Entwicklung einer Game AI zur realistischen Lasterzeugung in verteilten Massively Multiplayer Online Games

Damian A. Czarny, B. Sc.
Gutachter: Prof. Alejandro Buchmann[1], Ph.D.
Betreuer: Max Lehn[1], M. Sc. und Tonio Triebel[2], Dipl.-Inform.

[1] Databases and Distributed Systems, Technische Universität Darmstadt
[2] Department of Computer Science IV, University of Mannheim

Master-Thesis
Eingereicht am 30. September 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Entwicklung einer Game AI zur realistischen Lasterzeugung in verteilten Massively Multiplayer Online Games
Master-Thesis

Eingereicht von Damian A. Czarny, B. Sc.
Tag der Einreichung: 30. September 2012

Gutachter: Prof. Alejandro Buchmann[1], Ph.D.
Betreuer: Max Lehn[1], M. Sc. und Tonio Triebel[2], Dipl.-Inform.

[1] Databases and Distributed Systems, Technische Universität Darmstadt
[2] Department of Computer Science IV, University of Mannheim

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Databases and Distributed Systems (DVS)
Prof. Alejandro Buchmann

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung der Arbeit stimmt mit der gedruckten überein.

Darmstadt, den 30. September 2012

Damian A. Czarny, B. Sc.

Zusammenfassung

Peer-to-Peer-Architekturen bieten aufgrund ihrer verteilten Natur viele Vorteile gegenüber zentral organisierten Architekturen, wie beispielsweise klassischen Client/Server-Systemen. Die Vorteile die *Peer-to-Peer*-Architekturen bieten gehen allerdings mit einer erhöhten Komplexität des Systems und vielen neuen Herausforderungen bezüglich der Netzwerkkommunikation einher. Um eine Vergleichbarkeit verschiedener P2P-Implementierungen zu ermöglichen, wird am Informatik Fachgebiet *Databases and Distributed Systems* (DVS) der Technischen Universität Darmstadt an einer Benchmarking-Methodik mitgearbeitet die diese ermöglichen soll. Dazu wurde eine Evaluationsplattform entwickelt, die mit dem P2P-MMOG *Planet PI4* ein Benchmarking von *P2P-Gaming-Overlays* ermöglicht. Ein wesentlicher Bestandteil der Benchmarking-Methodik ist die synthetische Generierung einer realistischen Netzwerklast. *Planet PI4* soll u.a. dazu eingesetzt werden diese Netzwerklast für *P2P-Gaming-Overlays* zu erzeugen. Ziel dieser Master-Thesis ist die konkrete Erzeugung der Netzwerklast von *Planet PI4* mit dem Ansatz kontextsensitiver AI-Spieler, die stellvertretend für Menschen das Spiel spielen.

Das Resultat dieser Arbeit ist die Entwicklung und Implementierung einer *Game AI* für *Planet PI4*, die eine Aufteilung der Verantwortlichkeiten in drei hierarchisch angeordneten Ebenen vornimmt. Die derart entwickelte Architektur ermöglicht die zeitlich entkoppelte Ausführung der Komponenten auf den unterschiedlichen Ebenen. Weiterhin ermöglicht sie den Austausch bestimmter Komponenten oder der Ebenen-Implementierung ohne damit gleich das ganze System zu beeinflussen. Diese Eigenschaft und die fein-granulare Abkapselung einzelner Aufgabenbereiche innerhalb der Ebenen begünstigen die erforderliche einfache und gezielte Weiterentwicklung der *Game AI*.

Die drei wichtigstens Komponenten der *Game AI* sind das *Decision Making* mittels *Behavior Trees* zur Aktionsbestimmung, sowie die *Steering Pipeline* und das *Combat System* zur Kombinierung, Bereitstellung und Ausführung nicht-primitiver Verhaltensweisen. Die Verhaltensweisen der beiden Komponenten werden dabei der *Decision Making*-Komponente als Aktionsmenge bereitgestellt und ermöglichen dieser somit die indirekte Steuerung der Spielfigur in *Planet PI4*.

Zur Evaluation des eigenen Ansatzes wurden 13 Metriken zur Messung von Verhalten und Performance in neun unterschiedlichen Testszenarien eingesetzt. Die Testszenarien erfolgten dabei unter Einsatz des *Discrete Event Game Simulators*, der ebenfalls Teil der Evaluationsplattform von *Planet PI4* ist und für die Reproduzierbarkeit der erzeugten Last und somit auch der Evaluationsergebnisse der Master-Thesis sorgt. Zusammenfassend lassen die Evaluationsergebnisse auf die generelle Eignung der *Game AI*-Implementierung zur Generierung der erforderlichen Netzwerklast schließen. Zur Beurteilung der Ergebnisse diente auch der Vergleich zur Vorgänger-Implementierung, welcher ergab, dass die *Game AI*-Implementierung in vielen Bereichen, wie Reproduzierbarkeit, Konfigurierbarkeit und Realitätsgrad, bessere Ergebnisse erzielt als die Vorgänger-Implementierung, sich allerdings in Sachen genereller Spielstärke und Performance dieser knapp geschlagen geben muss.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Problemstellung und -beschreibung . . . 3
- 1.2 Zieldefinition . . . 5
- 1.3 Themeneingrenzung . . . 6
- 1.4 Aufbau der Arbeit . . . 6

2 Grundlagen **9**
- 2.1 Was ist ein Computerspiel? . . . 9
- 2.2 Academic AI vs. Game AI . . . 11
- 2.3 Der Entwurf eines Rationalen Agenten . . . 12
- 2.4 Steering Behaviors und das Agentenmodell nach Reynolds . . . 15

3 Verwandte Ansätze **17**
- 3.1 Methoden zur Lasterzeugung in Spielen . . . 18
- 3.2 Halo's AI-Architektur . . . 21
- 3.3 GOAP - Goal-Oriented Action Planning . . . 23
- 3.4 Verfahren zur Kombinierung von Steering Behaviors . . . 27
- 3.5 Vorgänger-Implementierung: DragonBot . . . 31

4 Konzept und Rahmenbedingungen der Game AI **33**
- 4.1 Einsatzumgebung Planet PI4 . . . 33
- 4.2 Funktionale Anforderungen der Game AI . . . 35
- 4.3 Konzept der Game AI . . . 36
- 4.4 Behavior Tree . . . 37

5 Design und Implementierungsdetails der Game AI **41**
- 5.1 Einbindung der Game AI in Planet PI4 . . . 41
- 5.2 Game AI-Architektur . . . 43
- 5.3 Operative Ebene . . . 47
  - 5.3.1 Steering Pipeline . . . 49
  - 5.3.2 Combat System . . . 54
- 5.4 Taktische Ebene . . . 55
  - 5.4.1 Blackboard Architektur . . . 57
  - 5.4.2 Behavior Tree Implementierung . . . 58
  - 5.4.3 Behavior Tags und Personality Selection . . . 63
  - 5.4.4 Der Behavior Tree der Game AI . . . 67
- 5.5 Strategische Ebene . . . 70
  - 5.5.1 Konfigurationsmöglichkeiten . . . 70
  - 5.5.2 Region Map und die strategische Zielbestimmung . . . 72

6 Evaluation **75**
- 6.1 Ziele und Rahmenbedingungen . . . 75
- 6.2 Eingesetzte Metriken . . . 76
- 6.3 Allgemeiner Aufbau und Ablauf der Testszenarien . . . 78
- 6.4 Durchgeführte Testszenarien . . . 79

6.4.1 1. Testszenario 80
6.4.2 2. Testszenario 81
6.4.3 3. Testszenario 83
6.4.4 4. Testszenario 84
6.4.5 5. Testszenario 86
6.4.6 6. Testszenario 87
6.4.7 7. Testszenario 88
6.4.8 8. Testszenario 89
6.4.9 9. Testszenario 91
6.5 Auswertung der Ergebnisse 92

7 Ausblick und Fazit **97**
7.1 Ausblick 97
7.2 Fazit 98

Abbildungsverzeichnis **101**

Tabellenverzeichnis **103**

Literaturverzeichnis **105**

1 Einleitung

*Man stelle sich folgendes fiktives Szenario vor: Es soll ein neues **M**assively **M**ultiplayer **O**nline **G**ame (MMOG) entwickelt werden.*

Der Begriff MMOG bezeichnet dabei die Art von Spielen, welche darauf ausgelegt sind, einer großen Anzahl an Spielern (*massively*) ein simultan ablaufendes gemeinsames Spielen (*multiplayer*) über das Internet zu ermöglichen (*online*). Hierfür wird den Spielern eine gemeinsame virtuelle, in vielen Fällen auch persistente Spielwelt bereitgestellt, in jener der Spieler dazu angehalten wird, sein Voranschreiten im Spiel mittels kooperativer oder kompetitiver Verhaltensweisen, die wiederum jeweils kleinere oder größere Untermengen anderer Spieler miteinschließen können, zu erzielen. Dieses allgemeine Konzept eines MMOGs lässt sich dabei auf ein breites Spektrum unterschiedlichster Spielkonzepte und somit quer über eine Vielzahl von Computerspiel-(Sub)Genres anwenden. Beispiele hierfür wären das ***F**irst-**P**erson **S**hooter* (FPS), das ***R**eal **T**ime **S**trategy* (RTS) oder das ***R**ole-**P**laying **G**ame* (RPG) Genre.

*Man stelle sich folgende Erweiterung für das Szenario vor: Das MMOG soll als dem Spiel zugrundeliegendes Netzwerk-Overlay eine **P**eer-**to**-**P**eer-Architektur verwenden.*

Die Verwendung einer P2P-Architektur hat den Vorteil, dass die System-inhärente Eigenschaft der dezentralen und somit verteilten Natur ausgenutzt werden kann, um ganz im Sinne eines MMOGs, einer möglichst großen Anzahl an Spielern ein gemeinsames spielen über das Netzwerk zu ermöglichen. Eine Alternative zu dezentralen Architekturen stellen dabei zentral verteilte Architekturen dar. Als klassisches Beispiel wären an dieser Stelle Client/Server-Systeme zu nennen, welche ebenfalls die zurzeit am häufigsten eingesetzte Netzwerkarchitektur in MMOGs darstellt.

Unter einem Client/Server-System versteht man, wie der Name bereits andeutet, eine generelle Zweiteilung in Dienstanbieter (Server) und Dienstempfänger (Client). In einem P2P-System hingegen sind alle Peers gleich, sie fungieren gleichzeitig als Client und als Server. Prinzipiell ist ein solches Verhalten auch in Client/Server-Systemen möglich, doch haben diese allgemein gemein, dass sie Funktionen logisch und physisch über mehrere Computer hinweg aufteilen, wobei jeder Rechner auf eine bestimmte Gruppe von Funktionen zugeschnitten ist. Diese Art der Verteilung wird in der Fachliteratur als „vertikale Verteilung“ bezeichnet.

Zwar existieren verschiedene Client/Server-Anordnungen, die definieren, welche Funktionen jeweils zum Client oder zum Server gehören, jedoch ist es vor allem in MMOGs üblich, dass der Server große Teile der Funktionalität eines Spiels beherbergt - einen sogenannten *fat Server* darstellt. Die Aufgaben eines solchen Servers wären dabei z.B. die Überprüfung und Berechnung der Auswirkungen der gewünschten Spielerinteraktionen auf die Spielwelt, sowie die Synchronisierung der einzelnen Spielersichten der Spielwelt untereinander. Logischerweise bleibt in so einem Fall nicht mehr viel Funktionalität für den Client übrig, sodass der Client des Spielers meistens aus nicht viel mehr besteht als einer graphischen Schnittstelle zum Server - ein sogenannter *thin Client*.

