technische universität
dortmund

Ein Konzept zur Unterstützung menschlicher Spieler in Browserspielen

Daniel Haus

Algorithm Engineering Report
TR09-2-011
Dez. 2009
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

technische universität
dortmund

Diplomarbeit

Ein Konzept zur Unterstützung menschlicher Spieler in Browserspielen

Daniel Haus

26. Juli 2009

Betreuer:
Prof. Dr. Rudolph
Dipl.-Inf. Preuß

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Parallel zur Ausbreitung des Internets in den vergangenen 10 Jahren ist im gleichen Zeitraum auch die Popularität von Computerspielen stark gewachsen. Besondere Beachtung verdient die Beliebtheit sogenannter MMOGs[1], zu denen auch Browserspiele zählen: Spiele, die in Erscheinung gewöhnlicher Internet-Seiten in gängigen Webbrowsern laufen. Die soziotechnische Architektur, die bei Browserspielen eingesetzt wird, ist dieselbe, wie in allen anderen modernen Internetanwendungen – insbesondere diese, die jüngst als „Web 2.0“ bezeichnet werden. Ein zentraler Webserver[2] beantwortet dynamisch und in Echtzeit die Anfragen, die viele voneinander ansonsten unabhängige Teilnehmer, welche weltweit verstreut vor ihren Rechnern sitzen, über entsprechende Clients, also Webbrowser, versenden. Dabei werden die einzelnen Teilnehmer identifiziert und verursachte Zustandsänderungen fließen in das System, ebenfalls in Echtzeit, ein. Details und eine genaue Definition von Browserspielen im Sinne dieser Arbeit sind in Abschnitt 2.1 ausgeführt. Erläuterungen zu einer möglichen, praxisgerechten Implementierung eines solchen Systems findet man in Abschnitt 3.2.

Das besondere Merkmal dieses Genres, die gleichzeitige Präsenz von Hunderten oder gar Tausenden von Spielteilnehmern, scheint auf den ersten Blick eine bisher sehr wichtige Disziplin der Computerspielentwicklung in dieser Art von Computerspielen überflüssig zu machen: künstliche Intelligenz bzw. computational Intelligence. Diese wird überwiegend dazu eingesetzt, dem Spieler würdige aber faire, gegnerische Mitspieler vorzutäuschen. Dank der weltweiten Vernetzung stehen für jedes Spiel rund um die Uhr genug reale Gegner bereit. Die Simulation derselben in Form von NPC[3] ist zum Spielen nicht mehr zwingend nötig. Natürlich eignen sich dennoch computergesteuerte Gegner hervorragend als „leichte Beute“, die den Frustrationsgrad der Mitspieler nicht unnötig steigen lässt. Jeder Spielteilnehmer möchte schließlich gewinnen oder sich zumindest als Gewinner fühlen.

[1] „Massive Multiplayer Online Game“ (zu Deutsch: „ Massen-Mehrspieler-Online-Gemeinschaftsspiel“)
[2] häufig auch mehrere gekoppelte Rechner um die Last zu verteilen
[3] „Non-player character“

Die Praxis zeigt, dass Browserspiele wunderbar ohne künstliche Intelligenz auskommen, da tatsächlich in diesem Genre bisher keine entsprechenden Techniken Anwendung finden. Das liegt einerseits am Know-How- und Budget-Mangel, da die Browserspiele der letzten Jahre und Jahrzehnte überwiegend von Schülern, Studenten und Hobbyprogrammierern im privaten Rahmen und unentgeltlich entwickelt wurden. Dies könnte sich in Zukunft ändern, sind in den letzten Jahren auffällig viele Spielestudios und Web-Agenturen in die Browserspiel-Branche eingestiegen, indem sie etablierte Spiele übernommen oder eigene Entwicklungen auf den Markt gebracht haben. Andererseits aber ist ein wichtiger Faktor, dass solche Spiele gegenüber anderen Spieltypen (insbesondere nicht-MMOGs) ohne maschinell simulierte Gegner überhaupt spielbar sind. Es stellt sich nun die Frage, ob künstliche Intelligenz in Browserspielen grundsätzlich unnötig ist oder ob sie dort vielleicht in einer ganz anderen Art und Weise Anwendung finden und zum Spielspaß beitragen kann.

Die in dieser Arbeit vorgeschlagenen Grundideen entstanden 2007 im Rahmen eines Seminars „KI/CI in Computerspielen“ [36] aus der Not heraus[4] und wurden dort kontrovers diskutiert.

1.1 Motivation und Hintergrund

Die meisten der derzeit angebotenen Browserspiele sind technisch sehr einfach gehalten und wenig benutzerfreundlich gestaltet. Die Mehrzahl kommt mit einer zweidimensionalen Land-, See- oder Weltraumkarte aus, auf der es Charaktere oder militärische Einheiten im Rundentakt zu navigieren gilt.

Ein realistisches Beispielszenario[5] für einen Spieleinsteiger, der mit seinem ersten Raumschiff, das er zum Spielbeginn geschenkt bekommt, einen kolonisierbaren Heimatplaneten finden soll, sieht folgendermaßen aus: Er bewegt das Raumschiff um wenige Kartenfelder, wartet dann einige Stunden auf den nächsten Spielrundenwechsel, mit dem die Batterien des Schiffs wieder aufgefrischt werden und kann erst dann die Reise um einige weitere Kartenfelder fortsetzen. Dies geschieht so lange bis er seinen Zielplaneten – häufig erst nach bis zu mehreren Tagen – erreicht hat. Größere Entfernungen werden hier oft zu einer tagelangen Tortur, besonders zum Spieleinstieg, bei dem die Ressourcen für gewöhnlich sehr knapp bemessen sind. Im fortgeschrittenen Spielverlauf, wenn dem Spieler mehrere Hundert Einheiten zur Verfügung stehen, schlägt die Langeweile in Überforderung um, wenn er den Überblick behalten und im gegebenen Zeitfenster alle Einheiten navigieren will.

Bei vielen Spielen des Genres kann die Abwesenheit einen Teilnehmer schon nach ein bis zwei Rundenwechseln weit zurückwerfen, wenn Gegner diese ausnutzen und systematisch die schlafenden Einheiten attackieren. Das mindert den Spielspaß des potenziellen Opfers oder fördert auf der anderen Seite sogar die Spielsucht, wenn Spieler um jeden Preis online

[4] zum Thema „CI & KI bei Browserspielen“ existierte schlicht kein Material
[5] wie etwa im Spiel „Star Trek Universe 2“ [30]

bleiben, um besondere Tages- und Nachtzeiten zum eigenen Spielvorteil auszunutzen oder den Mitstreitern ähnliche Chancen zu nehmen.

Für den Fall längerer Abwesenheit von Spielern bietet „Star Trek Universe“ den so genannten Urlaubsmodus als Lösung, in welchem das Konto des Spielteilnehmers vorübergehend stillgelegt wird. Eine Interaktion mit seinen Einheiten ist dann bis zum Beenden des Urlaubsmodus unmöglich. Entsprechend werden in diesem Zeitraum allerdings auch keine Ressourcen gefördert und die Schiffe nicht gewartet, wodurch der Spieler Zeit verliert.

Die beschriebenen Szenarien dienen als Beispiel für mögliche Einsatzgebiete von künstlicher Intelligenz in Browserspielen. Die umständliche Navigation lässt sich etwa vereinfachen, indem man dem Spieler einen programmierbaren Agenten zur Verfügung stellt, der für das Erreichen der Zielkoordinaten sorgt. Agenten mit komplexerer Logik, die durch den Spielteilnehmer angepasst werden könnte, wären sogar in der Lage große Zahlen von Spielfiguren simultan im Sinne des Spielers zu steuern, so das dieser den Überblick behalten würde. Im Falle eines Angriffs in Abwesenheit des angegriffenen Spielers wäre es möglich verschiedene Strategien – wie Angriff oder Flucht – zu benennen, von denen der Agent dann automatisch die erfolgversprechendere auswählen und anwenden würde.

1.2 Verwandte Arbeiten

Dem Autor ist es zu diesem Zeitpunkt nicht möglich, ein öffentlich zugängliches Browserspiel nach der dieser Arbeit zugrunde liegenden Definition (siehe Abschnitt 2.1) auszumachen, welches sich den Methoden der künstlichen Intelligenz bedient. Eine mögliche Erklärung ist, dass, wie bereits erwähnt, Browserspiele bisher von meist jüngeren Amateur- und Hobbyprogrammierern entwickelt wurden, denen diese Methoden nicht vertraut oder zu aufwändig einzusetzen sind. Die jüngsten Entwicklungen zeigen, dass mittlerweile auch kommerzielle Softwarestudios vermehrt Browserspiele entwickeln. Methoden und Techniken wie die oben beschriebenen finden aber offensichtlich auch hier (noch) keine Verwendung. Wissenschaftliche Arbeiten zum Thema „CI/KI in Browserspielen“ sind zu diesem Zeitpunkt ebensowenig bekannt.

1.3 Zielsetzungen

Im Rahmen dieser Arbeit soll erforscht werden, in welcher Weise sich Browserspiele um Methoden der künstlichen Intelligenz (kurz „KI“) und der computational Intelligence (kurz „CI“) ergänzen lassen, so dass Spielspaß und Motivation gesteigert werden und gleichzeitig die Fairness insbesondere gegenüber Spieleinsteigern erhöht wird.

1.3.1 Erhöhung des Spielspaßes und der Motivation

Das Konzept von Agenten, die durch den Spieler programmiert werden können, um zugewiesene Einheiten automatisiert zu steuern, kann die oben beschriebenen Probleme lösen. Angenommen ein Spieler hat eine Einheit an einen bestimmten, ferngelegenen Punkt auf der Karte zu navigieren, so beauftragt er nun einen Agenten mit der Steuerung, indem er diesem die Zielkoordinaten und eventuell einige Regeln für Ausnahmesituationen – beispielsweise feindliche Angriffe – gibt. Sobald die Zielposition erreicht ist, schickt der Agent dem Spieler eine Nachricht und beendet sein Programm, die „Taktik“. Verfolgt der Spieler mehrere unterschiedliche Strategien langfristiger Art, welche über Rundenwechsel hinaus gehen, mit vielen verschiedenen Einheiten und beauftragt hierzu programmierbare Agenten, so verliert er nicht so schnell den Überblick wie bei manueller Steuerung.

Durch Ereignisse wie Angriffe oder Ressourcenknappheit angestoßene Taktiken können Einheiten bei Abwesenheit des Spielers aber auch als automatische Selbstschutzmechanismen dienen. Droht Gefahr durch eine Übermacht feindlicher Schiffe in unmittelbarer Nähe, so löst der Agent zum Beispiel eine Fluchttaktik aus, die das gesteuerte Objekt zunächst in Sicherheit bringt. Eine alternative Taktik könnte Verstärkung anfordern und zum Angriff übergehen. Bei Knappheit von Rohstoffen oder Kapazitätsengpässen würde ein anderer Agent etwa Minen, Raffinerien oder Silos bauen lassen. Auf diese Weise entfallen in einigen Fällen mühsame, stupide, repetitive Aufgaben, der motivierende „schnelle Erfolg“ rückt in greifbare Nähe.

1.3.2 Verbesserung der Fairness zwischen den Spielern

Die Delegation von unangenehmen Aufgaben und der Zugewinn an Sicherheit können sich auch auf die Fairness zugunsten von Neueinsteigern eines solchen Spiels auswirken. Besitzt ein Spieler gerade seine ersten zwei bis drei Einheiten, ist er einerseits ein leichtes Opfer, andererseits wird das Spiel für ihn schnell beendet sein, sollte seine Wachsamkeit nachlassen.

Bei aktuellen Browserspielen bzw. MMOGs kommt es vor, dass Spieler beobachten und darauf warten, dass andere Mitspieler zu bestimmten Zeiten nicht aktiv am Rechner sind, zum Beispiel weil sie auf der Arbeit oder in der Schule sind oder schlafen. Die Angst, in solchen Momenten schutzlos ausgeliefert zu sein, etwas zu verpassen oder den Anschluss zu verlieren kann oft ernste soziale oder auch gesundheitliche Folgen für den Spieler haben. Mit entsprechend vorprogrammierten und im Notfall automatisch ausgelösten Taktiken können Einheiten in einem eingeschränkten Rahmen für sich selbst sorgen und damit solchen Gefahren entgegenwirken.

Durch die offene Architektur ist es für erfahrene Bastler ein Leichtes Skripte zu Entwickeln, die über unerlaubte[6] Zweitzugänge bei einem Browserspiel, welches von Haus aus

[6] aber schwierig automatisiert zu verifizierende

keine Automatisierungsmöglichkeiten vorsieht, Rohstoffe sammeln und Waren produzieren, um diese dann den Einheiten des Hauptkontos des realen Spielers zugänglich zu machen. Diese Vorgehensweise wird in der Regel nicht geduldet und ist bei fast jedem Spiel per Spielregel verboten. Sie stellt ein großes Problem dar, ist es doch sehr schwierig programmatisch festzustellen, ob die HTTP-Anfragen von einem realen Webbrowser stammen oder dieser von einem Skript simuliert wird. Stellt man ähnliche Methoden bewusst allen Spielern zur Verfügung, so kann die Fairness deutlich verbessert werden und vielleicht sogar den Betrügern eine legitime Alternative angeboten werden.

1.4 Aufbau der Arbeit

Um den Leser grundsätzlich in die Thematik einzuführen, wird in Kapitel 1 zunächst kurz auf die Motivation des Themenkomplexes eingegangen. Anschließend werden die grundlegenden Ziele, die durch den Einsatz von Methoden der künstlichen Intelligenz und der computational Intelligence in Browserspielen erreicht werden sollen, grob umrissen.

Kapitel 2 definiert den Begriff „Browserspiel“ im Sinne der Arbeit allgemein und grenzt das Genre gegenüber artverwandten Spieltypen ab. Im Anschluß werden einige Methoden der künstlichen Intelligenz und der computational Intelligence aufgegriffen, die im Kontext sinnvoll erscheinen und in der eigens für diese Arbeit entwickelte Beispielanwendung „Megaira“ [28] zum Einsatz kommen. Kapitel 3 umreißt die technische Architektur der Anwendung, die eingesetzten Entwicklungstechniken, Bibliotheken und die Test- und Server-Software.

Die ermittelten Ergebnisse und Antworten auf die zu Beginn aufgeworfenen Fragen werden in Kapitel 4 erläutert. Ferner werden hier neue Probleme aufgezeigt, die durch das neuartige Spielsystem entstehen können, und gleichzeitig Vorschläge für Lösungen derselben umrissen. Zum Abschluss werden weitere Methoden angesprochen, die thematisch sinnvoll erscheinen und die Beispielanwendung bereichern könnten, aber den Rahmen dieser Arbeit sprengen würden. Diese sind in Kapitel 5 nachzulesen.

Kapitel 2

Grundlagen

Um tiefer in die Materie einzusteigen, ist es nötig zunächst einige wichtige Grundbegriffe zu definieren (Abschnitt 2.1) und zugrundeliegende Methoden der künstlichen Intelligenz und der computational Intelligence zu umreißen. Das passiert in Abschnitt 2.2. Vor allem soll definiert werden, was im Allgemeinen überhaupt unter Browserspielen verstanden wird. Zwar werden einigen Abwandlungen dieses Genres gerne unter diesem Begriff zusammengefasst, aber vom Wesen her und im Kontext dieser Arbeit betrachtet, sind sie anderen Kategorien zuzuordnen.

2.1 Was sind Browserspiele?

Diese Arbeit betrachtet speziell serverseitige Browserspiele. Bevor die Techniken beschrieben werden, um die sich solche Spiele erweitern lassen, wird erläutert, was genau im Rahmen dieser Arbeit unter dem genannten Begriff zu verstehen ist.

Auch wenn einige Vertreter der Gattung schon länger existieren, so explodiert die Zahl der Spielteilnehmer in diesen Tagen deutlich. Die Zahl weltweit an Browserspielen teilnehmenden Menschen lässt sich auf mindestens mehrere zig Millionen schätzen. Allein das Spiel „Die Renaissance Königreiche“ [16] zählt laut seiner Startseite über zweieinhalb Millionen registrierte Teilnehmer. Bei „DarkOrbit“ [14] sind es sogar weit über 20 Millionen, von denen 40.000 bis 50.000 gleichzeitig angemeldet sind. Regelmäßig kommen neue Spiele mit neuen technischen Innovationen hinzu, und die Grenze zu herkömmlichen, mehrspielerorientierten Computerspielen verschwimmt zunehmend. So bietet die Weltraumsimulation „DarkOrbit“ eine 2.5D-Ansicht[1] mit Echtzeit-Schlachten und bezeichnet sich daher selbst auch als „Der Action-Shooter unter den Browsergames“ [14].

[1] isometrische 3D-Ansicht einer zweidimensionalen Weltraumkarte

2.1.1 Die gemeine Architektur von Browserspielen und Webanwendungen

Die Wikipedia [51] definiert bis heute als wohl einzige Enzyklopädie den Begriff „Browserspiel“:

> Ein Browserspiel (engl. „browser-based game“ oder „browser game“) ist ein Computerspiel, das einen Web-Browser als Benutzerschnittstelle benutzt.

Browserspiele sind Spiele in Form von Webanwendungen, die auf Webservern laufen und Internet-Browser als graphische Benutzerschnittstelle verwenden. Die Kommunikation zwischen Benutzerschnittstelle und Controller-Logik erfolgt über HTTP[2]. Das bedeutet in der einfachsten Form, dass der Spieler über seinen Browser eine Spielseite vom entfernten Server irgendwo im Internet anfordert, die dieser dann auf dem lokalen Rechner darstellt. Benutzereingaben wie Klicks auf Hyperlinks oder Formulareingaben werden in HTTP-Anfragen umgewandelt und dem Server zugestellt. Dieser antwortet dann wiederum mit einer aktualisierten Version der Spielseite, welche sowohl die Änderungen des anfragenden Spielers als auch die seiner Mitspieler und Gegner reflektiert.

Aus Benutzersicht ist beim heutigen Stand der Technik oft kaum noch ein Unterschied zwischen Internetanwendung und nativer, lokal installierter Software zu erkennen. Schnelle Antwortzeiten dank modernem Breitband-Internet, Techniken wie browserseitigem Scripting und AJAX[3] lassen nicht erkennen, dass die Daten oft vom anderen Ende der Erde geladen werden. Wie auch immer das Ergebnis aussieht, es handelt sich bei Browserspielen genau um die typische Client-Server-Architektur des World Wide Web, also um gewöhnliche Websites auf HTML-Basis mit einem beliebig gearteten Server-Backend[4].

2.1.2 Genres und Szenarien

Bei den ersten Browserspielen Anfang/Mitte der 90er Jahre handelte es sich um zweckentfremdete Foren-, Chat- und Newsgroup-Software, die als Kommunikationsplattform für klassische Rollenspiele diente. Speziell entwickelte Anwendungen entstanden später in Form von Strategiespielen, Wirtschaftssimulationen und Rollenspielen, die sich bis heute als Browserspiele im gemeinen Sinne durchgesetzt haben. So stehen hier in der Regel Handel mit verschiedenen Waren und/oder die Navigation militärischer Einheiten – sei es im Weltraum oder in Märchenwäldern – im Mittelpunkt.

Einer der ersten populären Vertreter seiner Art ist die Science-Fiction-Wirtschaftssimulation SOL [2], die bereits 1995 veröffentlicht wurde und bis heute noch von mehreren tausend Spielern gespielt wird [54].

[2] Hypertext Transfer Protocol
[3] „Asynchronous JavaScript and XML“ [50]
[4] meist „LAMP“ (Linux, Apache, MySQL, PHP/Perl)

Weltweit nehmen mindestens mehrere Millionen Menschen an Browserspielen teil. Allein das Spiel „Die Renaissance Königreiche“ [16] zählt laut seiner Startseite schon weit über zweieinhalb Millionen registrierte Teilnehmer.

2.1.3 Der Gemeinschaftsspielcharakter von MMOGs und Browserspielen

Das typische Merkmal bei Browserspielen ist der Gemeinschaftsspielcharakter, der gerade durch die technischen Gegebenheiten des Internets besonders begünstigt wird. So nehmen an einem erfolgreichen Browserspiel meist Hunderte, manchmal sogar Tausende bis Hunderttausende von Spielern parallel teil. Das deutschsprachige Spiel „Die Stämme“ zählt derzeit weit über 600.000 Teilnehmer [29].

Besonders wichtig ist in dieser Hinsicht vor allem die Kommunikation zwischen den Spielern. Browserspiele stellen in der Regel spezielle Kommunikationswege wie Chats und Kurznachrichtensysteme bereit. Auch aus diesem Aspekt sind NPC[5] für den Spielspaß wenig förderlich. Es existieren zwar angeblich Bemühungen[6], verbale, schriftliche Kommunikation zwischen Mensch und Maschine in Browserspielen unterzubringen. Jedoch ist so ein Vorhaben entweder mit einem ungeheuren Aufwand verbunden, oder das Ergebnis wird – zumindest beim heutigen Stand der Technik – sehr einseitig, eintönig und emotionslos ausfallen. Die wohl bekannteste und zugleich älteste Realisierung eines Systems zur verbalen Interaktion ist das 1966 von Joseph Weizenbaum entwickelte Programm ELIZA, welches ehrgeizige, gewinnstrebige Spielteilnehmer wohl eher nerven als beglücken dürfte. Im Rahmen dieser Arbeit werden wir nicht näher auf solche Systeme eingehen, sondern uns auf die automatisierte Steuerung von Einheiten konzentrieren. Weitere Informationen zu ELIZA sind [49] sowie [53] zu entnehmen.

Anders als in den meisten anderen gängigen Browserspielen hat der Spieler in dem Spiel „Die Kreuzzüge“ [37] ein konkretes Ziel vorgegeben. Er muss als erster über einen Ozean segeln und dann auf dem erreichten Kontinent eine oder mehrere Burgen für eine bestimmte Zeit halten. Der Spieler darf eine Seite des Ozeans – Norden oder Süden – wählen und bekommt zu Beginn eine Burg. Um das Spielziel zu erreichen schließt sich der Spieler mit anderen Teilnehmern in der Nähe[7] zu einer Allianz zusammen. Laut Handbuch des Spiels haben Einsteiger, die dieses nicht tun, eine sehr geringe Überlebenschance und praktisch keine Aussicht darauf, die Runde zu gewinnen. Browserspiele leben von der Gemeinschaft. Sie setzen häufig auf das Gruppieren vieler Spieler zu „Banden“, „Clans“, „Syndikaten“

[5] „Non-player character“

[6] Der Schöpfer des Spiels „Andromeda“ [33], das noch immer im Entwicklungsstadium und daher nicht öffentlich erreichbar ist, teilte dem Verfasser dieser Arbeit vor einiger Zeit mit, er arbeite an einer „ELIZA“-ähnlichen Implementierung zur Kommunikation in seinem Browserspiel.

[7] ausgehend von seiner Burg

oder „Allianzen“, wie im genannten Beispiel, und auf Absprachen und informellen Regeln innerhalb und zwischen den Gruppen.

Um große Zahlen an Spielern, welche gleichzeitig untereinander interagieren, parallel zu bedienen, wird der Verlauf der meisten Spiele in Runden organisiert. Der Übergang von einer Runde zur nächsten bezeichnet man gemeinhin als „Tick“. Bei Star Trek Universe finden Ticks 5 Mal täglich im 3 Stunden-Abstand statt. Die nächtliche Runde dauert dann entsprechend 12 Stunden [31]. Zu jedem Tick werden beispielsweise von Kraftwerken generierte Energien gutgeschrieben, Nahrungsmittel in den Kolonien verbraucht oder sich in der Konstruktion befindende Einheiten fertiggestellt.

Gegenüber anderen Genres wie Action-, Sport-, Geschicklichkeits- und Abenteuerspielen sind Browserspiele sehr langfristig ausgelegt und laufen oft Jahre. Das oben genannte Spiel [2] läuft schon seit 14 Jahren und ist heute immer noch aktiv.

2.1.4 Abgrenzung, Browserspiele im klassische Sinn, Browser-Plugins und -Erweiterungen

Sowohl dank der heutigen, ausgefeilten Scripting-Unterstützung der verbreiteten Browser, als auch der Möglichkeit diese mittels Plugins wie Adobe Flash [5], Sun Java [43] oder spezielleren Erweiterungen wie zum Beispiel Unity [45] zu erweitern, lassen sich beliebige Arten von Spielen im Browser ausführen. Es existieren mittlerweile Emulatoren für alte 8-Bit-Systeme, sowohl in Flash als auch in Java, unter denen sämtliche Spiele, die für die damaligen Systeme entwickelt wurden, gespielt werden können. Somit sind praktisch alle populären Spiele bis in die späten 80er Jahre im Webbrowser spielbar. Diese Arbeit beschäftigt sich nicht mit beliebigen Spielen, die im Browser lauffähig sind, sondern ausschließlich mit solchen, deren Logik hauptsächlich serverseitig abläuft und die sich den genannten Techniken höchstens zusätzlich aus Gründen der Benutzerfreundlichkeit bedienen.

