vorberechnet und in zweidimensionalen Arrays gespeichert. Wie bereits erwähnt, bringt das eine Ungenauigkeit der Werte mit sich. Daher muss ein Mittelweg zwischen Menge der gespeicherten Daten (Speicherbedarf) und Genauigkeit gefunden werden. Tatsächlich

[32] Dadurch wird der Panzer immer in die optimale Schussweite gezogen.

[33] Das geschieht aus taktischen Gründen, in der Hoffnung der Gegner stellt entweder das Feuer ein oder zerstört sich gleich mit, wodurch er doppelt bestraft würde.

bietet sich sogar eine Kombination verschiedener Auflösungen je nach Feldart an. Für den Prototyp haben alle Felder eine Auflösung von 1:4[34]. Das Resultat wird am Ende des Kapitels besprochen.

6.3.1 Landschaft

Die Landschaft wird zur Laufzeit, durch die Wahrnehmung der Panzer, erweitert. Erkannte Hindernisse werden mit ihren Typen in ein Array geschrieben. Es wird unterschieden zwischen „TERRAIN", „WATER" und „MOUNTAIN". Diese Informationen dienen zur Neuberechnung der Felder im nächsten Unterabschnitt.

6.3.2 Trieb

Dieses Feld enthält Informationen über die Felder der Welt. Wird beispielsweise ein Hindernis erkannt, wird in diesem Array das Feld, welches dem Hindernis zugeordnet wird, eingetragen. Des Weiteren werden hier die den Basen zugeordneten Felder verzeichnet. Ändert sich die Erkenntnis über die Welt, z.B. da eine Position untersucht wurde, die jedoch anders als erwartet keine Basis enthielt, wird das Feld aktualisiert. Diese Aktualisierung findet nur selten (über die Dauer des Spieles maximal dreimal) statt. Ist eine Aktualisierung von Nöten, werden die Informationen über die Landschaft aus dem Array des vorherigen Unterabschnitts verwandt.

6.3.3 Pheromone

In diesem Feld werden die Pheromon-negativ-Marker aus Abschnitt 5.5.4 eingetragen. In dieser Implementation verlieren Marker nie ihre Gültigkeit, daher werden die Werte einmalig berechnet und fest eingetragen.

Stößt ein Agent auf ein lokales Maximum, wird ein Marker gesetzt. Je nach Landschaft und Größe der Lücken in der Landschaft (siehe Abschnitt 5.4.1) sollten die von den Markern gesetzten Felder angepasst werden. Der Prototyp benutzt gleichbleibende Feldstärken. Als Resultat sind Agenten unter Umständen länger damit beschäftigt, aus Sackgassen zu entkommen. Auf der anderen Seite würden größere Felder an eigentlich günstigen Positionen die Agenten unnötig ablenken.

Eine Verbesserung würde erzielt, wenn der Agent in Abhängigkeit der verstrichenen Zeit seit dem letzten Marker, dynamisch stärkere Marker setzen würde. Bei Verstreichen einer längeren Zeit zwischen zwei Markern würde der Agent dann wieder schwache Marker setzen.

[34] Also entspricht eine Fläche mit vier mal vier Punkten einem Wert im Array.

6.3.4 Sonstiges

Die Felder der Panzer und Kisten sind sehr flüchtig. Die Wahrnehmung der Objekte im Sichtbereich ist eine Momentaufnahme und kann sich beim nächsten Update komplett verändern, wenn der Panzer sich z.B. dreht und die Objekte nicht mehr im Sichtbereich liegen.

Zu diesem Zweck gibt es ein weiteres Array, in das jeder Panzer seine aktuelle Wahrnehmung einträgt. Wird ein Objekt zweimal gesehen, wird der zweite Eintrag verworfen. Beim Eintragen der Objekte in die Karte, werden deren Felder gleich mit eingetragen. Um die Berechnung zur Laufzeit zu vermeiden, werden einmalig (zu Beginn der Spielrunde) zwei weitere Arrays angelegt. Das eine enthält das von Panzern erzeugte Feld, das andere das Feld für Materialkisten. Die Arrays entsprechen in ihrer Größe der doppelten Reichweite des Feldes des jeweiligen Objektes.

Wird nun ein Objekt erkannt, werden die Werte für die umliegenden Felder im Array aus den Objekt-Arrays gelesen und in das temporäre Array eingetragen. Die Angaben in diesem Array werden nach jeder Runde (also nach dem einmaligen Update jedes Panzers) verworfen und neu ermittelt.

6.4 Der Prototyp

Wie bereits erwähnt, lag bei der Implementation die Priorität in der Navigation. Zu Beginn einer Runde werden die im vorherigen Abschnitt beschriebenen Karten aufgebaut.

6.4.1 Navigation

Die Basen links und rechts der Startposition liegen gleich weit, die letzte Basis etwas weiter entfernt[35]. Die Panzer starten nebeneinander. Die rechten Beiden liegen somit stärker im Einzugsgebiet der rechten Basis, die linken Beiden zieht es zur linken Basis. Der mittlere Panzer bleibt unter anderem zur Verteidigung in der Basis stehen[36]. In Abbildung 6.3 ist diese Situation zu sehen[37]. Die blauen Punkte repräsentieren die Panzer. Der obere Teil stellt die Potentialfelder dar. Die grünen Kreise in den Ecken sind die möglichen Positionen der Basen, der rote Bereich ist als unpassierbar erkannt worden. Der untere Teil der Abbildung zeigt die von den Agenten aufgedeckte Karte. Die schwarzen Punkte stehen für die eigenen Panzer. Gelbe Bereiche sind Berge, blaue Bereiche stehen für Wasser.

Der mittlere, in der Basis verbliebene Panzer übernimmt eine weitere Aufgabe. Er durchläuft den aufgedeckten Bereich der Welt anhand der gesammelten Daten und versucht potentielle Lücken ausfindig zu machen. Dazu wendet er die Methoden aus Abschnitt 5.4.1

[35] Um den Faktor Wurzel 2 um genau zu sein.

[36] Eine bessere Taktik wäre das Patrouillieren vor der Basis, um Gegner früher zu erkennen. Außerdem nimmt bei einem in der Basis stehenden Panzer, der angegriffen wird, die Basis ebenfalls Schaden.

[37] Abbildung 6.1 zeigt die Situation in 3D.

an. Die gefundenen Lücken werden ebenfalls als unpassierbar in die Karten mit eingebaut. Dafür stehen die roten Punkte im unteren Bereich der Abbildungen 6.3 und 6.4.

In Abbildung 6.4 sind zwei Panzer auf einen gegnerischen Panzer gestoßen. Auf der rechten Seite der Abbildung (oben) ist der grüne Kreis zu sehen, der die optimale Schussweite repräsentiert. Der rote Punkt in der Mitte ist der gegnerische Panzer selbst. Außerdem ist zu sehen, dass das positive Feld zum Gegner hin stärker wird, so dass der eigene Panzer angezogen würde (siehe Abschnitt 2). Im unteren Bereich der Abbildung ist die bereits weiter aufgedeckte Welt zu sehen.

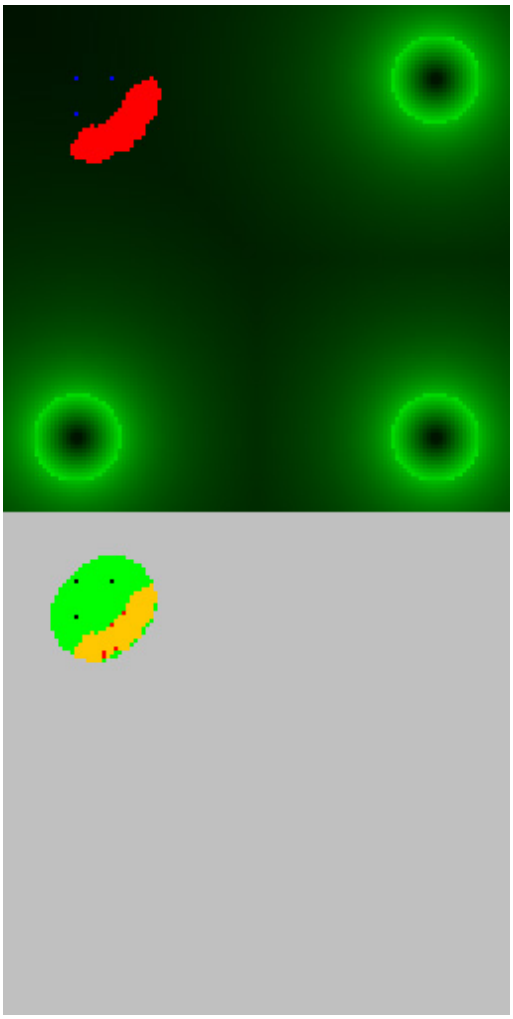

Abbildung 6.3: Sicht der Agenten zu Beginn einer Runde (Start oben links).

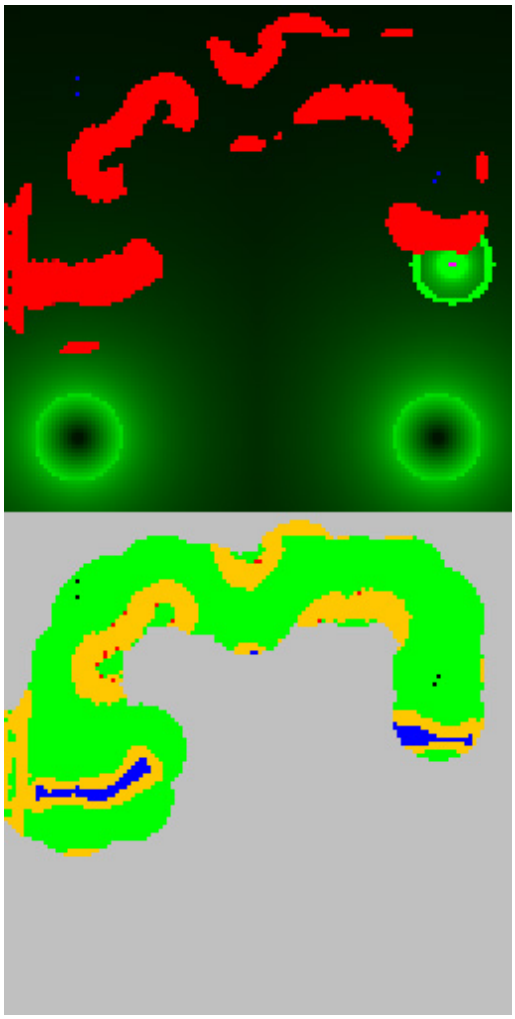

Abbildung 6.4: Zwei Agenten sind auf einen gegnerischen Panzer gestoßen.