Die Funktionalitätsfülle und der Aspekt, dass der Server als direkte zentrale Anlaufstelle für alle anderen Clients dient, lassen bereits erahnen, welch immense Anforderungen und damit auch Kosten auf Serverseite anfallen können. Man stelle sich beispielsweise die erzeugte Last eines *World of Warcraft* vor, wo mehrere hundert tausend Spieler gleichzeitig[1] versuchen in derselben Spielwelt zu spielen. Laut

[1] Zwar können alle Spieler gleichzeitig spielen, jedoch existieren Partionierungstechniken, wie *Leveling* oder *Sharding*, die die tatsächlich gleichzeitig zusammenspielende Menge an Spielern in ein und derselben partiellen Spielwelt reduzieren und beschränken [56].

einem Bericht von Vivendi war 2006 zur Bewältigung dieser Last eine Serverfarm für *World of Warcraft* im Einsatz, die stellenweise aus mehr als 1.900 parallel laufenden Servern bestand [43].

In P2P-Systemen erfolgt währenddessen eine sogenannte "horizontale Verteilung". Bei dieser Art der Verteilung können Client oder Server physisch in logisch gleichwertige Teile unterteilt werden. Vorteil dieser Einteilung ist, dass die Gesamtlast des Systems verteilt werden kann, indem jeder dieser gleichwertigen Teile nur auf einem eigenen Ausschnitt der vollständigen Datenmenge arbeitet. Werden weitere Daten aus anderen Ausschnitten benötigt, werden die dafür zuständigen Teile erfragt und die Daten entsprechend angefordert [54, S. 55-67]. Die derart realisierte, in weiten Teilen, gleichmäßige Verteilung der Last auf alle Clients, ermöglicht eine starke Reduktion der Serverlast, da diese nicht mehr den Großteil der Arbeit alleine übernehmen muss.

Zusammenfassend könnte zumindest in der Theorie die Verwendung von P2P-Technologien dazu beitragen, ein Spiel mit den enormen Spielerzahlen eines *World of Warcrafts* zu deutlich geringeren laufenden (Server-)Kosten betreiben zu können. Warum dies nicht bereits längst getan wird, ist dem Umstand geschuldet, dass besonders die verteilte Systemeigenschaft, die eigentlich zur Reduktion der Last dient, die Spielentwicklung vor eine Menge neuer Herausforderungen stellt, die bei der Entwicklung von zentral organisierter Software nicht oder nur in abgeschwächter Form auftreten. Als eine der dabei größten Herausforderungen erweist sich das Lösen des Problems des fehlenden gemeinsamen, globalen und synchronisierten Gesamtzustandes. Auf das Beispiel-Szenario übertragen bedeutet dies, dass sich die Bestandteile der Spiellogik des MMOGs über eine skalierbare Vielzahl von Peers erstreckt, die sich auf unterschiedlichen und weit voneinander entfernten Rechnern befinden können.

Man stelle sich folgende Erweiterung für das Szenario vor: Das auf einer P2P-Architektur aufsetzende MMOG soll im Third Person Shooter-Genre angesiedelt sein und in einer 3D-Open World spielen.

Mit anderen Worten soll ein Spiel entwickelt werden, wo der Spieler das Spielgeschehen aus der Schulterperspektive seiner Spielfigur wahrnimmt, mit dieser innerhalb einer frei erkundbaren, dreidimensionalen Spielwelt agiert und mit Schusswaffen andere Spieler oder computergesteuerte Gegner in Echtzeit bekämpft. Aufgrund der Echtzeitbedingung bzw. der allgemein erhöhten Spielgeschwindigkeit und der Notwendigkeit einer möglichst präzisen und unmittelbaren Steuerung, stellen MMOTPS eine besondere Herausforderung an die Performance des Spiels dar und verschärfen mit diesem Performance-Anspruch die eben vorgestellten Problembereiche verteilter Anwendungen nochmals.

Die Performance-Herausforderung für verteilte Anwendungen ergibt sich hierbei in erster Line nicht aus der Notwendigkeit der schnellen grafischen Darstellung des Spiels, sondern äußert sich in der erhöhten Verzögerungssensitivität der Netzwerkkommunikation. Diese Verzögerungssensitivität bzw. die daraus resultierende Anforderung einer geringen Latenz bei der Netzwerk-Kommunikation des Spiels, wobei die Latenz den Zeitraum zwischen einem verborgenen Ereignis und dem Eintreten einer sichtbaren Reaktion darauf bezeichnet, stellt die Netzwerkkomponente eines Spiels vor eine schwierige Aufgabe. Während bei Video- oder Telefoniesoftware eine Netzwerk-Paketverzögerung von unter 300 ms. benötigt wird, sind Onlinespiele bei einer Verzögerung jenseits von 100 bis 150 ms. kaum mehr spielbar. Wobei leichte Verzögerungen des Spielgeschehens, sogenannte *Lags*, in anderen Spielarten noch tolerierbar wären, kann in einem *Shooter* jede kleinste Verzögerung bereits über Sieg oder Niederlage entscheiden und somit einen unmittelbaren negativen Einfluss auf die Qualität der Spielerfahrung ausüben [7, S. 1].

Man stelle sich folgende abschließende Erweiterung für das Szenario vor: Es soll eine bestehende P2P-Implementierung für das zu entwickelnde MMOTPS gefunden werden, welches den hohen Performance-Anforderungen des Spiels gerecht wird.

Nachdem das zu entwickelnde Spiel feststeht, die Notwendigkeit einer leistungsstarken und skalierbaren Netzwerkkomponente verdeutlicht wurde und eine P2P-Architektur als mögliche Lösung hier-

für vorgestellt wurde, stellt sich nun die Frage, welche der zahlreichen P2P-Overlays, Architekturen oder konkreten Implementierungen die Richtige für diesen Einsatzzweck ist? Eine Beantwortung dieser Frage gestaltet sich allerdings durchaus problematisch. Nicht nur das zahlreiche unterschiedliche P2P-Implementierungen existieren, so konzentrieren sich diese, aufgrund der gegenüber Client/Server-Systemen erhöhten Komplexität, meistens noch auf einige ausgewählte Aspekte, wie der fairen Auslastung der Peers (Incentive Mechanisms), die Verhinderung von Cheating-Versuchen (Cheating Mitigation) oder den Informationsaustauschprozess zwischen den Peers (Game Event Dissemination) [14, 29].

Ein weiteres Problem, welches maßgeblich den Auswahlprozess einer geeigneten P2P-Implementierung erschwert ist, dass die meisten Forschergruppen eigene, oft speziell auf ihr Projekt zugeschnittene, Evaluierungstechniken einsetzen. So sieht man sich schnell unweigerlich mit einer Fülle an unterschiedlichen Methoden, angewandten Metriken, verwendeten Konfigurationen und eingesetzten Testumgebungen konfrontiert. Dieser Variantenreichtum und die mangelnde Allgemeingültigkeit der Evaluationsergebnisse erschweren die Vergleichbarkeit der P2P-Overlays untereinander entscheidend.

Aus diesem Grund wird am Informatik Fachgebiet *Databases and Distributed Systems* (DVS) der Technischen Universität Darmstadt an einer Benchmarking-Methodik für ***N**etworked **V**irtual **E**nvironments* (NVEs) mitgeforscht, die diese fehlende Vergleichbarkeit ermöglichen soll, indem es eine einheitliche und objektive Analyse und Bewertung unterschiedlichster P2P-Overlays gewährleistet [29, 17]. Die Benchmarking-Methodik ist ein wesentlicher Bestandteil der durch die Deutsche Forschungsgemeinschaft (DFG) geförderten Forschergruppe QuaP2P - "Verbesserung der Qualität von Peer-to-Peer-Systemen durch die systematische Erforschung von Qualitätsmerkmalen und deren wechselseitigen Abhängigkeiten.“ [42].

Die Benchmarking-Methodik von NVEs umfasst mit der Definition eines einheitlichen Funktionalitäts-Interfaces [17], der Identifizierung und Aufstellung von Qualitätsmetriken [30], dem Einsatz einer Evaluationsumgebung [28] und der Generierung einer realistischen Netzwerklast [30] vier wesentliche Komponenten. Der DVS-Fachgebiet entwickelte dazu eine Evaluationsplattform, die mit dem P2P-MMOG *Planet PI4* ein Benchmarking von *P2P-Gaming-Overlays* ermöglicht. Ein wesentlicher Bestandteil der Benchmarking-Methodik ist die synthetische Generierung einer realistischen Netzwerklast. *Planet PI4* soll u.a. dazu eingesetzt werden diese Netzwerklast für *P2P-Gaming-Overlays* zu erzeugen. Ziel dieser Master-Thesis ist die konkrete Erzeugung der Netzwerklast von *Planet PI4* mit dem Ansatz kontextsensitiver AI-Spieler, die stellvertretend für Menschen das Spiel spielen.

1.1 Problemstellung und -beschreibung

Es existieren zwar mehrere Methoden zur Erzeugung von Netzwerklasten, allerdings sind diese meistens entweder zu simpel oder für eine allgemeine Vergleichbarkeit unterschiedlicher P2P-Overlays untereinander zu spezifisch. Eine gängige Art ist beispielsweise die Nutzung der Generierung zufallsbasierter Bewegungsabläufe, um die dadurch entstehenden Positionsänderungen, welche jeweils an die anderen Peers weiterpropagiert werden müssen, zur Erzeugung der Netzwerklast zu benutzen. Dieser oder andere auf ihn beruhende Ansätze konzentrieren sich mit dem Bewegungsaspekt, nur auf einen von vielen wichtigen Aspekten eines Computerspiels und können so gesehen nur eine ungefähre Approximation der tatsächlich erzeugten Netzwerklast eines Computerspiels liefern.

Der von der Master-Thesis entwickelte Ansatz sollte dementsprechend keine zu starken Vereinfachungen vornehmen und möglichst alle relevanten Faktoren des Computerspiels zur Erzeugung der Netzwerklast beachten.

Die meisten anderen Ansätze beruhen auf der Erfassung und Auswertung echter Spielpartien anhand von sogenannten *Game Traces*, die wichtige Daten des Spiels aufzeichnen. Diese Aufzeichnungen oder auf ihnen beruhende Verhaltensmodelle werden dann zum automatisierten Nachspielen des Spiels eingesetzt. Dazu wird nicht wirklich das Spiel gespielt, sondern nur die relevanten Nachrichten zur Lasterzeugung generiert, wie z.B. Positionswechsel oder Todesmeldungen. Das Problem solcher Ansätze ist, dass sie entweder nicht skalieren, weil dazu erneut echte Spielerpartien mit der entsprechenden Spieleranzahl aufgezeichnet werden müssten, oder ebenfalls zu ungenau in der Erzeugung des Netzwerklast sind. Die Ungenauigkeit entsteht dabei häufig dadurch, dass sich bei der Aufzeichnung nur auf relevante Daten der Netzwerkebene beschränkt wird. Die damit gelernten Modelle können demzufolge nur Auskunft über Aspekte geben, wie die mittlere Anzahl an Todesmeldungen oder die bevorzugte Ansteuerung von Positionen. Die sich gegenseitig beeinflussenden Spielprozesse oder ineinandergreifenden Interaktionen der Spieler im Spiel, die zu Entscheidungen wie der Ansteuerung einer Position oder dem Sterben eines Spielers führen, werden dabei weitestgehend ignoriert.