2.2 Ausgewählte Methoden der KI/CI

In diesem Abschnitt soll kurz auf einige gängige Methoden der künstlichen Intelligenz und der Computational Intelligence eingegangen werden, deren Einsatz sich in MMOG-Browserspielen anbietet oder solche Spiele bereichern könnte.

2.2.1 Regelbasierte Systeme

Oben wurde bereits angedeutet, dass in Browserspielen Gegner in Form von NPCs weder unbedingt nötig, noch besonders kommunikativ sind. Die Kernidee, die hinter dieser Arbeit steht, beschreibt im Grunde einfache regelbasierte Systeme, die durch den Spieler aktiv erstellt und zur Anwendung gebracht werden.

if $|\{x|feindlich_u(x) \wedge sichtbar_u(x)\}| \geq 5$ **then**
  $Aktion_u \leftarrow fliehen$
end if

Algorithmus 2.1: Ein einfaches Beispiel für eine Regel in einem regelbasierten System

Regelbasierte Systeme, die sich seit den 1970/1980er Jahren bei Spieleentwicklern größter Beliebtheit erfreuen [34], sind im Prinzip kleine Logikprogramme, die aus einer Liste von Wenn-/Dann-Anweisungen bestehen. In Algorithmus 2.1 ist ein einfaches Beispiel für eine Regel zu sehen. Wenn sich fünf oder mehr feindliche Einheiten im Sichtbarkeitsradius der Einheit u befinden, so soll diese zur Aktion „Fliehen“ übergehen.

Regelbasierte Systeme können datengetrieben (engl. „forward chaining“[8]) oder zielgetrieben (engl. „backward chaining“[9]) sein [15]. Der verbreitetste Ansatz ist die Vorwärtsverkettung. Hier wird in drei Phasen vorgegangen: Zunächst wird versucht, passende Regeln für die im Arbeitsspeicher gegebenen Fakten zu finden. Dies geschieht, indem getestet wird, ob die Prämisse[10] der Regel zutrifft. Gibt es mehrere passende Regeln, so geht das System in die „Konfliktlösungsphase“ über. Nun werden sämtliche passende Regeln betrachtet und die passende ausgewählt. In der Praxis wird häufig einfach die erste passende Regel akzeptiert. Manchmal ist es auch sinnvoll, die Regel zufällig auszuwählen oder Regeln zu gewichten und danach zu sortieren. In der letzten Phase wird die gewählte Regel angewendet. Die Rückwärtsverkettung arbeitet genau umgekehrt. Zunächst wird ein erstrebenswertes Ziel gesucht, die Konklusion[11]. Dann werden diejenigen Regeln ausgewählt, die nötig sind, um das Ziel aus der aktuell gegebenen Situation zu erreichen.

Wenn es eine Standardtechnik der künstlichen Intelligenz in Computerspielen gibt, dann handelt es sich um regelbasierte Systeme. Fast alle Genres der Branche[12] bedienen sich dieser Technik seit es Computerspiele gibt. Ihr heftet zwar der Ruf an, sie sei ineffizient und umständlich zu implementieren, nicht zuletzt, weil ähnliche Lösungen meist mit Hilfe von Entscheidungsbäumen oder Automaten gefunden werden können [34]. Trotzdem handelt sich hier um die bekannteste und beliebteste Methode künstliche Intelligenz in Computerspielen zu implementieren. Als bekannte Vertreter für den Einsatz regelbasierter Systeme in Computerspielen seien etwa das beliebte „Age of Empires“ und die „Total War“-Reihe genannt. Meist finden regelbasierte Systeme implizit Verwendung in Computerspielen, ohne dass der Spieler davon weiß.

Die in Abschnitt 1.3 beschriebene Grundidee dieser Arbeit von programmierbaren Agenten basiert ebenfalls auf der Grundlage datengetriebener regelbasierter Systeme. Auch

[8] auch als „Vorwärtsverkettung“ bezeichnet
[9] auch als „Rückwärtsverkettung“ bezeichnet
[10] die Bedingung, der „IF“-Teil einer Regel
[11] der „THEN“-Teil einer Regel
[12] Regelbasierte Systeme werden auch außerhalb der Computerspielebranche in vielen „ernsten“ technischen Anwendungen eingesetzt.

bei physischer Abwesenheit des Spielers werden im Hintergrund taktische Aufgaben übernommen, sowie Spielelemente bzw. Einheiten gesteuert. Programme, die im Folgenden als „Taktiken“ bezeichnet werden, können vom Spieler erstellt und Einheiten zugewiesen werden. Es existieren Regeln, die in jedem Fall ausgeführt werden, im Kontext als „Aktionen“ bezeichnet, Variablenzuweisungen und bedingte Sprünge. Als mögliche Prämissen kann der Taktik-Entwickler Ausdrücke in Form von Gleichungen angeben. Hierzu stehen mehrere vorgegebene Variablen zur Verfügung, die dann mit anderen Ausdrücken verglichen werden, vergleichbar zur Prämisse in Algorithmus 2.1 auf der vorherigen Seite. Taktiken selbst können an Bedingungen geknüpft sein, die diese ereignisgesteuert, abhängig von der Situation auslösen, sogenannte „Trigger“. Insbesondere in Verbindung mit Auslösern und Bedingungen kann sich Fuzzy Logic als sehr nützlich erweisen, auf die im nächsten Unterabschnitt eingegangen wird. Auf Taktiken und deren Möglichkeiten und Implementierung in der Test- und Beispielanwendung „Megaira“ wird in Abschnitt 3.1 detailliert und praxisorientiert eingegangen.

2.2.2 Fuzzy Logic

Betrachten wir mögliche Prämissen genauer, so fällt auf, dass es nicht immer trivial ist, welche Bedingung in einer beliebigen Situation die richtige Aktion auslösen wird. In der Beispielregel in Algorithmus 2.1 auf der vorherigen Seite wird etwa angenommen, dass eine Situation genau dann gefährlich ist, wenn sich mindestens fünf feindliche Einheiten in Sichtweite des Subjekts befinden, so dass die Einheit flieht. In der Spielrealität sind oft komplexere Perspektiven wünschenswert. Beispielsweise könnten hier zusätzlich verschiedene weitere Gegebenheiten, wie die Anzahl der befreundeten Einheiten im Umkreis, die Ausrüstung (insbesondere Waffen und Schilde), der Grad der Beschädigung, geladene Güter, Treibstoff etc. betrachtet werden. Eine solche Detailtreue würde auf der anderen Seite das manuelle Modellieren von Taktiken sehr viel komplexer gestalten und mächtigere Eingabemöglichkeiten für Bedingungen und Ausdrücke voraussetzen.

Hier bietet es sich an, auf die sogenannte „Fuzzy Logic“ zurückzugreifen. Die „Theorie der Fuzzy Mengen“ wurde 1965 Lotfi Zadeh an der University of California Berkeley entwickelt und wird von ihm wie folgt erklärt [15]:

> Fuzzy logic is a means of presenting problems to computers in a way akin to the way humans solve them. [...] the essence of fuzzy logic is that everything is a matter of degree.

Auf Deutsch sinngemäß etwa:

> Fuzzy Logic ist ein Mittel, Computern Probleme so ähnlich zu präsentieren, wie Menschen sie lösen. [...] das Wesen der Fuzzy Logic ist, das alles eine Frage des Maßes ist.

Gegenüber der klassischen, booleschen Logik kennt die Fuzzy Logic nicht nur *wahr* ($x = 1$) und *falsch* ($x = 0$), sondern auch den fließenden Übergang dazwischen ($x \subseteq [0, 1] \subset \mathbb{R}$), wodurch sich auch Werte wie „ziemlich wahr“ oder „fast ganz falsch“ modellieren lassen.

Der Fuzzy-Inferenzprozess verläuft in drei[13] Stufen. Die erste Stufe nennt man „Fuzzifizierung“, hier werden scharfe Eingaben („Crisp Input“), etwa eine konkrete gegebene Körpergröße in cm, in unscharfe Ausprägungen („Fuzzy Input“), im Beispiel eine etwaige Einschätzung der Körpergröße („klein“, „mittelgroß“, „groß“, „sehr groß“, „riesig“), umgewandelt. Dies geschieht mit Hilfe vordefinierter Mengen, sogenannter Fuzzymengen. In der zweiten Stufe, der „Kombination“, wird nun mit Fuzzy-Regeln auf den (in der Regel mehreren) unscharfen Eingaben gearbeitet. Hier werden die verschiedenen fuzzifizierten Eingabewerte kombiniert betrachtet. Im Beispiel könnte das die Zusammenstellung „kleine Körpergröße“, „eher fettleibig“, „sportlich wenig aktiv“ sein. Das Ergebnis der zweiten Phase nennt man „Fuzzy Output“, im betrachteten Beispiel würde hier beispielsweise der Wert „etwas mehr Sport treiben“ stehen, welcher dann in der letzten Stufe mittels „Defuzzifizierung“ zurück in einen scharfen, exakten numerischen Wert gewandelt wird. Je nachdem, wie wir die Regeln ansetzen, könnte das im Beispiel „5× in der Woche Sport treiben“ sein.

Aus mathematischer Perspektive ist die Fuzzy Logic – wie die boolesche Logik – leicht über einen mengentheoretischen Ansatz zu veranschaulichen. Während bei der booleschen Logik ein Element x entweder genau oder genau nicht in einer „scharfen“ Menge A enthalten ist, formal Gleichung (2.1) zu entnehmen, so kann in der Fuzzy Logic ein Element zu einem gewissen Grad einer oder mehreren Mengen zugehörig sein. Das Maß $F(x)$, in dem ein Element x zu einer Menge F gehört, wird durch eine Zugehörigkeitsfunktion $F : X \to [0, 1] \subset \mathbb{R}$ ausgedrückt [40].

$$A(x) := 1_{x \in A} := 1_{A(x)} := \begin{cases} 1 & \text{, falls } x \in A \\ 0 & \text{, falls } x \notin A \end{cases} \tag{2.1}$$

Sehr viel anschaulicher werden die Fuzzymengen, wenn wir ihre Zugehörigkeitsfunktionen graphisch veranschaulichen. Die Zusammenfassung von Zugehörigkeitsfunktionen bezeichnet man als Fuzzyfunktion. Beispielgraphen zum oben angegebenen Beispiel der Körpergrößen sind in Abbildung 2.1 auf der nächsten Seite zu sehen. Hier werden Menschen von einer Körpergröße bis 170cm als nicht groß ($F_{groß}(170) = 0$), bis 180cm zunehmend als groß, von 180cm bis 190cm als groß ($F_{groß}(x) = 1,\ \forall x \in [180, 190]$) und darüber abnehmend als groß eingestuft.

Um auf Fuzzymengen zu arbeiten, die zweite Stufe des Fuzzy-Inferenzprozesses, führen wir nun kurz einige Operationen ein [40], wobei A, B und C unscharfe Mengen über X sind:

[13] hier zum besseren Verständnis leicht vereinfacht, der Prozess lässt sich auch noch etwas feiner aufgliedern

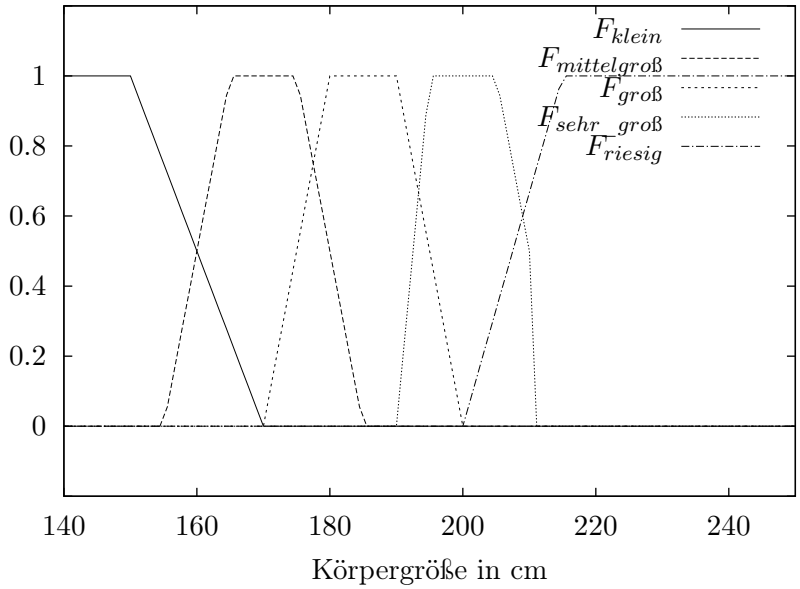

Abbildung 2.1: Fuzzyfunktion für Körpergrößen

1. **Vereinigung**

$$C := A \cup B \text{ mit } C(x) := \max\{A(x), B(x)\} \; \forall x \in X \tag{2.2}$$

2. **Durchschnitt**

$$C := A \cap B \text{ mit } C(x) := \min\{A(x), B(x)\} \; \forall x \in X \tag{2.3}$$

3. **Komplement**

$$C := A^C \text{ mit } C(x) := 1 - A(x) \; \forall x \in X \tag{2.4}$$

Mit Hilfe der definierten Operationen lassen sich nun ähnlich zur booleschen Logik Fuzzymengen zu Ausdrücken verknüpfen, so dass sich Regeln in der Form ähnlich zu *wenn A und B dann C* formulieren lassen. Da wir hier nun auf einem unscharfen Wertebereich arbeiten, erhalten wir gemäß den oben stehenden Definitionen kein klares *wahr* oder *falsch* sondern einen Wert $x \in [0, 1] \subset \mathbb{R}$, den es dann später zu interpretieren gilt. Um das Beispiel vom Anfang aufzugreifen, lässt sich folgende Gleichung aufstellen:

$$F_{sport\_treiben}(x) = F_{klein}(x) \cup F_{dick}(x) \cup F^C_{aktiv}(x)$$

Setzen wir nun den Zustand einer gegebenen Person x, näher spezifiziert durch $F_{klein}(x) = 0,6$, $F_{dick}(x) = 0,7$, $F_{aktiv}(x) = 0,25$, in die Formel ein, so erhalten wir:

$$F_{sport\_treiben}(x) = \max\{0.6, 0.7, 1 - 0,25\} = 0,75$$

Für Vereinigung, Durchschnitt und Komplement lassen sich neben den in Gleichung (2.2) auf der vorherigen Seite, Gleichung (2.3) und in Gleichung (2.4) eingeführten Operatoren auch viele weitere sinnvolle Operatoren definieren. Die Verallgemeinerung der Fuzzy-Schnittmengenoperation nennt man T-Norm, die verallgemeinerte Fuzzy-Vereinigungsoperation wird S-Norm oder T-Conorm genannt [40], auf die hier aber nicht näher eingegangen werden soll.

Auf Basis der Formel wird der Person in unserem Beispiel nahegelegt, 75% Sport zu treiben. Diesen Wert gilt es in der Defuzzifizierung, also der letzten Phase des Inferenzprozesses, zu interpretieren. Dies kann über eine einfache Funktion passieren. Nehmen wir an, es ist der Testperson möglich jeweils an jedem Tag einmal Sport zu treiben, so wäre eine mögliche Defuzzifizierungsfunktion die folgende[14]:

$$\lfloor F_(sport\_treiben)(x) * 7 \rfloor = \lfloor 0,75 \cdot 7 \rfloor = \lfloor 5,25 \rfloor = 5$$

Dem Subjekt sei also aufgrund seiner körperlichen Gegebenheiten zu fünf mal in der Woche Sport geraten. Je nach Anwendung existieren verschiedene Möglichkeiten der Defuzzifizierung. Benötigt man ein schlichtes Ja-/Nein-Ergebnis, also einen booleschen Wert, so bestimmt man häufig eine Grenze α, ab der das Ergebnis *wahr* ist, den so genannten α-Schnitt [40].

$$A^{\geq\alpha} := \{x \in X \mid A(x) \geq \alpha\}$$

Das Ergebnis für den errechneten Wert x ist für ein vorher festgelegtes α genau dann *wahr*, wenn $x \in A^{\geq\alpha}$.

Fuzzy Logic ist ein sehr praktisches Modell, um die unscharfe Denkweise von Menschen zu modellieren. Gerade wenn programmierbare Agenten entscheiden sollen, wie sich die Situation der zu steuernden Einheit verhält – etwa bezüglich der Einheit von Machtverhältnissen oder der Detektion von Notständen und Ressourcenengpässen – und welche auszuführende Taktik als Folge den größten strategischen Erfolg verspricht, bieten sich Verfahren auf Grundlage dieser Theorie an. Detaillierte Ausführungen zum Thema Fuzzy-Mengen und Fuzzy Logic sind dem Buch [32] zu entnehmen.

2.2.3 Machine Learning

Maschinelles Lernen bezeichnet allgemein das erfahrungsbasierte, künstliche Generieren von Wissen. Ein entsprechendes System wird in der Lernphase mit Beispielen gefüttert

[14] ähnlich zu, aber etwas einfacher als im Beispiel in [15]

und versucht auf deren Basis Regeln abzuleiten und zu verallgemeinern. Gesetzmäßigkeiten werden automatisch erkannt und können nach Abschluss der Lernphase automatisch angewendet werden.

In der Praxis existieren für das maschinelle Lernen sehr viele algorithmische Ansätze, die sich grob in folgende Kategorien einteilen lassen, wie in [36] sowie in [6] beschrieben:

1. **Überwachtes Lernen** (engl. „supervised learning“)

   Das System lernt durch einen externen Lehrer, welcher es in der Lernphase mit Beispielpaaren von Eingaben und den gewünschten passenden Ausgaben „füttert“. Ziel ist es, dass nach Abschluss der Trainingsphase eingehende Daten selbständig durch das System klassifiziert werden können.

2. **Unüberwachtes Lernen** (engl. „unsupervised learning“)

   Hier wird das System nur mit Eingaben gespeist, zu denen – im Gegensatz zum überwachten Lernen – keine erwünschten Ausgaben vorliegen. Der Algorithmus versucht nun, selbständig die eintreffenden Werte in Gruppen (engl. „Cluster“) zu unterteilen (engl. „Clustering“). Unüberwachtes Lernen eignet sich besonders für Anwendungsfälle, in denen Vorhersagen getroffen werden müssen.

3. **Bestärkendes Lernen** (engl. „reinforcement learning“)

   Beim bestärkenden Lernen entwickelt der Algorithmus sein Wissen durch Belohnung und Bestrafung, er lernt ausschließlich durch Interaktion mit der Umwelt. Für jede ausgeführte Aktion wird ermittelt, ob sie die Situation in Hinsicht auf ein gegebenes Ziel verbessert oder verschlechtert hat. Das Ziel ist hier, eine Strategie zu entwickeln, welche stets die Aktion wählt, die die Kosten (Bestrafungen) minimiert und Nutzen (Belohnungen) maximiert.

Dynamische Skripte

Dynamische Skripte basieren zum Teil auf regelbasierten Systemen, die wir bereits zu Beginn von Abschnitt 2.2 betrachtet haben, zu einem anderen Teil auf evolutionären Algorithmen und bestärkendem Lernen [41]. Hier werden Skripte – Sequenzen von Regeln – während der Laufzeit eines Spiels adaptiert, wie in [38] speziell für Strategiespiele vorgestellt wurde. Da die statischen Skripte in den meisten kommerziell erfolgreichen Spielen sehr lang und kompliziert sind[15], enthalten sie häufig Fehler und Schwächen, die vom Spieler nach einiger Zeit entdeckt und ausgenutzt werden können. Statische Skripte besitzen allerdings keinen Mechanismus dies zu erkennen und entsprechend zu reagieren. Dynamische Skripte realisieren laut Ponsen „die Online-Adaption geskripteter Gegner-KI“, lassen sich also dem Online-Lernen unterordnen. Online-Lernen bezeichnet den Vorgang, dass

[15] einfache Skripte sind hingegen sehr leicht durchschaubar

sich die künstliche Intelligenz während dem Spiel gegen den Menschen selbst bewertet und anpasst, während sich das System beim Offline-Lernen ohne menschliches Eingreifen gegen sich selbst weiterentwickelt.

Dynamische Skripte nach Ponsen [38] selektieren im ersten Schritt zufällig so lange Regeln aus einem vorgegebenen Regelbasis-Repertoire bis eine auf den aktuellen Zustand passt und zu einem Übergang führt. Das Vorgehen wird anschließend für den neuen Zustand wiederholt. Die zugrundeliegende Regelbasis besteht aus manuell mit Hilfe von bereichsspezifischem Wissen gefertigten Regeln. Jede Regel bekommt ein Gewicht, welches die Wahrscheinlichkeit, dass diese Regel ausgewählt wird, proportional beeinflusst. Nach jedem Gefecht wird die Regel aufgrund ihres Resultats in Hinblick auf das Spielziel neu gewichtet. Um Monotonie zu verhindern darf jede Regel für jeden passenden Zustand nur einmal angewendet werden. Sobald das Ende des Skripts erreicht wurde, geht es dazu über, durchgehend endlos Angriffe auf den Gegner auszuüben.

Zwar ist es momentan nicht vorgesehen, dass die Skripte in der Beispielanwendung „Megaira“ selbständig im laufenden Spiel lernen und sich adaptieren, doch ist es prinzipiell sehr leicht möglich ein entsprechendes System einzugliedern. Entsprechende Mittel sind in Kapitel 3 ausgeführt. Für die Experimente in Abschnitt 4.1 wurde eine kleine API entwickelt, die es erlaubt, Einheiten zu erstellen, Taktiken zuzuweisen, diese wiederholt auszuführen und gegebenenfalls zwischen den Durchläufen Parameter zu verändern. Hierzu mehr in Abschnitt 3.4.

2.2.4 Evolutionäre Algorithmen

Evolutionäre Algorithmen (kurz „EA“) sind Algorithmen, die in einer Weise lernen, welche mit der biologischen Evolution vergleichbar ist. Ein evolutionärer Algorithmus versucht zunächst einfache Lösungen für ein gegebenes Problem zu finden, im Algorithmus „Individuen“ genannt, welche dann hinsichtlich der Zielerreichung bewertet werden [36]. Diese Bewertungsmethode wird auch als Fitnessfunktion bezeichnet. Aus der Population, der Menge der bewerteten Lösungen, werden nun einige Individuen gruppenweise herausgenommen und zu neuen Individuen kombiniert. Diesen Vorgang bezeichnet man als „Rekombination“ (engl. „Crossover“). Die gewonnenen potenziellen Lösungen werden im Anschluss verändert („Mutation“) und analog zu den ursprünglichen Individuen neu bewertet. Die besten Lösungen werden ausgewählt („Selektion“) und anschließend zur neuen Population, auf der der Algorithmus wiederholt wird. Dies geschieht so lange bis ein vorgegebenes Kriterium für die Terminierung erfüllt ist. Prinzipiell handelt es sich hierbei also um ein Optimierungsverfahren. Evolutionäre Algorithmen werden sowohl für Permutationsprobleme, als auch für diskrete Probleme und auch für reellwertige Probleme verwendet. Eine tiefergehende Einführung in das Gebiet der evolutionären Algorithmen bietet [23].

2.2.5 Wegfindung

Wegfindung ist ein elementares Problem bei der Computerspielentwicklung, welches meistens mit dem A*-Algorithmus oder einer Abwandlung desselben gelöst wird. Ziel ist es hierbei, einen möglichst günstigen Weg von einem Punkt zu einem anderen zu finden. Der A* Algorithmus wurde erstmals 1968 in [27] beschrieben. Algorithmus 2.2 zeigt eine Skizze der Funktionsweise nach [15].