6.4.2 Pheromon-negativ-Marker im Einsatz

Abbildung 6.5 und 6.6 zeigen eine andere Welt. Dieses Mal gibt es nur einen Panzer, mit der Startposition unten rechts. Im Vergleich der beiden Teile der Abbildung 6.5 ist zu sehen, wie dem Wasser negative Felder zugeordnet werden.

Im Zentrum des gelben Quadrates befindet sich ein Panzer. Er steuert direkt auf das Wasser und somit eine Blockade zu (siehe Abschnitt 5.4.1). Abbildung 6.6 zeigt, dass der Panzer mit Hilfe eines Markers die Blockade aufgelöst hat. Würde er sich das nächste Mal wieder auf diesem Weg durch die Landschaft bewegen, würde er direkt in die richtige Richtung abgelenkt.

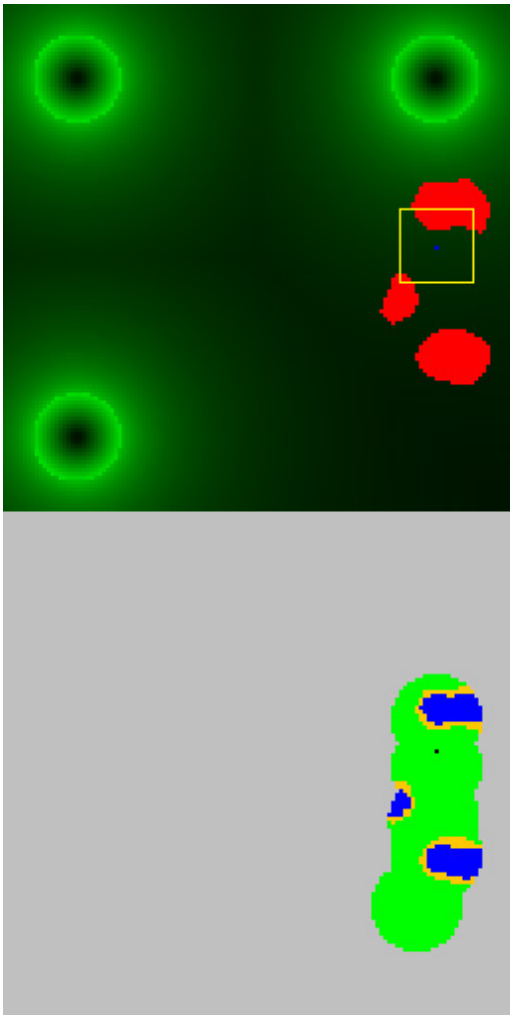

Abbildung 6.5: Ein Agent im Spiel (Start unten rechts).

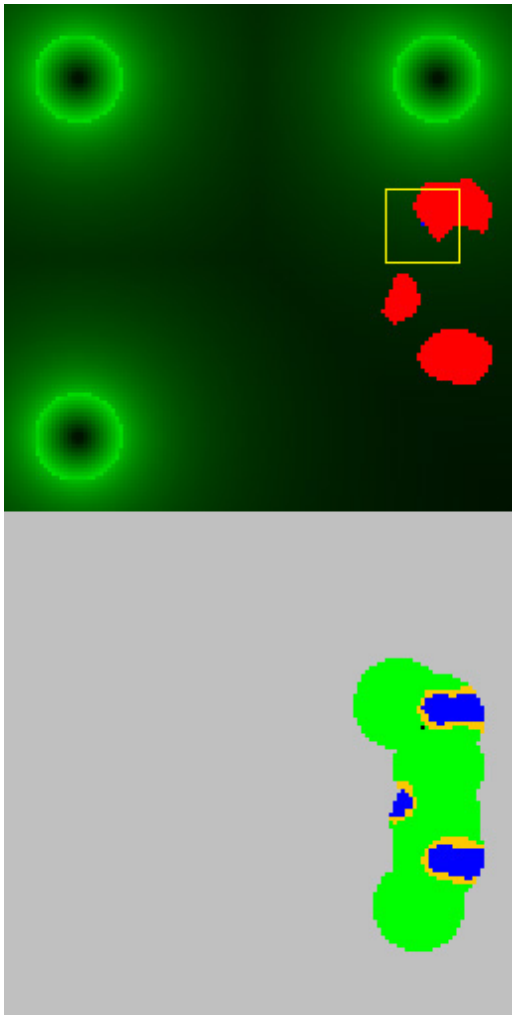

Abbildung 6.6: Lücke durch Marker geschlossen.

Abbildung 6.7 zeigt dieselbe Situation in 3D. Der blaue Panzer bewegt sich um das Hindernis herum[38].

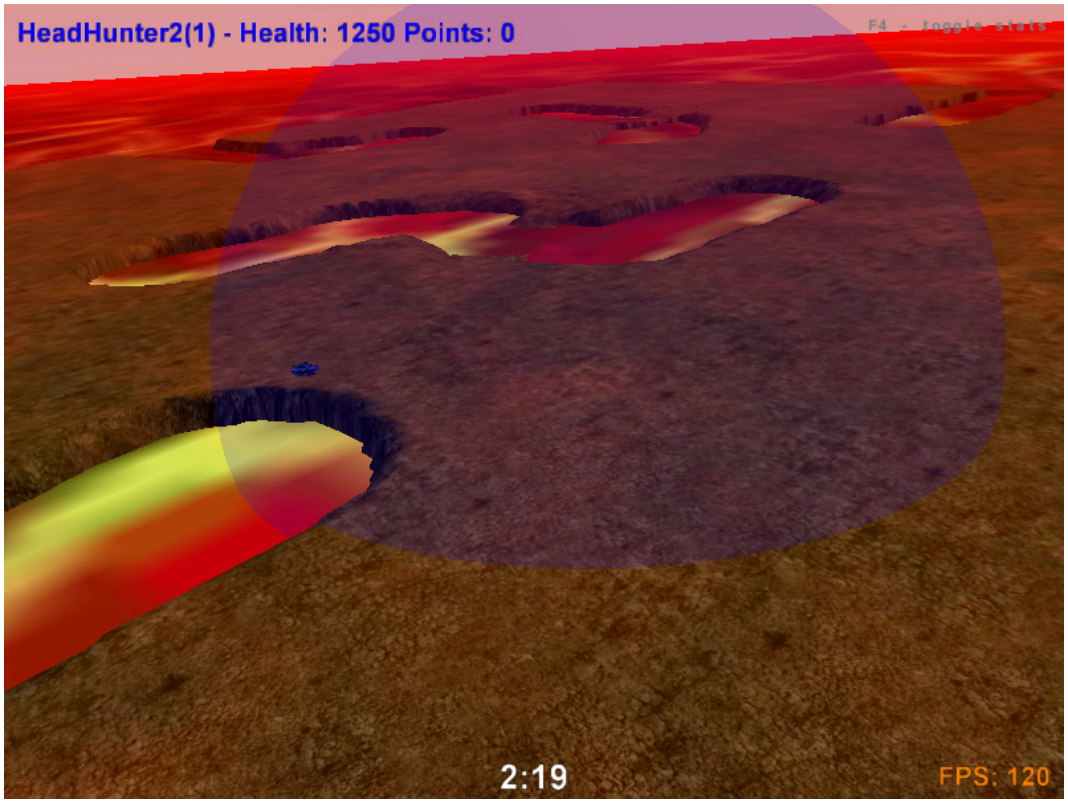

Abbildung 6.7: TankArena: Ein Panzer bewegt sich um unpassierbares Gelände herum.

6.5 Zusammenfassung

Insgesamt läuft der Prototyp sehr flüssig. Im Vergleich zu anderen KI-Spielern, die allesamt auf Wegfinde-Algorithmen basieren, steht der Prototyp sogar recht gut da. Er nahm zwar an keinem Wettbewerb teil[39], unabhängige Tests zeigten jedoch, dass er oftmals bessere Ergebnisse in Bezug auf die Landschaftserkennung und Navigation liefert. Da der Prototyp nur die Navigation bzw. das Fluchtverhalten implementiert und sich bei einem Angriff lediglich verteidigt, stehen seine Chancen im offenen Kampf eher schlecht.

Der große Vorteil der Potentialfelder, dass auf den Einsatz von Such-Algorithmen verzichtet werden kann, äußert sich sehr positiv beim Vergleich der Laufzeiten. Andere KIs

[38] In diesem Fall handelt es sich nicht um Wasser, sondern um Lava. Jedoch ändert das nichts am Verhalten des Agenten, es ist lediglich eine alternative Darstellung.

[39] Im Praktikum traten die KI-Spieler der einzelnen Gruppen gegeneinander an.

brauchten gerade im späteren Verlauf einer Runde deutlich länger. Zu diesem Zeitpunkt ist bereits der Großteil der Karte aufgedeckt und der Suchraum entsprechend groß. Die KI mit den Potentialfeldern hingegen, weist einen durchgehend gleichbleibenden, niedrigen Zeitbedarf auf. Ähnliches gilt für den Speicherbedarf. Zwar benötigen die Potentialfelder gleich zu Beginn etwas mehr Platz, dafür steigt der Bedarf nicht weiter an, so dass nach wenigen Minuten der Speicherbedarf ausgeglichen ist. Zum Ende einer Spielrunde, liegt der Bedarf der anderen KIs sogar deutlich höher.

Würde weiterer Aufwand in das Feintuning der einzelnen Felder gesteckt und die Angriffsstrategie verbessert[40], hätte der Prototyp gute Chancen einen Wettbewerb als Sieger zu verlassen.

[40] bzw. implementiert

Kapitel 7

Dynamische Implementation

In diesem Kapitel soll auf Grundlage der Informationen und Erkenntnisse der vergangenen Kapitel die Implementation der Potentialfelder in einer dynamischen Welt erfolgen.

Das Vorgehen folgt dabei der gleichen Struktur wie schon in Kapitel 6 bei der statischen Implementation.

7.1 Die Welt

Bei der Welt, in der sich die Agenten dieser Anwendung befinden, handelt es sich um die mehrfach angesprochene „Hunters and Deers"-Welt (siehe Kapitel 1.1.2).