Der von Master-Thesis entwickelte Ansatz sollte dementsprechend skalierbar im Hinblick auf die Spielerzahl sein und sich nicht ausschließlich auf für die Netzwerkebene relevante Daten beschränken. Deswegen soll der Ansatz direkt auf der Spielebene ansetzen und in diesem Sinne auch die Erfassung und Abbildung sich gegenseitig beeinflussender kausaler Prozesse im Spiel ermöglichen, die zur Erzeugung der Netzwerklast als wichtig erscheinen.

Der Ansatz der Nutzung computergesteuerter kontext-sensitiver Spieler zur Erzeugung der Netzwerklast ist ein Konzept, welches genau diesem beschriebenen Anforderungsprofil gerecht zu werden scheint und deswegen im Rahmen dieser Master-Thesis umgesetzt werden soll. Hierbei handelt es sich im Rahmen der Lasterzeugung von P2P-Systemen um einen noch recht jungen und unerprobten Ansatz, weswegen eine weitere Aufgabe dieser Master-Thesis, neben der Implementierung, auch die Evaluation bzw. Feststellung der generellen Eignung dieses Ansatzes zur Generierung einer realistischen Netzwerklast ist.

Die Entwicklung computergesteuerter kontext-sensitiver Spieler ist Vergleichbar mit der Entwicklung von NPCs in herkömmlichen Computerspielen. In aller Regel übernimmt dabei eine sogenannte **A**rtificial **I**ntelligence (AI)-Komponente des Computerspiels oder auch kurz *Game AI* genannt die Steuerung dieser NPCs. Der Prozess der Entwicklung einer *Game AI* ist hierbei stark vom jeweiligen konkreten Computerspiel abhängig. Das liegt vorrangig daran, dass die *Game AI*, wie der menschliche Spieler auch, zum Spielen konkrete Informationen über das Spiel im Allgemeinen, wie z.B. über das Spielziel oder die Spielmechanik, als auch Kenntnis über das genaue Steuerungs- und Bedieninterface im Speziellen benötigt. Daher ist die Entwicklung zur Generierung der Netzwerklast (4. Komponente der Benchmarking-Methodik) eng mit der konkret verwendeten Evaluationsumgebung (3. Komponente der Benchmarking-Methodik) verbunden.

Als Evaluationsumgebung für *P2P-Gaming-Overlays* wird das Spiel *Planet PI4* [28] eingesetzt und stetig weiterentwickelt. *Planet PI4* ist ein auf P2P basierender MMOTPS und ist in einer 3D-Weltraumwelt angesiedelt. Das Spiel entspricht dabei weitestgehend dem fiktiv zu entwickelten Spiel des Beispielszenarios aus der Einleitung. Dies bedeutet im Besonderen, dass mit *Planet PI4* ein Spiel eingesetzt wird, welches aufgrund seiner hohen Verzögerungssensitivität, hohe Ansprüche an die zu evaluierenden *P2P-Gaming-Overlays* stellt. Die Evaluationsumgebung enthält darüber hinaus noch einen *Discrete-Event Simulator*, der eine vollständige Kontrolle der Umgebung ermöglicht. Somit kann, anders als bei der ebenfalls möglichen Evaluierung im einem realen Netzwerk, die Reproduzierbarkeit der Umgebungseinflüsse gewährleistet werden, die z.B. bei einer Evaluation übers Internet, wegen mangelnder Kontrollierbarkeit des Internets, nicht garantiert werden könnte.

1.2 Zieldefinition

Das übergeordnete Ziel der vorliegenden Master-Thesis ist zusammenfassend die synthetische Generierung einer realistischen Netzwerklast zu Evaluationszwecken verschiedenartiger *P2P-Gaming-Overlays* mittels einer auf besonderen Qualitätsmerkmalen ausgerichteten Benchmarking-Methodik. Dazu soll unter Verwendung der Spiel- und Evaluationsumgebung *Planet PI4* der Ansatz der kontext-sensitiven AI-Spieler, im Nachfolgenden auch als Bots bezeichnet, umgesetzt werden. Bei der Generierung einer realistischen Netzwerklast wurden mit der **Reproduzierbarkeit**, der **Skalierbarkeit**, dem **Realitätsgrad** und der **Konfigurierbarkeit** vier wichtige Kriterien herausgearbeitet, dessen Erfüllung das vorrangige Ziel der Master-Thesis darstellt.

Das erste Kriterium der **Reproduzierbarkeit** bezeichnet dabei den Umstand, dass es mit dem zu verwirklichenden Ansatz möglich sein soll, unter Zuhilfenahme des *Discrete-Event Simulators*, ein für ein Testszenario wiederholbares Verhalten zu generieren, sodass jedes *P2P-Gaming-Overlays* unter denselben Voraussetzungen evaluiert und beliebig oft wiederholt werden kann.

Das nächste Kriterium **Skalierbarkeit** bedeutet, dass die Lösung mittels Bots performant genug sein muss, um eine für ein MMOG ausreichend große Anzahl an Spielern zu unterstützen. Ein Bot vertritt im realen Testszenario einen menschlichen Spieler pro Peer bzw. Rechner, sodass diesem ausreichend Ressourcen zur Verfügung stehen sollten. Doch soll es darüber hinaus auch möglich sein eine Evaluation mithilfe des Simulators, welcher die Netzwerkkommunikation der Peers simuliert und kontrolliert, auf einem einzelnen Rechner auszuführen. Die Lösung sollte demnach hardwareschonend genug sein, um eine Simulation mit einer möglichst großen Anzahl an Spielern in einer vertretbaren Zeit auf einem Rechner zu gewährleisten.

Als drittes und vielleicht wichtigstes Kriterium soll ein möglichst hoher **Realitätsgrad** in Bezug auf das Verhalten der Bots erzielt werden. Dabei wird der angestrebte Realitätsgrad nicht primär an Aspekten wie Spielstärke oder der Nachahmung menschlicher kooperativer oder kompetitiver Verhaltensweisen gemessen. Stattdessen geht es vielmehr um die Nachbildung eines menschenähnlichen kontrollierbaren Spielverhaltens, welches in seiner Gesamtheit über alle Bot-Instanzen hinweg betrachtet, eine möglichst genaue Annäherung an von ausschließlich menschlichen Spielern verursachten Netzwerklasten darstellt. Erst die Generierung realistischer Lasten ermöglicht nämlich eine aussagekräftige Bewertung der Qualitätsmerkmale von P2P-Overlays im Rahmen der Benchmarking-Methodik.

Allen voran für die Erfüllbarkeit der beiden letztgenannten Kriterien Skalierbarkeit und Realitätsgrad wird eine ausreichende **Konfigurierbarkeit** bzw. Kalibrierung des Bots benötigt und stellt somit das vierte und letzte Kriterium der Zieldefinition dar. Hierbei müssen einerseits Wege zur Anpassung der Performance an die benötigte Spieleranzahl gefunden werden, anderseits Konfigurations-Mechanismen entwickelt werden, die ein Variieren der Netzwerklast durch spürbare Veränderungen des Spielverhaltens ermöglichen. Dies schließt Einstellungsmöglichkeiten der Verhaltensweisen einzelner Bot-Instanzen, bestimmter Untermengen davon, oder gar der Gesamtheit aller Bot-Instanzen mit ein.

Neben der Erfüllung der eben vorgestellten vier Kriterien soll aufgrund der gewonnenen Erfahrungswerte und Erkenntnisse aus der Vergangenheit mit diesem Ansatz (siehe dazu Abschnitt 3.5), nicht eine bloße Bot-Implementierung umgesetzt werden, sondern vielmehr eine stabile *Game AI*-Architektur geschaffen werden. Diese soll als Grundlage für Analyse und Entwicklung weiterer konkreter Bot-Implementierungen oder gezielter Verbesserungen der bestehenden Bot-Implementierung genutzt werden. Dabei soll größtmöglichen Wert auf Wiederverwendbarkeit und Austauschbarkeit der Teilkomponenten der *Game AI*-Architektur gelegt werden.

1.3 Themeneingrenzung

Die tatsächlichen Aufgaben und somit möglichen Komponenten einer vollständigen *Game AI* sind umstritten und können je nach eingesetztem Computerspiel stark variieren. Gegenstand der Entwicklung der Master-Thesis soll deswegen eine *Game AI* sein, die sich Hauptsächlich auf den Entscheidungsprozess der Auswahl der als nächsten auszuführenden Aktion (Decision Making) beschränkt. Dabei werden alle relevanten Spielaspekte umfasst, mit denen sich auch ein menschlicher Spieler von *Planet PI4* in der Regel auseinandersetzt. Mit anderen Worten geht es um die bestmögliche Steuerung des Raumschiffes von *Planet PI4*, sowie der Umsetzung und Unterstützung von Entscheidungsprozessen zur Evaluierung der bestmöglichen Spielweise die einerseits zum Gewinnen des Spiels führt, andererseits eine möglichst realistische Nachbildung der Netzwerklast erlaubt. Demzufolge sind andere Aspekte die ebenfalls zu einer *Game AI* gehören können, wie z.B. Animations- und Kamerakontrolle, Natural Language Processing, sowie die Tool-Entwicklung zur Unterstützung der *Game AI* nicht Gegenstand dieser Master-Thesis.

Weil es sich bei der zugrundeliegenden Implementierung der Master-Thesis um eine Neu-Entwicklung handelt und es sich bei der Entwicklung einer vollständigen *Game AI* um eine komplexe und arbeitsintensive Arbeit handelt, die in aller Regel von einem größeren Team und in einem größeren Zeitraum entwickelt wird, können erweiterte Aspekte wie *Learning*, *Player Prediction* und *Opponent Modelling* nur rudimentär oder gar nicht berücksichtigt werden. Mit der Schaffung einer durchdachten *Game AI*-Architektur soll allerdings die Grundlage für Weiterentwicklungen geschaffen werden, die sich u.a. auch mit diesen Themen eingehender auseinander setzen können.

Abschließend sei noch erwähnt, dass bei der Bot-Implementierung von der Annahme ausgegangen wird, dass eine Bot-Instanz jeweils auf einem separaten Rechner läuft und somit Aspekte wie Ressourcen-Teilung oder *NPC Host Allocation* keine Rollen spielen.

1.4 Aufbau der Arbeit

In Kapitel 2 „Grundlagen“ werden allgemeine und elementare Themen behandelt, die für das weitere Verständnis der Arbeit erforderlich sind. Dabei werden einerseits grundlegende Definition zu den Begriffen „Computerspiel“ und „Game AI“ präsentiert, die zur besseren Einordnung und Erfassung der Aufgabenstellung beitragen. Andererseits werden mit der Vorstellung des Entwurfs des rationalen Agenten und des Agentenmodells nach Reynolds, in diesem Kapitel bereits erste abstrakte Lösungskonzepte präsentiert, auf denen der eigene konkrete Ansatz beruht.