Eingabe: Startknoten s, Zielknoten g

Ausgabe:

1: $O \leftarrow \{s\}$ // Liste der noch offenen Knoten
2: $C \leftarrow \emptyset$ // Liste der bereits untersuchten Knoten
3: **while** $O \neq \emptyset$ **do**
4: $c \leftarrow$ Knoten aus O mit geringsten Kosten
5: **if** $c = g$ **then**
6: **return** // Pfad gefunden
7: **else**
8: $C \leftarrow C \cup \{c\}$
9: **for all** a mit $adjazent(a, c)$ **do**
10: **if** $a \notin O \ \wedge \ a \notin C \wedge$ frei(a) **then**
11: $O \leftarrow O \cup \{a\}$
12: berechne Kosten für a
13: **end if**
14: **end for**
15: **end if**
16: **end while**

Algorithmus 2.2: A*-Algorithmus

Das Spiel „Megaira“ ist ein kachelbasiertes Spiel, in dem jedes Feld von höchstens einer Einheit besetzt sein darf. Einheiten können gegnerische Einheiten blockieren, indem sie diese umzingeln oder sich ihnen in den Weg stellen. Aus diesen Gründen ist für das Spiel ein Wegfindungsalgorithmmus, hier eine leicht abgewandelte Form des in Algorithmus 2.2 beschriebenen A*-Algorithmus, unverzichtbar. Zwar sieht der Spieler nicht genau den Weg, den die Einheit zurücklegt, allerdings werden hierüber die Kosten berechnet und der bewegten Einheit entsprechend Treibstoff abgezogen. Auch für die Benutzerschnittstelle wird eine weitere Variante genutzt, um die von der aktuell gewählten Einheit erreichbaren Fel-

der zu berechnen und hervorzuheben. Sie wird angewendet, falls die Aktion „bewegen“ ausgewählt wird, wie in Abbildung 3.18 auf Seite 40 gezeigt wird[16].

[16] In Megaira dürfen sich Einheiten auch diagonal bewegen, daher die Erreichbarkeit aller freien Felder in einem quadratischen Gebiet um die Einheit.

Kapitel 3

Architektur und Ausstattung von Megaira

Um die in Kapitel 1 und Kapitel 2 vorgestellten und vorgeschlagenen Techniken und Ideen zu implementieren, zu testen und bewerten zu können, wurde für diese Arbeit eigens ein Browserspiel namens „Megaira“ konzipiert und implementiert. Das Spiel bildet eine gute Grundlage, um einige der genannten Methoden und Verfahren umzusetzen. Insbesondere die Idee der programmierbaren Agenten stand bei der Umsetzung der Methoden der künstlichen Intelligenz und der computational Intelligence im Mittelpunkt. Die Software liegt dieser Arbeit sowohl im Quellcode als auch binär bei, siehe hierzu Anhang A und B.

Dieses Kapitel geht genauer auf die technischen und strukturellen Eigenschaften, Methoden und Voraussetzungen der Beispielanwendung „Megaira“ ein. Zunächst werden in Abschnitt 3.1 Inhalt, Funktionsumfang und Aufbau der Anwendung beschrieben. Dann folgen Erläuterungen zur externen Architektur, also der eingesetzten Plattform und den Abhängigkeiten sowie den Bibliotheken in Abschnitt 3.2. In Abschnitt 3.3 wird schließlich detailliert auf das Klassenmodell und die Datenbankschemata eingegangen und die interne Funktionsweise erörtert.

3.1 Die Ausstattung von Megaira

Beim browserbasierten Strategiespiel „Megaira“ [28], welches speziell für diese Arbeit entwickelt wurde, handelt es sich in erster Linie um eine flexible Spieleplattform[1], mit der einfach verschiedene Browserspiele umgesetzt werden können. Für diese Arbeit halten wir uns an ein maritimes Szenario. Es gibt zwei verschiedene Schiffstypen, Bauschiffe („Builder“) und Patrouillenboote („Patrol Boat“), sowie Bohrinseln, „Oil Rig“. Bauschiffe sind mit Raketen bewaffnet, können Bohrinseln und Patrouillenboote bauen und sich, sofern sie genügend Öl im Tank haben, um bis zu 5 Felder je Runde bewegen. Patrouillenboote können

[1] im Sinne des verbreiteteren, englischen Begriffs „Engine“ bzw. „Gameengine“

je Runde die doppelte Entfernung zurücklegen und sind zusätzlich mit Maschinengewehren bewaffnet. Ölbohrinseln hingegen sind unbeweglich, können sich nicht verteidigen und fördern bei jedem Rundenwechsel 50 Einheiten Öl, die sie an andere Einheiten in Reichweite transferieren können.

Bei dem Spiel „Megaira“ existieren zwei Sorten von Ressourcen. Die wichtigste ist Öl, welches regelmäßig zu jedem Rundenwechsel durch die Bohrinseln gefördert wird und an Schiffe in Reichweite transferiert werden kann. Schiffe sind auf Öl angewiesen, um sich fortbewegen zu können. Zusätzlich existiert noch Munition in Form von Raketen und Patronen für Maschinengewehre.

Rundenwechsel finden bei Megaira rund um die Uhr im Abstand von jeweils 15 Minuten statt. Zu jedem „Tick“ werden Ressourcen abgerechnet, die rundenbasierte Einschränkung der möglichen zurücklegbaren Entfernung jeder Einheit zurückgesetzt, produzierte Güter gutgeschrieben und die nächsten Schritte aller aktiven und zugewiesenen Taktiken ausgeführt.

3.1.1 Datengetriebene Browserspiel-Plattform

Seine Flexibilität erlangt „Megaira“ durch sein offenes Datenmodell. Einheiten, Modelle (Typen für Einheiten), Aktionen, Ressourcen, Vorräte, Kapazitäten, Taktiken, Regeln für den Bau von neuen Einheiten durch bestehende Einheiten und viele weitere Details werden in der Datenbank gespeichert und sind beliebig veränder- und erweiterbar. Die grundlegenden Objekte des Spiels und die groben Zusammenhänge zwischen diesen sind in Abbildung 3.1 auf der nächsten Seite abgebildet. Neue Modelle für Einheiten mit neuen Fähigkeiten lassen sich sehr schnell und einfach anlegen. Ebenso ist das Hinzufügen von neuen Ressourcentypen sowie Bau- und Produktionsregeln sehr einfach und Bedarf keinerlei Eingriffe in den Programmcode. Je nach Art der Anpassung müssen gegebenenfalls einige Piktogramme und Grafiken hinzugefügt werden. Auf die Erweiterbarkeit von Megaira wird in Zusammenhang mit dem Klassenmodell und der Datenbankstruktur in Abschnitt 3.3 detailliert eingegangen.

3.1.2 Die Brücke

Nach der Anmeldung im Anmeldefenster des Spiels, in der sich der Spieler mit seiner persönlichen Kombination aus Benutzernamen und Passwort identifiziert, wird dieser zur Brücke weitergeleitet. Die Brücke ist die Hauptansicht von Megaira und in Abbildung 3.2 auf Seite 24 zu sehen. Von hier aus kann der Spieler komfortabel das Spielfeld erkunden, Einheiten auswählen und kommandieren, Informationen über eigene und feindliche Einheiten einholen und seinen eigenen Einheiten Taktiken zuordnen. Eigene Einheiten werden durch einen grünen, schwach leuchtenden Rahmen symbolisiert, fremde Einheiten sind rahmenlos.

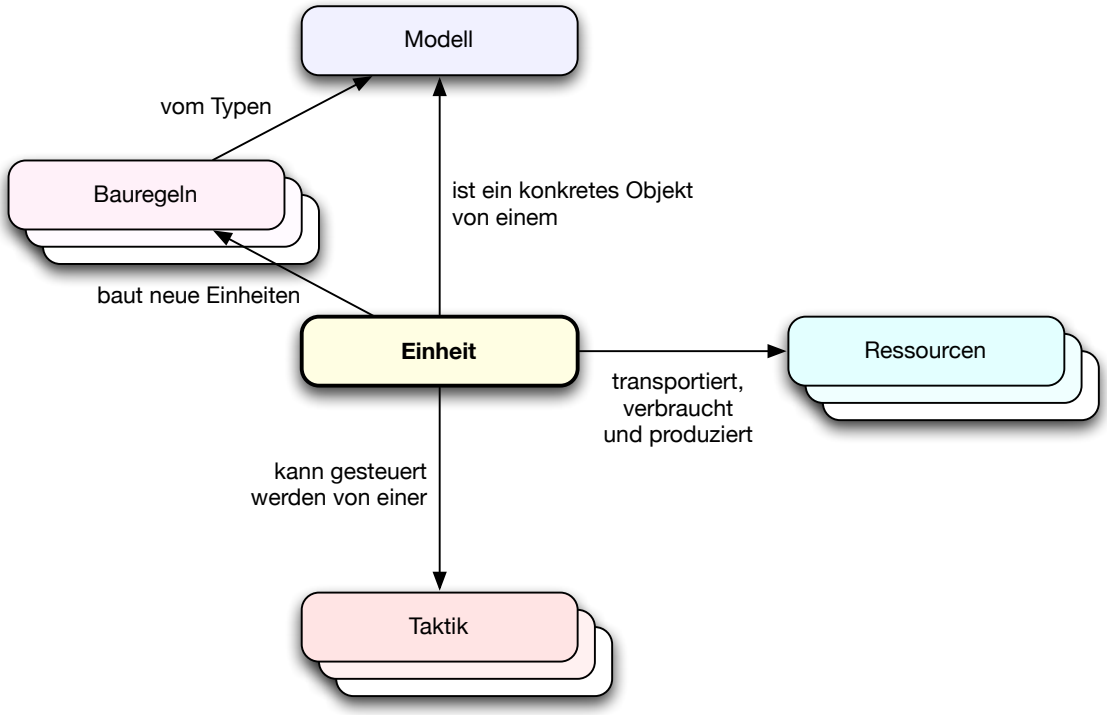

Abbildung 3.1: Der Zusammenhang der elementaren Spielobjekte in Megaira

Klickt der Spieler mit dem Mauszeiger auf eine Einheit, so wird diese selektiert und durch einen leuchtenden, dünnen Rahmen hervorgehoben. Handelt es sich um seine eigene Einheit, so ist der Rahmen grün gefärbt, bei feindlichen Einheiten hat dieser die Farbe Rot wie in Abbildung 3.3 auf der nächsten Seite abgebildet ist. Das Einheiten-Informationsfenster im unteren mittleren Bereich des Browserfensters wird daraufhin mit den Daten der gewählten Einheit aktualisiert. Dort sind neben einer vergrößerten Darstellung des Symbols für die Einheit ihr Name und ihr Besitzer sowie ihre Koordinaten und die geladenen Rohstoffe und Munition zu sehen.

Das Aktionsmenü in der rechten unteren Ecke des Fensters reflektiert die Aktionen, die durch den Spielteilnehmer von der gewählten Einheit ausgeführt werden können. In Abbildung 3.4 auf Seite 25 sehen wir die Aktionen, die ein gerade gewähltes Patrouillenboot ausführen kann. Die ersten beiden Punkte sind allgemein für jede Einheit verfügbar. Über das erste Symbol lässt sich der Bildschirmausschnitt um die gewählte Einheit zentrieren, mit dem zweiten Knopf wird die aktuell gewählte Einheit deselektiert. Das dritte Symbol aktiviert den Bewegungsmodus der Einheit. Über den letzten Knopf in Abbildung 3.4 auf Seite 25 gelangt man in ein Untermenü zur Waffenauswahl, um dann mit der gewählten Waffe eine gegnerische Einheit für den Angriff zu markieren.

Klickt man für eine ausgewählte Einheit im Aktionsmenü auf den „Bewegen“-Knopf, in Abbildung 3.4 auf Seite 25 der Pfeil oben rechts, so wird der Bewegungsmodus aktiviert.

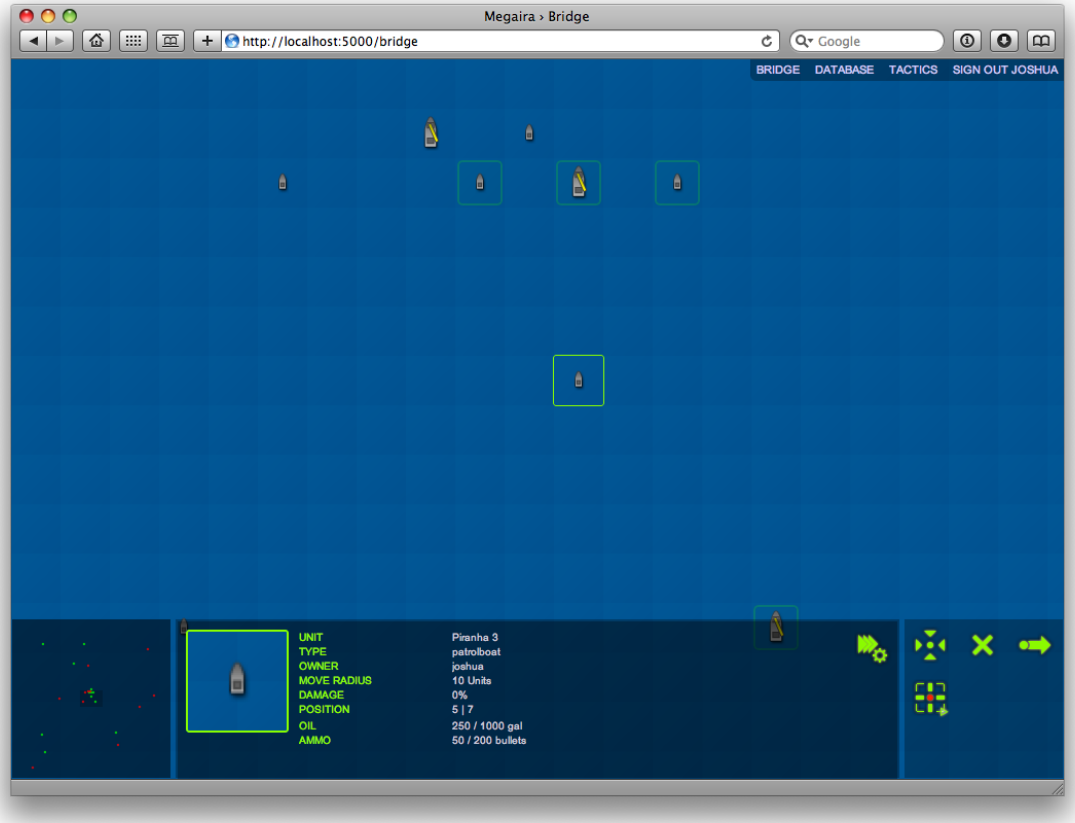

Abbildung 3.2: Die Brücke von „Megaira“: Spielfeld, Radar, Einheiten-Informationsfenster und Aktionsmenü

Abbildung 3.3: Die Brücke: auf der linken Seite hat der Spieler seine eigene Einheit ausgewählt, auf der rechten ist die eines Gegners selektiert.

Abbildung 3.4: Das Aktionsmenü: Ein ausgewähltes Patrouillenboot kann in der Ansicht zentriert, deselektiert, fortbewegt werden oder eine Einheit attackieren. Das kleine Dreieck im letzten Menüpunkt deutet auf ein Untermenü zur Waffenauswahl hin.

Abbildung 3.5: Das Aktionsmenü: Untermenü zur Waffenauswahl – zur Auswahl stehen ein Maschinengewehr und Raketen.

Wie in Abbildung 3.18 auf Seite 40 zu sehen ist, werden sämtliche erreichbaren Felder grün hervorgehoben. Der Akteur klickt nun auf das Feld, auf das sich die Einheit bewegen soll und das System führt das Kommando aus, zieht die benötigten Ressourcen von den Reserven des Bootes ab und aktualisiert den Kartenausschnitt in der Hauptansicht.

Angriffe auf gegnerische Einheiten lassen sich über den entsprechenden Knopf im Aktionsmenü – in Abbildung 3.4 das Fadenkreuz auf der linken Seite – initiieren, nachdem man im erscheinenden Untermenü die gewünschte Waffe ausgewählt hat. Das Untermenü zur Waffenauswahl ist in Abbildung 3.5 dargestellt. Hier hat der Spieler die Wahl zwischen einem Maschinengewehr und Raketen. Ist die gewünschte Waffe ausgewählt, so werden alle erreichbaren Ziele rot hervorgehoben, wie in Abbildung 3.6 auf der nächsten Seite zu sehen ist. Der angreifende Spielteilnehmer klickt nun auf die zu attackierende Einheit, und die Aktion wird durchgeführt.

Ressourcentransfers in Megaira funktionieren ähnlich wie die zuletzt beschriebenen Angriffe. Über das Aktionsmenü in Abbildung 3.7 auf der nächsten Seite wird der Ressourcentransfer initiiert und die zu übertragende Ressource ausgewählt. Im Anschluss erscheint ein Dialog, in den die Anzahl der zu transferierenden Einheiten des Rohstoffs einzugeben

Abbildung 3.6: Die Brücke: Auswahl der Zieleinheit zum Angriff. Das Patrouillenboot rechts (grün umrandet) kann zum Angriff zwischen den drei rot unterlegten gegnerischen Schiffen wählen.

Abbildung 3.7: Die Brücke: Ressourcentransfer im Aktionsmenü. Links das Aktionsmenü der Ölbohrinsel, rechts das Untermenü mit der Auswahl der zu übertragenden Ressource

Abbildung 3.8: Die Brücke: Der Bau von Einheiten durch Bauschiffe

ist. Schließlich werden sämtliche empfangsbereiten Einheiten hervorgehoben, so dass der Spieler – analog zur Zielauswahl bei den Angriffen – die empfangende Einheit auswählen kann.

Der Schiffsbau durch Einheiten mit entsprechenden Fähigkeiten, im Beispielszenario die Bauschiffe, welche Ölbohrinseln und Patrouillenboote bauen können, wird nach einem ähnlichen Schema wie bei den bereits beschriebenen Aktionen durchgeführt. Der Spielteilnehmer wählt im Aktionsmenü das Symbol „Bauen“ (ein kleiner Hammer), dann im Untermenü die zu konstruierende Einheit und schließlich ein freies Feld im Umkreis, auf dem die Einheit gebaut wird, dargestellt durch gelbe Kästchen wie in Abbildung 3.8.

Taktiken kann man Einheiten zuweisen, indem man auf das entsprechende Symbol oben rechts im Informationsfenster (links vom Aktionsmenü) klickt und dann aus der erscheinenden Auswahlliste die gewünschte Taktik auswählt. Für die ausgewählte Einheit können nur Taktiken angezeigt und zugewiesen werden, die dem angemeldeten Spieler gehören und speziell für das Modell der betreffenden Einheit deklariert wurden. Da nur stets eine Einheit gleichzeitig ausgewählt sein kann, sind Massenzuweisungen von Taktiken

nicht möglich. Nähere Informationen zu Taktiken und deren Funktionsweise im nächsten Unterabschnitt und in Abschnitt 3.3.

Die Radaransicht unten Links, siehe Abbildung 3.17 auf Seite 39, zeigt einen Ausschnitt des theoretisch unbegrenzt großen Spielfelds in einem sehr großen Maßstab, so dass der Spieler leicht die Lage in der Umgebung überblicken kann. Klickt man auf eine beliebige Position in der Radaransicht, so zentriert sich der sichtbare Ausschnitt im Hauptfenster, sowie die Radaransicht um die angeklickten Koordinaten.

Im unteren mittleren Bereich befindet sich rechts oben in der Nähe des Aktionsmenüs ein Knopf, über den der aktuell ausgewählten Einheit eine vorher im Taktikeditor erstellte Taktik zugewiesen werden kann (ein dreifacher Pfeil mit einem kleinen Zahnrad).

3.1.3 Taktiken und der Regeleditor

Wie bereits angedeutet, lassen sich in Megaira Einheiten automatisieren, indem der Spielteilnehmer Taktiken erstellt und diese den Einheiten zuweist. Um die Lernkurve möglichst flach zu halten und einer breiten Zahl von Spielern den Einsatz von Taktiken zu ermöglichen – auch denjenigen, die keine Programmierer sind oder werden möchten – bietet das Spiel einen Regeleditor an, welcher es erlaubt beliebige Taktiken einfach mit der Maus zusammenzubauen. Der Taktikeditor mit einer kleinen Beispieltaktik ist in Abbildung 3.9 auf der nächsten Seite abgebildet.

Taktiken erlauben es dem Spieler, Einheiten automatisiert zu steuern, auch während seiner physikalischen Abwesenheit. Gesteuert werden können selbstverständlich nur diejenigen Einheiten, die dem Spieler selbst gehören. Da einzelne Schritte – genauer gesagt nur die Aktionsschritte der Taktiken – Kosten verursachen und die Ausführung der Taktiken an die Rundenwechsel des Spiels, die alle 15 Minuten stattfinden, gekoppelt sind, werden Taktiken im Alltag nicht in jedem Fall der klügste und schnellste Weg zur Zielerreichung sein. Die Fähigkeiten der Taktiken können allerdings in einigen speziellen Situationen die Fähigkeiten eines Spielers übertreffen. Gerade in unübersichtlichen Szenarien kann ein Taktikschritt schnell eine sichere Position oder auch das Freund-/Feindverhältnis zuverlässig berechnen. Eine Angriffsaktion sucht – sofern die Einheit über mehrere Waffensysteme verfügt – automatisch diejenige Waffe mit der höchsten Wirkung aus. Taktiken können in einigen Fällen auch asynchron über Auslöser („Trigger“) angestoßen werden. Der erste Schritt wird in diesem Fall sofort ausgeführt, auch wenn gerade kein Rundenwechsel stattfindet. Trigger erlauben damit also dem Spieler, direkt zu reagieren und mindestens einen Taktikschritt mehr auszuführen als wenn er die Taktik regulär, ohne Auslöser anstoßen ließe. Dieser Mechanismus eignet sich insbesondere, um seine Einheiten bei Angriffen in Sicherheit zu bringen, oder für automatische Abwehrsysteme. Auf Taktiken, Taktikschritte und Auslöser wird im folgenden Unterabschnitt genauer eingegangen.

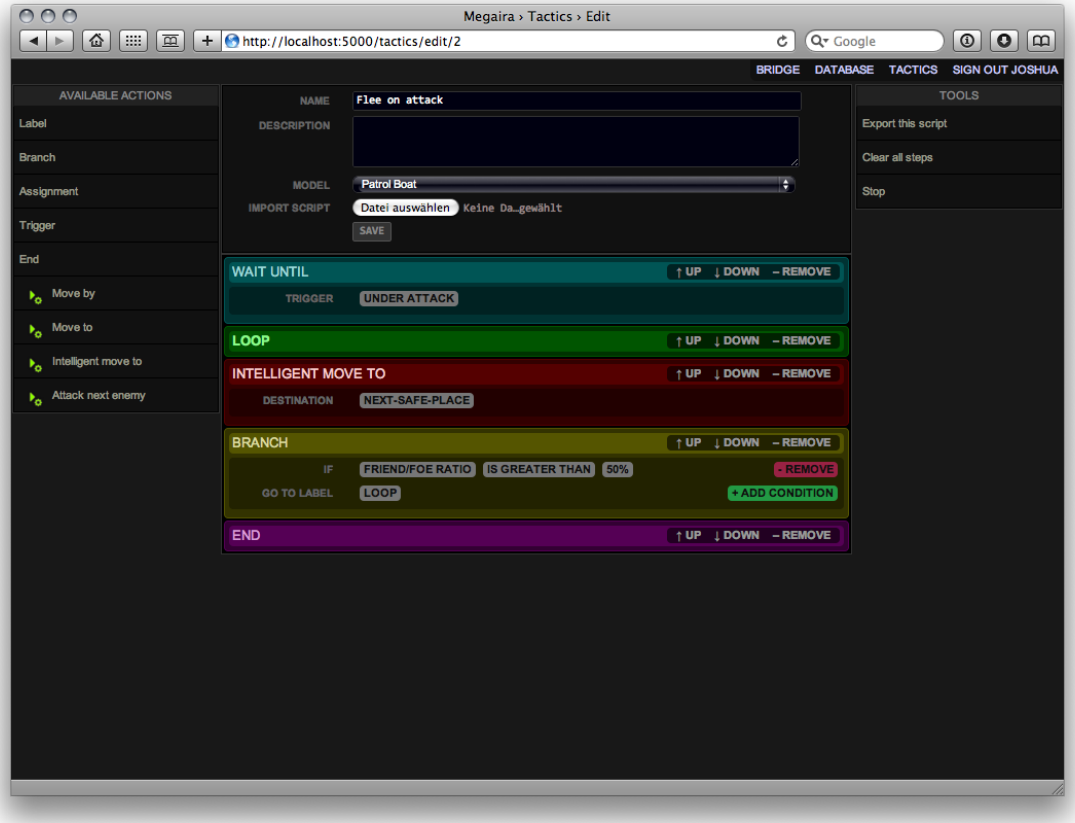

Abbildung 3.9: Der Taktik-Editor von „Megaira“.