Genauer betrachtet, handelt es sich um einen Wald mit Bäumen und Gras. Außerdem gibt es in der Welt noch Rehe, die Gras fressen und Jäger, die Rehe jagen. Um die Landschaft in ihrer Dimension kompakt zu halten, ist der Wald zusätzlich rund herum von Wasser umgeben. Somit ist die Welt eine Insel.[41]

Für die Umsetzung dieser Arbeit wurde größtenteils mit dem in Abschnitt 2.1.2 erwähnten Observer gearbeitet. Abbildung 7.1 zeigt einen Ausschnitt aus der Welt. Darin sind ein Reh (rechts) und ein Jäger (links) zu sehen. Die gelben Kreise[42] repräsentieren die Sichtbereiche, die roten Kreise die Angriffs-Reichweiten. Die Sichtweite der Rehe beträgt 400, die der Jäger 600 Einheiten. Die Angriffs- bzw. Schussreichweite der Jäger beträgt 300 Einheiten. Beim Reh entspricht der rote Kreis ebenfalls der Reichweite in der es Gras fressen kann. Die dunkelgrünen Objekte sind Bäume, die hellgrün schraffierten sind Grasflächen. Am linken Rand sind eine Küstenlinie der Insel und dahinter Wasser zu sehen. Sowohl Rehe als auch Jäger meiden das Wasser.

Aus der Abbildung geht hervor, dass zwar der Jäger das Reh sieht, aber nicht umgekehrt. Daher nähert sich der Jäger dem Reh bis er in Schussreichweite ist. Da der Sichtbereich des

[41] Die Nutzung der in diesem Kapitel entwickelten Agenten in einer räumlich nicht begrenzten Welt ist ebenfalls möglich. Die Welt ist nur der Übersicht halber beschränkt worden.

[42] In der Abbildung sind die Kreise nur teilweise als runde Linien zu sehen.

Rehs größer ist als die Schussreichweite des Jägers, wird das Reh rechtzeitig zu flüchten beginnen.

Weitergehende Information, insbesondere zur Implementation des Gameserver-Prototyps, befinden sich in Anhang A.

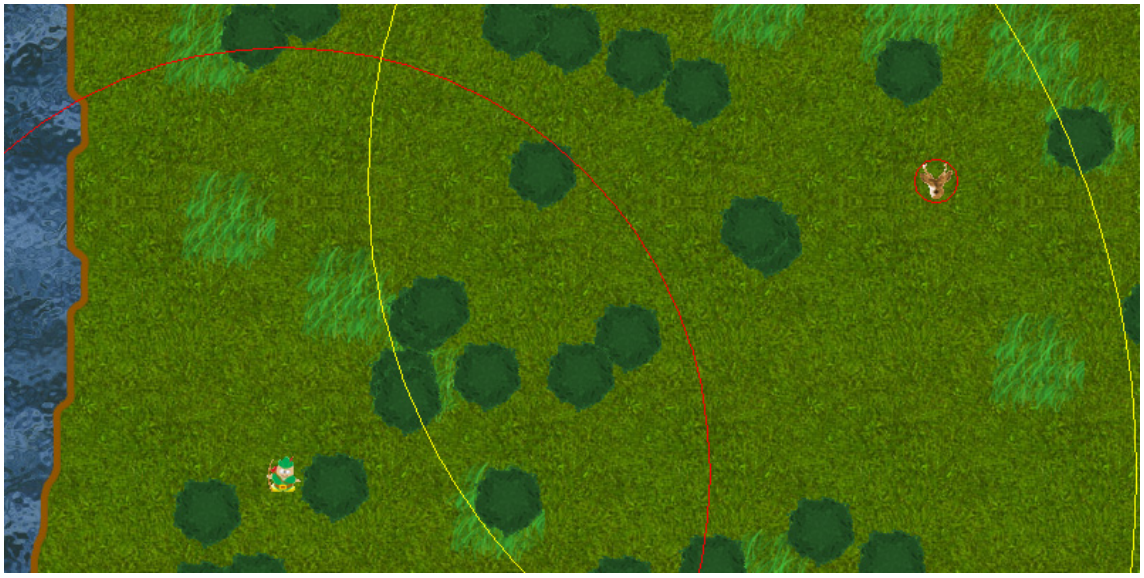

Abbildung 7.1: Ausschnitt der „Hunters and Deers”-Welt im 2D Observer.

7.2 Identifikation der Felder

Zunächst werden alle Felder samt ihrer Ausrichtungen identifiziert. In diesem Fall muss zwischen den Feldern, die die Jäger nutzen, und den Feldern, die die Rehe nutzen, unterschieden werden.

Hinweis zu den Abbildungen: Rote Felder haben eine abstoßende, grüne Felder eine anziehende Wirkung. Der gelbe Punkt in der Mitte ist der jeweilige Agent. Der gelbe Kreis stellt den Sichtbereich des Agenten dar.

Zunächst die gemeinsamen Felder:

1. Bäume: Für alle Agenten gilt, dass Bäume nicht passierbar sind. Daher werden alle Bäume im Sichtbereich mit einem stark negativen Feld mit kurzer Reichweite belegt. Die roten Punkte in den Abbildungen 7.2, 7.4 und 7.5 sind die Bäume aus Sicht der Agenten.

2. Landschaft: Die Beispielwelt besteht der Einfachheit halber aus einer flachen Insel. Aus diesem Grund gibt es zwar keine Berge, aber eine Küstenlinie und Wasser. Diese

Bereiche werden mit Feldern maximaler Stärke und sehr kurzer Reichweite belegt. In Abbildung 7.2 ist die Küstenlinie (rot) am unteren Rand zu sehen.

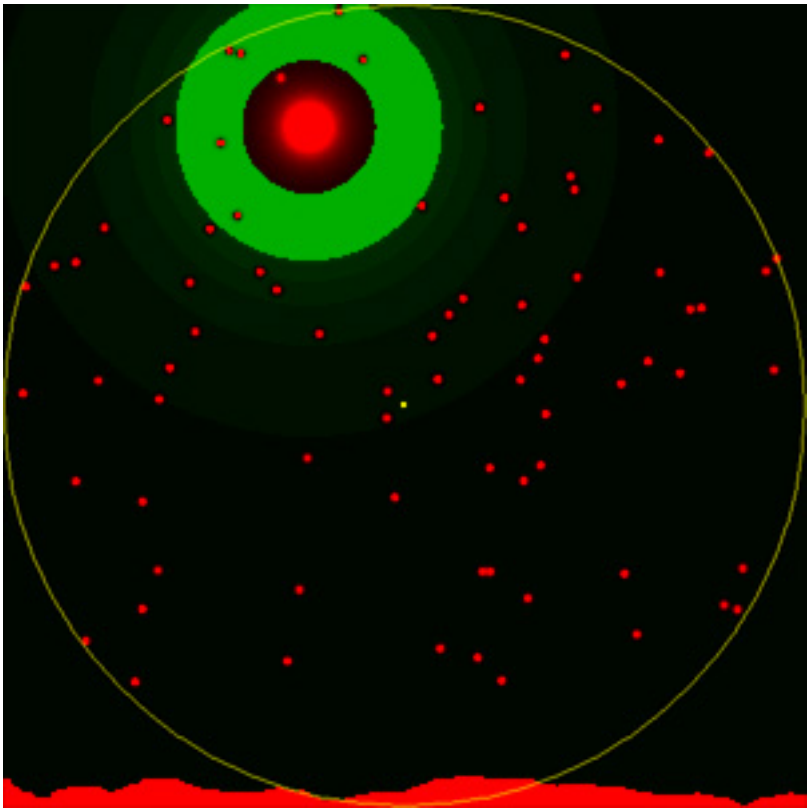

Abbildung 7.2: Sicht eines Jägers: Reh am oberen Rand.

Als nächstes die Felder der Jäger:

1. Rehe (lebendig): Die Jäger wollen von den Rehen angezogen werden. Daher belegen sie ihre Beute mit einem anziehenden Feld. Da sie jedoch nicht unnötig nah an das Reh heran müssen, wird ein Bereich zwischen 100 und 200 Einheiten von der Mitte des Rehs entfernt mit einem gleichbleibenden Feld belegt. Dieser Bereich entspricht der optimalen Schussweite. Der Bereich zwischen der optimalen Schussweite und dem Reh wird mit einem abstoßenden Feld ausgefüllt, sodass der Jäger in der optimalen Schussweite bleibt. Abbildung 7.2 zeigt das beschriebene Feld am oberen Rand.

2. Rehe (tod): Hat ein Jäger ein Reh erlegt, muss er es natürlich noch einsammeln. Dazu werden tote Rehe mit stark anziehenden Feldern belegt. Die Felder wirken sich stärker als die Felder für lebendige Rehe aus, damit die Jäger zunächst die bereits erlegten Rehe einsammeln, bevor sie weiter jagen. Abbildung 7.3 zeigt ein totes Reh im Sichtbereich eines Jägers.

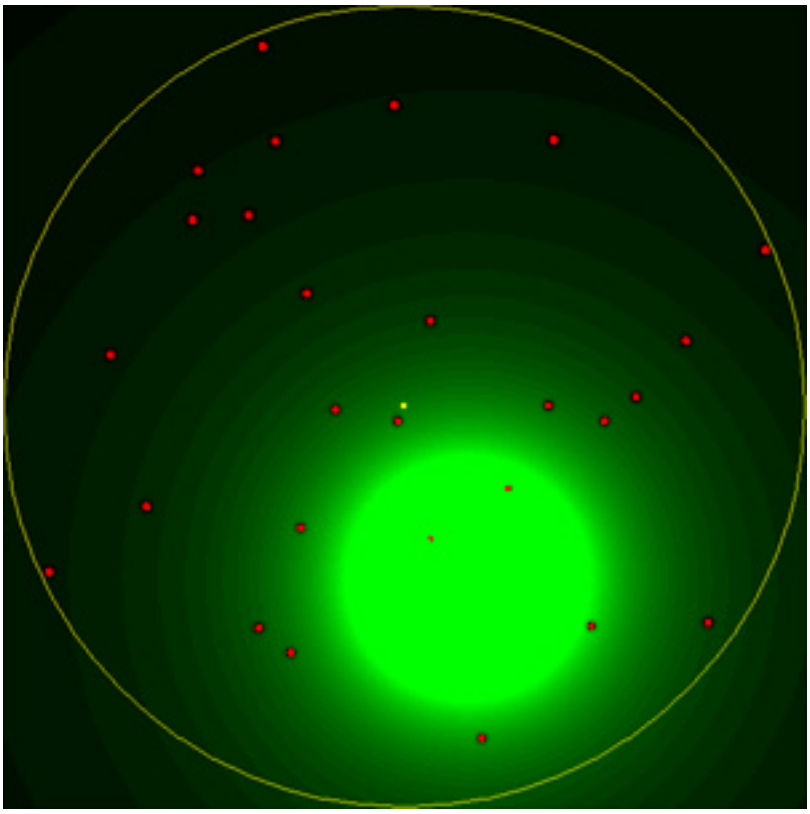

Abbildung 7.3: Sicht eines Jägers: Totes Reh.