Kapitel 3 „Verwandte Ansätze“ stellt alternative Konzepte oder konkrete Verfahren zum eigenen Ansatz vor. Im ersten Teil des Kapitels werden die in der Einleitung erwähnten unzureichenden alternativen Ansätze zur realistischen Lasterzeugung detaillierter vorgestellt und konkrete Projekte aufgeführt in denen sie eingesetzt werden. Der restliche Teil des Kapitels behandelt Konzepte, Methoden oder konkrete Verfahren zur Entwicklung einer *Game AI* oder deren wichtigsten Komponenten. Der eigene Ansatz einer *Game AI* nimmt eine Trennung der Verantwortlichkeiten und eine Einordnung dieser in unterschiedliche Ebenen vor. Bei der Vorstellung der Alternativen in diesem Kapitel wird dieser Aufteilung Rechnung getragen, indem ebenfalls Ebenen-gezielt verwandte Ansätze vorgestellt werden.

Kapitel 4 „Konzept und Rahmenbedingungen der Game AI“ und Kapitel 5 „Design und Implementierungsdetails der Game AI“ stellen die System-Kapitel der Arbeit dar und dienen der Beschreibung des eigenen Ansatzes. Das erste System-Kapitel spezifiziert mit der genauen Spielbeschreibung von *Planet PI4* die konkrete Einsatzumgebung und das tatsächliche Einsatzfeld der *Game AI*. Des Weiteren erfolgt die Vorstellung des allgemeinen *Game AI*-Konzepts zur Verdeutlichung der bei der Implementierung verfolgten Design-Prinzipien. Das zweite System-Kapitel geht daraufhin ins Detail und konkretisiert das zuvor

vorgestellte *Game AI*-Konzept auf Klassenebene. Ergänzt werden die Ausführungen mit der detaillierten Behandlung von wichtigen Implementierungsdetails der einzelnen Komponenten und Besonderheiten der *Game AI*-Implementierung.

In Kapitel 6 „Evaluation“ erfolgt die Beschreibung der vorgenommen Validierung des eigenen Ansatzes in Bezug auf die Eignung zur Erzeugung einer realistischen Netzwerklast. Zur Beurteilung der Ergebnisse wird vor allem ein direkter Vergleich mit der Vorgänger-Implementierung herangezogen.

Abgeschlossen wird die Arbeit mit dem Kapitel 7 „Ausblick und Fazit“ indem einerseits die Ergebnisse der Evaluation und der Arbeit zusammengefasst, andererseits Bereiche für mögliche Verbesserungen und ansetzende Weiterentwicklungen vorgestellt werden.

2 Grundlagen

Das Ziel dieses Kapitels ist die Schaffung einer für das weitere Verständnis der Master-Thesis ausreichenden Grundlagenbasis. Hierzu wird im Abschnitt 2.1 „Was ist ein Computerspiel?“ eine Definition des Begriffs Computerspiel präsentiert, die versucht die Quintessenz zahlreicher Definitionen zu erfassen und in einer für den Rest dieser Ausarbeitung gültigen Form zu vereinen. Dies ist wichtig, weil mit *Planet PI4* ein Computerspiel als direkte Einsatzumgebung für die zu entwickelnde *Game AI* eingesetzt wird und somit eine genaue Kenntnis dieser notwendig ist.

Dieselbe Argumentation lässt sich ebenfalls auf den Begriff *Game AI* anwenden, welche die zu verwirklichende Komponente dieser Master-Thesis bezeichnet, und deswegen im nächsten Abschnitt 2.2 „Academic AI vs. Game AI“ einer eingehenden Betrachtung unterzogen wird. Die Präsentation einer für den Rest dieser Ausarbeitung gültigen Definition ist wie zuvor auch der wichtigste Bestandteil des Abschnittes.

Der Abschnitt 2.3 „Der Entwurf eines Rationalen Agenten“ stellt ein übergeordnetes weit verbreitetes Model zur Beschreibung intelligenten Verhaltens in virtuellen Umgebungen vor. Dieses Modell beschreibt die allgemeine Architektur der entwickelten *Game AI* und bildet somit die Grundlage des in Kapitel 4 vorgestellten eigenen Systemansatzes und der daraus resultierenden konkreten Implementierung. Das Konzept findet ebenfalls Verwendung als Grundlage der in Abschnitt 3.2 und 3.3 vorgestellten verwandten Ansätze.

Das entwickelte *Game AI*-Konzept der Master-Thesis stellt darüberhinaus eine Erweiterung des reynoldschen Modells dar, welches in das Modell eines rationalen Agenten aus Abschnitt 2.3 integriert wird. Aus diesem Grund stellt der letzte Abschnitt 2.4 „Steering Behaviors und das Agentenmodell nach Reynolds“ dieses Modell vor und erläutert das, für das weitere Verständnis dieser Arbeit wichtige, Konzept der *Steering Behaviors*.

2.1 Was ist ein Computerspiel?

In den 70er Jahren des vergangenen 20. Jahrhunderts entwickelte der spätere Atari-Gründer Nolan Bushnell ein Spiel namens *Pong* und markierte damit ein Ereignis in der Geschichte, welches aus heutiger Sicht als die Geburtsstunde der Computerspielbranche angesehen werden kann [57]. *Pong* war allerdings nicht das erste bekannte Computerspiel. Bereits 1952 entwickelte A. Sandy Douglas auf einem EDSAC[2]-Computer, dem ersten speicher-basierten Rechner der Welt, an der Cambridge Universität ein *Tic Tac Toe*-Spiel mit dem Namen *OXO* [11]. Auf Grund der Tatsache, dass die Entwicklung des Spiels nicht primär der Unterhaltung diente, sondern der Verifikation einiger von ihm aufgestellten Thesen im Bereich der Mensch-Maschinen Interaktion und lange Zeit keinem breiten Publikum außerhalb der Universität vorgestellt wurde, gilt stattdessen öfters auch *Tennis for Two* als das erste Computerspiel, das vom amerikanischen Physiker William Higinbotham 1958 konstruiert wurde [15, S. 1] [58, S. xvii]. Daneben gibt es durchaus noch weitere Entwicklungen die den Anspruch erheben das erste Computerspiel zu sein [8] [57], allerdings mit einer jeweils deutlich kleineren Anhängerschaft als die beiden eben vorgestellten.

Ganz gleich welchen Titel man als das erste richtige Computerspiel ansieht, alle hatten eins gemeinsam: Sie waren nur einem sehr eingeschränkten, meist wissenschaftlichen oder akademischen Kreis von Menschen zugänglich. *Pong* jedoch bediente zum ersten Mal erfolgreich eine spielende und vor allem

[2] EDSAC steht für Electronic Delay Storage Automatic Calculator.

zahlende breite Öffentlichkeit. Eine Öffentlichkeit die auf einer einheitlichen und damals erstmal günstigen Spiele-Hardware spielen konnte. Der Erfolg von *Pong* führte erst zu einer stetig wachsenden Zahl von Variationen des Spiels, dann zu einer Lawine von Kopierversuchen Dritter und endete schließlich mit der Entwicklung neuer Spielkonzepte und passender Hardware. Mit anderen Worten: Durch den Erfolg von *Pong* entstand ein neuer Massenmarkt, der zu Beginn des 21. Jahrhundert mehrere Hundert Millionen Spieler weltweit umfasst und einer stetig wachsenden Computerspielbranche Jahresumsätze im zweistelligen, wenn nicht gar dreistelligen, Billionen Dollar Bereich beschert[3] [58, S. 34fff.] [13, S.11].

Obwohl es sich bei der Bezeichnung „Computerspiel“ um einen sehr weit verbreiteten und geläufigen Begriff handelt, stellt sich überraschenderweise eine präzise Beantwortung dieser Frage als recht problematisch heraus. Der Grund hierfür ist jedoch genau der Gleiche, wie für die fälschliche Annahme einer einfachen Beantwortung: Dadurch dass der Begriff eine derart große Verbreitung und flexible Verwendung besitzt, gibt es auch eine unüberschaubare Vielfalt an unterschiedlichsten Entwicklungen weltweit, die sich als Computerspiel bezeichnen. Dieser Umstand und das Fehlen einer einheitlichen Definition des Begriffs „Computerspiel“ sind auch die wesentlichen Gründe dafür, warum die zu Beginn dieses Abschnittes erwähnte Debatte um das erste Computerspiel bis heute so kontrovers diskutiert wird. Die Ursache dafür wiederum resultiert maßgeblich daraus, dass die Definition eines Computerspiels untrennbar mit der Auffassung darüber verbunden ist, was eigentlich unter einem „Spiel“ verstanden wird. Darunter kann je nach subjektiver Auffassung alles Mögliche verstanden werden.

In [47] wird versucht sich genau dieser Problemantik anzunehmen. Nach der Untersuchung und des Vergleichs einer Vielzahl unterschiedlicher Definitionen, aus den verschiedensten Bereichen des Spielens, wird darin folgende, auf das wesentliche beschränkte, Definition eines Spiels präsentiert:

> *A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome [47, S. 80].*

Die Hauptaussagen dieser Definition sind dabei folgende:

- Die Tatsache dass ein Spiel ein System ist und somit erst das Zusammenwirken der einzelnen Teile ein komplexes Ganzes ergibt.
- Das ein oder mehrere Spieler mit diesem System zum Erlangen einer Spielerfahrung interagieren müssen.
- Ein Spiel eine Instanz eines künstlichen Konfliktes ist, welcher allerdings an Konflikten aus der wirklichen Welt angelehnt sein kann.
- Das Regeln sowohl das Verhalten des Spielers beschränken, als auch das Spiel als Ganzes definieren.
- Das jedes Spiel quantifizierbare Ergebnisse liefert, sowie das Erreichen bestimmter Ziele verfolgt.

Aufbauend auf dieser Definition eines Spiels werden in [47, S. 87ff.] vier spezielle Eigenschaften vorgestellt, deren intensivere Ausprägung, nicht allerdings ihre bloße Existenz, ein Computerspiel von anderen Spielformen abhebt:

- **Immediate but narrow interactivity:** Ein Computerspiel ermöglicht eine direkte und unmittelbare Form der Interaktivität, allerdings ist diese durch Eingabegeräte und durch die Komplexität des Spiels stehts beschränkt.

[3] Zum Vergleich: Um laut [48] in die Reihe der zehn erfolgreichsten Kinofilme aller Zeiten aufgenommen zu werden, werden momentan etwas mehr als 900 Millionen Dollar Einnahmen benötigt. Das momentan erfolgreichste Computerspiel *Modern Warfare 3* hat innerhalb von nur 16 Tagen seit der Veröffentlichung eine Milliarde US-Dollar Umsatz eingefahren [12].

- **Manipulation of information:** Ein Computerspiel basiert immer auf der Manipulation von Daten bzw. Informationen, wie z.B. Ein- und Ausgabedaten, Hardwaresteuerungssignale, Interne Programmlogik, Algorithmen, Bilder, Video, Audio, Animationen, 3D-Grafiken usw..

- **Automated complex systems:** Ein Computerspiel ist ein automatisch ablaufendes komplexes System, welches alleine für das (nahezu) direkte Fortschreiten des Spielgeschehens zuständig ist. In einem nicht auf einem Computer umgesetzten Brettspiel beispielsweise, muss der Spieler selbst für das Fortschreiten des Spiels sorgen, indem er z.B. seine Figuren bewegt oder Spielereignisse anderen Spielern vorträgt. In einem Computerspiel würden solche Prozesse automatisiert ablaufen und somit dem Spieler weitestgehend abgenommen.

- **Networked communication:** Eine große Anzahl an Spielen wird nicht ausschließlich alleine gespielt, sondern mit anderen Spielern zusammen. Computerspiele ermöglichen es, dass die Spieler über eine große Distanz miteinander kommunizieren und interagieren, d.h. miteinander spielen, können. Moderne Computerspiele erlauben es ebenfalls, dass eine große Menge an Spielern dies zusammen unternehmen kann.