Taktikschritte

In der linken Spalte des Editors sind alle möglichen Schritte für eine Taktik aufgelistet. Durch das Anklicken einer der Einträge wird der entsprechende Schritt der aktuell bearbeiteten Taktik angehängt. Zur Auswahl stehen benannte Sprungziele („Label“), bedingte und unbedingte Sprünge („Branch“) und Auslöser („Trigger“), sowie ein Befehl, um das aktuell laufende Skript zu beenden („End“). Weitere Befehle können über die Datenbank respektive über die interaktive Python-Shell angelegt werden. In unserer Beispielkonfiguration existiert ein Befehl zum relativen Bewegen von Einheiten („Move by“), Bewegen von Einheiten auf gegebene Zielkoordinaten („Move to“), Bewegen auf intelligente, logische Zielpositionen („Intelligent move to“) und ein Befehl, um die nächstgelegene Einheit anzugreifen („Attack“). Jeder Schritt kann im Regeleditor über kleine Knöpfe auf der rechten Seite (Abbildung 3.10 auf der nächsten Seite) vor seinen vorhergehenden Schritt, hinter seinen nachfolgenden Schritt verschoben, oder gelöscht werden. Auf Funktionsweise und Bedeutung der einzelnen Taktik-Schritte werden wir im Folgenden tiefer eingehen. Einzelne Schritttypen unterscheiden sich im Taktikeditor farblich: Labels sind grün, Trigger türkis, Sprünge gelb, der End-Befehl lila, alle übrigen Aktionen rot eingefärbt.

Abbildung 3.10: Taktik-Schritt „Label“ im Taktikeditor.

Abbildung 3.11: Taktik-Schritt „Label“ im Taktikeditor beim Bearbeiten des Namens.

- **Label** ist der einfachste Typ für Taktik-Schritte. Schritte von diesem Typen dienen als Sprungziele für Verzweigungen und sind beliebig betitelbar. In Abbildung 3.10 ist ein Label namens „Loop“ im Taktikeditor zu sehen. Durch einen Mausklick auf den Titel des Labels verwandelt sich der Text in ein editierbares Textfeld und es erscheinen zwei Knöpfe, „save“ und „cancel“, über die der neue Name zugewiesen oder der Vorgang abgebrochen werden kann (Abbildung 3.11). Bei der Ausführung von Taktiken werden Labels einfach übersprungen ohne Kosten (in Form von Laufzeit sprich „Ticks“) zu verursachen.

- **Branch** ist der Taktikschritt für alle bedingten und unbedingten Sprünge. In Abbildung 3.12 auf der nächsten Seite ist eine Repräsentation des entsprechenden Bedienelements im Taktikeditor zu sehen. Aus einer Klappliste ist eines der gegebenen Labels als Sprungziel auszuwählen. Existieren keine Sprungziele oder ist keins ausgewählt, so wird bei der Ausführung dieses Taktikschritts der Sprung einfach ignoriert. Mit dem Knopf „add condition“ kann der Verfasser der Taktik dem Sprung beliebig viele Bedingungen hinzufügen, welche dann per logischem *Und*[2] verknüpft werden. Jede Bedingung besteht aus drei Teilen: eine Linke Seite, meist eine Variable oder ein vom System berechneter Wert, ein Komparator und eine rechte Seite, die über die Vergleichsoperation im Sinne der linken Seite mit derselben verglichen wird. Tiefergehende Informationen zur Funktionsweise und Möglichkeiten der Sprungbedingungen sind im letzten Teil von Abschnitt 3.3 ausgeführt.

- **Trigger** pausieren die laufende Taktik und warten auf das Eintreten eines konfigurierbaren Ereignisses. Sie arbeiten asynchron. Das bedeutet, dass die Taktik sofort beim Eintreten der erwarteten Situation (zum Beispiel „Feind nähert sich“ oder „Einheit wird angegriffen“) den Folgeschritt ausführt, danach aber dann wieder synchron weitergeführt wird. Abbildung 3.13 auf der nächsten Seite zeigt einen Auslöser im Taktikeditor. Auslöser können an jeder Stelle der Taktik stehen, werden aber häufig am Anfang eingesetzt, wie in Abbildung 3.9 auf der vorherigen Seite zu sehen ist.

[2] Eine logische Oder-Verknüpfung lässt sich realisieren, indem man einfach mehrere Bedingungen mit demselben Sprungziel hintereinander setzt.

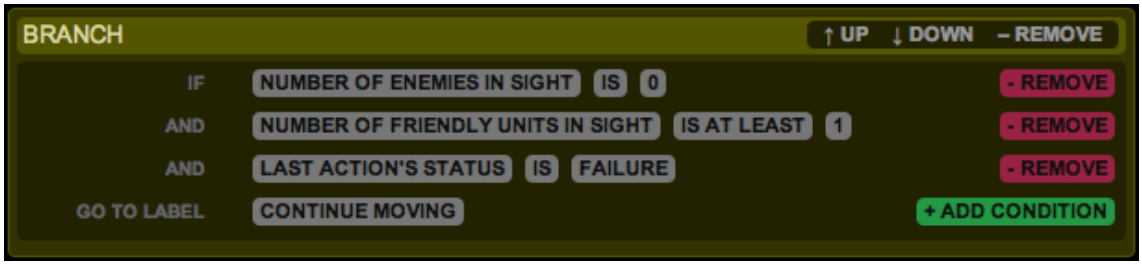

Abbildung 3.12: Taktik-Schritt „Branch“ mit drei Bedingungen im Taktikeditor.

Abbildung 3.13: Taktik-Schritt „Trigger“ im Taktikeditor.

- **End** beendet die aktuelle Taktik und hebt die Zuordung zur gesteuerten Einheit auf. Wird das Ende einer Taktik erreicht, wird automatisch der End-Befehl ausgeführt, auch wenn er an dieser Stelle nicht explizit angegeben ist.

- **Move by** ist ein einfacher Befehl um eine Einheit relativ zu ihrer aktuellen Position zu bewegen. Der einzige Parameter ist ein Vektor, die gewünschte relative Bewegung der gesteuerten Einheit, wie in Abbildung 3.14 abgebildet ist. Liegt die Zielposition außerhalb der Reichweite der Einheit, so wird das System versuchen, diese an den zum Ziel nächstgelegenen erreichbaren Punkt zu bewegen.

- **Move to** bewegt im Gegensatz zu „Move by“ die Einheit absolut. Analog zu „Move by“ wird hier versucht eine möglichst nahegelegene Zielposition zu erreichen, sollte das Ziel besetzt oder außer Reichweite sein.

- **Intelligent move to** stellt verschiedene Algorithmen bereit und ist leicht um neue Ziele zu erweitern. Hier wird die zugeteilte Einheit nicht an eine physische, sondern an eine logische Position bewegt. Zu diesem Zeitpunkt existieren drei logische Ziele: Der nächste „sichere“ Ort, die nächste gegnerische Einheit und die nächste befreundete Einheit, wie in Abbildung 3.15 auf der nächsten Seite dargestellt ist. Hierbei wird als

Abbildung 3.14: Taktik-Schritt „Move by“ im Taktikeditor.

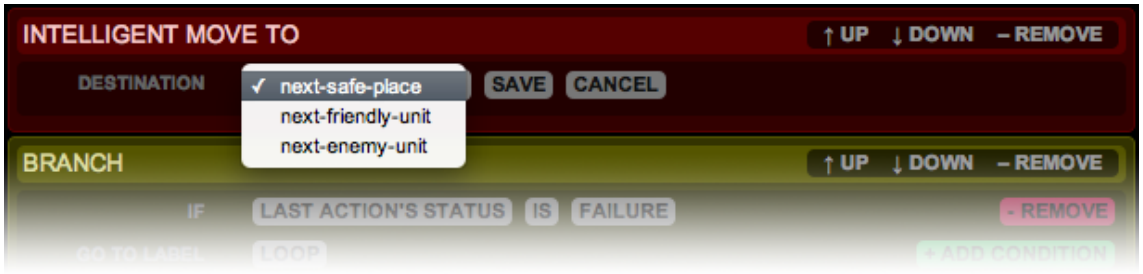

Abbildung 3.15: Taktik-Schritt „Intelligent move to“ im Taktikeditor.

Abbildung 3.16: Taktik-Schritt „Attack“ im Taktikeditor.

nächster sicherer Ort eine erreichbare Position gesucht, bei der möglichst keine oder wenige gegnerische Einheiten in Sichtweite sind.

- **Attack** veranlasst die gesteuerte Einheit die nächstgelegene gegnerische Einheit, die sich in Angriffsreichweite befindet, zu attackieren. Hierbei wird automatisch die Waffe ausgewählt, die den größten Schaden anrichtet. Dieser Taktikschritt besitzt keine Parameter (Abbildung 3.16).

Jeder Taktikschritt kann einen Wert zurückgeben. Beispielsweise wird bei einem Angriff Auskunft darüber erteilt, ob die gegnerische Einheit zerstört wurde, oder bei einer Bewegungsaktion, ob die exakte Zielposition erreicht wurde, oder ob die Einheit nur in die Nähe des angegebenen Ziels bewegt werden konnte. Anhand dieses Rückgabewerts kann der Entwickler der Taktik mittels eines bedingten Sprungs verzweigen und etwa die „fehlgeschlagene“ Aktion wiederholen, oder weitere Maßnahmen hinzuziehen.

Rückgabewerte werden in einem entsprechenden Register des Zustandsobjekts der ausgeführten Taktik gespeichert. Taktikschritte können beliebige weitere Register anlegen und verwenden. Auf technische Details hierzu wird in Abschnitt 3.3 weiter eingegangen.

Sofern man die oben beschriebenen Auslöser („Trigger“) ausblendet, wird von jeder laufenden, zugewiesenen und aktiven Taktik mit jedem Rundenwechsel der nächste Schritt abgearbeitet. Auslöser versetzen die Taktik für die zugewiesene Einheit in einen Wartezustand. Jeder Aktionsschritt verursacht Kosten in Form von Zeiteinheiten, „Ticks“. Allerdings können diese Kosten je nach Aktion variieren und reelle Werte annehmen. Die Abarbeitung der Schritte während eines Rundenwechsels funktioniert dann wie in Algorithmus 3.1 auf der nächsten Seite ausgeführt. Die Obergrenze für die Anzahl der Taktikschritte s_{max} verhindert eine Endlosschleife im System, wenn während eines Schleifendurchlaufs einer Taktik keine Zeitkosten verursacht werden.

$s_{max} \leftarrow 15$ // Obergrenze für die Anzahl der Schritte je Tick und Taktik
$t_{tick} \leftarrow 1.0$ // Restliche Zeit für Taktik in diesem Tick
$s \leftarrow 0$ // Abgearbeitete Schritte so weit
while $t_{tick} > 0 \wedge s < s_{max}$ **do**
  $t \leftarrow step()$ // führe nächsten Schritt aus und berechne Kosten
  if $t = Nil$ **then**
    return
  end if
  $t_{tick} \leftarrow t_{tick} - t$
  $s \leftarrow s + 1$
end while

Algorithmus 3.1: Abarbeitung der Taktikschritte während eines Rundenwechsels

Die in Abbildung 3.9 auf Seite 29 gegebene Taktik „Flee on Attack“ ist ein einfaches Beispiel dafür, wie ein Spielteilnehmer seine Einheiten - insbesondere während seiner Abwesenheit vom Spiel – schützen kann. Zunächst wird gewartet bis die gewählte Einheit angegriffen wird. Ist dies der Fall, so beginnt eine Schleife, in der der Agent versucht, die gesteuerte Einheit in Sicherheit zu bringen. Ist das Ziel „Freund/Feind verhältnis $\geq 50\%$“ erreicht, beendet sich die Taktik und wird von der Einheit gelöst. Das bedeutet, die Zuweisung der Taktik zu der Einheit wird aufgehoben. Taktik sowie Zuweisungen bleiben natürlich für alle anderen Einheiten aktiv.

Weitere Eigenschaften von Taktiken

Neben den Taktikschritten haben Taktiken ein paar weitere Eigenschaften. Der Spieler kann sie benennen, Beschreibungen hinzufügen, und er muss angeben, auf welches Modell sich die Taktik bezieht. Da verschiedene Einheiten unterschiedliche Fähigkeiten haben, ist nicht jede Taktik für jeden Typen sinnvoll. Eine „Suchen und Angreifen“-Taktik ist beispielsweise für eine Bohrinsel unpassend.

Um Taktiken auszutauschen, extern zu bearbeiten oder an maschinelle Lernsysteme (etwa evolutionäre Algorithmen, siehe Abschnitt 2.2) anzubinden, erlaubt die Plattform Megaira den Import und Export von Taktiken in Form von Textdateien. Der Inhalt der exportierten Textdatei der Taktik, die in Abbildung 3.9 auf Seite 29 zu sehen ist, ist Listing 3.1 zu entnehmen.

Einzelne Taktiken können global für alle Einheiten angehalten werden. Ändert der Spielteilnehmer irgendeine Einstellung einer Taktik oder eines Taktikschritts, so wird die geänderte Taktik automatisch angehalten, um Überschneidungen oder die Ausführung unfertiger Taktiken zu verhindern. Unabhängig davon, ob einer Einheit eine Taktik zugewiesen

ist oder nicht und unabhängig davon, ob die Taktik läuft oder pausiert kann der Spieler seine Einheit jederzeit (parallel) manuell steuern.

```
# exported megaira tactic
# owner: joshua
# date:  2009-07-15 17:38:15.097220

.name  Flee on attack
.desc  None
.model patrolboat

    wait until under-attack

loop:
    intelli_moveto (destination=next-safe-place)
    go to [loop] if friend-foe-ratio > 50%
    end
```

Listing 3.1: Megaira: exportiertes Taktik-Skript

3.2 Die Architektur

Ein Browserspiel benötigt – wie die meisten webbasierten Anwendungen – eine offene und flexible Plattform. Erweiterbarkeit ist wichtig, so dass die Spielteilnehmer hin und wieder mit neuen Funktionen, Möglichkeiten und Herausforderungen überrascht werden können. Wartbarkeit ist ein weiterer, essenzieller Aspekt. Während viele Spiele anderer Genres bisher auf optischen Medien vertrieben werden, die nach der Produktion unveränderbar[3] sind, haben Browserspiele den großen Vorteil, dass der Programmcode eines Spiels zentral in einer gemeinsamen Infrastruktur lebt und jederzeit simultan für alle Spieler in einem Zug modifiziert werden kann. Ein weiteres Problem, welches sich häufig insbesondere bei MMOG[4] und damit auch bei Browserspielen stellt, ist das gleichzeitige Interagieren großer Zahlen von Mitspielern. Dies impliziert Skalierbarkeit, Modularität und Flexibilität der Spielplattform, besonders in Bezug auf die Installations- und Konfigurationsmöglichkeiten.

Um die genannten Anforderungen zu erfüllen, haben sich so genannte MVC-Frameworks[5] durchgesetzt. Sie repräsentieren das Paradigma der möglichst strikten Trennung von Dar-

[3] Es existiert natürlich grundsätzlich die Möglichkeit, Aktualisierungen über das Internet bereitzustellen, doch handelt es sich hierbei in der Praxis meist um kleinere Fehlerbehebungen. Zudem kann man Spieler nicht zwingen, regelmäßig zu aktualisieren und Installationen auf Rechnern ohne Internetzugang sind nicht oder nur umständlich möglich.

[4] „Massive Multiplayer Online Game“ (zu Deutsch: „ Massen-Mehrspieler-Online-Gemeinschaftsspiel“)

[5] „MVC“ steht für „Model-View-Controller“, zu Deutsch: „Modell/Präsentation/Steuerung“

stellung, Logik und Datenmodell. Das Fundament von „Megaira“ basiert auf diesem Prinzip.

3.2.1 Der Anwendungsstapel

Für die Beispiel- und Testanwendung „Megaira“ kommt eine zeitgemäße, flexible Auswahl an Frameworks auf Basis der betriebssystemunabhängigen Programmiersprachen Python [24] und JavaScript [22] zum Einsatz, die einerseits die Entwicklung der Anwendung auf einer sehr hohen Abstraktionsebene ermöglichen ohne sich mit komplizierten, technischen Interna aufzuhalten, andererseits Benutzerfreundlichkeit ermöglichen und in den Mittelpunkt stellen, welche für anhaltende Spielmotivation ausgesprochen wichtig ist. Zudem wird in der gesamten Architektur durchweg auf Erweiterbarkeit, Wartbarkeit und Skalierbarkeit gesetzt.

Das Pylons Webanwendungs-Framework

Serverseitig baut Megaira auf dem freien MVC-Framework Pylons [11] auf, das HTTP-Anfragen, die es von einem gewöhnlichen HTTP-Server[6] übergeben bekommt, je nach angefragter URL[7] an ein entsprechendes, vom Entwickler bereitgestelltes Steuerungs-Objekt (engl. „Controller“) weiterleitet. Dieses kümmert sich dann sowohl um Sicherheitsaspekte wie Authentifizierung und Zugriffskontrolle - hier mittels AuthKit [25] – als auch um die konkrete Anwendungslogik. Je nach Art der angefragten Aktion lädt die passende Methode des Controllers entweder ein HTML-Template, füllt dieses mit den Variablen im aktuellen Kontext aus und schickt die Webseite an den Browser zurück, oder es serialisiert angefragte Daten als JSON[8]-Objekt, welches dann im Hintergrund direkt an die laufende JavaScript-Routine im Webbrowser übergeben wird, ohne dass die gesamte HTML-Seite neugeladen werden muss. Die zuletzt beschriebene Technik wird allgemein als AJAX[9] bezeichnet.

Pylons unterstützt den WSGI[10]-Standard [21], der es erlaubt Python-basierte Web-Frameworks an Webserver wie den verbreiteten Apache HTTP Server[11] anzubinden und dabei Middleware-Komponenten transparent zwischenzuschalten, die spezielle Aufgaben erledigen. Im Fall von Megaira sind dies „AuthKit“ [25] für Autorisierung und Authentifizierung, „ToscaWidgets“ [46] zum Generieren, Validieren und Auswerten von HTML-

[6] in unserem Fall dem populären Projekt der Apache Software Foundation [7]

[7] „Uniform Resource Locator“, umgangssprachlich auch „Internetadresse“

[8] „JavaScript Object Notation“, JavaScript-kompatibles Datenformat

[9] AJAX steht für „Asynchronous JavaScript and XML“ [50], eine JavaScript-Routine im Browser fragt selbständig über ein XmlHttpRequest-Objekt beim Server an und bekommt die Ergebnisse in Form von XML-Quellcode (ursprünglich, heute überwiegend HTML und JSON) an eine bereitgestellte Callback-Funktion geliefert.

[10] WSGI steht für „Web Server Gateway Interface“ [21]

[11] mittels „mod_wsgi“ [20]

Formularen sowie Komponenten zum URL-Routing („Routes“ [9]), zur Sessionverwaltung und zum Caching („Beaker“ [10]).

Dank der „Python Paste“ Bibliothek steht unter Pylons eine Python-Shell bereit, über die ein Administrator Vollzugriff auf alle Module, Klassen und Objekte im Kontext der laufenden Anwendung hat. Somit kann jederzeit administrativ in Echtzeit in das Spielgeschehen eingegriffen werden, sollten beispielsweise Softwarefehler entdeckt oder Betrugsversuche aufgedeckt werden.

Elixir, die Datenbankabstraktionsschicht

Elixir [18] ist eine deklarative Datenbankabstraktionsschicht, die auf dem Datenbank-Toolkit und ORM[12] SQLAlchemy [12] aufsetzt. Das Projekt wurde vom Autor dieser Arbeit mitkonzeptioniert und -entwickelt.

```
1  from elixir import *
2
3  class Director(Entity):
4      name = Field(Unicode(60))
5      movies = OneToMany('Movie', inverse='director')
6
7  class Movie(Entity):
8      title = Field(Unicode(60))
9      description = Field(Unicode(512))
10     releasedate = Field(DateTime)
11     director = ManyToOne('Director', inverse='movies')
12     actors = ManyToMany('Actor', inverse='movies')
13
14 class Actor(Entity):
15     name = Field(Unicode(60))
16     movies = ManyToMany('Movie', inverse='actors')
```

Listing 3.2: Elixir: Datenmodell spezifizieren

Der Entwickler legt nach gewohnter, objektorientierter[13] Methodik ein Klassensystem mit Eigenschaften und Beziehungen zwischen den Klassen an. Elixir generiert für die gegebenen Objekte und die eingesetzte SQL-Datenbank SQL-Befehle, Datenbanktabellen, Indizes und alles was für die Umsetzung nötig ist. Anfragen an die Datenbank werden komplett über Python-Code formuliert, so dass der Entwickler kein SQL schreiben muss und zudem unabhängig vom später eingesetzten Datenbanksystem entwickeln und selbiges

[12] „Object Relational Mapper“ bezeichnet ein System, das automatisch Objekte (im Sinne der OOP) als Entitäten in relationalen Datenbanken persistiert und Relationen zwischen diesen Objekten verwaltet, ohne dass der Entwickler SQL beherrschen muss.

[13] im Python-Stil

austauschen kann. Megaira wird beispielsweise lokal unter SQLite [3] entwickelt, da SQLite sehr viel einfacher zu handhaben ist als die meisten „großen“ SQL-Datenbanksysteme. Die öffentliche Version wird aus Gründen der Skalierbarkeit mit MySQL [42] betrieben.

In Listing 3.2 auf der vorherigen Seite ist ein Beispiel für die Verwendung von Elixir gegeben. Um Elixir mitzuteilen, dass das Datenmodell fertig spezifiziert ist, führt man das Kommando `setup_all()` aus. Ein einfaches `create_all()` generiert nun die nötigen Tabellen in der angebundenen Datenbank. Um Relationen in die Datenbank einzufügen genügt es, die entsprechenden Python-Objekte anzulegen und gegebenenfalls die Datenbank-Transaktion abzuschließen, wie in Listing 3.3 zu sehen ist.

```
1  ridley_scott = Director(name=u"Ridley Scott")
2  blade_runner = Movie(title=u"Blade Runner",
3                       year=1982,
4                       director=ridley_scott,
5                       actors=[
6                          Actor(name=u"Harrison Ford"),
7                          Actor(name=u"Rutger Hauer"),
8                          Actor(name=u"Sean Young")
9                       ])
10 session.commit()
```

Listing 3.3: Elixir: Objekte erstellen

Datenbankanfragen werden nun ebenfalls in Python ausgeführt, ohne dass auf die Flexibilität und Systemnähe von SQL verzichtet werden muss, wie in Listing 3.4 demonstriert wird. Hier werden alle Filmtitel aus der Datenbank abgefragt, die im Jahr 1982 erschienen oder deren Titel auf „Runner“ enden (oder beides).

```
1  movies = Movie.query().filter(
2                  or_(Movie.name.like('%Runner',
3                      Movie.year==1982))).all()
```

Listing 3.4: Elixir: Objekte abfragen

Selbstverständlich sind sehr viel mächtigere und komplexere Anfragen möglich. Fast beliebige SQL-Anfragen können auf der SQLAlchemy-Ebene über die Python-kompatible Syntax realisiert werden. Auch sämtliche Funktionen, die das unterliegende DBMS[14] anbietet, stehen über die Schnittstelle bereit. Unterstützt werden die meisten verbreiteten relationalen Datenbanken wie MySQL, PostgreSQL, Oracle, SQLite, SQL Server und viele weitere.

[14] Datenbank-Management-System

Neben den üblichen Relationen („many-to-many“, „many-to-one“, „one-to-many“, und „one-to-one“) unterstützt Elixir auch verschiedene Formen der polymorphen Vererbung: „single“, „concrete“ und „multi“. Im Modus „single“ Teilen sich alle erbenden Klassen und die ursprünglichen Elternklasse eine Tabelle in der Datenbank. Die Spalten der Tabelle reflektiert dann die Vereinigung aller Eigenschaften der repräsentierten Klassen. Bei „concrete“ hingegen bekommt jede Klasse ihre eigene Tabelle zugewiesen. Im Fall „multi“[15] bekommt jede erbende Klasse eine Tabelle, deren Spalten ausschließlich die Eigenschaften repräsentieren, die zu den geerbten der Elternklasse hinzukommen, plus einen Verweis auf den Datensatz in der Tabelle der Elternklasse. Bei Datenbankanfragen im letzten Fall werden alle betroffenen Tabellen über ein SQL-`JOIN`-Konstrukt verbunden.

Tiefergehende Informationen zu Funktionalität, Datenbankanfragen und Kompatibilität mit verschiedenen Datenbanken sind [18] und [12] zu entnehmen.