3. Jäger: Da es sich in der Gruppe besser jagen lässt, werden anderen Jägern umgekehrte, negative Felder zugeordnet. Diese haben nicht die Absicht die Jäger zu einander zu treiben. Sie sollen vielmehr die Gruppe zusammen halten, indem der Agent in immer negativere Bereiche kommt, je weiter er sich von einem anderen Jäger entfernt. Damit die Jäger jedoch nicht zu dicht zusammen stehen, ist der Bereich von 5 bis 25 Einheiten um andere Jäger als neutral durch den Wert 0 definiert. Da Jäger nicht an derselben Stelle stehen dürfen, wird der Bereich 5 Einheiten um einen anderen Jäger als stark negatives Feld definiert.

4. Gras: Jäger haben in dieser Implementation kein Interesse an Gras und somit auch kein Feld für diese Objekte.

Fehlen noch die Felder der Rehe:

1. Rehe: Auch Rehe wollen sich in Gruppen bewegen.[43] Dazu verwenden sie für andere Rehe ein Feld, welches dem Feld der Jäger für ihre Artgenossen entspricht.

[43] Das mindert unter anderem die Gefahr von einem Jäger erschossen zu werden, da mehr Ziele zur Auswahl stehen.

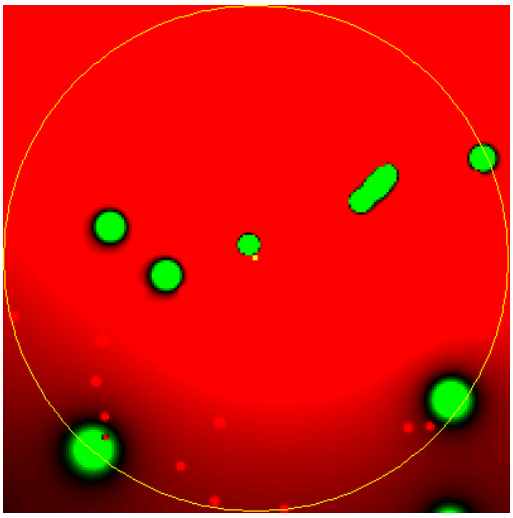

Abbildung 7.4: Sicht eines Rehs: Jäger am oberen Rand.

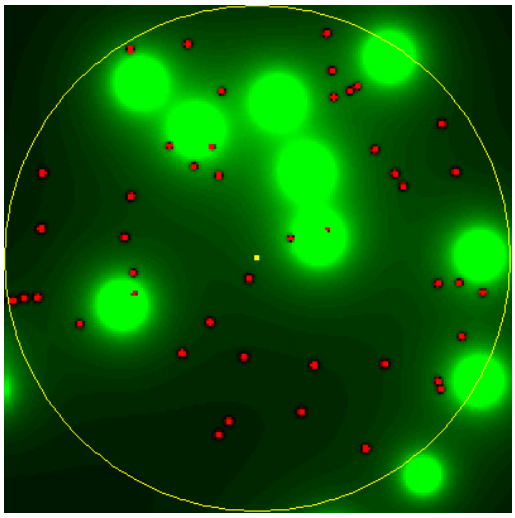

Abbildung 7.5: Sicht eines Rehs: Gras und Bäume.

2. Jäger: Da Rehe vor Jägern Angst haben[44], werden diese mit einem stark negativen Feld belegt, welches die Reh-Agenten zur Flucht veranlasst. Abbildung 7.4 zeigt am oberen Rand einen Jäger im Sichtbereich eines Rehs. Die grünen Bereiche sind Grasflächen, die das Reh auf der Flucht vor dem Jäger fressen kann, um seine Lebensenergie wieder aufzufüllen.

3. Gras: Jeder Grasfläche, die dem Agenten bekannt ist[45], wird ein anziehendes Feld zugeordnet. In Abbildung 7.5 ist die Sicht eines Rehs mit zwölf Grasflächen zu sehen. Außerhalb des Sichtbereichs (gelber Kreis) sind ebenfalls Grasflächen eingetragen. Die Felder der einzelnen Grasflächen addieren sich, sodass der Agent zunächst in Richtung der meisten Grasflächen und dann gezielt auf eine spezielle Grasfläche zusteuert.

7.3 Implementation der Felder

Im Gegensatz zu der statischen Implementation in Kapitel 6 werden alle Werte zur Laufzeit berechnet und nicht zwischengespeichert. Das hat den Vorteil, dass die Agenten einen geringen Speicherbedarf haben.

[44] Es hätte auch so modelliert werden können, dass Rehe vor allem Angst haben, was fremd ist, jedoch bietet der Prototyp des Gameservers keine weiteren Objekte.

[45] Die Agenten merken sich die Positionen der Grasflächen, auch wenn sie den Bereich verlassen haben.

Die Kostenersparnis beim Speicherbedarf geht zu Lasten der Rechenleistung. Wie bereits angedeutet, ist das jedoch bei der aktuellen Java-Version kein Problem mehr. Die notwendigen Berechnungen[46] werden zügig abgearbeitet. Zum Vergleich wurde testweise eine Implementation der Felder unter Zuhilfenahme der Optimierung mittels Manhattan-Distance (siehe Abschnitt 5.5.2) durchgeführt. Dazu wurde mit deutlich mehr Objekten getestet als in der Welt realistisch ist.[47] Als Ergebnis dieses Tests war kein messbarer Unterschied festzustellen.

Da die Agenten außerdem pro Berechnung[48] lediglich neun Werte ermitteln müssen, kann mit Ausnahme des Punktes „Berechnung nur wenn nötig" auf die Optimierungen aus den Abschnitten 5.5.1 und 5.5.2 verzichtet werden.

7.4 Der Prototyp

Als nächstes werden die interessanten Aspekte bei der Implementation des Prototyps besprochen.

7.4.1 Grundsätzliches

Die Implementation der Potentialfelder ist komplett unabhängig vom Gameserver. Für die Nutzung mussten keine speziellen Veränderungen vorgenommen werden. Dem Gameserver erscheinen die Agenten wie jeder andere Client auch.

Jegliche Verwaltung von Informationen zu den Potentialfeldern geschieht in der Agenten-Anwendung. Obwohl die Agenten derselben Laufzeit-Umgebung theoretisch Zugriff auf alle anderen Agenten haben, tauschen sie keine Informationen über Felder oder ähnliches aus. Die Agenten teilen sich lediglich aus Performancegründen dieselbe Umgebungs-Instanz (siehe Abschnitt 4.1.4).

7.4.2 Die Klasse „WorldObject"

Die Agenten kennen eine Klasse WorldObject. Jedes vom Server beschriebene Objekt in der Welt erzeugt eine Instanz dieser Klasse. Innerhalb dieser Klasse gibt es eine Methode mit dem Namen „getDesire". Mit Hilfe dieser Methode kann der Agent unter Angabe einer Position und seines eigenen Typs ermitteln, welchen Wert das Feld dieses Objektes für ihn hat.

[46] Auch die Berechnung einer Wurzel verursacht keine großen Kosten.

[47] Eine realistische Anzahl an Objekten, die gleichzeitig im Sichtbereich liegen, liegt zwischen 10 und 40. Die Vergleichstests wurden mit 1.000, 5.000 und 10.000 Objekten durchgeführt.

[48] Diese Berechnung wird nur etwa alle 2 Sekunden durchgeführt.

7.4.3 Die Klasse „Vision”

Die bereits in Abschnitt 4.2.1 angesprochene Klasse Vision liefert einem Agenten alle Objekte in seinem Sichtbereich. Sie stellt den Ausgleich einer Unzulänglichkeit des Gameserver-Prototyps dar. Normalerweise würde dieser nämlich jedem Agenten die Objekte in dessen Sichtbereich nennen.

Bei der Aktualisierung der Liste werden gleichzeitig die ebenfalls in Abschnitt 4.2.1 genannten Beliefsets „enemies”, „friends”, „prey” und „food” aktualisiert. Verlässt ein Objekt den Sichtbereich eines Agenten, wird an dessen Stelle ein „Dummy”-Objekt in die Beliefsets aufgenommen. Diese Objekte enthalten sämtliche unveränderlichen Informationen[49] des Ursprungs-Objekts. Zusätzlich werden die veränderlichen Werte[50] gespeichert. Somit merkt sich der Agent gewissermaßen die letzte bekannte Position dieses Objektes. Also muss z.B. ein Reh erst wieder in Sichtweite einer gemerkten Grasfläche gelangen, um zu erfahren, ob sie bereits von einem anderen Reh gefressen wurde.

7.4.4 Pheromon-negativ-Marker

Die im Abschnitt 5.5.4 eingeführten Pheromon-negativ-Marker werden von den Agenten dieser Anwendung in zwei Fällen genutzt:

Verlassen lokaler Maxima

Die Jäger werden lediglich von Rehen bzw. anderen Jägern beeinflusst.[51] Diese kommen jedoch im Vergleich zu Grasflächen nur selten vor. Daher würden die Jäger-Agenten an einem Ort stehen bleiben, bis zufällig ein Reh oder ein anderer Jäger vorbeikommt.

Die Position eines solchen Agenten kann als lokales Maximum (siehe Abschnitt 5.4.1) aufgefasst werden. Zur Lösung dieses Problems wurden die Pheromon-negativ-Marker entwickelt. Stellt nun ein Agent fest, dass er sich in einem lokalen Maximum befindet, erzeugt er an Ort und Stelle einen solchen Marker. Der Marker veranlasst den Agenten in eine zufällige Richtung zu flüchten.[52]

Nach Ablauf einer vorgegebenen Zeit (25 Sekunden) verliert der Marker seine Gültigkeit und wird entfernt. Ist der Agent mit seiner neuen Position ähnlich unzufrieden, wiederholt er den Vorgang solange, bis er ein Reh oder einen anderen Jäger findet.