Seit den 1980er Jahren wird oftmals plattformspezifisch zwischen einem Computer- und Videospiel unterschieden. Ein Computerspiel bezeichnet dabei ein Spiel, welches auf einem Personal Computers (PC) gespielt wird, wohingegen ein Videospiel ein Spiel kennzeichnet, dass für irgendeine Spielekonsole entwickelt wurde. Eine Spielekonsole ist dabei ein meist geschlossenes Hardwaresystem, welches primär zum Spielen konzipiert wurde und einen Anschluss an ein TV-Gerät voraussetzt. Im Rahmen dieser Abschlussarbeit wird auf diese oder andere Unterscheidungen von Computerspielen verzichtet und mit dem Begriff Computerspiel alle Formen digitaler Spiele zusammengefasst, die obiger allgemeinen Definition eines Computerspiels genügen.

2.2 Academic AI vs. Game AI

Künstliche Intelligenz (KI) bzw. *Artificial intelligence* (AI) bezeichnet in erster Linie einen Wissenschaftszweig der Informatik, in dessen Zentrum der Betrachtung die (automatisierte) Schaffung von Intelligenz bzw. das (optimale) Lösen von intelligenten Problemen steht. Unter einem intelligenten Problem versteht man dabei eine zu lösende Aufgabenstellung, welche Aufgrund ihrer Komplexität nicht durch einfaches Ausprobieren aller Möglichkeiten gelöst werden kann [36, S. 1].

Die traditionelle KI (Academic AI) hat im Laufe der Zeit den Begriff der Intelligenz aus einer Vielzahl unterschiedlicher Richtungen untersucht. Je nach Interpretation des Begriffs verfolgte die KI dabei einen anderen Weg zur Erforschung und Erschaffung intelligenter Systeme. Eine konkrete Definition der KI ist aufgrund dieser zeitlich geprägten Auslegung, was Intelligenz bedeutet, als durchaus problematisch anzusehen. Stuart Russell versucht jedoch in [46, S. 17f] den verschiedenen Entwicklungsströmen der KI gerecht zu werden und leitet daraus vier übergeordnete Auffassungen ab. Sie definieren ab wann ein künstliches System als intelligent bezeichnet werden kann.

Demnach ist ein System genau dann intelligent, wenn dieses entweder

- menschlich denken,
- rational denken,
- menschlich handeln oder
- rational handeln kann.

Rationalität meint in diesem Kontext die Fähigkeit, das Richtige oder das Bestmögliche im Sinne einer vorgegebenen Richtlinie, der maximierbaren Nutzenfunktion, zu tun. Somit konzentriert sich die KI

unter Betrachtung der Rationalität auf das optimale Lösen eines Problems. Das Entscheidende bei dieser Betrachtung ist die Vernachlässigung des Prozesses zur Entstehung des Lösungsweges. Der Lösungsweg muss nur optimal unter der Nutzenfunktion sein, wie jedoch die Lösung erreicht wurde, ob durch Berechnungen, logisches Schließen oder auf menschenähnliche Weise, ist nebensächlich. Nach Russell bildet diese Auffassung von Intelligenz das rationale Handeln am besten ab und ist somit seiner Meinung nach die aktuell erstrebenswerteste Richtung der KI-Forschung. Russell begründet diese Annahme indem er aufführt, dass rationales Denken hingegen immer auf korrekten logischen Schlussfolgerungen beruht und diese alleine nicht ausreichen um das ganze Spektrum der Rationalität abzudecken. So gibt es häufig Situationen, wo man zwar nicht das beweisbar Korrekte tun kann, jedoch etwas getan werden muss. Wenn im weiteren Verlauf dieser Ausarbeitung von der traditionellen KI die Rede ist, dann ist dabei die akademische Betrachtung des rationalen Handelns gemeint [46, S. 58-62].

Game AI ist ein Sammelbegriff für den praktischen Einsatz einiger Teilbereiche der traditionellen KI und anderen Teilgebieten der Informatik in Computerspielen. Somit können Computerspiele als mögliches Anwendungsszenario für die von der KI entwickelten intelligenten Systeme angesehen werden. Im strikt akademischen Sinne bezeichnet der Begriff *Game AI* ausschließlich Systeme die für ein intelligentes Verhalten von NPCs zuständig sind. Die *Game AI*-Entwicklung ist allerdings keine primär akademisch betriebene Ausübung, sondern wie in Abschnitt 2.1 erwähnt, übernahm nach dem Erfolg von *Pong* die Industrie maßgeblich die Weiterentwicklung von Computerspielen, sodass der Begriff *Game AI* im Kontext von Computerspielen hauptsächlich von der Industrie geprägt worden ist. So fasst diese unter dem Begriff einer *Game AI* sämtliche Techniken und Systeme zur Lösung jeglicher Art von Steuerungs- und Kontrollproblemen zusammen. Beispielsweise würden die Berechnungen, die Steuerung und das Abspielen passender Greif-Animationen eines NPCs nach einem Gegenstand unter Umständen in den Aufgabenbereich einer *Game AI* fallen. Außerhalb von Computerspielen würde dies eher als ein inverses Kinematik-Problem der Kontrolltheorie aufgefasst werden [15, S. 9ff] [51, S. 4ff].

Die größte Abgrenzung zwischen industriell-geprägter und akademischer Sichtweise liegt allerdings im unterschiedlichen Problemlösungsansatz begründet. In der Industrie steht nicht das optimale Lösen eines KI-Problems im Vordergrund, sondern eine Problemlösung die auf die Maximierung des Unterhaltungswertes ausgerichtet ist. Die Problemlösung muss sich dem zufolge in erster Linie an den Spielspaß, die Hardwareleistung der Kundenzielgruppe und an die Ziele des Spieldesigns ausrichten. Im Extremfall interessiert sich eine *Game AI* überhaupt nicht für optimale Ergebnisse. So könnte die aktuelle Schwierigkeitsmodellierung vorgeben, in bestimmten Situationen nicht optimale Aktionen zu wählen, um den Spieler nicht zu überfordern. Auch kann im Sinne der Abwechslung die Wahl nicht optimaler Ergebnisse sinnvoll erscheinen. In der akademischen Welt hingegen wünscht man sich beweisbar wiederholbare optimale Ergebnisse, die darüber hinaus am besten generell anwendbar sind [51, S. 4ff] [35, S. 4fff].

Im Rahmen dieser Ausarbeitung wird prinzipiell vom akademischen Verständnis einer *Game AI* ausgegangen.

2.3 Der Entwurf eines Rationalen Agenten

Stuart Russell beschreibt einen Agenten als

> *Alles, was eine Umgebung (Environment) über Sensoren (Sensors) wahrnehmen kann und in dieser Umgebung über Aktuatoren (Actuators) handelt [46, S. 55].* [4]

Dieses einfache Konzept auf eine *Game AI* übertragen bedeutet, dass ein Agent gefilterte Informationen, im weiteren Verlauf als Wahrnehmungen (*Perceptions*) bezeichnet, über seine Umgebung, die Spielwelt,

[4] Die englischen Begriffe in den Klammern des Zitats gehören nicht zum Original. Sie wurden an dieser Stelle zur Verständniserleichterung der Abbildungen 1 und 2 eingefügt, da diese die englischen Begriffe verwenden.

bekommt und dann anhand dieser Informationen entscheidet, welche Aktion als nächstes auszuführen ist. Der Auswahlprozess der als nächstes auszuführenden Aktion wird in *Game AI*-Bereich als *Decision Making* bezeichnet. Weiter geht man üblicherweise davon aus, dass ein Agent zwar seine eigenen Aktionen wahrnehmen kann, allerdings nicht immer auch dessen Wirkungen, d.h. man geht von einer Umgebung mit unvollständigen Informationen aus. Abbildung 1 veranschaulicht diesen Zusammenhang.

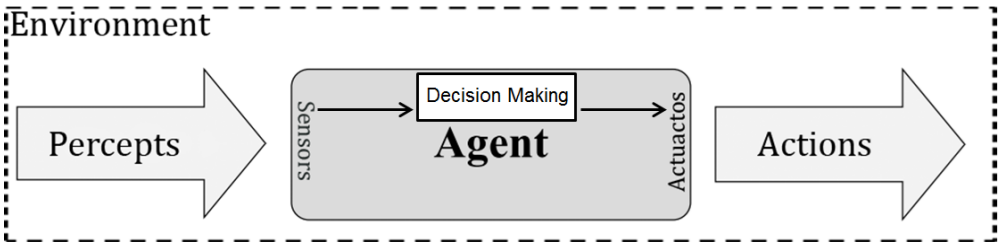

Abbildung 1: Der schematische Aufbau eines simplen rationalen Agenten. Angelehnt an [46] und [36].

Ein rationaler Agent ist in diesem Sinne ein Agent der das Richtige bzw. das Bestmögliche tut. Um entscheiden zu können was das Beste ist, braucht der Agent eine Leistungsbewertungsfunktion (*Utility Function*), die eine Bewertung seiner Aktionen ermöglicht. Wie diese auszusehen hat ist Anwendungsabhängig. Wichtig ist allerdings eine nicht ergebnisorientierte Funktion zu verwenden, sondern eine die den vom Entwickler gewünschten Weg zum Ziel belohnt. Ein rationaler Agent zeichnet sich dadurch aus, dass diese Leistungsbewertung unter Betrachtung seiner Wahrnehmungen, eventuellen Vorwissen und den wahrscheinlichen Auswirkungen seiner möglichen nächsten Aktionen zu maximieren ist. Rationalität sollte in der Anwesenheit von unvollständigem oder unsicherem Wissen nicht mit Perfektion verwechselt werden. Perfektion maximiert immer die tatsächliche Leistung, Rationalität maximiert dagegen die erwartete Leistung unter Berücksichtigung des aktuellen Wissens. Anders ausgedrückt versucht ein rationaler Agent in jeder durch die Wahrnehmung gegebenen Situation die Aktion zu wählen, die in Zukunft den größten Nutzen im Sinne der Nutzenfunktion verspricht.

Generell werden mindestens fünf Agententypen unterschieden, die, mit Ausnahme des *Lernenden Agenten*, als aufeinander aufbauende Prinzipien betrachtet werden können. Nach Russel in [46, S. 72-82] könnte eine mögliche Unterscheidung wie folgt definiert werden:

- **Einfache Reflex-Agenten**: Aktionsauswahl basiert nur auf aktueller Wahrnehmung und wird durch *Condition-Action-Rules* modelliert.

- **Modellbasierte Reflex-Agenten**: Es wird ein interner Zustand und ein Weltmodell verwaltet, um nicht beobachtbare Aspekte zu modellieren. Nicht beobachtbare Aspekte sind beispielsweise gegnerische Aktionen außerhalb des Wahrnehmungsbereiches des Agenten.

- **Zielbasierte Agenten**: Anhand des Weltmodells und des internen Zustandes beeinflusst eine Zielfunktion die Aktionsauswahl. Suchen und Planen sind wichtige Mittel um Aktionsfolgen zu finden, die Ziele erreichen.

- **Nutzenbasierte Agenten**: Zielerfüllung wird auf den Nutzen analysiert, um zwischen Aktionen oder Aktionsketten die Ziele erfüllen zu unterscheiden. Eine Nutzenfunktion bildet einen Zustand auf eine, den Nutzen darstellende, reelle Zahl ab.