JavaScript, AJAX und die Bibliothek jQuery

Der browserseitige Programmcode vom Megaira ist entsprechend den aktuellen Gegebenheiten der gängigen Webbrowser in JavaScript verfasst. Da leider mittlerweile unzählige Kombinationen aus Browser-Software und -Versionen im Einsatz sind, die sich aus Sicht der Softwareentwickler in wichtigen Details zum Teil stark unterscheiden, setzt Megaira auf die JavaScript-Bibliothek jQuery [39], welche einfache Schnittstellen zu vielen häufig genutzten Funktionen bereitstellt, die sonst speziell für verschiedene Browsertypen separat zu entwickeln wären. Als Beispiele seien hier etwa die Durchführung von AJAX-Anfragen oder DOM-Manipulationen angeführt, die auch in aktuellen Browsern noch immer nicht einheitlich standardisiert funktionieren.

In Megaira setzen alle Bedienelemente, insbesondere die Brücke (bzw. das Spielfeld) und der Taktikeditor verstärkt auf clientseitiges Scripting, um den Bedienkomfort zu maximieren. Um an Bandbreite und Datentransfervolumen zu sparen, die Serverlast einzugrenzen und parallel die Zugriffs- und Wartezeiten seitens der Spieler möglichst gering zu halten, werden häufig die gerade benötigten Daten nachgeladen und in der bereits angezeigten Seite aktualisiert. Das ist insbesondere bei großen Teilnehmerzahlen wichtig.

Bei der Navigation durch die Karte auf der Brücke, siehe Abbildung 3.2 auf Seite 24, wird auf „Drag’n Drop“ gesetzt. Das bedeutet, dass der Spieler sich durch die Karte bewegen kann, in dem er einen Teil derselben mit der Maus „anfasst“ und dann die ganze Karte verschiebt. Der sichtbare Ausschnitt auf dem Bildschirm passt sich dann der neuen Position der Karte an. „Scrollen“ über die Pfeiltasten der Tastatur ist zusätzlich auch möglich.

An vielen Stellen und besonders beim Taktik-Editor, auf den näher in Abschnitt 3.1 eingegangen wurde, kommen „Inline-Editoren“ zum Einsatz. Der Benutzer klickt einen zu ändernden Wert in einer Anzeige an, ändert Ihn durch ein Auswahl-Steuerelement oder

[15] auch als „joined“ bezeichnet

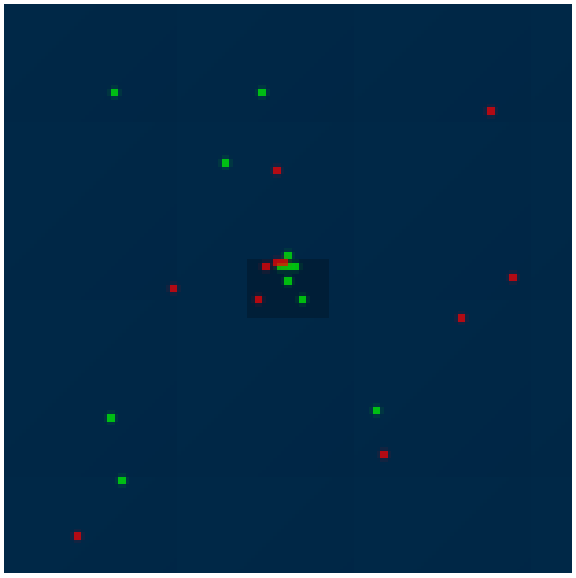

Abbildung 3.17: Die Radaransicht auf der Brücke von „Megaira“

per Eingabe über die Tastatur, klickt auf den Speichern-Knopf und der Wert wird in der Datenbank auf dem Server sofort geändert, ohne dass eine einzige HTML-Seite geladen werden musste. Dies spart Bandbreite, Ressourcen beim Betreiber und Nerven beim Spieler.

3.2.2 Darstellung: HTML/DOM, Canvas, CSS

Gemäß dem Verständnis des Begriffs „Browserspiel“ im Kontext dieser Arbeit werden die Inhalte in Megaira komplett in HTML[16] ausgegeben, die Darstellung erfolgt durchgängig in CSS[17]. Für die Radarübersicht wird das HTML-Canvas-Element[18] eingesetzt, das eine Zeichenfläche bereitstellt, in der die Einheiten in einem wesentlich größeren Maßstab als auf der großen Spielfeldkarte eingezeichnet werden, wie in Abbildung 3.17 zu sehen ist. Einheiten werden durch Grafiken[19] dargestellt, die je nach Status einen CSS-Rahmen bekommen. Markierungen, die Ziele hervorheben – entweder Angriffsziele oder erreichbare Kartenfelder, wie in Abbildung 3.18 auf der nächsten Seite zu sehen ist, kommen ganz ohne Grafiken aus. Sie bestehen aus HTML-`DIV`-Elementen, die über CSS entsprechend eingefärbt und dargestellt werden. Auch dadurch wird in der Masse einiges an Bandbreite und Ladezeit eingespart.

[16] XHTML 1.0 Transitional [48]

[17] CSS 2.1 [47] mit wenigen CSS 3-Eigenschaften, die allerdings bei nicht CSS-3-kompatiblen Browsern höchstens unbedeutende kosmetische Folgen haben

[18] Der Einsatz des Canvas-Elements bricht leider den ansonsten von „Megaira“ eingehaltenen XHTML-1.0- (bzw. HTML4-)Standard [48]. Das Element wird zwar erst mit HTML5 eingeführt werden, allerdings verstehen die aktuellen stabilen Versionen der verbreiteten standardkonformen Webbrowser wie Firefox [35], Safari [8] oder Chrome[26] bereits damit umzugehen.

[19] überwiegend im 24-Bit PNG-Format mit Alphakanal

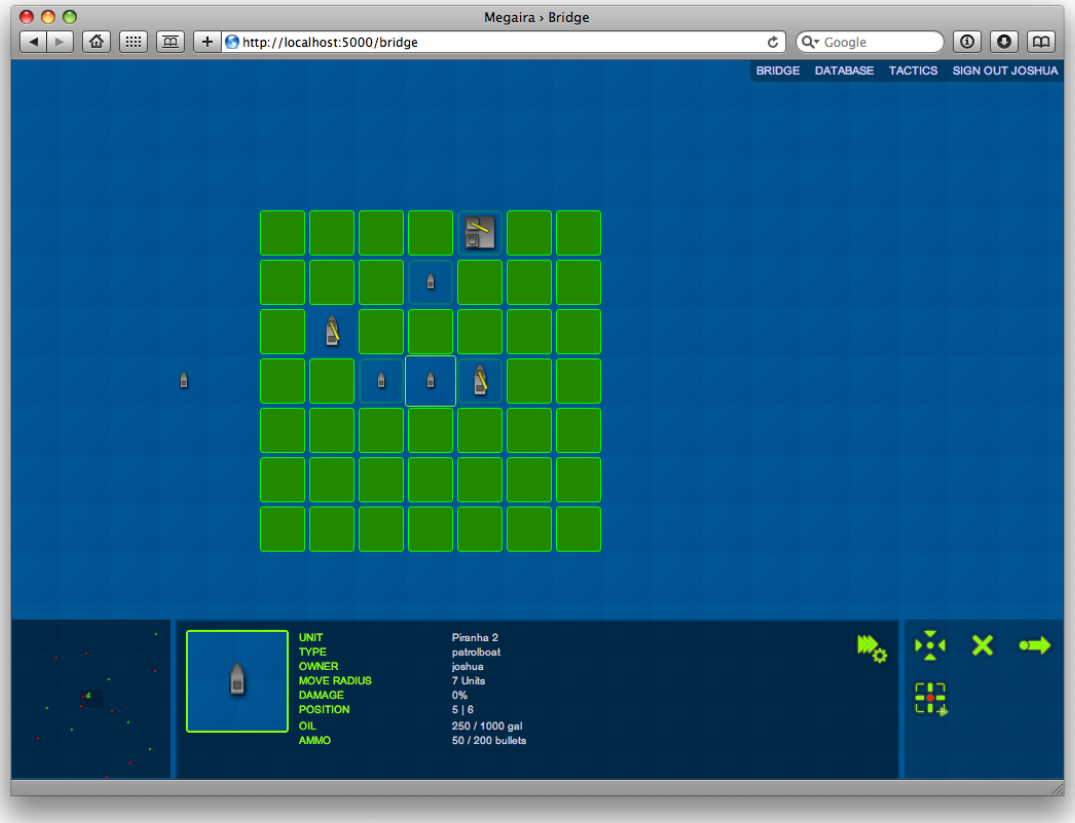

Abbildung 3.18: Einheiten bewegen in Megaira

Bei der Entwicklung von Megaira wurde überwiegend Apple Safari (Version 4.0) [8] als Browser unter Mac OS X eingesetzt. Gelegentlich kam auch Mozilla Firefox (Version 3.5) [35] zum Einsatz. Grundsätzlich wurde auf Client- wie auf Serverseite überwiegend auf Standardkonformität und Plattformunabhängigkeit geachtet, wordurch das Spiel auf allen modernen, standardkonformen Webbrowsern unter den gängigen Betriebssystemen korrekt dargestellt werden sollte.

3.2.3 Umgebung, Einsatz und Testverfahren

Einer Internetanwendung wird naturgemäß vorausgesetzt, dass sie rund um die Uhr durchgängig weltweit zu erreichen ist. Um diese Anforderung zu erfüllen benötigt man die entsprechende Hard- und Software sowie eine zuverlässige Internetanbindung. In diesem Unterabschnitt werden wir näher auf die softwaretechnischen Einzelheiten des Spiels eingehen. Hardwareseitig kommt für die Produktion ein Rootserver bei einem großen deutschen Hoster zum Einsatz, welcher unter der bekannten Linux-Distribution „Ubuntu“ betrieben wird. Die Entwicklungsumgebung läuft komplett unter Mac OS X.

Serverkonfiguration, Apache, MySQL und mod_wsgi

Wie zu Beginn von Abschnitt 3.2 beschrieben wurde, basiert Megaira auf dem Webanwendungs-Framework Pylons [11]. Da zu diesem Zeitpunkt für das Spiel kein dedizierter Server bereitsteht und die Zugriffszahlen noch sehr überschaubar sind, teilt sich die Anwendung einen Rechner mit mehreren anderen Webanwendungen und Internetpräsenzen. Diese Funktionalität nennt man virtuelles Hosting, die nötige Infrastruktur wird vom Apache HTTP Server bereitgestellt.

Um die Pylons-Anwendung in den Apache-Server zu integrieren gibt es verschiedene Möglichkeiten. Die sauberste basiert auf der Integration über die WSGI-Schnittstelle, auf die schon in Abschnitt 3.2 eingegangen wurde. Das Apache-HTTP-Server-Modul mod_wsgi [20] stellt für die Konfiguration des Servers Direktiven zur Verfügung, die eine Einbindung von WSGI-kompatiblen Diensten und deren Konfiguration erlauben. Beispielsweise lassen sich Anzahl der Prozesse und Threads, verschiedene Begrenzungen der Systemressourcen, Timeouts und Verzeichnisse konfigurieren.

Damit die Rundenwechsel regelmäßig und zuverlässig alle 15 Minuten durchgeführt werden, bedient sich die Anwendung des auf der jeweiligen Serverumgebung installierten Cron-Dienstes[20]. Ein Bash-Skript namens `tick.sh` wird von diesem in regelmäßigen Abständen aufgerufen und startet dann die Globale `tick()`-Funktion mittels eines Paster-Kommandos. Mehr Informationen zur praktischen Python Paste Bibliothek, welche in Verbindung mit Pylons unverzichtbar ist, finden sich in [13].

[20] hier „Vixie Cron“

3.3 Das Datenbankmodell

Wie schon zu Beginn von Abschnitt 3.1 beschrieben, liegt bei der Entwicklung von Megaira ein wichtiger Schwerpunkt auf der Erweiterbarkeit und Wiederverwendbarkeit der Plattform. Zu diesem Zweck wird versucht, den überwiegenden Teil der Daten in der Datenbank unterzubringen und möglichst wenig fest einzuprogrammieren. Dieser Abschnitt geht auf das Klassen- und Datenbankmodell der Anwendung ein.

Fast alle Klassen von Megaira verfügen über Methoden zur Konvertierung der Objekte in einfache, menschenlesbare Stringrepräsentationen `__repr__()`, um das direkte Arbeiten am laufenden Spiel[21] in einer Python-Konsole zu erleichtern. Viele Klassen besitzen zusätzlich eine Methode `jsonify()`, die die Repräsentation der Objekte in JSON vereinfacht, so dass diese mittels AJAX-Anfragen direkt an eine JavaScript-Routine im Browser übergeben werden können (siehe hierzu auch Abschnitt 3.2).

3.3.1 Einheiten, Modelle und Ressourcen

Das Grundgerüst des Datenbankmodells des Spiels ist in Abbildung 3.19 auf der nächsten Seite zu sehen. Einige Methoden und Eigenschaften einiger Klassen sind in dieser Darstellung zu Gunsten der Übersichtlichkeit ausgeblendet. Die Notation ist nah am entsprechenden Python-Code gehalten, was insbesondere an den Datentypen zu erkennen ist. Kleingeschriebene Typen, wie etwa `bool`, `int`, `string`[22] und `object` repräsentieren die gleichnamigen Python-Standarddatentypen. Großgeschriebene Bezeichner, so etwa `Unicode`, `Integer`, `SmallInteger` oder `DateTime` hingegen beschreiben die Python-Äquivalente der SQL-Datentypen aus Elixir bzw. SQL-Alchemy, die dann automatisch für das unterliegende Datenbanksystem übersetzt werden. Weitere großgeschriebene Typen wie `Unit` oder `Resource` beziehen sich auf die gleichnamigen Klassen, welche größtenteils auch in den Klassendiagrammen in Abbildung 3.19 auf der nächsten Seite und Abbildung 3.20 auf Seite 48 abgebildet sind.

Spieler

Die Klasse `Player` kapselt den Spieler und alle seine persönlichen Eigenschaften. In diesem Fall beschränken sich diese auf die Anmeldedaten, also Benutzername, Passwort und seine E-Mail-Adresse. Sie bildet die Grundlage für Authentifizierung und Autorisierung. In späteren Versionen werden weitere Felder wie beispielsweise für persönliche Voreinstellungen hinzukommen. Die im Diagramm einzige sichtbare Relation der Klasse ist die Zuordnung von vielen Einheiten zu genau einem Spieler.

[21] selbstverständlich Administratoren und Entwicklern vorbehalten
[22] wobei mit `string` in der Regel auch Unicode-Strings (Typ `unicode`) gemeint sind

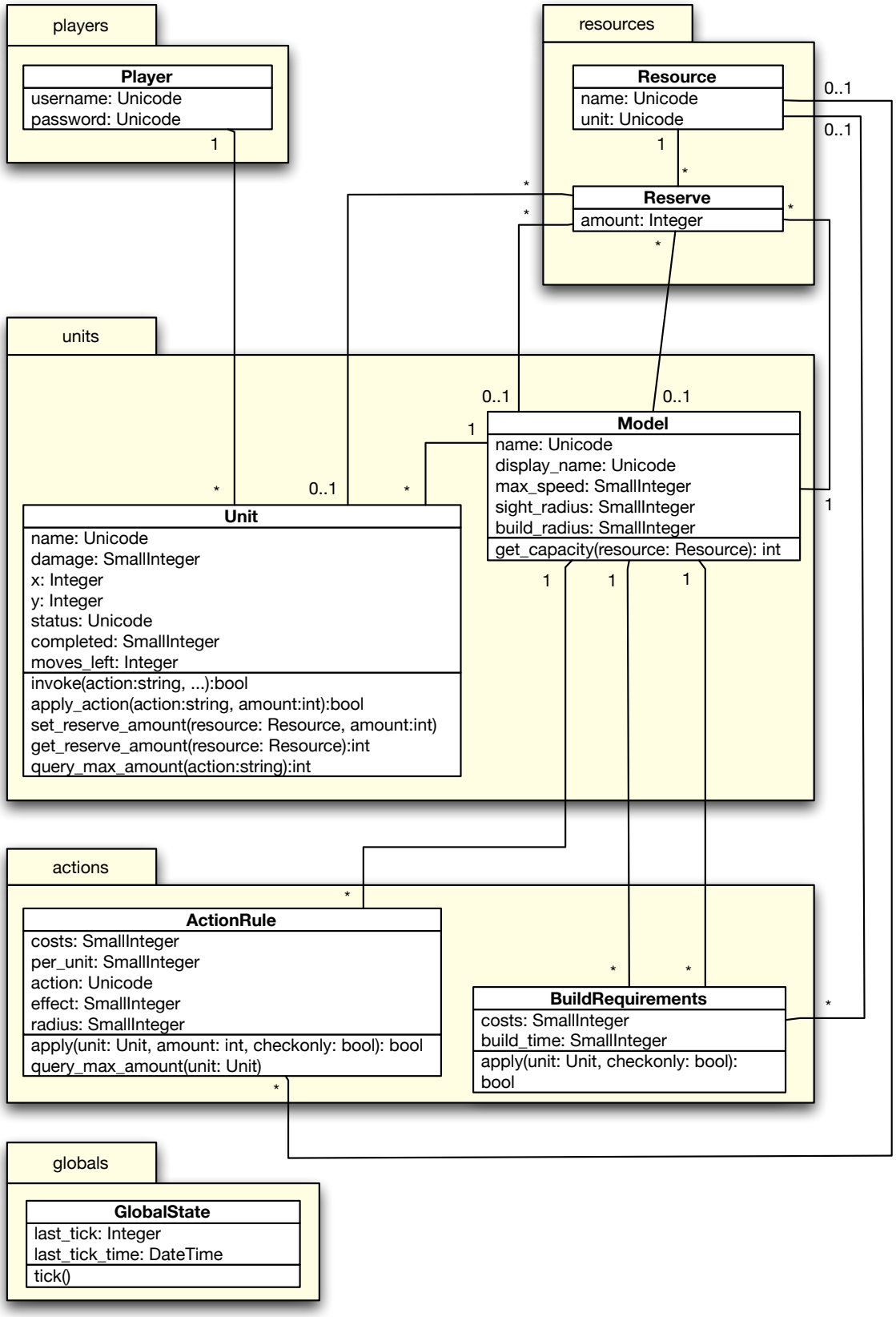

Abbildung 3.19: Das Klassen- und Datenbankmodell von Megaira: Modelle, Einheiten, Ressourcen, Spieler, Aktionsregeln und Bauvorschriften

Einheiten und Modelle

Das Paket `units` enthält die Klassen `Model` und `Unit`. Objekte der Klasse `Model` sind Modelle für sämtliche Arten von beweglichen sowie unbeweglichen Einheiten. In der Beispielkonfiguration von Megaira gibt es die drei Modelle „Builder“, „Patrolboat“ und „Oil Rig“. Ein Modell hat einen Namen (Eigenschaft `name`), der es eindeutig repräsentiert und aus dem auch der Dateiname für das repräsentierende Piktogramm generiert wird, und einen Anzeigenamen (Eigenschaft `display_name`), so wie er den Spielteilnehmern angezeigt wird. Über das Feld `max_speed` wird angegeben, über wie viele Kartenfelder sich die Einheit in jeder Runde maximal bewegen[23] darf, sofern die benötigten Ressourcen dazu ausreichen. Über `sight_radius` wird spezifiziert, in welchen Umkreis eine Einheit „sehen“ kann, also feindliche und freundliche Einheiten erkennt, während `build_radius` bestimmt, in welchem Radius neue Einheiten positioniert werden dürfen, die die betreffende Einheiten erzeugt hat, sofern sie überhaupt fähig ist, selbst Einheiten zu konstruieren. Das Feld `sight_radius` ist insbesondere in Verbindung mit Taktiken und dort speziell bei den Sprungbedingungen wichtig, da hier Ausdrücke wie „Anzahl feindlicher Einheiten in Sichtweite“ oder „Verhältnis Freund/Feind in Sichtweite“ zur Verfügung stehen. Die Methode `get_capacity(resource: Resource)` liefert die jeweilige grundsätzliche Ladekapazität der Einheiten des Modells für einen gegebenen Rohstoff. Über eine 1-zu-n-Relation des Modells zu beliebig vielen Reserve-Objekten wird festgehalten, welche Mengen an Rohstoffen eine Einheit je Runde fördern oder produzieren kann. Eine Ölbohrinsel im Beispielszenario fördert 50 Gallonen Öl, die zu jedem Rundenwechsel gutgeschrieben und an andere Einheiten übertragen werden können. Die technische Realisierung von Rohstoffen und Reserven in Megaira wird im nächsten Unterabschnitt erläutert.

Ressourcen

```
1 from megaira.model.ressources import Resource
2
3 oil = Resource(name=u"oil", unit=u"gal")
4 mgammo = Resource(name=u"machinegun-ammo", unit=u"bullets")
5 missiles = Resource(name=u"missiles", unit=u"ea")
```

Listing 3.5: Megaira: Ressourcen anlegen

In den meisten Strategiespielen spielen Rohstoffe, Waren, Treibstoffe und Munition fundamentale Rollen, so auch in Megaira. Jedoch sind Anzahl und Arten hier nicht durch die Plattform fest vorgegeben, sondern können frei angelegt und beliebig erweitert werden. Die nötigen Klassen finden sich im Paket `resources`. Die Klasse `Resource` repräsentiert beliebige Arten von Rohstoffen. Im Beispielszenario handelt es sich hierbei um Öl, Raketen

[23] auch diagonale Bewegungen sind in Megaira möglich

und Patronen für Maschinengewehre. Jede Ressource hat zwei Eigenschaften, erstens einen Namen und zum Zweiten eine Maßeinheit. In unserem Beispiel wird Öl in Gallonen gemessen, Maschinengewehrmunition in Patronen und Raketen in „Stück“ (engl. beziehungsweise militärisch „ea“), siehe Listing 3.5.

Die Zweite wichtige Klasse im Paket `resources` heißt `Reserve` und verkörpert eine – meist „greifbare“ – Menge einer bestimmten Ressource, etwa 10 Gallonen Öl oder 50 Schuss Munition. Hat ein Bauschiff 20 Raketen geladen, so wird dies über ein Objekt dieser Klasse manifestiert. Die Ladekapazitäten einer Einheit bezüglich einer Ressource werden ebenfalls über die Klasse `Reserve` gespeichert. Beispielsweise kann jede Ölbohrinsel die gleiche, große Menge an Öl speichern, während Patrouillenboote neben kleineren Mengen Öl auch Maschinengewehrmunition transportieren können. Wird nun eine Rakete abgefeuert, Öl verbraucht oder Munition von einem Schiff an ein anderes, das diese Art von Ressource laden kann, abgegeben, dann wird das entsprechende `Reserve`-Objekt um den passenden Betrag verringert und gegebenenfalls das korrespondierende Objekt der empfangenden Einheit um denselben Betrag erhöht. Ladekapazitäten werden hierbei selbstverständlich beachtet und die Existenz negativer Reserven ist weder vorgesehen noch möglich.

Aktionen und Bauvorschriften

Ausgesprochen wichtig für ein Spiel sind natürlich die Aktionen, die vom Spielteilnehmer respektive seinen Einheiten ausgehen. Für diese existiert das Paket `actions`, welches zwei Klassen beinhaltet. Die erste der beiden Aktionsklassen `ActionRule` spiegelt sämtliche im Spiel vorkommenden Aktionen wieder mit Ausnahme der Bauvorschriften. Bauvorschriften werden in Objekten der Klasse `BuildRequirements` reglementiert.

Der Programmcode, welcher bei Ausführung einer beliebigen Aktion zum Tragen kommt, ist nicht in der Datenbank festgelegt. Das hat den Grund, dass es einerseits sehr umständlich ist, konkreten Programmcode[24] in einer Datenbank unterzubringen, andererseits hat dies kaum Vorteile, da sich die grundsätzlichen Aktionen von Spiel zu Spiel wenig unterscheiden. Es existieren Operationen zum Fortbewegen von Einheiten, zum Abfeuern von Munition auf andere Einheiten, sowie zum Transfer von Ressourcen. Was allerdings über das Datenbankmodell konfigurierbar ist, sind die Bedingungen, unter denen bestimmte Aktionen vollzogen werden können und deren Konsequenzen, etwa im Verbrauch der Ressourcen.