[49] z.B. ein Identifier oder der Typ

[50] Dazu gehören die momentane Position und der Status (also lebendig oder tod).

[51] Die Auswirkung der Felder von Bäumen wird in diesem Beispiel wegen Geringfügigkeit außer Acht gelassen.

[52] In Abschnitt 7.5 wird beschrieben, wieso das so ist.

Flucht vor Gegnern

Da es in der Welt anfangs viele Grasflächen gibt, laufen Rehe selten in lokale Maxima.[53] Zusätzlich zu der Anwendung im vorherigen Unterabschnitt nutzen sie die Marker jedoch auch bei Feindkontakt. Sieht ein Reh einen Jäger, wird es sofort vor ihm flüchten. Sobald der Jäger den Sichtbereich des Rehs verlässt, nimmt das Reh den Jäger jedoch sehr schnell nicht mehr als Bedrohung war. Damit der Reh-Agent dem Jäger nicht direkt wieder entgegen läuft, werden auf der Flucht in Abständen von 5 Sekunden Marker gesetzt. Diese sorgen dafür, dass das Reh eine Chance hat den Sichtbereich des Jägers zu verlassen.

7.5 Implementation des „Walking"-Plans

In Kapitel 4 wurde vorerst auf eine ausführliche Beschreibung des „Walking"-Plans verzichtet. Da nun das erforderliche Wissen über die Welt und das Konzept der Potentialfelder bekannt ist, soll dies an dieser Stelle nachgeholt werden.

Theoretisches Vorgehen

Zunächst ermittelt der Agent die Liste der Objekte im Sichtbereich. Dazu befragt er seine Instanz der in Abschnitt 7.4.3 beschriebenen Klasse. Anhand dieser Liste werden dann die acht Bereiche um den Agenten herum auf ihren Wert hin untersucht. Findet der Agent einen maximalen Wert, ist dieser Bereich sein nächstes Ziel. Sollte er mehrere Bereiche mit dem gleichen Wert finden, entscheidet er per Zufall wohin er als nächstes geht. Die Aufteilung in Bereiche geschieht wiederum durch Rasterisierung.

Im vorherigen Kapitel wurden die Positionen der Agenten rasterisiert. Das heißt die Agenten haben sich immer von einem Rasterfeld zum nächsten bewegt. Das wäre vergleichbar mit einem Menschen, der einen Gehweg entlang geht und dabei immer nur auf den Mittelpunkt einer Gehwegplatte tritt, was jedoch wenig realistisch aussehen würde.

Die Agenten in diesem Kapitel legen das Raster hingegen über ihre aktuelle Position. Dadurch bewegt sich der Agent zwar immer noch in dem Muster fort, jedoch tritt er jetzt quasi auf die Kanten der Gehwegplatten. Hat der Agent die nächste Position ermittelt, wird die gewünschte Bewegung an den Gameserver übertragen. Noch bevor der Agent diese Position erreicht hat, wird die nächste Position ermittelt und übertragen. Das Ergebnis dieses Vorgangs ist eine flüssigere Bewegung.

Implementation

Der Algorithmus in 7.1 zeigt die Rasterisierung des Agenten. In Zeile 2 wird der Skalierungsfaktor festgelegt. Dieser gibt die Entfernung zwischen den Positionen an. Während in den

[53] Es sei denn es gibt so viele Rehe, dass die Grasflächen nicht schnell genug nachwachsen können.

Zeilen 3 und 4 die obere linke Ecke[54] als erste Position festgelegt wird, wird in den Zeilen 5 und 6 die untere rechte Ecke als die letzte Position bestimmt.

Die Variable „temporaryPosition" ist ein zweidimensionaler Vektor mit Werten für x und y.[55] In Zeile 8 wird der x-Wert der Variablen initialisiert. Der Bereich von Zeile 9 bis 22 wird solange durchlaufen, bis der x-Wert dem in Zeile 5 berechneten Maximalwert entspricht (genau drei Mal). In Zeile 10 wird in jedem Durchlauf der äußeren Schleife der y-Wert der Variablen „temporaryPosition" gesetzt. Die Schleife von Zeile 11 bis 20 wird solange durchlaufen, bis der y-Wert dem in Zeile 6 berechneten Maximalwert entspricht, also drei Mal pro Durchlauf der äußeren Schleife. Durch die Verschachtelung wird diese Schleife insgesamt neun Mal durchlaufen.

Innerhalb der inneren Schleife wird der Wert der jeweiligen Position ermittelt. Zu Beginn hat „desire" den Wert 0. Es wird geprüft ob der neu ermittelte Wert größer als der bisherige ist oder ob er gleich dem bisherigen Wert ist, dann spielt auch der Zufall eine Rolle. Die Variable „zufall" besitzt bei jedem Vergleich einen zufälligen Wahrheitswert. Trifft eine der beiden Bedingungen zu, wird die momentane Position als nächste Position und der momentane Wert als neues „desire" vermerkt.

In den Zeilen 19 und 21 werden nach jedem Schleifendurchlauf die entsprechenden Werte auf die neue Position gesetzt.

Am Ende steht die nächste Position fest. Für den Fall, dass die neue Position der aktuellen Position entspricht[56], wird wie in Abschnitt 7.4.4 beschrieben ein Marker gesetzt, um den Agenten voranzutreiben.

[54] Die obere linke Ecke ist in diesem Fall im Bezug auf das Welt-Koordinatensystem gemeint.

[55] Von oben betrachtet entspricht die x-Koordinate links und rechts und die y-Koordinate vor und zurück.

[56] Das heißt, es wurde keine bessere als die aktuelle Position gefunden.

Algorithm 7.1 Die Rasterisierung der Bewegung in einer dynamische Welt.

```
1  // Initialisierung der Werte
2  scale = 15
3  startX = currentPosition.x - scale
4  startY = currentPosition.y - scale
5  endX = currentPosition.x + scale
6  endY = currentPosition.y + scale
7  // Umliegende Felder betrachten
8  temporaryPosition.x = startX
9  WHILE temporaryPosition.x <= endX DO
10     temporaryPosition.y = startY
11     WHILE temporaryPosition.y <= endY DO
12        tmpDesire = getDesire(temporaryPosition);
13        IF tmpDesire > desire OR
14           (tmpDesire == desire AND zufall == true)
15        THEN
16           desire = tmpDesire
17           nextMovePosition = temporaryPosition
18        END IF
19        temporaryPosition.y = temporaryPosition.y + scale
20     END WHILE
21     temporaryPosition.x = temporaryPosition.x + scale
22 END WHILE
```

7.6 Test

Die erste Version des Prototyps war noch recht gierig, was den Speicherbedarf und die Rechenzeit anging. Nachdem sich jedoch herausgestellt hat, dass die angedachten Optimierungen aus dem Abschnitt 5.5 keinen wirklichen Vorteil bringen, konnte die Anzahl der Agenten pro Jadex-Instanz in der zweiten Version verdoppelt werden.

7.6.1 Testumgebung

Um die Anwendung unter weniger optimalen Bedingungen zu testen, wurden als Testrechner handelsübliche PCs verwendet. Die Ausstattung und Rechenleistung dieser Testrechner entspricht einem gängigen Office-PC. Es wurden drei PCs über ein Gigabit-Netzwerk miteinander verbunden. Auf einem PC lief der Gameserver auf den beiden übrigen sollten jeweils eine Jadex-Instanz mit der maximal möglichen Anzahl Agenten laufen. Wie sich jedoch herausstellte, sind die Testrechner zum Betrieb des Gameservers ungeeignet.

Es wurde daher nur mit einem Testrechner und einer einzigen Jadex-Instanz getestet. Unter diesen Bedingungen war es mühelos möglich 120 Agenten zu starten, die sich auch beim Server anmelden konnten. Jedoch hat das den Gameserver so stark ausgelastet, dass das Verhalten der Agenten nicht beurteilt werden konnte.

7.6.2 Entwicklungsumgebung

Da die Entwicklung des Prototypen auf einem von der Ausstattung her deutlich besseren PC erfolgte, wurde dort ein weiterer Test durchgeführt. Aber auch hier waren die Bedingungen nicht optimal, da nun der Gameserver, der Observer und die Jadex-Instanz gemeinsam auf einem einzigen Computer betrieben werden mussten.

Zum Starten der verschiedenen Agenten wurde ein Manager mit zwei Beliefs entwickelt. Diese Beliefs sind Maximalwerte für die Anzahl von Agenten. Der Manager ist problemlos in der Lage die Welt mit 10 Jägern und 60 Rehen zu betreiben. Sobald ein Jäger ein Reh erschießt, erzeugt der Manager ein neues Reh.

Für die Anzahl an möglichen Agenten waren etwa 100 angepeilt. Wird der Manager mit dem Auftrag gestartet, nur Rehe zu erzeugen, schafft er es problemlos 150 davon zu erzeugen, die sich alle am Gameserver anmelden. Das liegt daran, dass für jedes von einem Jäger erlegte Reh ein Spieler beim Server abgemeldet werden muss. Im gleichen Atemzug erzeugt der Manager einen neuen Reh-Agenten, der wiederum am Server angemeldet wird. Wird also auf die Jäger verzichtet, kann der Gameserver-Prototyp eine größere Menge Agenten bedienen. Jedoch erreicht das Nachrichtenaufkommen zwischen Gameserver, Observer und Jadex-Instanz ab etwa 90 Agenten eine solche Masse, dass der Gameserver mit der Verarbeitung nicht mehr nachkommt.

Während der Tests sind die Abbildungen 7.6 und 7.7 entstanden.

7.7 Zusammenfassung

Der Prototyp ist in der Lage die an ihn gestellten Anforderungen zu erfüllen. Unter besseren Betriebsbedingungen wäre er vermutlich sogar in der Lage die Anforderungen zu übertreffen.