- **Lernende Agenten**: Lernen erlaubt in unbekannten Umgebungen eine gegenüber dem Ausgangswissen größere Kompetenz aufzubauen.

Prinzipiell kann man einen *Nutzenbasierten Agenten*, egal ob mit oder ohne Lernaspekt, als den Agenten bezeichnen der einem rationalen, und somit intelligenten, Agenten am nächsten kommt. Im weiteren

Verlauf wird mit Agent genau dieser Typ bezeichnet. Abbildung 2 fasst die wichtigsten abstrakten internen Komponenten dieser Agentenart nochmal zusammen.

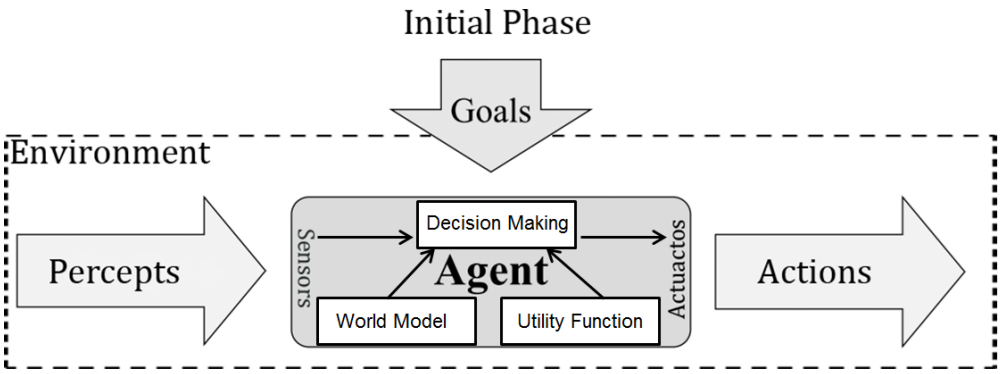

Abbildung 2: Der *Nutzenbasierte Agent* und seine wichtigsten Komponenten als Erweiterung des simplen Agenten aus Abbildung 1. Angelehnt an [46] und [36].

Ian Millington und John Funge bezeichnen den Agentenentwurf als einen *Bottom-Up*-Entwurf, wo das endgültige Spielverhalten aus dem Zusammenspiel der Agenten untereinander resultiert. Im Vergleich dazu würde ein nicht Agentenbasierter Entwurf einem *Top-Down*-Entwurf entsprechen, wo ein einzelnes System das Verhalten aller NPCs zentral berechnet und die auszuführenden Aktionen lediglich, wenn überhaupt, an diese zur Ausführung weiter gibt [35, S. 11]. Aus diesem Grund ist dieser Ansatz, auch als ***M**ulti**a**genten **S**ystem* (MAS) bezeichnet, als ein dezentrales System anzusehen. Dies ist eine für die Entwicklung der *Game AI* wichtige Erkenntnis, die mit *Planet PI4* für ein Spiel entwickelt wird, welches mit P2P auf eine ebenfalls dezentrale Systemarchitektur setzt.

Mateas schlussfolgert in [34], dass dem Konzept des *Zielbasierten Agenten* bzw. seiner Erweiterungen eine tragende Schlüsselrolle bei der Entwicklung einer guten *Game AI* zukommt. Begründet wird diese Hypothese indem aufgeführt wird das die wichtigste Aufgabe einer *Game AI* die Realisierung von für den Spieler nachvollziehbaren und autonom wirkenden Verhalten von NPCs darstellt. Glaubwürdige NPCs sind die Grundlage zur Bewertung von intelligenten Verhalten, denn erst wenn das Verhalten in einen für den Spieler nachvollziehbaren Kontext gestellt wird, kann dieser zwischen intelligenten und nicht intelligenten Verhalten unterscheiden. Eine Möglichkeit für die Realisierung der Glaubwürdigkeit von NPCs ist die Verwendung des eben vorgestellten Agentenkonzeptes, welches NPCs als Ziele verfolgende Agenten definiert, die auf Grundlage ihrer aktuellen Ziele und ihrem Wissen über die Welt, gewisse sichtbare Aktionen zur Zielerreichung ausführen. Der menschliche Verstand kann durch Abstraktion komplexe Systeme, wie andere Menschen, leichter durchschauen und somit dessen Wirkung bzw. im Fall von NPCs deren Verhalten nachvollziehen. Indem der Spieler von der technischen Realisierung eines NPCs abstrahiert und diesen stattdessen als einen Ziele verfolgenden Agenten betrachtet, kann der Spieler das Verhalten der NPCs besser bewerten und leichter nachvollziehen. Laut Mateas sind für den menschlichen Spieler glaubwürdige und intelligente NPCs ein wichtiger Faktor zur Schaffung einer in sich realistischen und lebendigen Spielwelt, die wiederum große Bedeutung für den Spielspaß eines Spieles hat.

Bei der Generierung einer realistischen Netzwerklast geht es zwar nicht primär um Faktoren wie den Spielspaß oder der Erzeugung glaubhaften NPC-Verhaltens aus Sicht eines menschlichen Spielers, jedoch basiert das Konzept des zielbasierten Ansatzes auf einer typisch menschlichen Abstraktionsweise zur Situationsbewältigung und erhebt dabei einen gewissen Realismus-Anspruch bei der Verhaltensgenerierung. Diese beiden Eigenschaften spielen bei der möglichst genauen Erfassung und Nachahmung der menschlichen Spielart eine zentrale Rolle, die wiederum die Grundlagen für eine realistische Generierung der Netzwerklast darstellen.

2.4 Steering Behaviors und das Agentenmodell nach Reynolds

Steering is the reactive, non-deliberative movement of physical agents [3, S. 2].

Der Begriff *Steering* bzw. *Steering Behaviors* im Kontext von autonomen Agenten wurde maßgeblich durch die Arbeit von Reynolds [44] aus dem Jahr 1999 geprägt. *Steering* bezeichnet darin die mittlere der drei hierarchisch angeordneten Ebenen der von Reynolds unternommenen Unterteilung und Strukturierung zur Aufgabe der Bewegungssteuerung eines autonomen Agenten in einer virtuellen Umgebung, welche in Abbildung 3 dargestellt ist.

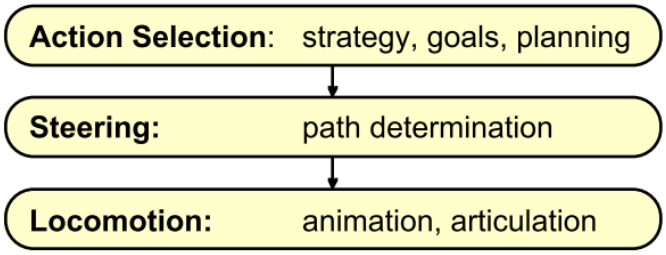

Abbildung 3: Das Modell nach Reynolds zur hierarchischen Unterteilung der Verantwortlichkeiten zur Bewegungssteuerung eines autonomen Agenten. Entnommen aus [44].

Die beiden anderen, die *Steering*-Ebene einschließenden, Ebenen sind zum Einem die darüber angesiedelte Ebene der *Action Selection* und zum anderen die *Locomotion*-Ebene darunter. Wenn die *Steering*-Ebene nach dem Zitat von [3, S. 2] als die „non-deliberative"-Ebene angesehen werden kann, d.h. eine Ebene darstellt, die ihre Entscheidungen ohne eines vorangestellten Abwägungsprozesses bestimmt, dann kann im Gegensatz dazu die darüber liegende Ebene der *Action Selection* als die „deliberative"-Ebene im Modell von Reynolds bezeichnet werden. Vergleichbar mit der allgemeinen Aufgabe des *Decision Making* im Konzept des rationalen Agenten aus Abschnitt 2.3 muss in dieser Ebene eine abwägende Entscheidung unter Einbeziehung etwaiger Ziele, Pläne oder Strategien darüber getroffen werden, welche Aktion als nächstes am besten auszuführen sei. Ist eine Aktion in einem Simulationszyklus bestimmt, wird diese zur Realisierung an die *Steering*-Ebene weitergereicht [44].

Die *Steering*-Ebene ermittelt dabei ob die Aktionsausführung direkt unternommen werden kann, die Aktion in weitere Unteraktionen zerlegt werden kann oder zusätzliche Aktionen vorher eingeschoben werden müssen, weil z.B. vorher noch einem Hindernis ausgewichen werden muss. Für diese Aufgabe greift die *Steering*-Ebene auf sogenannte *Steering Behaviors* zurück. *Steering Behaviors* sind reaktive Verhaltensweisen auf Bewegungsebene, die unter Einbeziehung lokaler Umgebungsinformationen als Eingabe, einen *Steering*- bzw. Bewegungsvektor als Ausgabe produzieren [3]. In [44] präsentiert Reynolds u.a. mit *seek*, *arrival*, *flee*, *pursuit*, *evasion*, *wander* und *obstacle avoidance* eine Basismenge von *Steering Behaviors*. Die einzelnen *Steering Behaviors* können dabei aufeinander aufbauen wie z.B. *arrival* auf *seek* aufbaut, indem es dem *seek*-Verhalten ein Abbremsen vorm Erreichen des Ziels einfügt. Weiter lassen sich ebenfalls beliebige Kombinationen anstellen, die zu komplexen und eindrucksvollen *Steering Behaviors* führen können. Ein Beispiel hierfür ist die Nachbildung des Schwarmverhaltens von Vögeln durch das berühmte *Flocking-Steering Behavior* von Reynolds. Nichtsdestotrotz stellen *Steering Behaviors* in der Regel voneinander unabhängige Berechnungen an, deren Ergebnisse sich mitunter auch untereinander wiedersprechen können. Aus diesem Grund ist eine weitere wichtige Aufgabe dieser Ebene die geschickte Auswahl und Kombination der einzelnen *Steering Behaviors*. Damit der unteren *Locomotion*-Ebene letzten Endes ein einziges *Steering*-Ziel übergeben werden kann, welches durch einen zusammenfassenden oder ausgewählten *Steering*-Vektor repräsentiert wird.

Bei der untersten Ebene der *Locomotion* handelt es sich um eine reine Ausführungsebene. Sie besitzt die Kontrolle über den Körper bzw. den Gegenstand der Bewegungsausführung. Im Kontext des rationalen Agenten aus Abschnitt 2.3 würde es sich hierbei um die Schnittstelle zwischen der *Decision Making*-Komponente und den Aktuatoren des Agenten handeln, die für die Umsetzung der gewünschten Aktion durch tatsächliche Handlungen in der virtuellen Agentenumgebung zuständig sind. Dabei können die Aufgaben dieser Ebene abhängig von der angewandten Definition einer *Game AI* (siehe Abschnitt 2.2) oder der konkret im Einsatz befindlichen *Game AI*-Architektur vielfältiger Natur sein. Aufgrund dessen kann je nach Auffassung das Aufgabenspektrum sich von einer reinen Konvertierung des *Steering*-Vektors in konkrete Steuerungssignale über die Definition der Aktuatoren bzw. der Bewegungssteuerungskomponenten samt dazu passendem Animationshandling erstrecken. Ein abschließendes Beispiel soll das Konzept der drei Ebenen zusammenfassend verdeutlichen.