Die Klasse, mit der die Aktionsregeln gespeichert festgelegt werden, hat fünf hierfür wichtige Felder und eine N-zu-1-Beziehung zur Entität `Resource`. Das Feld `costs` gibt an, wie viele Maßeinheiten des verknüpften Rohstoffes je Aktion verbraucht werden. Beim Feuern von Raketen könnten dies beispielsweise zwei Raketen sein, die zum Ausführen der Aktion vorhanden sein müssen und als Folge der Aktion von den Reserven abgezogen

[24] mit Ausnahme von Taktik-Skripten

werden. Die Eigenschaft `effect` ist eine Art Multiplikator für den Effekt der Aktion. Im gegebenen Beispiel in Listing 3.6 auf der nächsten Seite würde ein Einschlag der beiden Raketen einen Schaden verursachen, der um den Faktor 5 größer ist als der „Einheitsschaden“. Gerade für Kampfszenarien hat die Eigenschaft `radius` eine wichtige Bedeutung. Sie gibt Auskunft darüber, in welchem Radius um die angreifende Einheit feindliche Einheiten angreifbar sind. In unseren „Doppelraketen“-Beispiel würde die Einheit des Modells, dem die angegebene Regel zugeordnet ist, Schiffe im Umkreis von sieben Felder[25] attackieren können.

```
from megaira.model.resources import Resource
from megaira.model.actions import ActionRule

rule = ActionRule(
    costs = 2,
    resource = missiles, # siehe oben
    effect = 5,
    radius = 7,
    action = u'attack.missile',
    per_unit = 1,
)
```

Listing 3.6: Megaira: Aktionsregeln

Das letzte Feld, `per_unit`, ist dazu gedacht, dass in bestimmten Fällen die Kosten etwas feiner Kontrolliert werden können. Seien für eine Aktion „bewegen“ als Kosten zwei Gallonen Öl angegeben, das Feld `per_unit` aber auf den Wert 3 gesetzt, so würde eine Bewegung der Einheit über sechs Felder genau vier Gallonen Öl kosten, errechnet aus der Formel in Gleichung (3.1), wobei a der Betrag („amount“) der zurückgelegten Felder und R die zugrundeliegende Regel ist. Die genannten Werte eingesetzt erhalten wir Gleichung (3.2).

$$k(a, R) = \frac{costs_R \cdot \mathrm{a}}{per\_unit_R} \tag{3.1}$$

$$k(6, R) = \frac{costs_R \cdot \mathrm{a}}{per\_unit_R} = \frac{2 \cdot 6}{3} = 4 \tag{3.2}$$

In Megaira können Einheiten andere Einheiten bauen. Um genau zu definieren, welche Modelle Einheiten welcher Modelle bauen dürfen und welche zeitlichen und materiellen Kosten dadurch verursacht werden, existiert die Klasse `BuildRequirements`, die analog zu `ActionRule` für die Aktionsregeln die Bauvorschriften kapselt. Eine Bauvorschrift hat zwei

[25] auch hier wieder diagonal möglich

N-zu-1-Beziehungen zur Klasse `Model` im Paket `units`. Zum Einen muss dasjenige Modell angegeben werden, das den Bauauftrag ausführt, im Beispiel in Listing 3.7 auf der nächsten Seite ist dies das Bauschiff, zum Anderen ist natürlich zu spezifizieren, Einheiten welchen Typs gebaut werden dürfen. Hier baut ein Bauschiff („Builder“) ein Patrouillenboot und verbraucht dazu 375 Gallonen des Rohstoffes Öl.

In der Beispielkonfiguration sind die Bauvorschriften minimal gehalten, da gerade drei Modelle existieren. Anzahl der Modelle und Bauregeln sind allerdings praktisch unbegrenzt und beliebig erweiterbar, bis hin zu komplexen Technologiebäumen (engl. „Techtrees“).

```
1 from megaira.model.acgtions import BuildRequirements
2
3 requirements = BuildRequirements(
4     builder = builder,
5     target = patrolboat,
6     costs = 375,
7     resource = oil,
8 )
```

Listing 3.7: Megaira: Bauvorschriften

Globale Objekte

Zu Beginn von Abschnitt 3.1 und an einigen anderen Stellen dieser Arbeit wird unter anderem Beschrieben, dass in Megaira Rundenwechsel existieren, zu denen regelmäßig Ressourcen gefördert und abgerechnet und diverse andere Aufgaben ausgeführt werden. Von der Klasse `GlobalState` im Paket `globals` existiert stets genau ein Objekt, welches den Zeitpunkt der letzten Ausführung des Rundenwechsels speichert. Die einzige Methode der Klasse heißt `tick()`. Diese führt den Rundenwechsel durch.

3.3.2 Die Implementierung der Regelbasen

Nachdem wir in Abschnitt 3.1 gesehen haben, wie Taktiken mit Hilfe des den Taktikeditors angelegt, bearbeitet und Einheiten zugewiesen werden können, und nachdem beschrieben wurde, wie das Modell der Anwendung intern aufgebaut ist und funktioniert, wird in diesem Unterabschnitt auf das Datenbankmodell für die Taktiken eingegangen, dessen Klassendiagramm in Abbildung 3.20 auf der nächsten Seite abgebildet ist.

Die aus dem letzten Unterabschnitt bekannten Klassen `Player`, `Unit` und `Model` sind hier stark vereinfacht dargestellt, auf sie wurde bereits näher eingegangen. Das zentrale Objekt im Diagramm ist die Klasse `Tactic`. Hier werden Name (Feld `name`) und Beschreibung (Eigenschaft `description`) der Taktik gespeichert, sowie ein Indikator `running`, ob die

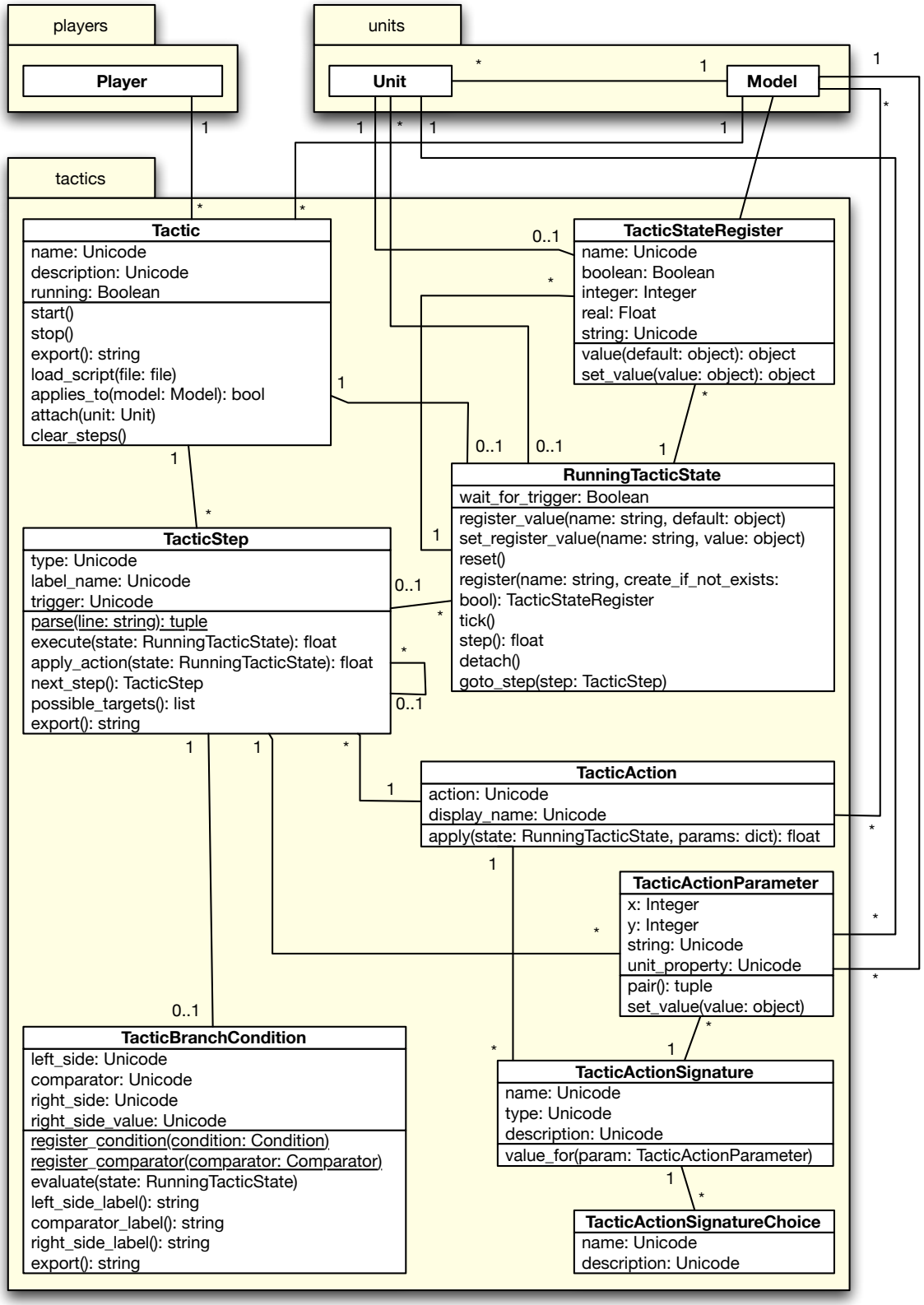

Abbildung 3.20: Das Klassen- und Datenbankmodell der Taktiken in Megaira

Taktik global läuft oder pausiert[26]. Die Methoden `start()` und `stop()` erlauben das globale Pausieren und Fortsetzen der Taktik für alle zugeordneten Einheiten. `export()` liefert eine Repräsentation der kompletten Taktik in Form einer Textdatei, wie in Abschnitt 3.1 beschrieben, die mittels `load_script(file)` wieder importiert werden kann. Ein Beispiel einer exportierten Taktik ist in Listing 3.1 auf Seite 33 zu sehen. Sämtliche Schritte der Taktik können mittels `clear_steps()` entfernt werden. Die Methode `applies_to(model)` erteilt Auskunft darüber, ob die Taktik für ein gegebenes Objekt einsetzbar ist. Ist dies der Fall, so kann sie mit `attach(unit)` einer Einheit zugeteilt werden.

Wird eine Taktik einer Einheit zugeordnet, so wird eine neue Instanz der Klasse `RunningTaticState` erzeugt, welche mit der entsprechenden Taktik und der zu steuernden Einheit in Relation gesetzt wird. Das Zustandsobjekt kann beliebig viele Register der Klasse `TacticStateRegister` besitzen, die das Speichern verschiedener Typen wie Strings, booleschen Werten, Zahlen etc. erlauben. Die oben beschriebene Übertragung der Rückgabewerte des letzten Taktikschritts zum nächsten erfolgt genau über diesen Mechanismus im Register mit dem Namen `last-actions-status`. Der Zugriff auf die Register erfolgt indirekt über die Methoden `register_value(name)` (lesend) und `set_register_value(name, value)` (schreibend). Das Registerobjekt zu einem gegebenen Namen liefert und erstellt bei Bedarf `register(name)`. Zu jedem Rundenwechsel führt das System für jedes Zustandsobjekt die Methode `tick()` aus, welche die Ausführung des nächsten Schritts anstößt, sofern die Taktik weder global pausiert ist noch nach Ausführung eines Trigger-Schritts auf ein Ereignis oder eine Spielsituation wartet. Der Name des Auslösers, auf den die Taktik in diesem Fall wartet, wird im Feld `wait_for_trigger` gespeichert, das ansonsten leer ist.

Die Implementierung der Taktikschritte `TacticStep` kommt ohne Vererbung aus. Ursprünglich war die Umsetzung auf diese Weise etwas einfacher und flexibler, möglicherweise wird das Modell in zukünftigen Versionen umgestellt werden und jeder Taktikschritt von einer Basisklasse erben. Die Art des Schritts wird über die Eigenschaft `type` festgelegt, welche als Wert eine der Konstanten `TacticStep.ACTION`, `TacticStep.BRANCH`, `TacticStep.LABEL`, `TacticStep.END` oder `TacticStep.TRIGGER` annehmen kann. Das Feld `trigger` speichert den Bezeichner des Ereignisses, bei dem der Auslöser evaluiert werden soll. Bei Ausführung eines Trigger-Schrittes wird dieser Wert in das Feld `wait_for_trigger` des `RunningTaticState`-Objekts übernommen. Die Methode `next_step()` liefert den Folgeschritt der Taktik, die übrigen Methoden werden zum Import und Export der Taktiken (`parse(line)` sowie `export()`) und zur Ausführung von Aktions-Schritten (Methode `apply_action(state)`) eingesetzt.

Bedingte Sprünge haben beliebig viele Bedingungen der Klasse `TacticBranchCondition`, die je Bedingung drei oder vier Werte enthält. Der erste Wert ist die Linke Seite des Ausdrucks `left_side`. Hier steht einer von mehreren auszuwählenden Ausdrücken, welcher bei Prüfung der Bedingung ausgewertet und verglichen wird. Mögliche Werte sind etwa

[26] etwa währenddem sie erstellt oder modifiziert wird

„Anzahl feindlicher Einheiten in Sichtweite“, „Freund/Feind-Verhältnis“ oder „Status der Ausführung des zuletzt ausgeführten Befehls“. Als zweiter Wert wird einer der gängigen Vergleichsoperatoren gewählt („ist“, „$\neq$“, „$\leq$“, „$\geq$“, „$<$“, „$>$“, ...). Mittels diesem wird dann die linke Seite mit der rechten Seite (Eigenschaft `right_side`), entweder eine von mehreren vordefinierten Konstanten („wahr“, „falsch“, „Fehlschlag“, „Erfolg“) oder ein frei einzugebender Wert (z.B. „78%“), der dann in das Feld `right_side_value` geschrieben wird, verglichen. Gibt der Ersteller der Taktik für einen bedingten Sprung mehrere Bedingungen an, so werden diese durch ein logisches *Und* verknüpft.

3.4 Die API für die Experimente

Um die Funktionalität der Taktiken, der Implementierung derselben und deren Möglichkeiten zu untersuchen, wie in Abschnitt 4.1 beschrieben ist, wurde eine kleine Mini-API[27] entwickelt, welche praktische Funktionen zur schnellen Konstruktion von Testszenarios bereitstellt und gestattet, bequem einzelne, in Klassen gekapselte Versuche aufzustellen, auszuführen und auszuwerten. Die Experimente werden unabhängig vom laufenden Spiel in einer eigenen Umgebung („Sandbox“) durchgeführt.

Wir wollen dieses Hilfsmittel kurz anhand eines praktischen Beispiels beschreiben, dessen Quellcode in Listing 3.8 gegeben ist. Zu Beginn des Skriptes wird die Basisklasse für alle Experimente `Experiment` importiert. Das Experiment `Experiment1`, welches von dieser Klasse erbt, definiert bzw. überschreibt die Anzahl der maximal auszuführenden Iterationen[28], so dass bei durchführung dieses Versuchs maximal 50 Schleifendurchgänge wiederholt werden. In der Methode `setup_scene()` generiert der Versuch sein Szenario. Zwei Spieler werden erstellt, jeder von ihnen bekommt ein Schiff, dem jeweils eine eigene Taktik zugewiesen wird. Die Taktiken werden über mehrzeilige Strings „inline“ übergeben.

Die Methode `run()` schließlich führt das Experiment durch. Die Szene wird zuerst zurückgesetzt, dann mittels der beschriebenen Methode neu aufgebaut. Einheiten beider Spieler werden nun gezählt und über die Standardausgabe ausgegeben. Anschließend wird iteriert. Entweder so lange, bis die oben angegebenen 50 Durchläufe erreicht sind, oder bis sich die Anzahl der Einheiten eines der beiden Spieler ändert. Die Anzahl der ausgeführten Schritte sowie jeweils die Anzahl der Einheiten werden ausgegeben.

```
1 from megaira.experiments import Experiment
2
3
4 class Experiment1(Experiment):
5     iterations = 50
6
```

[27] API = „Application Programming Interface“, zu Deutsch kurz „Programmierschnittstelle“
[28] das sind die simulierten Ticks

```
7       def setup_scene(self):
8           self.a = self.create_player(u'A')
9           self.b = self.create_player(u'B')
10          b = self.a.create_unit(u'A', u'patrolboat', (-2, 0))
11          p = self.b.create_unit(u'B', u'builder', (2, 0))
12
13          # A: seek and destroy
14          self.assign_tactic(self.a.load_tactic(u"""
15              .name Seek and destroy next enemy
16              .model patrolboat
17
18              loop:
19                  intelli_moveto (destination=next-enemy-unit)
20                  go to [loop] if enemies-in-sight = 0
21
22              attack:
23                  attack
24                  go to [end] if last-action-status = success
25                  intelli_moveto (destination=next-enemy-unit)
26                  go to [attack] if enemies-in-sight > 0
27                  go to [loop]
28
29              end:
30                  end
31          """), p)
32
33          # B: run and hide
34          self.assign_tactic(self.b.load_tactic(u"""
35              .name  Run on attack
36              .model builder
37
38              wait:
39                  wait until under-attack
40
41              loop:
42                  intelli_moveto (destination=next-safe-place)
43                  go to [wait]
44          """), b)
45
46      def run(self):
47          self.reset()
48          self.setup_scene()
49
```

```
50          c0 = self.count_units(self.a),  self.count_units(self.
               b)
51          print "#A = %d vs. #B = %d" % c0
52          for i in self.iterate():
53              c1 = self.count_units(self.a),  self.count_units(
                   self.b)
54              if c1 != c0: # abort if there's a change
55                  break
56              print i
57
58          print "#A = %d vs. #B = %d" % c1
59
60  Experiment.register(Experiment1)
```

Listing 3.8: Megaira: ein grundlegendes Experiment auf Basis der Versuchs-API

Zuletzt wird der Versuch bei der Basisklasse registriert, um in die Ausführung aller Versuche integriert zu werden.

Nicht nur über die Taktik-Import/Export-Schnittstelle der Web-Oberfläche von Megaira lassen sich Algorithmen zum maschinellen Lernen, evolutionäre Algorithmen und ähnliche Verfahren anwenden, sondern natürlich auch über die Schnittstelle für die Experimente. Da die Schnittstelle isoliert vom Spiel verläuft ist Online-Lernen ohne weitere Modifikationen nicht direkt möglich, jedoch lassen sich Taktiken durch entsprechende Versuchsszenarien automatisch optimieren und dann später ins laufende Spiel importieren.

Verschiedene Experimente, die im Rahmen dieser Arbeit mit Hilfe der beschriebene Methoden durchgeführt und bewertet wurden, sind in Abschnitt 4.1 eingehend beschrieben.

Kapitel 4

Versuche und Ergebnisse

Das folgende Kapitel ist drei Abschnitte unterteilt. Der erste Abschnitt beschreibt, erklärt und bewertet Versuche, die mit Hilfe von Megaira, welches ausführlich in Kapitel 3 beschrieben ist, durchgeführt wurden, um festzustellen, ob die in Kapitel 1 und Kapitel 2 vorgestellten Methoden sinnvoll anwendbar sind und die These, dass die genannten spielerunterstützenden Verfahren in Browserspielen von Vorteil sein können, unterstützen.

Im zweiten Teil wird versucht, die Fragen, die zu Beginn aufgeworfen wurden, insbesondere bezüglich Abschnitt 1.3 zu beantworten. Der letzte Teil schließlich geht auf die etwaigen Probleme ein, die sich in Folge der neu eingeführten Funktionen ergeben können und versucht Ideen für mögliche Lösungsansätze anzubieten.

4.1 Experimente mit Taktiken

Auf Basis der in Abschnitt 3.4 beschriebenen Test-API wurden einige Experimente durchgeführt. Im Folgenden werden wir die durchgeführten Versuche in Bezug auf Aufbau, Durchführung und Verhalten sowie die Ergebnisse der einzelnen Experimente beschreiben und bewerten.

4.1.1 Experiment 1: Funktionalität der Taktiken und deren Implementierung

Im ersten Experiment soll zunächst geprüft werden, ob die Taktiken überhaupt sinnvoll arbeiten und sich in Bezug auf die vorgestellten Methoden der künstlichen Intelligenz und der computational Intelligence für weitere Versuche eignen. Hierzu generieren wir für das erste Experiment einen eher einfachen Versuchsaufbau, nämlich den aus dem letzten Kapitel, Listing 3.8 auf Seite 50.

Aufbau

Für den ersten Versuch werden zwei Spieler erstellt, genannt „A“ und „B“. Spieler A bekommt ein gleichnamiges Patrouillenboot mit Standardausrüstung[1], während Spieler B ein Bauschiff unterstellt wird, ebenfalls mit Standardausrüstung[2]. Schiff „A“ startet an Position $(-2, 0)$, während die Startposition von Schiff „B“ etwas weiter rechts auf dem Feld $(2, 0)$ liegt.

Jedes der beiden Schiffe bekommt eine individuelle Taktik zugeordnet. Schiff A, das Patrouillenboot, wird der Jäger, das Bauschiff die Beute. Die Taktik von Schiff A versucht nun das nächstgelegene feindliche Schiff ausfindig zu machen und anschließend darauf zu feuern, bis dieses entweder zerstört oder außer Reichweite ist. Im letzteren Fall wird die Suche wiederholt, ansonsten die Taktik beendet. Schiff B hingegen wird von der Taktik zunächst in einen Wartezustand versetzt, bis die Einheit durch einen Gegner angegriffen wird. Tritt dieses Ereignis ein, so versucht das Schiff, sich in Sicherheit zu bringen, in dem es diejenige Position in Reichweite ausmacht, von der die feindlichen Schiffe am weitesten entfernt sind, und sich dorthin begibt. Im Anschluss wird wieder zum beschriebenen Wartezustand zurückgekehrt.

Ziel

Ziel dieses Versuchs ist es, herauszufinden, ob die Jäger-Taktik des Patrouillenbootes fähig ist, die Flucht-Taktik des Bauschiffes zu schlagen und dieses innerhalb der vorgegebenen 50 Rundenwechsel zu zerstören. Besonderes Interesse liegt hierbei auch auf der Frage, wie viele Schritte nötig sind, bis der Versuch zum Ende kommt, und welche Wege die Schiffe während der Jagd zurücklegen.

Durchführung

Während jeder Iteration wird die Anzahl der Einheiten jedes Spielers mit der entsprechenden Zahl der vorherigen Iteration verglichen. Wird eine Veränderung festgestellt, so bricht der Versuch ab. Da es zu Beginn je Spieler genau ein Schiff gibt und im Versuch keine Schiffe gebaut werden, ist in diesem Fall eine Einheit zerstört worden.

Wie oben beschrieben starten die Schiffe von den Positionen $(-2, 0)$ und $(2, 0)$. Beide sind völlig unbeschädigt. Das Bauschiff B geht direkt in den Wartezustand auf ein Angriffsereignis über, während das Patrouillenboot A direkt beim ersten Tick mit der Suche nach der nächsten feindlichen Einheit beginnt und sich auf das zum Ziel benachbarte Feld

[1] Die Standardausrüstung für Patrouillenboote in Megaira sind 250 Gallonen Öl und 100 Schuss Maschinengewehr-Munition. Jeder Schritt in ein benachbartes Feld kostet 5 Gallonen Öl, jede mit dem Maschinengewehr abgeschossene Salve 3 Schuss Munition, verursacht aber 5 Schadenseinheiten.

[2] Bauschiffe haben standardmäßig 5500 Gallonen Öl geladen und verbrauchen je zurückgelegtem Feld 10 Einheiten davon.

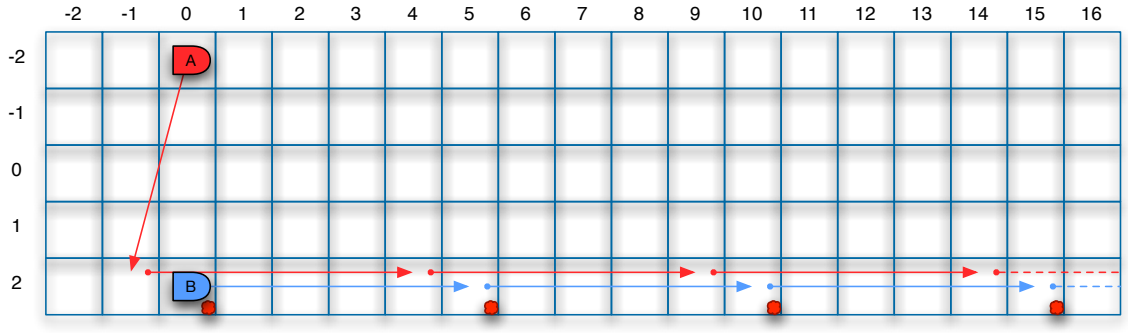

Abbildung 4.1: Schematische Darstellung eines Ausschnitts des Verlaufs des Experiments (aus Platzgründen sind x- und y-Achse vertauscht). Das jagende Patrouillenboot ist rot (A), das gejagte Bauschiff blau (B) markiert.