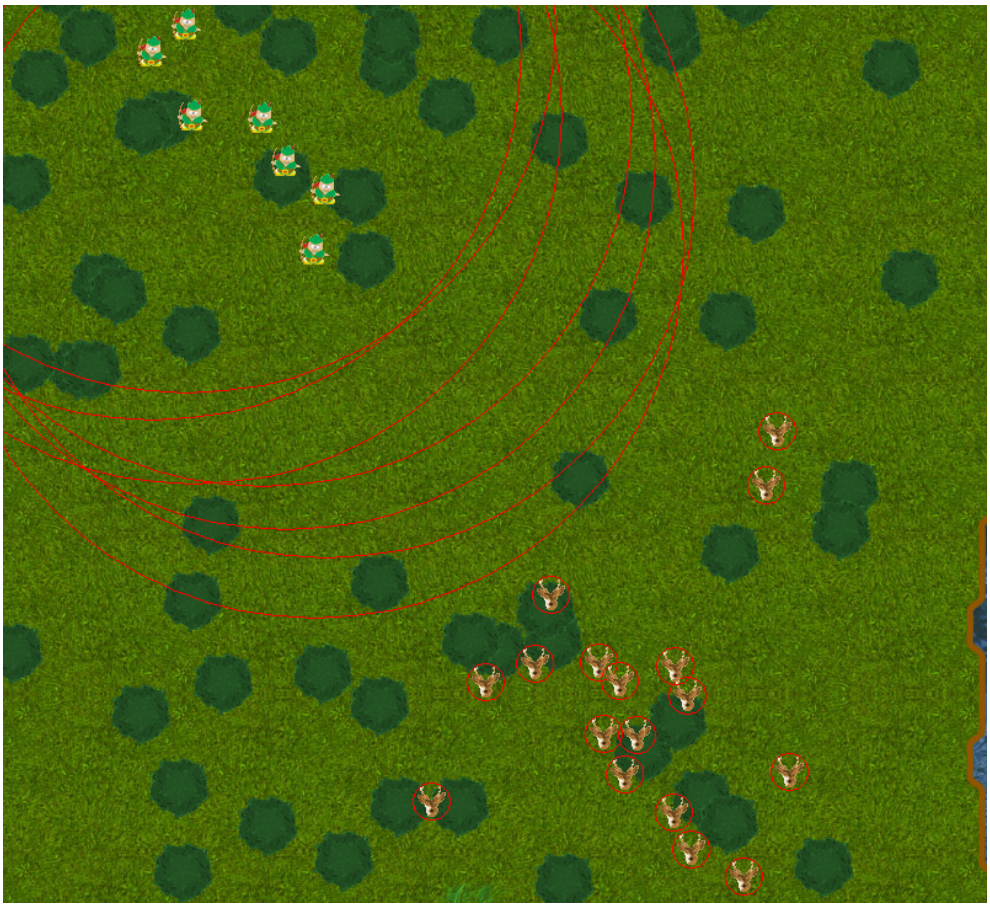

Abbildung 7.6: Massenjagd: Eine Gruppe Jäger jagd eine Gruppe Rehe.

Zu Beginn dieser Arbeit wurde mit einem Prototyp auf Basis einer A*-Wegfindung experimentiert. Bei Einsatz dieser Technik in Jadex trat schon bei 15 bis 20 Agenten oftmals ein Speicherüberlauf[57] in der Jadex-Instanz auf. Zwar hätte der Prototyp sicherlich verbessert werden können, aber nach den jetzigen Erkenntnissen kann eine entsprechende Agenten-Anzahl nahezu ausgeschlossen werden.

Die Anwendung der Potentialfelder ist insgesamt sehr leicht durchzuführen. Der Vorgang ist dabei immer der gleiche:

1. Felder identifizieren

2. Felder implementieren

[57] Heap-Space overflow

3. Problem der lokalen Maxima lösen[58]

Wie bereits im letzten Kapitel besteht jedoch auch bei dieser Anwendung weiterer Optimierungsbedarf bei der Feineinstellung der Feldstärken.

Die während der Tests im vorigen Abschnitt entstandene Abbildung 7.6 zeigt eine Gruppe von sieben Jägern, die sich zusammengeschlossen haben. Sie sind auf eine Gruppe Rehe gestoßen. In Abbildung 7.7 sind die Jäger auf Schussreichweite an die Rehe herangekommen und haben bereits sechs Rehe erfolgreich erlegt. Der Übersicht halber wurden die gelben Kreise (Sichtbereiche) ausgeblendet.

Abbildung 7.7: Massenjagd: Die Jäger haben die Gruppe Rehe eingeholt.

[58] Da es für dieses Problem wie bereits erwähnt noch keine beste Lösung gibt, kann allenfalls versucht werden, es zu umgehen.

Kapitel 8

Zusammenfassung

Abschließend werden nun nochmal alle Erkenntnisse der vorherigen Kapitel zusammengefasst. Danach folgt eine Bewertung der Implementation in Bezug auf Verhalten und Performance der Agenten und zum Schluss noch ein Ausblick über offene Fragen und Möglichkeiten.

8.1 Die statische Implementation

Die Welt aus Kapitel 6 ist in ihrer Größe beschränkt. Die Agenten, die in ihr leben und agieren, bewegen sich während der gesamten Dauer einer Spielrunde in denselben Bereichen. Daher bietet sich eine statische Lösung, bei der die Agenten ihre Umwelt nach und nach besser kennen lernen, an. Zu diesem Zweck wurden die Felder, wie in Abschnitt 5.5.1 beschrieben, aufgeteilt.

Die Effizienz dieser Lösung könnte durch den Einsatz eines geeigneten Suchverfahrens durchaus gesteigert werden. Da sich die Felder nur selten verändern und dynamische Felder ebenfalls zwischengespeichert werden, existiert zu jedem Zeitpunkt eine vollständig bewertete Karte der bislang aufgedeckten Welt. Wie bereits in Abschnitt 5.1 erwähnt, könnte ein modifizierter A*-Algorithmus den aktuellen Sichtbereich nach dem besten Wert durchsuchen. Ein Agent würde somit seltener in Sackgassen laufen und falls doch, würde er sie früher wahrnehmen und umkehren. Zusätzlich könnte der Agent diese Lücken füllen, sodass er beim nächsten Mal nicht wieder in dieselbe Sackgasse läuft. Dabei sollte jedoch der in Abschnitt 5.5.4 angesprochene Nachteil berücksichtigt werden, dass sich eine bestehende Sackgasse durch eine Veränderung der Welt auch auflösen könnte.

Die statische Implementation erfolgte jedoch nur, um die unterschiedlichen Einsatzmöglichkeiten der Potentialfelder zu zeigen.

8.2 Die dynamische Implementation

Schwerpunkt dieser Arbeit war die Kombination eines Agentensystems mit dem Konzept der Potentialfelder in einer dynamischen Welt. Wie sich gezeigt hat, bietet sich diese Lösung in einer Echtzeitumgebung durchaus an.

Die Implementation aus Kapitel 7 verzichtet gänzlich auf den Einsatz von Suchverfahren. Dazu musste jedoch eine Lösung für das Problem der lokalen Maxima aus Abschnitt 5.4.1 gefunden werden. Wie bereits in Abschnitt 5.5.4 erwähnt gibt es keine beste Lösung.

Im Gegensatz zu den Agenten aus Kapitel 6, bewegen sich die Agenten in der „Hunters and Deers"-Welt frei und wechseln regelmäßig ihren Aufenthaltsort. Außerdem hatten es die Agenten in dem Kapitel nur selten mit dynamischen Feldern zu tun und selbst wenn, waren es Felder von gegnerischen Spielern, die es auszuschalten galt. Die Agenten dieser Welt besitzen jedoch auch Felder für Agenten der gleichen Art, sodass sich Gruppen bilden können. Daraus ergibt sich eine ständige Veränderung der Werte im Sichtbereich, was ein Zwischenspeichern der Werte überflüssig macht, da sie beim nächsten Zugriff bereits mit hoher Wahrscheinlichkeit ungültig wären.

In Abschnitt 5.1 wurde das Vorgehen der Agenten mit dem Suchverfahren „optimistisches Bergsteigen" verglichen. Wird dazu noch das in Abschnitt 5.4.1 angesprochene Verfahren der Pheromon-Spur betrachtet, ist eine Ähnlichkeit zu einem weiteren Suchverfahren zu erkennen. Es handelt sich dabei um die Simulated-Annealing-Suche. Dieses Verfahren kombiniert das optimistische Bergsteigen mit einem zufälligen Weitergehen, „so dass man sowohl Effizienz als auch Vollständigkeit erhält" ([RN04] S. 155).

8.3 Bewertung

Der in dieser Arbeit entstandene Prototyp ist insgesamt recht stabil und das Verhalten der Agenten plausibel. Im Folgenden werden einige Beispiele näher betrachtet:

Gruppenbildung

In Bezug auf die Bildung von Gruppen zeigte sich sogar emergentes Verhalten. Treffen z.B. zwei Rehe aufeinander, sorgen die entsprechenden Felder für eine Anziehung. Befinden sich jedoch zusätzlich Grasflächen in der Nähe der einzelnen Rehe, sind deren Felder stärker, sodass die Rehe zunächst das Gras fressen. Dabei ist es gut möglich, dass sich die Rehe aus den Augen verlieren. Hat eines der Rehe die Grasflächen in seiner Umgebung gefressen, wird das Feld des anderen Rehs schließlich dominant. Das Reh bewegt sich also auf die zuletzt bekannte Position des anderen Rehs zu. Für den Beobachter scheint dieses Verhalten so, als ob sich die Rehe zunächst aus der Ferne betrachten und nur langsam gegenseitig annähern. Trifft ein Reh hingegen auf eine Gruppe anderer Rehe, ist die Anziehungskraft

stärker, da sich die Felder der einzelnen Rehe addieren. Für den Beobachter sieht es dann so aus, als würde ein Reh zu einer Gruppe anderer Rehe eher Vertrauen aufbauen.[59]

Informationsaustausch

Ein weiterer Vorteil der Gruppenbildung ist, dass die Gruppenmitglieder ihr Wissen über Grasflächen außerhalb der einzelnen Sichtbereiche teilen. Dabei werden diese Informationen jedoch nicht direkt ausgetauscht. Wenn sich ein Reh von Grasfläche zu Grasfläche durch die Welt bewegt, kann es vorkommen, dass bereits bekannte Grasflächen wieder aus dem Sichtbereich verschwinden. Wären dann alle Grasflächen im Sichtbereich verbraucht, würde das Reh sich an eine gemerkte Position erinnern und zu ihr zurückkehren. Bilden nun zwei Rehe eine neue Gruppe und eines der Rehe kennt keine Grasflächen mehr, folgt es automatisch dem anderen Reh, sofern dieses noch Grasflächen kennt. Sind auch dessen Grasflächen aufgebraucht und ist kein neues Reh zu der Gruppe hinzugestoßen, bewegt sich die Gruppe anhand der Marker aus Abschnitt 7.4.4 so lange zufällig durch die Welt, bis neue Grasflächen gefunden wurden.