In einem FPS könnte die *Action Selection*-Ebene zu dem Schluss kommen einen etwas weiter entfernten Gegner anzugreifen, weil sich in der aktuellen Situation kein anderer Gegner in der unmittelbaren Nähe befindet. Dieses Ziel wird von der *Steering*-Ebene unterteilt in die Bestimmung der kürzesten Route durch das Level zur gegnerischen Position und der Verfolgung dieser Route. Da ein Gegner in der Zwischenzeit aller Wahrscheinlichkeit nicht auf der Stelle stehen bleibt, wird die gegnerische Position durch eine Prognose der zukünftigen Position des Gegners ersetzt. Auf dem Weg zur wahrscheinlich zukünftigen Position des Gegners muss immer wieder im ganzen Level verstreuten statischen Hindernissen ausgewichen werden. Hierzu muss die Verfolgung der Route zur prognostizierten Position des Gegners hin und wieder unterbrochen werden und durch temporäre Ausweichbewegungen ersetzt werden, die im Anschluss daran wieder fortgesetzt werden kann. Während der Verfolgung des Pfades oder dem Ausweichen von Hindernissen werden die dazu berechneten *Steering*-Vektoren an die *Locomotion*-Ebene weitergeben, die diese letzten Endes in eine konkrete Fortbewegung und Ausführung von Bewegungsanimationen des virtuellen Körpers des Agenten umsetzt.

3 Verwandte Ansätze

Das Ziel dieser Master-Thesis ist die realistische Lasterzeugung durch Verwendung computergesteuerter Spieler bzw. einer *Game AI*, welche die Steuerung dieser Spieler übernimmt. Wie bereits in der Einleitung angedeutet, ist dies im Kontext der Evaluation von P2P-Overlays nur eine von vielen Möglichkeiten zur Lasterzeugung. Abschnitt 3.1 „Methoden zur Lasterzeugung in Spielen“ stellt alternative Methoden und Verfahren zur Lasterzeugung in anderen Projekten vor.

Der Rest dieses Kapitels behandelt Konzepte, Methoden oder konkrete Verfahren zur Entwicklung einer *Game AI* oder deren wichtigsten Komponenten. Mit dem Konzept eines rationalen Agenten aus 2.3 wurde bereits ein allgemeingültiges Modell vorgestellt, das als Grundlage zur weiteren Konzeptionierung einer *Game AI*-Architektur herangezogen werden kann. Die in Abschnitt 5.2 präsentierte *Game AI*-Architektur der im Rahmen dieser Master-Thesis umgesetzten *Game AI* stelle genau solch eine Konkretisierung dar. Aus diesem Grund wird in Abschnitt 3.2 „Halo’s AI-Architektur“ exemplarisch eine mögliche alternative Konkretisierung des allgemeinen Architekturansatzes präsentiert, wie diese auch tatsächlich mit Halo in einem Computerspiel eingesetzt wird.

Das *Decision Making* ist das Herzstück beim Entwurf eines rationalen Agenten und deswegen auch eines der wichtigsten Komponenten des eigenen Ansatzes. Bei der Vorstellung der fünf Agententypen des vorherigen Abschnitts 2.3, war jedes Mal die Art und Weise, wie die nächste Aktion des Agenten ausgewählt werden soll, das ausschlaggebende Unterscheidungskriterium. Mit anderen Worten: Die Agententypen oder konkrete *Game AI*-Implementierungen unterscheiden sich hauptsächlich bezüglich ihrer *Decision Making*-Komponente. Es existiert eine Vielzahl von Verfahren zur Lösung dieses Problems oder deren Unterstützung. Einige in der Computerspielbranche populäre Verfahren sind in Tabelle 1 aufgelistet.

Decision Making Verfahrensbezeichnung	**Bekannte Verwendungen in Computerspielen**
Rule-Based System	Baldur Gate, Virtua Fighters 2 und Halo
(Hierarchical) Finite State Machines	Warcraft III, Quake und Halo
Decision Trees	Black & White 2
Hierarchical Task Network Planning	Killzone 2 und Unreal Tournament
Goal-Oriented Action Planning	F.E.A.R., S.T.A.L.K.E.R. und Empire: Total War
Behavior Trees	Halo 2, Crysis und Crysis 2
Scripting Systeme	Neverwinter Nights und Unreal Tournament
Scheduling Systeme	Elder Scrolls IV: Oblivion

Tabelle 1: Der Einsatz von *Decision Making*-Verfahren in modernen Computerspielen.

Die in Tabelle 1 präsentierte Auflistung erhebt dabei keinen Anspruch auf Vollständigkeit und könnte somit noch um zahlreiche Variationen oder Erweiterungen, wie z.B. *Goal Trees* [24], eine *Decision Tree*-Erweiterung, oder Kombinationen bekannter Ansätze, wie z.B. das *Knowledge-Based Behavior System* [26], ein *Decision Tree/Finite State Maschine*-Hybrid, beliebig erweitert werden.

Aufgrund der großen Menge an unterschiedlichen *Decision Making*-Systemen wird in Abschnitt 3.3 „GOAP - Goal-Oriented Action Planning“ mit der Vorstellung des GOAP-Ansatzes exemplarisch ein konkretes *Decision Making*-System detailliert vorgestellt. GOAP wurde aus der Kandidatenmenge gewählt,

weil es ein Verfahren darstellt, welches im Vergleich zum *Behavior Tree*-Ansatz, einen völlig anderen Lösungsansatz mit unterschiedlichen Vor- und Nachteilen verfolgt. Die Unterschiedlichkeit zu *Behavior Trees* ist deswegen interessant, weil der eigene in den Kapiteln 4 und 5 vorgestellte Ansatz des *Decision Making* darauf basiert.

Das *Decision Making* ist nicht die einzig wichtige Aufgabe einer *Game AI*. Wie bereits in Kapitel 2 erwähnt, ist das entwickelte *Game AI*-Konzept der Master-Thesis ebenfalls eine Erweiterung des reynoldschen Modells aus Abschnitt 2.4 und somit die Umsetzung und Kombinierung von *Steering Behaviors* eine weitere wichtige Aufgabe der entwickelten *Game AI*. Abschnitt 3.4 „Verfahren zur Kombinierung von Steering Behaviors“ präsentiert alternative Methoden zur Kombinierung von *Steering Behaviors*.

Der letzte Abschnitt 3.5 „Vorgänger-Implementierung: DragonBot“ präsentiert den von der Vorgänger-Implementierung umgesetzten Ansatz. Des Weiteren erfolgt in hier eine Untersuchung der Erfüllung der in der Einleitung definierten Ziele der Master-Thesis. Die Vorgänger-Implementierung spielt vor allem in der Evaluation des eigenen Ansatzes eine wichtige Rolle.

3.1 Methoden zur Lasterzeugung in Spielen

Neben dem relativ neuen und im Rahmen dieser Master-Thesis zu realisierenden Ansatz der Verwendung von computergesteuerten kontext-sensitiven Spielern existieren derweil noch zwei weitere Ansätze zur Erzeugung einer realistischen Netzwerklast, der sich auch der Großteil der bisherigen Verfahren zuordnen lässt.

Der erste Ansatz bezeichnet dabei die direkte Benutzung von statischen Game-Traces. Game-Traces sind dabei als Aufzeichnungen tatsächlicher von Menschen durchgeführter Spielpartien zu verstehen. Diese Game-Traces können unterschiedlichste Informationen enthalten, wie z.B. sämtliche durchgeführte Aktionen eines jeden Spielers zu jedem Zeitpunkt im Spiel oder die jeweils variierende Anzahl der Nachbarn des vom Spieler repräsentierten Peers. Zumeist beschränken sich allerdings die Game-Traces auf von der Applikationsebene und somit von der konkreten Spielmechanik unabhängigen Informationen, wie z.B. der Positionsangabe eines jeden Spielers zu jedem Zeitpunkt im Spiel. Die zwei größten Nachteile des Einsatzes von statischen Game-Traces sind, neben dem kosten- und zeitaufwendigen Erstellungsprozess oder der Abhängigkeit des Vorhandenseins, entsprechend der eigenen Wünsche, passender Game-Traces, die nicht vorhandene Skalierbarkeit und Konfigurierbarkeit/Parametrisierbarkeit der erzeugten Last. Die Vorteile sind hingegen die gewährleistete Reproduzierbarkeit der erzeugten Last, sowie die berechtigte Annahme dass diese so erzeugte Last per se als realistisch angesehen werden kann.

In [2] wird die Evaluierung des darin vorgestellten Anti-Cheating-Protokolls für P2P-Overlays mit eben jenem Ansatz der direkten Verwendung von statischen Game-Traces realisiert. Dazu werden Simulationen basierend auf den Game-Traces von Spielpartien mit jeweils 10, 25, 50, und 75 Spielern des Spiels *XPilot* [52] durchgeführt. Pro Simulation wird anhand der Daten eines Game-Traces das Spiel nachgespielt, es werden allerdings zusätzlich vereinzelt Cheat-Versuche eingefügt. Cheat-Versuche sind in diesem Kontext unerlaubte Lese- oder Schreiboperationen oder gar Blockierungsversuche der zwischen den Peers ausgetauschten Spielnachrichten.

Die Alternative zum ersten Ansatz ist die Ableitung von sogenannten *Mobility*- oder *Behavior*-Modellen anhand einer Vielzahl von zugrunde liegenden Game-Traces. Mit einem solchen Modell kann dann die entstehende Last, entsprechend eingestellter Parameter, wie z.B. der Spieleranzahl, approximiert werden. Der Vorteil dabei ist, dass nachdem ein Modell einmal aufgestellt worden ist, das eigentliche Spiel zur Lasterzeugung nicht mehr benötigt wird. Die Last wird derweilen nur noch simuliert bzw. berechnet.

Die einfachste und bekannteste Form eines solchen Modells ist das *Random Waypoint Mobility Modell* (RWP) [23].

Beim RWP-Modell wird jeweils zufällig ein Punkt auf der Karte ausgewählt und sich zu diesem geradewegs mit konstanter Geschwindigkeit hinbewegt. Am ausgewählten Punkt angekommen wird eine gewisse Zeit abgewartet, um im Anschluss daran erneut einen zufälligen Punkt auszuwählen und sich zu diesem hinzubewegen. Dieser Vorgang wird dann entsprechend endlich oft wiederholt. Abbildung 4 (a) zeigt wie ein durch das RWP-Modell entstandenes Bewegungsverhalten für eine *Quake 2*-Partie (FPS-Spiel) aussehen könnte. Abbildung 4 (b) zeigt hingegen wie das Referenz-Bewegungsverhalten einer echten Spielepartie *Quake 2* im Vergleich dazu auszusehen hat. Der Vergleich beider Abbildungen zeigt, dass das RWP Model, obwohl es sich mit der Nachbildung der Bewegung nur auf einen Aspekt bei der Generierung der Netzwerklast beschränkt, dennoch große Ungenauigkeiten aufweist. Die zwei auffallendsten Mängel sind dabei die fehlende oder fehlerhafte Erfassung von **P**oints **o**f **I**nteressts (POI) und die Beschränkung auf ausschließlich gerade Bewegungsabläufe.