$(2, -1)$ bewegt. Eine schematische Darstellung des Verlaufs der ersten sieben Schritte des Experiments ist in Abbildung 4.1 zu sehen.

Im nächsten Schritt eröffnet das Patrouillenboot A den Angriff auf das Bauschiff B. In Abbildung 4.1 sind die Detonationen durch kleine rote Punkte rechts unten in den entsprechenden Kacheln symbolisiert. Der Schaden der getroffenen Einheit B steigt von 0% auf 10%. Sie beendet in Reaktion auf den Angriff den Wartezustand sofort und ergreift die Flucht zu den Koordinaten $(2, 5)$.

Das Patrouillenboot A zieht daraufhin nach und begibt sich wieder in unmittelbare Nähe des Bauschiffs B, auf $(2, 4)$. Es wiederholt den Angriff und erhöht damit die Beschädigung seines Opfers auf 20%. Das Spiel setzt sich fort, bis das Bauschiff schließlich nach 20 Rundenwechseln an Position $(2, 45)$ zerstört wird.

Ergebnis und Bewertung

Die ursprünglich gesetzte Grenze von 50 Runden reicht völlig aus, um zu erkennen, dass die Jagd-Taktik des Patrouillenbootes das Bauschiff zerstört und somit gegen die Flucht-Taktik desselben gewinnt. Beide Taktiken sind sehr einfach gehalten, wobei der Jäger-Taktik ein paar kleine Optimierungen gestattet wurden. Jäger sind in dem vorgestellten System eher im Nachteil, da sie abwechselnd verfolgen und angreifen müssen. Jede der beiden Aktionen verbraucht jeweils einen ganzen Zeitschritt. Gejagte können jeden Zeitschritt zur Flucht nutzen und zusätzlich durch Auslöser („Trigger“) asynchron reagieren, also zwischen den Rundenwechseln Schritte einlegen, die den Aktionen des Jägers direkt folgen. Im gegebenen Szenario ist allerdings das gejagte Schiff seinem Jäger bezüglich Geschwindigkeit unterlegen: Das Patrouillenboot kann in jedem Zeitschritt 10 Felder zurücklegen, während sich das Bauschiff nur halb so schnell bewegen kann.

Das durchgeführte Experiment zeigt also, dass Taktiken in Megaira durchaus fähig sind, gegeneinander – und damit grundsätzlich (technisch) auch gegen menschliche Spieler – anzutreten.

4.1.2 Experiment 2: Erfolg der Taktiken in Abhängigkeit des Abstands der Einheiten

Im zweiten Experiment beobachten wir, welche Konsequenzen die Veränderung des ursprünglichen Abstands zwischen den beiden Einheiten aus Experiment 1 hat. Wir werden beide Einheiten schrittweise voneinander weg bewegen, die Folgen überwachen und auswerten.

Aufbau

Der Versuchsaufbau gleicht prinzipiell dem Aufbau des letzten Experiments. Wir führen hier nun mehrere Durchgänge aus, bei denen jeweils die Startposition der beiden Einheiten geändert wird. Bei jedem Durchgang starten jeweils beide weiter voneinander entfernt. Als Abbruchkriterium dient auch hier wieder die Zerstörung der gejagten Einheit oder das Erreichen der 50. Runde. Zu Beginn jedes Versuchsdurchgangs wird jede Einheit um ein Feld von der anderen weg bewegt. So startet das Bauschiff A im ersten Durchlauf an der Position $(1, 0)$ und das Patrouillenboot B bei $(-1, 0)$. Vor der zweiten Ausführung setzen wir Schiff A auf $(2, 0)$ und Schiff B auf $(-2, 0)$, genau wie in Experiment 1. Entsprechend folgen dann die Positionen $(3, 0)$ bzw. $(-3, 0)$, $(4, 0)$ und $(-4, 0)$ bis hin zu $(50, 0)$ und $(-50, 0)$. Wir führen also $n = 50$ Versuche durch für $1 \leq i \leq 50 \in \mathbb{N}$ mit den Positionen $(-i, 0)$ für den Jäger und $(i, 0)$ für das gejagte Objekt bei jedem Versuche i. Somit beträgt der Abstand zwischen den Schiffen im Durchlauf i genau $2i - 1$ Kartenfelder.

Wichtig hierbei ist zu erwähnen, dass in diesem Versuch die Aktion „bewege Dich zur nächsten feindlichen Einheit“ die durch Administratoren konfigurierbare Sichtweite des Subjekts ignoriert[3]. Einerseits erscheint dies unfair, weil die flüchtende Einheit auch nur Gegner im entsprechenden Radius sehen kann. Andererseits darf ein realer Spieler auch beliebige Kartenausschnitte überblicken und darauf hin entscheiden, wohin er seine Schiffe bewegt. Verschiedene Vorexperimente haben gezeigt, dass die Jagd vorschnell beendet ist, wenn der Gejagte das Sichtfeld des Jägers verlässt, also das Verhältnis von Sichtweite zu Reichweite zu gering ist. Der Jäger wird in diesem Fall das Bauschiff nie zerstören können. Zum Vergleich werden wir einige Stichproben ohne diese Modifikation durchführen.

Ziel

Ziel des Experiments ist zu sehen, wie sich der Erfolg der Verteidigung mit zunehmendem Abstand verhält. Sollte der Abstand eine gewisse Größe erreicht haben, dürfte es für den Jäger immer schwieriger werden die Beute zu erreichen. Wir werden im folgenden Experiment versuchen zu zeigen, dass bei genügend großem Abstand der Jäger die Beute nicht mehr einholen kann. Hierzu führen wir das Experiment 50 Mal durch.

[3] bzw. ist die Sichtweite unbegrenzt

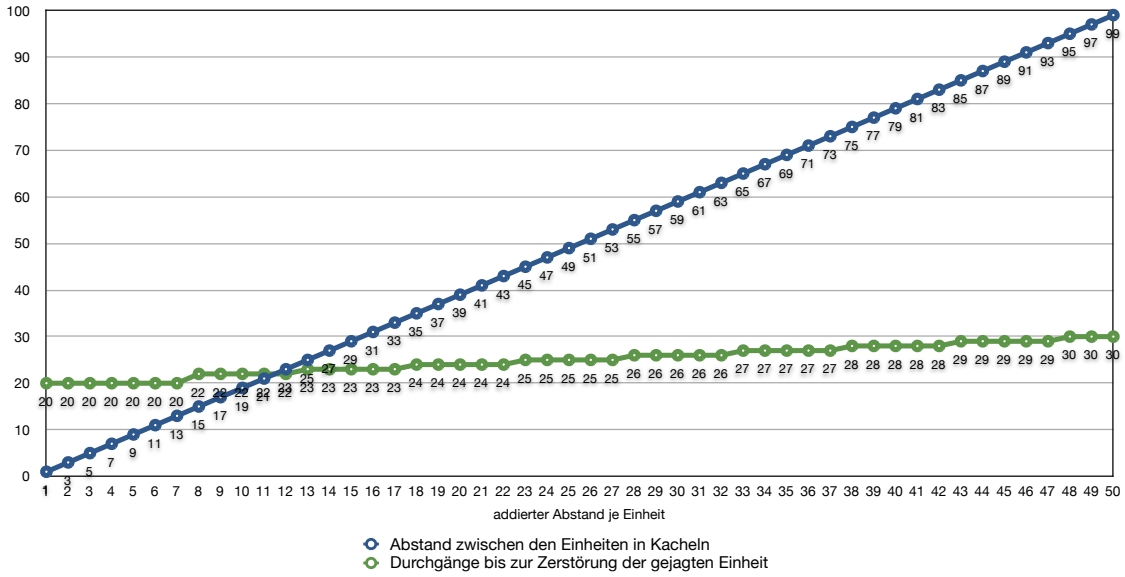

Abbildung 4.2: Diagramm mit den Ergebnissen des zweiten Experiments, Die blaue Linie stellt den Abstand zwischen den beiden Einheiten dar, die grüne die Anzahl der nötigen Durchgänge bis das Patrouillenboot das Bauschiff zerstört hat.

Durchführung

In den ersten sieben Durchläufen benötigt die jagende Einheit jeweils genau 20 Durchführungen bis sie ihr Ziel, die Zerstörung des Bauschiffs, erreicht hat. Danach steigt die Anzahl der benötigten Iterationen auf 22 an, wie Abbildung 4.2 zeigt. Ab hier erhöht sich die Zahl der nötigen Durchgänge jeweils immer nach fünf Versuchen um genau einen Durchlauf, bis hin zum letzten Versuch, bei dem die Jagd über genau 30 Runden läuft. Es scheint, als brauchte der Jäger mit steigendem Abstand verhältnismäßig wenig zusätzliche Schritte. Nach Abbildung 4.2 pendelt sich das Verhältnis offensichtlich ein.

Stichprobenartig wird zuletzt noch ein Versuch für $i = 75$ durchgeführt, bei dem die Schiffe vor Beginn auf die Positionen $(75, 0)$ und $(-75, 0)$ gesetzt werden, also mit 149 Kacheln Abstand. Dieser soll die Vermutung belegen, dass auch später keine große Abweichung von der Regel zu erwarten ist. In dieser Ausführung brauchen die Taktiken 35 Ticks bis zur Zielerreichung. Dies unterstützt genau die Vermutung, dass sich die Versuchsergebnisse linear fortsetzen.

Die Durchführungen, bei denen wir den Sichtbarkeitsradius für die oben genannte Aktion einschränken, zeigen tatsächlich, dass die jagende Einheit schon bei $i = 1$ – also dem Abstand von einer freien Kachel zwischen den beiden Schiffen – sehr geringe Chancen hat, das Ziel zu zerstören. Wir setzen den Sichtbarkeitsradius auf 15 Felder[4].

[4] analog zu den übrigen Aktionen von Einheiten sind auch hier diagonale Schritte möglich, zählen also einfach

Den ersten Treffer erzielt das Patrouillenboot nach dem zweiten Takt. Der Schaden des Bauschiffs steigt auf 3%. Im Abstand von zwei Takten folgen regelmäßig weitere Einschläge, die den Schaden um jeweils 3% erhöhen. Nach erst 50 Runden nimmt das Schiff einen Schaden von 75%. Führen wir den Versuch für weitere $i \in \{2, 5, 6, 7, 8, 9, 10\}$ durch, so entdecken wir die erste Abweichung bei $i = 8$. Hier passiert bis zum letzten Tick nichts. Die Einheiten stehen still, es wird nicht gefeuert. Die Versuche $i > 8$ verlaufen analog.

Ergebnis und Bewertung

Wie wir in Abbildung 4.2 auf der vorherigen Seite sehen können, steigt die Anzahl der nötigen Ticks bis zum Schluss nur sehr langsam an. Eine Stagnation des Erfolgs der Jägertaktik tritt offensichtlich nicht ein. Selbst bei einem Abstand von 149 Kacheln folgen die Ergebnisse noch dem erkannten Muster. Der Grund für dieses Verhalten liegt ganz klar in der Aufhebung der Restriktion des Sichtbereichs der Such-Aktion der Jagd-Taktik. Diese kann Dank der Modifikation immer den nächsten Gegner lokalisieren, sofern ein solcher existiert.

Ziehen wir nun die Beobachtungen des Versuchsteils hinzu, in dem diese Restriktion gültig ist, so fällt auf, dass hier ab einem gewissen Punkt der Abstand zwischen den Einheiten so groß wird ($2 \cdot 8 - 1 = 15$), dass das angreifende Schiff die Beute aufgrund des zu geringen Sichtbereichs von 15 Kacheln nicht mehr sehen kann und aus diesem Grund auch nicht angreift. Hieraus lässt sich den Schluss ziehen, dass die Parameter für die Aktionen vom Spieladministrator sorgfältig gewählt sein wollen.

4.1.3 Experiment 3: Erfolg von einer Defensivtaktik im Vergleich zur Situation ohne programmierbare Agenten

Das dritte Experiment soll den Vorteil von Defensivtaktiken, die schließlich im Mittelpunkt der Idee der programmierbaren Agenten von Megaira stehen, belegen und damit direkt auf die Beantwortung der aufgeworfenen Fragen dieser Arbeit hinarbeiten. Wir werden beobachten, ob ein ungeschütztes Schiff, das ohne Verteidigungstaktik widerstandslos zerstört werden würde, durch eine entsprechende Fluchttaktik gerettet werden und gegebenenfalls den Angreifer sogar in einen Hinterhalt locken und zerstören kann.

Aufbau

Der Versuchsaufbau sieht zunächst ein Jäger- sowie ein Beuteschiff vor, ähnlich wie in den beiden vergangenen Versuchen. Diesmal handelt es sich allerdings um zwei gleich starke Patrouillenboote. Das jagende Schiff startet bei Position $(-5, -5)$, sein potenzielles Opfer bei $(0, 0)$. Zusätzlich existiert nun eine Gruppe von Einheiten, die dem gejagten Schiff freundlich gesinnt ist und außerhalb der Sichtweite dem Angreifer auflauert. Der Hinterhalt besteht aus vier mit Maschinengewehren bewaffneten Patrouillenbooten und aus vier mit

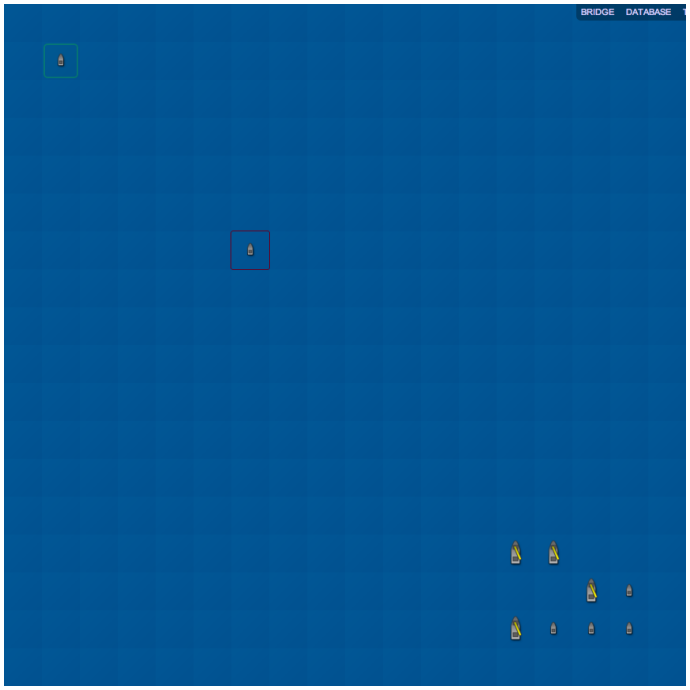

Abbildung 4.3: die Ausgangssituation von Experiment 3: links Oben der Angreifer (grün hervorgehoben), in der Mitte sein Opfer (rot umrahmt), unten rechts der auf den Angreifer wartende Hinterhalt.

Raketen beladenen Bauschiffen. Sämtlichen acht Schiffen sind aktivierte Taktiken zugewiesen, so dass diese auf Feindaktivität im Sichtbereich warten und bei Sichtkontakt sofort losfeuern. Das zum Angreifer nächste Schiff dieser Gruppe befindet sich an der Position $(7, 8)$. Die übrigen Einheiten rechts daneben oder darunter, wie in Abbildung 4.3 dargestellt ist.

Die Taktik des Patrouillenbootes, welches als Lockvogel dient, ist Listing 4.1 auf der nächsten Seite zu entnehmen. Das Objekt wartet per „Trigger“ auf einen Angriff, sucht, sobald dieser erfolgt, die nächstgelegene freundliche Einheit und nähert sich ihr so weit wie möglich. Anschließend geht das Schiff in den Wartemodus über – bis zur nächsten Bewegung einer gegnerischen Einheit in Sichtweite. Bei Erkennung einer solchen wird wieder versucht, die nächste freundliche Einheit anzusteuern und damit der Flucht-Vorgang erneut wiederholt, für den Fall, dass die eigenen Einheiten sich bewegt haben.

Der Versuch besteht aus zwei Stufen. Im ersten Teil bekommt das gejagte Objekt keine Defensivtaktik zugewiesen. Wir stellen uns vor, der Spielteilnehmer, der für die Einheit verantwortlich ist, sei gerade nicht am Bildschirm. Im zweiten Teil wird die Defensivtaktik zugewiesen und aktiviert. Das angreifende Objekt wird in beiden Fällen manuell gesteuert.

Ziel

Ziel des Versuchs ist es zu zeigen, dass ein Schiff, das nicht durch eine entsprechende Taktik abgesichert ist, seinem Angreifer hilflos ausgeliefert ist. Wir möchten sehen, ob durch Zuweisung eines entsprechenden Skripts nicht nur das Schiff gerettet, sondern eventuell sogar der Angreifer zerstört werden kann. Der Nutzen für den Einsatz von Verteidigungstaktiken bei Abwesenheit oder Unaufmerksamkeit des Spielers soll damit deutlich werden.

Durchführung

Der erste der beiden Versuchsteile ist sehr einfach. Der Jäger greift sein Opfer mit dem Maschinengewehr an. Die Ausgangssituation ist wie in oben beschrieben in Abbildung 4.3 auf der vorherigen Seite dargestellt. Das angegriffene Schiff bleibt regungslos, da es ja weder von einer Taktik beschützt wird, noch sein Besitzer anwesend ist. Die angreifende Einheit braucht seine Position nicht anzupassen, denn das Opfer bleibt in Reichweite. Jeder Schuss verursacht 6% Schaden. Nach 17 Treffern ist das Opfer zerstört. Der Angreifer ist noch immer unbeschadet.

Im zweiten Teil bekommt die zu attackierende Einheit die Verteidigungstaktik zugewiesen, die in Listing 4.1 skizziert ist. Hier ändert sich der Ablauf. Zuerst feuert der Angreifer auf das Patrouillenboot. Bei diesem wird die Fluchttaktik aktiviert und es bewegt sich zur Gruppe, auf das Feld $(7, 7)$. Hier ist es für den Angreifer nicht mehr erreichbar. Er versucht sich jetzt seinem Opfer soweit zu nähern, dass er es weiter beschießen kann. Das attackierende Schiff bewegt sich an die Position $(4, 4)$, von der es das Patrouillenboot theoretisch erreichen könnte. Allerdings löst es die „Trigger“ der Einheiten des Hinterhalts aus, worauf es von allen vier Bauschiffen und den übrigen vier Patrouillenbooten simultan beschossen und sofort zerstört wird.

```
1     wait until under-attack
2
3 follow:
4     intelli_moveto (destination=next-friendly-unit)
5     wait until enemy-approaching
6     go to [follow]
```

Listing 4.1: Megaira: Fluchttaktik

Ergebnis und Bewertung

Das Experiment verlief erfolgreich. Wie wir zeigen wollten, konnte das ungeschützte Schiff vom Angreifer sehr schnell zerstört werden. Der Spielteilnehmer, dem die angegriffene Einheit gehört, würde nach der nächsten Anmeldung am Spielsystem feststellen, dass er die Einheit verloren hat. Solche Szenen treten bei Browserspielen häufig auf. Im zweiten

Teil sehen wir, dass die Defensivtaktik das Patrouillenboot in Sicherheit bringt und die dort wartenden Einheiten den sich nähernden Gegner – ebenfalls automatisiert – zerstören. Der abwesende Spieler kann sich und seine Einheiten also in Sicherheit wiegen.

4.2 Lassen sich Methoden der KI/CI in Browserspielen sinnvoll einsetzen?

In Kapitel 1 wurden einige allgemeine Probleme bei Browserspielen angesprochen und in Abschnitt 1.3 grob angedeutet, wie diese unter Zuhilfenahme von Methoden der künstlichen Intelligenz und der computational Intelligence zu lösen sein könnten. Anschließend sind wir auf bekannte und verbreitete Techniken und Verfahren eingegangen und haben anhand der Beispielanwendung „Megaira“ gezeigt, wie die theoretischen Vorschläge und Lösungen in der Praxis realisierbar sind. Zuletzt wurden Tests und Experimente vorgestellt, durchgeführt, beobachtet und bewertet, die die beschriebenen Methoden umsetzten und erprobten.

4.2.1 Warum maschinell simulierte Gegner bei Browserspielen wenig sinnvoll sind

Wie in der Einleitung und in Abschnitt 2.1 erwähnt, leben Browserspiele von der Gemeinschaft, der Masse an realen Spielern. Verschwörungen und Bildung von Clans und Allianzen stehen im Vordergrund, unterstützt von Kommunikation zwischen den Spielern. Gewissermaßen kann man Browserspiele als eine alternative, spielerischere Form von sozialen Netzwerken begreifen. Computergesteuerte Gegner, NPCs, können an diesem Geschehen schwerlich bis unmöglich teilhaben und stellen bezüglich den genannten, wichtigsten Aspekten von Browserspielen keine oder keine nennenswerte Bereicherung dar. Selbstverständlich ließen sich einfache NPCs als leichte Beute in solchen Spielen unterbringen. Doch würden diese aus den genannten Gründen eine untergeordnete Rolle spielen. Die Praxis zeigt schließlich, dass hier KI-Gegner seit Existenz des Genres bis heute nicht an Relevanz gewinnen konnten.

4.2.2 Automatisierte Steuerung von Einheiten des Spielers durch den Spieler selbst

Als besonders interessant und spannend stellt sich das Erweitern der Browserspiele um programmierbare Agenten heraus, die mühsame Aufgaben übernehmen, auf Ereignisse und Situationen reagieren und grundsätzlich in verschiedenen Szenarien den Spieler bei seinen Bemühungen, die Spielziele umzusetzen, unterstützen. Während der Implementierung der Taktiken in Megaira und bei den ersten Versuchen, diesen mehr oder weniger sinnvolle

Aufgaben zuzuteilen, war bereits deutlich zu sehen, dass es sich hierbei um ein mächtiges Hilfsmittel handelt, welches dem Spieler unzählige[5] Möglichkeiten bieten kann.

Die Experimente in Abschnitt 4.1 vermitteln einen Eindruck dessen, was mit Taktiken machbar ist und wie sich diese in der Praxis verhalten. Der Taktikeditor von Megaira, der in Abbildung 3.9 vorgestellt wurde, bietet eine Übersicht über alle möglichen Taktikschritte und motiviert zum kreativen und intuitiven Programmieren der Einheiten.

4.2.3 Automatische Verteidigung/Kampfführung auch während physikalischer Abwesenheit des Spielers

In Experiment 3 in Abschnitt 4.1 wurde verdeutlicht, wie sich Einheiten bei Abwesenheit des Spielers mit Hilfe der vorgestellten Taktiken beschützen lassen und wie Kampfhandlungen insbesondere zur Verteidigung automatisiert werden können. Ein angreifendes Schiff wurde in eine Falle gelockt und dann zerstört. Würde der angreifende Gegner in dieser Situation mit ähnlich ausgerüsteten[6] Schiffen nachrücken, um wiederholt anzugreifen, so scheiterten diese genau wie ihr Vorgänger.

Die Art und Weise, in der die programmierbaren Agenten in „Megaira“ umgesetzt sind, ist für Angriffstaktiken zwar geeignet, bietet aber die größeren Vorteile in Verbindung mit Verteidigungsstrategien. Die asynchrone Funktionsweise der Auslöser/„Trigger“ unterstützt besonders deren reaktiven Charakter. Angegriffene oder sich durch Feindaktivität in Bedrohung befindliche Einheiten können ohne Verzögerung auf Drohkulissen reagieren. Aktive Taktiken, wie die in Experiment 1 beschriebene „Such- und Zerstör“-Taktik, warten nach der Ausführung jedes einzelnen Schritts auf den nächsten Rundenwechsel, der bei Megaira in 15-Minuten-Intervallen durchgeführt wird. Einerseits ist dies ein gravierender Nachteil für den Angreifer. Andererseits lassen sich auf diese Art und Weise solche Taktiken schwieriger umsetzen, die sofort alle Einheiten der Gegner zerstören, noch bevor diese überhaupt Zeit hatten, sich einen Überblick über das Spiel zu verschaffen und eigene Verteidigungsstrategien zu entwickeln.