Nachteil beim Einsatz von Pheromon-negativ-Markern

Es gibt jedoch auch negative Beobachtungen im Verhalten der Agenten. Die zufällige Bewegung durch die Marker aus Abschnitt 7.4.4 führt gelegentlich dazu, dass ein Jäger sich von einem Reh, welches sich kurz außerhalb seines Sichtbereichs befindet, entfernt. Das ist nicht ungewöhnlich, da er das Reh noch gar nicht wahrgenommen hat. Ein Beobachter dieser Szene bzw. ein menschlicher Spieler könnte sich jedoch darüber wundern. Eine Möglichkeit, dieses Verhalten zu verbessern, wäre es, den Jäger, sofern er keine Beute sieht, eine längere Zeit warten zu lassen, bevor er einen Marker setzt. Dabei stellt sich aber die Frage, wie lange er warten soll und im ungünstigen Fall könnte er wieder kurz vor Eintreffen eines Rehs die Position verlassen. Eine mögliche Lösung für dieses Problem findet sich im nächsten Abschnitt.

8.4 Ausblick

Während der Implementation des Prototyps wurden mehrere Möglichkeiten für dessen Erweiterung identifiziert.

[59] Das gleiche Verhalten gilt natürlich für die Jäger entsprechend.

Wissensnutzung

Die in dieser Arbeit umgesetzten Verhaltensmuster der Agenten sind auf das Jäger-Beute-Szenario beschränkt. Für den im letzten Unterabschnitt angesprochenen Nachteil beim Einsatz der Marker und ähnlicher Fälle könnte das Wissen eines Charakters über seine Umwelt genutzt werden.

Wie in Abschnitt 7.2 beschrieben, ignorieren die Jäger die Grasflächen völlig. Ein guter Jäger kennt jedoch das Verhalten seiner Beute. Daher sollte er wissen, dass Rehe Gras fressen. Wenn ein Jäger also keine Beute sieht, sollte er nicht zufällig irgendwo nach ihr suchen, sondern dahin gehen, wo es wahrscheinlich ist, sie zu finden. Um dieses Verhalten zu erzielen, könnte über eine Finite State Machine (Endlicher Automat) der Zustand des Agenten in die Berechnung der Felder einfließen. In diesem Fall würden Grasflächen ein anziehendes Feld bekommen, sofern der Jäger keine Beute sieht.

Gruppenverhalten

Wie im vergangenen Abschnitt beschrieben, funktioniert das Bilden von Gruppen schon recht gut. Jedoch besteht beim Verhalten der Gruppenmitglieder noch Verbesserungsbedarf.

In [Rey87] wird das so genannte Flocking (Schwarmverhalten) von einzelnen Agenten (Boids) in einer Gruppe beschrieben. Dabei wird ein realistisches Gruppenverhaltung durch die Einhaltung von nur drei Regeln erzielt:

1. Separation oder „Collision Avoidance”
   Vermeide die Kollision mit anderen Gruppenmitgliedern.

2. Alignment oder „Velocity Matching”
   Bewege dich wie dein Nachbar/Versuche dich mit der gleichen Geschwindigkeit zu bewegen wie dein Nachbar.

3. Kohäsion oder „Flock Centering”
   Versuche in der Nähe deiner Nachbarn zu bleiben.

Für ein besseres Gruppenverhalten könnten die Felder so anpasst werden, dass diese drei Regeln eingehalten werden.

In dieser Arbeit stand die Navigation der Agenten im Vordergrund. In Unterabschnitt 5.4.2 wird auf den Unterschied der Addition von Werten und dem Bilden eines Maximums eingegangen. Für die Navigation werden die Felder addiert, damit sich der Agent zunächst in die positivste Richtung und dann auf ein konkretes Ziel zubewegt. Um die oben genannten Regeln abdecken zu können, müssten die Agenten zum einen über die Felder ihrer Nachbarn ein Maximum bilden und zum anderen verschiedene Felder kombinieren. Bei der richtigen Feineinstellung der einzelnen Felder sollte sich ein realistischeres Gruppenverhalten, als das in dieser Arbeit erzeugte, zeigen.

Werkzeug zur Erzeugung neuer Charaktere

Ein großer Vorteil beim Einsatz von Jadex ist die Modularisierung durch Capabilities (siehe Unterabschnitt 3.3.2). Dadurch ist es möglich, bestimmte Verhaltensmuster wiederzuverwenden. Es würde sich daher anbieten, eine Verhaltensdatenbank anzulegen, sodass neue Charaktere aus bestehenden Komponenten zusammengesetzt werden könnten. Fehlt ein bestimmtes Verhalten, müsste lediglich die Datenbank um diese Komponente erweitert werden.

Ein Ziel dieser Arbeit war es, eine generische Basis für die Erzeugung neuer NPCs zu schaffen. Dieses Ziel wurde somit erreicht. Als nächstes könnte ein Werkzeug entwickelt werden, welches über die Nutzung der Verhaltensdatenbank die Erzeugung neuer Charaktere ermöglicht. Sinnvollerweise würde es in die in Unterabschnitt 2.1.3 erwähnte WebApplikation integriert werden und ist prädestiniert für eine eigenständige Bachelorarbeit.

Anhang A

Hunters and Deers

„Hunters and Deers" ist der Name eines Prototyps für die Timadorus-Welt. Die folgenden Abschnitte erläutern in Kürze die Hintergründe der einzelnen Komponenten.

A.1 Timadorus

Hinter dem Namen „Timadorus" verbirgt sich zum einen die Idee zu einer prinzipiell uneingeschränkt realistischen Welt. Zum anderen ist er Teil eines Projektmoduls an der Hochschule für Angewandte Wissenschaften Hamburg. Der volle Name des Moduls lautet „Der Weg nach Timadorus" (siehe [TIMa]).

Ziel des Projektes ist der Lernerfolg der Studierenden. Dennoch soll im Laufe der Zeit eine funktionierende Umgebung entstehen, die nach Möglichkeit als Basis für weitere Abschlussarbeiten dienen kann. Weitere Informationen zum Projekt finden sich auf der offiziellen Seite unter [TIMb].

A.2 Gameserver

Der Prototyp des Gameservers ist einzig für die Anwendung dieser Arbeit entstanden. Daher wurden lediglich die Teile implementiert, die für deren Durchführung relevant sind. Gewisse Aspekte wurden dabei vernachlässigt, obwohl sie sicherlich interessant wären. Da jedoch die Entwicklung des Gameservers nicht Bestandteil dieser Arbeit ist, muss darauf verzichtet werden.

Entsprechend dem Hunter-Prey-Szenario erhielt der Prototyp seinen Namen[60]. Der Server unterstützt die Anmeldung mehrerer Clients, jedoch ohne Authentifizierung. Mit dem Server verbundene Clients können sich frei bewegen und erhalten Informationen über Aktionen

[60] HuntersAndDeersServer

anderer Clients. Die Verwaltung der Welt-Objekte geschieht in Listen, die allen Clients zur Verfügung stehen.

Die Anforderungen für die Durchführung dieser Arbeit unterscheiden sich jedoch stark vom eigentlichen Konzept.

A.2.1 Konzept

Der Server:

- prüft Aktionen und Nachrichten der Clients und reagiert ggf. auf Fehler und Betrugsversuche
- verwaltet die Sichtbereiche der Clients und übermittelt die für den Client interessanten Informationen
- ist frei von Regeln zur Verwaltung von Objekten (künstliche Intelligenz)

A.2.2 Umsetzung

Der Server:

- leitet alle Nachrichten an alle Clients weiter
- liefert einmalig eine Liste aller Objekte an einen Client aus und bietet weitere Informationen auf Anfrage
- prüft lediglich bei konkreten Aktionen auf Objekte, ob ein Client dazu überhaupt berechtigt ist
- enthält Regeln für die Reaktion von z.B. Gras
- verwaltet für alle Objekte Attribute wie Ausdauer und Lebensenergie

Die Clients:

- müssen die Relevanz eingehender Nachrichten beurteilen
- müssen ihren Sichtbereich selbst verwalten

A.3 Gameclient 3D

Dieser Client war ursprünglich als Möglichkeit zur Beobachtung der Welt gedacht. Die Entwicklung dieses Clients wurde jedoch nicht weitergeführt, da eine bessere Lösung gefunden wurde. Der Client ist dennoch funktionstüchtig und kann zum Betrachten der Welt genutzt werden.

A.4 Observer

Der Observer ist eine 2D-Anwendung, die es erlaubt einen besseren Überblick über das Geschehen zu erhalten. Da der Server alle Informationen an jeden Client schickt, ist er ebenfalls als gewöhnlicher Client entwickelt worden.

Der Observer bietet die Möglichkeit weite Teile der Welt zu überblicken und Objekte in ihr zu verfolgen. Dazu zeigt er die Objekte mit ihren Bewegungen und Sichtbereichen an.

Anhang B

TankArena

„TankArena” ist der Name eines Prototyps für ein Framework zur Simulation von Intelligenz in einer unbekannten Welt. Der Einsatz dieses Frameworks soll Studenten die Anwendung von Methoden der Künstlichen Intelligenz näher bringen.

Die aktuelle Version des Frameworks kann unter [Can] bezogen werden.

B.1 Die Welt

Die Welt wurde absichtlich einfach gehalten. Die Flächen auf denen sich die Spieler bewegen ist flach und ohne Steigungswinkel. Es gibt jedoch auch Berge und Wasser, welche als Hindernisse dienen. Während Berge lediglich unpassierbar sind und umfahren werden müssen, stellt Wasser eine Gefahr für den Spieler dar. Bewegt sich ein Spieler ins Wasser, wird er zerstört.

Weitere Objekte in der Welt sind die jeweilige Basis, sowie die Spieler eines Teams. Um den Aufwand überschaubar zu halten, können sowohl Teile der Basis, als auch andere Spieler „durchfahren” werden (es findet keine Kollisionsabfrage statt).

Die Welt besitzt eine simple Physik. Bewegungen sind lediglich auf Bodenhöhe möglich, ein echtes „Fallen” der Spieler gibt es nicht. Lediglich die Geschosse der Spieler unterliegen den Gesetzen der Natur („gravity-only”).

B.2 Die Teams

Es gibt maximal vier Teams mit bis zu fünf Spielern, also höchstens 20 Spieler in einer Spielwelt.

Die Positionen der Basen sind fest vorgegeben (in jeder Ecke der Welt eine Basis). Die Farben und somit die Position der Heimatbasis werden dem Team zu Beginn einer Runde

per Zufall zugeteilt. Bei weniger als vier Teams ist somit erst nach Erkundung der Welt eine Aussage über die Position von gegnerischen Basen möglich.

Jeder Spieler eines Teams kann durch eine eigene KI gesteuert werden.

B.3 Besondere Merkmale

Die Spieler kennen zu Beginn einer Runde lediglich ihre Startposition. Der Sichtbereich eines Spielers ist eingeschränkt. Bewegt sich ein Spieler durch die Welt, erhält er automatisch Informationen über die Objekte in seinem Sichtbereich und kann entsprechende Entscheidungen treffen.

Wird ein Spieler angegriffen, verliert er Lebenspunkte. Er startet mit 100 Punkten. Ein Treffer kann maximal 75 Punkte abziehen (Volltreffer), die Höhe des Schadens nimmt vom Zentrum der Explosion eines Geschosses linear bis zu einer Grenze von 7,5 Einheiten ab. Somit muss ein Spieler mindestens zweimal getroffen werden, bevor er ausscheidet. Zehn Sekunden nach dem Ausscheiden eines Spielers, wird er wieder in seinen Hangar in der Basis zurückgesetzt. Während der Wartezeit kann der Spieler nichts tun.

An der Position, an der der Spieler ausgeschieden ist, bleibt eine Kiste mit Reparaturmaterial zurück. Sammelt ein Spieler diese Kiste ein, werden seine Lebenspunkte regeneriert (auf 100 Punkte).

Die Basen können ebenfalls angegriffen werden. Für die Hangars der Basen gilt bezüglich des Schadens das gleiche Muster wie für die Spieler. Ein Hangar hat zu Beginn 250 Lebenspunkte. Sobald die Lebenspunkte eines Hangars auf 0 sinken, wird dieser Hangar zerstört. Der Spieler dieses Hangars hat nach seiner Zerstörung keine Möglichkeit der Rückkehr mehr.

Sind alle Hangars einer Basis zerstört (sowie ggf. alle Spieler des Teams), scheidet das zugehörige Team aus.

Geschossen wird mittels Angabe von Kraft und Winkel. Die Geschosse beschreiben eine Parabelkurve und explodieren bei Kontakt mit dem Untergrund bzw. einer Basis, jedoch nicht bei Kontakt mit einem Panzer.

„Friendly Fire” ist aktiv, also können auch Spieler des eigenen Teams angegriffen und zerstört werden.

B.4 Punktesystem

Verschiedene Ereignisse werden durch Vergabe von Punkten belohnt/bestraft:

- Zerstörung eines Spielers, gegnerisches Team: +200
- Zerstörung eines Spielers, eigenes Team: -200

- Verlust eines Spielers: -200
- Zerstörung eines Hangars, gegnerisches Team: +500
- Zerstörung eines Hangars, eigenes Team: -500
- Verlust eines Hangars: -500
- Einsammeln einer Materialkiste: +125

B.5 Siegbedingungen

Der Schiedsrichter (Prof.) hat jederzeit die Möglichkeit (unter Angabe von Gründen) ein Team zu disqualifizieren. Fehlerhafte Programmierung (z.B. Endlosschleifen, Exceptions, unerlaubter Eingriff in die Spiellogik) führen ebenfalls zur Disqualifizierung eines Teams. (Im Zweifelsfall entscheidet immer der Schiedsrichter.)

B.5.1 Erreichen einer Punktzahl

Erreicht ein Team eine vorgegebene Punktzahl, wird dieses Team zum Sieger erklärt. Die Höhe dieser Punktzahl wird vom Schiedsrichter (Prof.) vorgegeben und kann während einer Runde entsprechend angepasst werden.

B.5.2 „Last man standing”

Wurden alle gegnerischen Basen zerstört, gewinnt das Team mit der letzten noch vorhandenen Basis.

B.5.3 Ablauf eines Timers

Es gibt die Möglichkeit eine maximale Spiellänge zu bestimmen. Nach Ablauf der vorgegebenen Zeit wird die Runde beendet und die aktuellen Punkte aufaddiert. Dabei werden die Lebenspunkte der Basen mit dem Faktor 10 verrechnet. Für jeden Panzer im Spiel (gerade aktiv) erhält das Team 100 Punkte.

Literaturverzeichnis

[AP07] ALEXANDER POKAHR, Lars B.: *Jadex User Guide*. http://freefr.dl.sourceforge.net/project/jadex/jadex/0.96/userguide-0.96.pdf. Version: Juni 2007

[BHW05] BORDINI, Rafael H. ; HÜBNER, Jomi F. ; WOOLDRIDGE, Michael: *Programming Multi-Agent Systems in AgentSpeak using Jason: A Practical Introduction with Jason*. John Wiley & Sons Ltd., 2005 (Wiley Series in Agent Technology). – ISBN 9780470029008

[BP] BRAUBACH, Lars ; POKAHR, Alexander: *Jadex*. http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview

[Can] CANOW, Carsten: *TankArena*. http://lunatic-soft.de/tankarena/

[CT04] CHRISTIAN THURAU, Christian B.: *Learning human-like Movement Behavior for Computer Games*. http://aiweb.techfak.uni-bielefeld.de/files/papers/Thurau2004-LHL.pdf. Version: 2004

[DAR] *Project Darkstar*. http://www.projectdarkstar.com/

[Dav05] DAVISON, Andrew ; MCLAUGHLIN, Brett (Hrsg.): *Killer Game Programming in Java*. Sabastopol, CA : O'Reilly Media, Inc., 2005. – ISBN 0596007302

[DeL08] DELISLE, Robert K.: Beyond A*: IDA* and Fringe Search. In: JACOBS, Scott (Hrsg.): *Game Programming Gems 7*. Boston, Mass. : Cengage Learning Services, 2008. – ISBN 9781584505273, S. 289–294

[FOW] *Nebel des Krieges*. http://de.wikipedia.org/wiki/Fog_of_War

[Hag09a] HAGELBÄCK, Johan: *A Multi-Agent Potential Field Based Approach for Real-Time Strategy Game Bots*. Ronneby, Sweden, School of Engineering, Blekinge Institute of Technology, Sweden, Diss., 2009. http:

//www.bth.se/tek/jhg.nsf/bilagor/JHG_1_Lic_Thesis_pdf/$file/JHG_1.Lic.Thesis.pdf. – ISBN: 9789172951600

[Hag09b] HAGELBÄCK, Johan: *Using Potential Fields in a Real-time Strategy Game Scenario (Tutorial)*. http://aigamedev.com/open/tutorials/potential-fields/. Version: 2009

[HB] HÜBNER, Jomi F. ; BORDINI, Rafael H.: *Jason*. http://jason.sourceforge.net/JasonWebSite/Jason%20Home.php

[HJ08a] HAGELBÄCK, Johan ; JOHANSSON, Stefan J.: *Dealing with Fog of War in a Real Time Strategy Game Environment*. http://www.bth.se/tek/jhg.nsf/bilagor/HagelbackJohanssonCIG08_pdf/$file/HagelbackJohanssonCIG08.pdf. Version: 2008

[HJ08b] HAGELBÄCK, Johan ; JOHANSSON, Stefan J.: *The Rise of Potential Fields in Real Time Strategy Bots*. http://www.bth.se/tek/jhg.nsf/bilagor/AIIDE08_pdf/$file/AIIDE08.pdf. Version: 2008

[HJ08c] HAGELBÄCK, Johan ; JOHANSSON, Stefan J.: Using Multi-agent Potential Fields in Real-time Strategy Games. In: PADGHAM, Müller P. Parkes (Hrsg.): *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. Estoril, Portugal, 2008, 631-638

[HJ09] HAGELBÄCK, Johan ; JOHANSSON, Stefan J.: *A Multi-agent Potential Field based bot for a Full RTS Game Scenario*. http://www.bth.se/tek/jhg.nsf/bilagor/AIIDE09_pdf/$file/AIIDE09.pdf. Version: 2009

[Jad] ; Telecom Italia SpA (Veranst.): *Jade*. http://jade.tilab.com/

[LP04] LIN PADGHAM, Michael W. ; WOOLDRIDGE, Michael (Hrsg.): *Developing intelligent agent systems - A practical guide*. West Sussex, England : John Wiley & Sons Ltd., 2004 (Wiley Series in Agent Technology). – ISBN 0470861207

[Rao96] RAO, Anand S.: *AgentSpeak(L): BDI Agents speak out in a logical computable language*. 1996

[RD08] RABIN, Steve ; DELP, Michael: Designing a Realistic and Unified Agent-Sensing Model. In: JACOBS, Scott (Hrsg.): *Game Programming Gems 7*. Boston, Mass. : Cengage Learning Services, 2008. – ISBN 9781584505273, S. 217–228

[Rey87] REYNOLDS, Craig W.: *Flocks, Herds, and Schools: A Distributed Behavioral Model*. http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf. Version: 1987

[RN04] RUSSEL, Stuart ; NORVIG, Peter: *Künstliche Intelligenz: Ein moderner Ansatz*. 2. München, Germany : Pearson Studium, 2004. – ISBN 9783827370891

[TIMa] *Projektmodul: Der Weg nach Timadorus*. `http://www.informatik.haw-hamburg.de/fileadmin/Homepages/ProfFaehnders/wahl/PO_BHNK.pdf`

[TIMb] *Der Weg nach Timadorus*. `http://www.timadorus.org/`

[UL08] UWE LÄMMEL, Jürgen C.: *Künstliche Intelligenz*. 3. München, Germany : Hanser Verlag, 2008

[Woo02] WOOLDRIDGE, Michael: *An Introduction to MultiAgent Systems*. West Sussex, England : John Wiley & Sons Ltd., 2002. – ISBN 9780471496915

Abbildungsverzeichnis

Algorithmenverzeichnis

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 01. Februar 2010 Carsten Canow