4.2.4 Programmierbare Agenten als Hilfsmittel zur Steuerung großer Zahlen von Einheiten

In Experiment 3 wurde eine ganze Gruppe verschiedener Einheiten automatisiert. Obwohl diese sich nicht vom Fleck bewegt hat, erzielte sie einen großen Erfolg durch die unmittelbare Zerstörung eines angreifenden Schiffs. Wären diese Einheiten manuell nacheinander durch einen Spielteilnehmer gesteuert worden, so hätte der Angreifer eine reale Chance gehabt entweder weiter zu feuern oder die Flucht zu ergreifen. Dadurch, dass die Auslö-

[5] insbesondere, wenn vielfältige mögliche Aktionsschritte und diverse Modelle mit unterschiedlichen Ausrüstungen und Fähigkeiten durch den Spieladministrator definiert sind und bereit stehen

[6] hier speziell bezüglich der Panzerung

ser simultan reagieren und alle Schiffe aus dem Hinterhalt sofort angreifen lassen, wird die sich nähernde gegnerische Einheit unmittelbar zerstört. Der gegnerische Spieler hat keine Chance sich zu verteidigen. Der Vorteil, den programmierbare Agenten in diesem Zusammenhang bieten, wird offensichtlich.

Der Nutzen, den Taktiken in Verbindung mit großen Zahlen von Einheiten eines einzelnen Spielteilnehmers haben, ist auch aus einer anderen Perspektive nicht zu unterschätzen. Hat ein Spieler nach einiger Zeit eine große Menge an Objekten erschaffen oder gesammelt, wird die Situation zunehmend unübersichtlich. Er kann nicht permanent jeder einzelnen Einheit seine Aufmerksamkeit widmen, sie aktiv beschützen oder zum eigenen Spielvorteil steuern. Weist er nun entsprechende Taktiken zu – sei es wie im vorigen Unterabschnitt beschrieben zur Verteidigung oder auch zu anderen Zwecken – so kann er seine Errungenschaften halten oder gar effektiv einsetzen und für sich arbeiten lassen.

4.3 Neue Probleme, die resultieren könnten und mögliche Lösungen

Jede Änderung bringt eine neue Situation und damit oft auch verbundene Nachteile. Das Hinzufügen von neuen Funktionen, wie programmierbaren Agenten in ein bestehendes Spielkonzept, ist ein drastischer Eingriff und verlangt besondere Überlegungen und Vorkehrungen zur Aufrechterhaltung der Spielbalance.

4.3.1 Das Spiel spielt sich selbst und wird schnell langweilig

Bei dem Gedanken, dass Spielelemente nicht mehr von den Spielern direkt gesteuert werden, sondern über programmierte Skripte automatisiert agieren, kann der Eindruck entstehen, dass ein solches Spiel für den Spieler selbst schnell langweilig wird[7]. Denke man an zwei gegeneinander antretende Schachcomputer, ein Szenario, das für Wissenschaftler und Schachprofis interessant sein mag, aber den durchschnittlichen Gelegenheitsspieler eher abschrecken dürfte.

Betrachtet man die Umsetzung der Taktiken in „Megaira“, so wird klar, dass programmierbare Agenten durchaus parallel zum Spielbetrieb ihre Arbeit verrichten können, während die automatisierten Einheiten zusätzlich manuell steuerbar bleiben – entsprechend konzipierte Taktiken vorausgesetzt. Gemäß der altbekannten These „Computer sind dumm und schnell“ ist die einzige wirkliche Bedrohung für den Spaß des menschlichen Spielers die Geschwindigkeit, mit der NPCs und programmierbare Agenten ihre Spielzüge umsetzen, legt man ein einfaches System wie das vorgestellte zugrunde. Die Geschwindigkeit ist hier jedoch auf einen Ausführungsschritt je Viertelstunde begrenzt, wodurch diese Ge-

[7] ein Einwand, der unter anderem bei der ersten Vorstellung der Idee der programmierbaren Agenten im Rahmen des Seminars „KI/CI in Computerspielen“ [36] vorgebracht und diskutiert wurde

fahr unterbunden wird. Die Ausnahme der Einschränkung bilden die Auslöser, die ohnehin nur reagieren können. Taktiken können also zusätzlich und spielerunterstützend eingesetzt werden, ohne denselben zu ersetzen.

Die Möglichkeit, dass Spielteilnehmer ausgeklügelte Taktiken entwerfen, die dann ausschließlich gegeneinander antreten, und gleichzeitig das aktive, direkte Spiel aufgeben, besteht weiterhin. Lockerte man die Restriktion der Verzögerung der Ausführung von Taktiken oder höbe man sie komplett auf, so könnte diese Erscheinung als eine Verlagerung des Spielniveaus in eine höhere Ebene betrachtet werden. Langeweile wäre selbst in diesem Fall nicht zu befürchten. Der Erfolg des Spiels „Core War“[8] aus den frühen 80er Jahren, in dem die Spieler Assemblerprogramme in einer virtuellen Maschine gegeneinander antreten lassen, belegt das. Mehr Informationen zu diesem Klassiker sind unter anderem [19] und [52] zu entnehmen. Der Spielinhalt ist zwar ein völlig anderer, doch die Motivation des „gegeneinander Programmierens“ ist prinzipiell dieselbe. Vergleichbar sind ebenfalls die weltweit beliebten Roboterfußballturniere der Forschungsinitiative „RoboCup“ [44]. Browserspiele sind im Vergleich zu den genannten Beispielen von langfristiger Natur, geht es hier nicht um einzelne Turnierwettkämpfe sondern um eine große, theoretisch niemals endende Spielentwicklung. Ferner haben sie den gravierenden Vorteil, dass jeder Mensch weltweit, ortsunabhängig[9] an ihnen teilnehmen kann und dies bei – im Vergleich zu „RoboCup“ – sehr geringem Material- und Transportaufwand.

4.3.2 Spieleinsteiger noch stärker benachteiligt

Spieleinsteiger bekommen durch Taktiken einen fairen Vorteil, da sie die wenigen Einheiten, die sie in der Regel zu Beginn des Spiels haben, mit Hilfe von Defensivtaktiken besser schützen können. Dagegen spricht natürlich, dass es nur darauf ankommt, wie ausgefeilt die Strategien und (gegebenenfalls programmierten) Taktiken der Mitstreiter sind.

Zusätzlich entsteht die Barriere, dass die Neueinsteiger, die das Taktiksystem nutzen möchten, zu allererst lernen müssen, wie sie ihre Skripte programmieren können. Bei der Überzahl der existierenden herkömmlichen Spiele ist dies eine nicht triviale Anforderung, insbesondere für Menschen, die noch nie programmiert haben oder es nicht möchten.

Auf die Einschränkungen von Offensivtaktiken sind wir im vorhergehenden Unterabschnitt bereits eingegangen und haben festgestellt, dass dies kein tatsächlicher Nachteil ist. Das Erlernen der Konstruktion von Taktiken in „Megaira“ stellt an durchschnittlich begabte Spieler keine großen Herausforderungen. Das System setzt nicht, wie die meisten anderen Spiele, auf eine herkömmliche Skriptsprache, sondern kommt mit einem visuellen, benutzerfreundlichen Regeleditor – eingehend beschrieben in Abschnitt 3.1, der kinderleicht zu bedienen ist, und weder Kenntnisse von Programmiersprachen noch das erlernen spezieller APIs voraussetzt.

[8] später auch bekannt unter dem Namen „Core Wars“
[9] Internetzugang vorausgesetzt

4.3.3 Spieler werden zu Programmierern

Zuletzt steht die Kernfrage im Raum, wie sich die vorgestellten Methoden auf den wesentlichen Aspekt, den Sinn und Zweck eines Computerspiels auswirken: Erhöht sich der Spielspaß durch die Bereitstellung eines Systems für programmierbare Agenten? Aufgrund des hoch subjektiven und philosophischen Charakters dieser Frage ist es sehr schwierig, eine pauschale Antwort zu geben. Empirische Studien an großen Menschenmengen hätten den Rahmen dieser Arbeit bei weitem gesprengt. Der Reiz an der schnellen und einfachen Programmierung von Agenten, dem Machtgefühl, durch etwas versteckte, im Voraus geleistete Arbeit einen Effekt zu erzielen, der mit dem gewöhnlichen, manuellen Steuern der Spielelemente nicht herbeizuführen wäre, ist sehr verlockend und wurde bei der Entwicklung von „Megaira“ und während den Experimenten deutlich spürbar. Voraussetzung, um einer möglichst breiten Masse an Spielern mit diesen Mitteln Freude zu bereiten, ist hier freilich die Bereitstellung von benutzerfreundlichen Werkzeugen, die keine bis wenig Vorkenntnisse und einen minimalen Lernaufwand erfordern.

Kapitel 5

Ausblick

Die in dieser Arbeit vorgestellte Software „Megaira“ ist ein vollständiges, spielbares Browserspiel, welches alle Funktionen besitzt, damit verschiedene Teilnehmer Einheiten bauen, navigieren und gegeneinander kämpfen lassen können. Die Spieler können ihre eigenen Taktiken erstellen und sie den Spielobjekten zuweisen, um diese automatisiert steuern zu lassen. Das Spiel ist flexibel und erweiterbar, wie in Abschnitt 3.3 detailliert beschrieben wurde. Trotzdem gibt es sehr viele Punkte, die noch verbesserungswürdig und ausbaufähig sind.

Die vorgestellten Aktionen sind nur eine Minimalausstattung. Insbesondere für die Taktiken sind weitergehende Aktionen wünschenswert. Die Registerspeicher, die durch Aktionsschritte gesetzt und dann von Sprungbedingungen ausgewertet werden können – siehe Abschnitt 3.3, dürften hierzu noch intensiver genutzt werden. Darüber hinaus wäre ein Taktikschritt wünschenswert, der die gesteuerte Einheit veranlasst, weitere Objekte zu konstruieren. In Verbindung mit einer Sprungbedingung, die Ressourcenbedarf und Produktionsmengen auswertet, könnte man hier bei Bedarf etwa Silos und Tanks bauen oder neue Bohrinseln und Minen errichten. Besonders in Verbindung mit der in Abschnitt 2.2 vorgestellten Fuzzy Logic könnten sich hierdurch selbstregulierende Systeme kreieren lassen, so dass die Transportfähigkeit durch die Rohstoffbestände gesichert wäre und überschüssige geförderte Ressourcen nicht verfallen würden.

Der 15-Minutentakt der Rundenwechsel, an den die Ausführung der Skripte gekoppelt ist, ist, wie in Abschnitt 4.3 angesprochen, nicht in jeder Hinsicht optimal. Die Angriffstaktik, die in Experiment 1 in Abschnitt 4.1 zur Zerstörung seines Opfers genau 20 Rundenwechsel[1] brauchte, würde im realen Spielablauf volle fünf Stunden in Anspruch nehmen.

Weitere Ausstattungen, die „Megaira“ vermissen lässt, sind die Mehrspielerfunktionen, die in Browserspielen besonders wichtig sind. Hierzu zählt zum Beispiel ein umfassendes Nachrichtensystem, über das die Spieler untereinander kommunizieren können. Auch die

[1] Aus zeitlichen Gründen wurden die Rundenwechsel bei der Durchführung der Experimente mit einer deutlich höheren Frequenz angestoßen.

Taktiken sollten von einer solchen Einrichtung Gebrauch machen können, um den Spieler in Notsituationen zu warnen oder ihn dann benachrichtigen zu können, wenn ein Ziel erreicht ist oder ein bestimmtes Ereignis eintritt. Eine Unterstützung von Allianzen wäre schon in der Form wünschenswert, dass nicht alle Einheiten, die nicht dem angemeldeten Spieler gehören, als feindlich angezeigt und in den Taktiken entsprechend als Bedrohung gewertet werden. Die vorgestellte Jägertaktik greift nämlich auch befreundete Mitspieler an, da das Spiel nicht zwischen Freund und Feind unterscheidet. Zur Unterstützung der zentralen These dieser Arbeit waren die beschriebenen Funktionsmerkmale nicht zwingend notwendig, doch könnten sie das vorgestellte Spiel erheblich Verbessern.

Zuletzt ist es – wie in Abschnitt 2.2 angemerkt – leicht möglich, maschinelle Lernsysteme oder evolutionäre Algorithmen an das Taktiksystem von „Megaira“ anzubinden. Sowohl Online- als auch Offline-Lernverfahren – im selben Abschnitt beschrieben – könnten sehr reizvoll aber auch anspruchsvoll in der Umsetzung sein. Taktiken, die sich selbst weiterentwickeln, dürften das Spielniveau auf eine noch höhere Stufe steigern. In wie fern das den Spielspaß erhöhen könnte, wäre dann zu beobachten. „Megaira“ bietet jedenfalls eine solide Ausgangsplattform, um diese Fragen weitergehend zu untersuchen.

Anhang A

Weitere Informationen

Die Quelldateien der Anwendung „Megaira“ liegen der Arbeit auf einer CD-ROM bei. Hinweise zu den Systemvoraussetzungen und eine Installationsanleitung sind Anhang B zu entnehmen.

Anhang B

Installationsanleitung für die Beispielanwendung „Megaira“

B.1 Systemvoraussetzungen:

Die Installation von Megaira setzt einige korrekt eingerichtete und funktionierende Pakete voraus. Die nötige Software ist im Folgenden aufgelistet:

- Ein UNIX-kompatibles[1] Betriebssystem (getestet unter Mac OS X 10.5 „Leopard“ und Ubuntu Linux 8.04 „Hardy Heron“).
- Python 2.5 [24]
- SQLite 3 [3] oder ein anderes Datenbanksystem mit entsprechend eingerichtetem Python DB-API 2.0 Interface (hier: „pysqlite“ [1])
- virtualenv[2] [4]

B.2 Installation des Binärpakets

Um Megaira auf einem beliebigen unterstützen System zu installieren, legen wir zunächst eine virtuelle Python-Umgebung an und aktivieren diese:

```
1 virtualenv env
2 source env/bin/activate
```

Dann wird das binäre Megaira-Paket installiert:

[1] Microsoft Windows sollte theoretisch auch als Serverplattform funktionieren, wurde allerdings nicht getestet.

[2] Strenggenommen ist virtualenv nicht unbedingt nötig, wenn man Administrator-Rechte auf dem System hat, erspart einem aber die Installation der Anwendung und Abhängigkeiten in Systemverzeichnisse.

```
easy_install /pfad/zu/Megaira-{VERSION}-py2.5egg
```

Jetzt müssen wir Paster [13] anweisen, eine neue Konfigurationsdatei zu generieren. Das passiert mit dem folgenden Kommando:

```
paster make-config megaira config.ini
```

Nun lässt sich die Anwendung über die generierte Datei `config.ini` beliebig konfigurieren. Die Software sollte – sofern SQLite 3 und pyqlite installiert sind und der Port 5000 frei ist – ohne irgendwelche Änderungen an der Konfigurationsdatei lauffähig sein.

Nun richten wir die Datenbank ein. Möglicherweise wird der nächste Befehl einige Unicode-Warnungen ausgeben, die wir getrost ignorieren dürfen:

```
paster setup-app config.ini
```

Nun starten wir den Server:

```
paster serve --reload config.ini
```

Öffnen wir nun einen unterstützten Webbrowser, am besten Firefox [35] oder Safari [8], und navigieren diesen auf die Adresse `http://localhost:5000/`, so erscheint der Anmeldedialog von Megaira.

Hier geben wir den Benutzernamen „joshua“ und das Passwort „secret“ ein, um zur Brücke zu gelangen. Hat das alles geklappt, ist die Grundinstallation erfolgreich abgeschlossen.

B.3 Installation des Quellcodes

Um eigene Experimente aufzusetzen oder einfach mit dem Code herumzuspielen, ist alternativ auch die Installation des Quellcodes möglich. Die Installation erfolgt analog zu den oben beschriebenen Schritten, nur dass hier statt der Installation des Binärpakets von Megaira die Quellen ausgepackt und eingerichtet werden. Falls nicht bereits vorhanden, ist hierzu noch das Paket „Paver“ [17] mittels `easy_install paver` zu installieren.

```
tar xfzv /pfad/zu/Megaira-{VERSION}.tar.gz
cd Megaira-{VERSION}
paver develop
```

Der Quellcode ist jetzt fertig eingerichtet und kann im laufenden Betrieb des Servers geändert werden. Dieser wird Änderungen an den Dateien automatisch erkennen und sie sofort nachladen.

Abbildungsverzeichnis

Algorithmenverzeichnis

Listings

Literaturverzeichnis

[1] *pysqlite*. http://pysqlite.org/, 2009. [Online; Stand 21. Juli 2009].

[2] *SOL*. http://www.freeport.de/Sol/, 2009. [Online; Stand 19. Juni 2009].

[3] *SQLite Home Page*. http://www.sqlite.org/, 2009. [Online; Stand 9. Juli 2009].

[4] *virtualenv*. http://pypi.python.org/pypi/virtualenv, 2009. [Online; Stand 21. Juli 2009].

[5] Adobe Systems Incorporated: *Adobe Flash*. http://www.adobe.com/de/products/flash/, 2009. [Online; Stand 25. Juni 2009].

[6] Alpaydin, Ethem: *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[7] Apache Software Foundation, The: *Apache HTTP Server*. http://httpd.apache.org/, 2009. [Online; Stand 19. Juni 2009].

[8] Apple Inc.: *Apple – Safari*. http://www.apple.com/de/safari/, 2009. [Online; Stand 26. Juni 2009].

[9] Bangert, Ben: *Routes*. http://routes.groovie.org/, 2008. [Online; Stand 25. Juni 2009].

[10] Bangert, Ben und Mike Bayer: *Beaker*. http://beaker.groovie.org/, 2009. [Online; Stand 25. Juni 2009].

[11] Bangert, Ben, Philip Jenvey und James Gardner: *Pylons*. http://www.pylonshq.com/, 2009. [Online; Stand 19. Juni 2009].

[12] Bayer, Mike: *SQLAlchemy*. http://www.sqlalchemy.org/, 2009. [Online; Stand 19. Juni 2009].

[13] Bicking, Ian: *Python Paste*. http://pythonpaste.org/, 2008. [Online; Stand 14. Juli 2009].

[14] Bigpoint GmbH: *DarkOrbit*. http://www.darkorbit.de/, 2009. [Online; Stand 20. Juli 2009].

[15] Bourg, David M. und Glenn Seemann: *AI for Game Developers*. O'Reilly Media, Inc., 2004.

[16] Celsius Online SARL: *Die Renaissance Könige reiche*. http://www.diekoenigreiche.com/, 2009. [Online; Stand 20. Juli 2009].

[17] Dangoor, Kevin: *Paver*. http://www.blueskyonmars.com/projects/paver/, 2008. [Online; Stand 21. Juli 2009].

[18] de Menten, Gaetan, Daniel Haus und Jonathan LaCour: *Elixir*. http://elixir.ematia.de/, 2009. [Online; Stand 19. Juni 2009].

[19] Dewdney, A. K.: *Computer Recreations: In the game called Core War hostile programs engage in a battle of bits*. Scientific American, 250(5):14–22, Mai 1984. Description of a computer game in which short pieces of code reproduce and compete for occupation of memory.

[20] Dumpleton, Graham: *mod_wsgi*. http://www.modwsgi.org/, 2009. [Online; Stand 25. Juni 2009].

[21] Eby, Phillip J.: *PEP 333 – Python Web Server Gateway Interface v1.0*. http://www.python.org/dev/peps/pep-0333/, 2009. [Online; Stand 25. Juni 2009].

[22] ECMA: *ECMAScript Language Specification*. http://bclary.com/2004/11/07/ecma-262.html, 1999. [Online; Stand 12. Januar 2004].

[23] Eiben, Agoston E. und J. E. Smith: *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[24] Foundation, Python Software: *Python Programming Language – Official Website*. http://www.python.org/, 1990-2009. [Online; Stand 9. Juli 2009].

[25] Gardner, James: *AuthKit*. http://authkit.org/, 2009. [Online; Stand 19. Juni 2009].

[26] Google: *Google Chrome: Ein neuer Webbrowser für Windows*. http://www.google.com/chrome/?hl=de, 2009. [Online; Stand 26. Juni 2009].

[27] Hart, P. E., N. J. Nilsson und B. Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, 1968.

[28] Haus, Daniel: *Megaira*. `http://megaira.ematia.de/`, 2009. [Online; Stand 19. Juni 2009].

[29] InnoGames GmbH: *Die Stämme*. `http://www.die-staemme.de/`, 2009. [Online; Stand 25. Juni 2009].

[30] Jakob, Daniel: *Star Trek Universe*. `http://www.stuniverse.de/`, 2009. [Online; Stand 25. Juni 2009].

[31] Jakob, Daniel: *Star Trek Universe Wiki*. `http://wiki.stuniverse.de/Beginner:Daten#Ticks`, 2009. [Online; Stand 25. Juni 2009].

[32] Klir, George J. und Bo Yuan: *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall PTR, May 1995.

[33] Krauss, Hans Jürgen: *Andromeda*. `http://www.andromedateam.de/`, 2009. [Online; Stand 20. Juli 2009].

[34] Millington, Ian: *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[35] Mozilla Europe und Mozilla Foundation: *Der Webbrowser Firefox | Schneller, sicherer & anpassbar | Mozilla Europe*. `http://www.mozilla-europe.org/de/firefox/`, 2009. [Online; Stand 26. Juni 2009].

[36] Müller, Tim-Hendrik, Tobias Hein, Frank Behler, Sebastian Schnelker, Andreas Thom, André Barthelmes, Jan Quadflieg, Simon Wessing, Dominik Opolony, Holger Danielsiek, Daniel Haus, Patrick Szcypior, Moritz Hofmann und Armin Büscher: *Computational Intelligence bei Computerspielen, Seminarausarbeitungen*. `http://ls11-www.cs.uni-dortmund.de/people/rudolph/teaching/seminars/CICS/SS2007/seminar.jsp`, 2007.

[37] Perlowski, Markus: *Die Kreuzzüge*. `http://www.die-kreuzzuege.de/`, 2009. [Online; Stand 13. Juli 2009].

[38] Ponsen, Marc und Pieter Spronck: *Improving Adaptive Game AI with Evolutionary Learning*. In: *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, Seiten 389–396, University of Wolverhampton, England, 2004.

[39] Resig, John: *jQuery*. `http://jquery.com/`, 2009. [Online; Stand 26. Juni 2009].

[40] Rudolph, Prof. Dr. Günter: *Fundamente der Computational Intelligence*. `http://ls11-www.cs.uni-dortmund.de/people/rudolph/teaching/lectures/FCI/WS2006-07/lecture.jsp?userLanguage=de`, 2006.

[41] Spronck, Pieter, Marc Ponsen, Eric Postma und Ida Sprinkhuizen-Kuyper: *Adaptive Game AI with Dynamic Scripting.* Machine Learning: Special Issue on Computer Games, 63(3):217–248, 2006.

[42] Sun Microsystems, Inc.: *MySQL :: Die populärste Open-Source-Datenbank der Welt.* http://www.mysql.de/, 2009. [Online; Stand 9. Juli 2009].

[43] Sun Microsystems, Inc.: *Sun Java.* http://java.sun.com/, 2009. [Online; Stand 25. Juni 2009].

[44] The RoboCup Federation: *RoboCup Official Site.* http://www.robocup.org/, 2009. [Online; Stand 25. Juli 2009].

[45] Unity Technologies ApS: *Unity.* http://unity3d.com/, 2009. [Online; Stand 25. Juni 2009].

[46] Valverde Gonzalez, Alberto: *ToscaWidgets.* http://toscawidgets.org/, 2009. [Online; Stand 25. Juni 2009].

[47] W3C: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* http://www.w3.org/TR/CSS2/, 2009. [Online; Stand 26. Juni 2009].

[48] W3C: *XHTML^{TM} 1.0 The Extensible HyperText Markup Language (Second Edition).* http://www.w3.org/TR/xhtml1/, 2009. [Online; Stand 26. Juni 2009].

[49] Weizenbaum, J.: *Eliza - a computer program for the study of natural language communication between man and machine.* Comm. A.C.M., 9(1):36–45, Januar 1966.

[50] Wikipedia: *Ajax (Programmierung) — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Ajax_(Programmierung)&oldid=61203133, 2009. [Online; Stand 19. Juni 2009].

[51] Wikipedia: *Browserspiel — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Browserspiel&oldid=60104831, 2009. [Online; Stand 26. Mai 2009].

[52] Wikipedia: *Core War — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Core_War&oldid=55865867, 2009. [Online; Stand 25. Juli 2009].

[53] Wikipedia: *ELIZA — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=ELIZA&oldid=59247974, 2009. [Online; Stand 20. Juli 2009].

[54] Wikipedia: *Freeport.de SOL — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/w/index.php?title=Freeport.de_SOL&oldid=55179285`, 2009. [Online; Stand 19. Juni 2009].

Index

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Mörfelden-Walldorf, den 26. Juli 2009

Daniel Haus