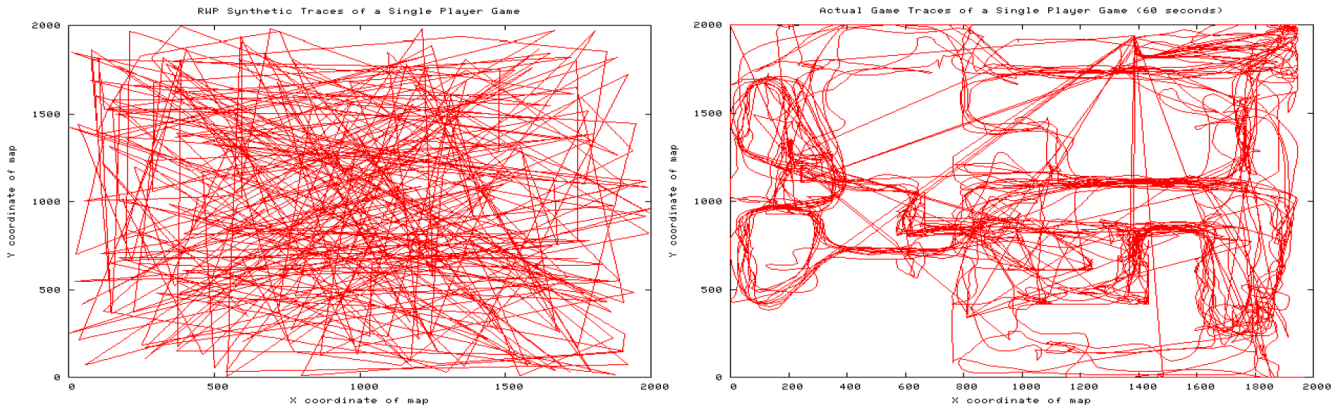

Abbildung 4: (a) Nachbildung des Bewegungsprofils der *Quake 2*-Partie aus (b) mit dem RWP-Modell. (b) Bewegungsprofil einer echten *Quake 2*-Partie. Beide Grafiken entnommen aus [53].

Aufgrund dieser oder weiterer Mängel des RWP-Modells wurden zahlreiche Modifikationen entwickelt und in anderen Projekten wie VON [19] oder MOPAR [60] eingesetzt. Einige Beispiele für derartige Modifikationen sind beispielsweise die Hinzunahme von zufälligen Interaktionen, wie *Hits* oder *Shoots*, oder Ereignissen, wie *Deaths* oder *Respwans*. Einige Modifikationen bzw. Erweiterungen des RWP-Modells unterstützen auch die Ansteuerung von zuvor aus echten Game-Traces ausgemachten POIs oder ungerade bzw. kurvige Bewegungsabläufe. Das ***N**etworked **G**ame **M**obility **M**odel* (NGMM) [53] ist eine solche RWP-Modell-Erweiterung, die die beiden Letzt genannten Modifikationen vornimmt.

Zum Erreichen der POI-Ansteuerung wird in einem vorgelagerten *Pre-Processing*-Schritt eine möglichst große Menge an Referenz-Game-Traces verarbeitet und anhand eines Popularitätsmodells die Punkte auf der Karte bestimmt, die in den Game-Traces besonders häufig besucht worden sind. Aus diesen Daten wird daraufhin die Welt in stationäre und mobile Punkte bzw. Areale unterteilt. Stationäre Areale sind die identifizierten POIs und dienen als Ansteuerungspunkte für das zugrundliegende RWP-Modell. Der *Random Walks* zwischen den stationären Arealen ist allerdings nicht mehr komplett zufällig. Zur Auswahl wird nämlich eine Verteilungsfunktion verwendet, die die zuvor gemessene Popularität eins POIs und die Entfernung zur aktuellen Position berücksichtigt. Die Unterstützung von nicht ausschließlich geraden Bewegungsabläufen wird erreicht indem in den mobilen Arealen, die sich jeweils zwischen Paaren von stationären Arealen befinden, die Methode der kleinsten Quadrate angewandt wird. Dabei wird versucht eine möglichst genaue Kurvenanpassung an die tatsächlichen Positionen der Game-Traces, die sich innerhalb eines mobilen Areals befinden, vorzunehmen und die somit gefundene Kurve als Bewegungs-

grundlage zu verwenden. Abbildung 5 zeigt ein mit NGMM entstandenes Bewegungsprofil, welches unter denselben Bedingungen wie zuvor auch die Abbildungen 4 (a) und (b) entstanden sind.

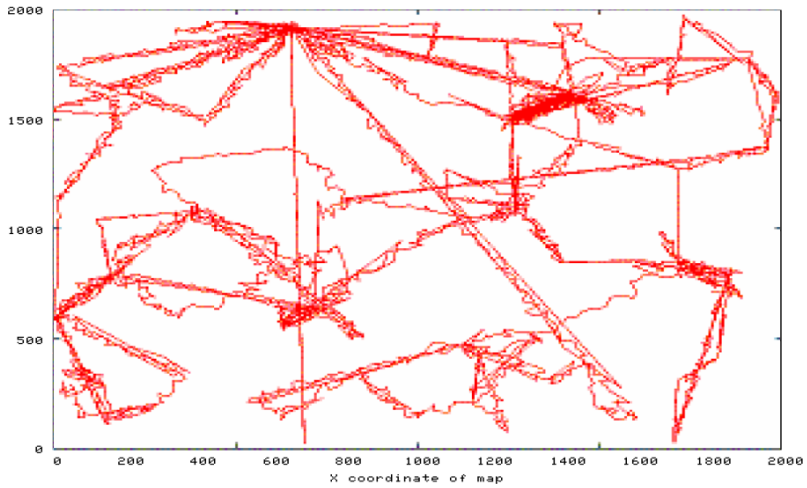

Abbildung 5: Nachbildung des Bewegungsprofils der *Quake 2*-Partie aus 4 (b) mit dem NGMM-Modell. Grafik entnommen aus [53].

NGMM oder vergleichbare Lösungen dieses Ansatzes erfüllen zumeist mit der Reproduzierbarkeit und der Skalierbarkeit zwei der vier in der Einleitung vorgestellten Bewertungskriterien zur Erzeugung einer realistischen Netzwerklast. Die Reproduzierbarkeit ist dabei durch den deterministischen Erzeugungsprozess der Daten gegeben und die Skalierbarkeit durch die allgemeine Anwendbarkeit des Modells gesichert. Die Möglichkeit der Skalierbarkeit in Bezug auf Spieleranzahl oder diverser anderer Parameter, wie z.B. bei NGMM der zu verwendeten probabilistischen Verteilungsfunktion, lassen ebenfalls auf eine gewisse Konfigurierbarkeit des Ansatzes schließen. Womit drei der vier Bewertungskategorien erfüllt scheinen. Der Nachteil dieses Ansatzes ist jedoch die weiterhin zu starke Simplifizierung der Lasterzeugung von Computerspielen, die mit einer Ungenauigkeit der Ergebnisse und somit einem verbesserungswürdigen Realitätsgrad einhergeht. Selbst aufwendigere Modelle wie das NGMM, das einige Aspekte der Applikationsebene wie z.B. POIs berücksichtigt, weißt immer noch erkennbare Ungenauigkeiten auf, wie u.a. der Vergleich von Abbildung 5 und Abbildung 4 (b) exemplarisch veranschaulicht.

Ein Computerspiel besteht, wie unter 2.1 aufgeführt, aus weit mehr als Bewegungsabläufen oder simplen zufälligen Verhaltensmustern. Gerade die Komplexität der Spielwelt und die Vielzahl an Interaktionsmöglichkeiten, sowie besonders das Zusammenspiel der Interaktionen untereinander zeichnen moderne Computerspiele aus und müssen dementsprechend in der Lasterzeugung berücksichtigt werden. Besonders der letztgenannte Punkt verlangt nach der Notwendigkeit der Reaktionsfähigkeit, statt der in *Mobility*- bzw. *Behavior*-Modellen modellierten Form des blinden Aktionismus, welcher zu zufälligen und somit voneinander losgelösten Spielaktionen führt.

Der Begriff Reaktion beschränkt sich in diesem Kontext nicht nur auf Aktionen, wie das Erwidern gegnerischen Feuers, sondern erfasst auch den Entschluss zu kooperativen Verhalten. Beabsichtigt beispielsweise eine Gruppe von Spielern möglichst nah beieinander zu bleiben, um sich zusammen dem Gegner zu stellen, so ist dies ebenfalls als eine Spielaktion zu bewerten und sollte dementsprechend von einem Modell zur Lasterzeugung erfasst werden. Das kooperatives Verhalten signifikante Auswirkungen auf die erzeugte Last haben kann, lässt sich u.a. an der Tatsache festmachen, das im erwähnten Beispiel die Kollektivbildung unmittelbare Konsequenzen auf die Anzahl der Nachbarn hätte, was wiederum maßgeblich die Menge der auszutauschenden Nachrichten innerhalb einer AOI bestimmt. Eine weitere Schwäche dieses Ansatzes ist es, dass die Modelle auf einer endlichen Menge von Spielinstan-

zen bzw. Game-Traces beruhen. Spielsituationen die in dieser Menge von Spielinstanzen nicht enthalten waren oder nur in einer unzureichenden Anzahl, werden in einem zusammenfassenden oder möglichst allgemeingültigen Modell nicht erfasst [30].

Um den aufgeführten Nachteilen entgegenzuwirken wird mit dem Ansatz der computergesteuerten Spieler in *Planet PI4* ein anderer Ansatz zur Lasterzeugung verfolgt. Ein erster Realisierungsversuch in *Planet PI4* wurde mit der Vorgänger-Implementierung bereits in Abschnitt 3.5 vorgestellt. Darüber hinaus existieren nur vereinzelt andere Projekte wie *Colyseus* [5] oder *Donnybrook* [6] die ebenfalls auf Bot-Nutzungen setzen. Sowohl *Colyseus* als auch *Donnybrook* benutzen *Quake 3* als Evaluationsumgebung und setzen dabei beide auf modifizierten Standard *Quake 3*-Bots zur Lasterzeugung. *Colyseus* verwendet zusätzlich dazu noch *Quake 2* und ein Eigenentwicklung, sowie entsprechende Bots zu Evaluationszwecken. In *Colyseus* verwenden die modifizierten *Quake 3*-Bots benutzen zur Fortbewegung ein *Obstacle-Sensitive Mobility Model* basierend auf Voronoi-Diagrammen. Die Ausführung mancher Aktionen, wie der Kampfentschluss oder die Ansteuerung von POIs, sind von Wahrscheinlichkeiten abhängig die aus echten Game-Traces ermittelt worden sind. Andere Aktionen hingegen basieren völlig auf den Entscheidungen des Bots, so z.B. das konkrete Kampfverhalten, welches Aspekte wie Ausweichbewegungen oder Zielen und Schießen miteinschließt.

Was für Modifikationen in *Donnybrook* eingesetzt werden ist nicht näher angegeben. Da es sich hierbei um ein Folgeprojekt von *Colyseus* handelt, kann davon ausgegangen werden, dass entweder dieselben oder ähnliche Bots zum Einsatz kommen. Erwähnenswert ist hingegen, dass in *Donnybrook* der Bot-Ansatz nicht ausschließlich zur Lasterzeugung oder Evaluationszwecken Verwendung findet. *Donnybrook* präsentiert mit sogenannten *interest sets* eine Alternative zu rein auf den Sichtradius beschränkten AOIs. Spieler die sich im *interest set* eines Peers befinden bilden eine echte Untermenge der aktuell sichtbaren Spieler und repräsentieren weiter die Spieler, dessen Aktionen aktuell die größte Aufmerksamkeit des durch den Peer vertretenen Spielers genießen. In Anlehnung an die Limitierung der menschlichen Aufmerksamkeit auf einige wenige unterschiedliche Aspekte gleichzeitig, wird die Anzahl der Spieler mit maximal fünf im *interest set* entsprechend klein gehalten. Die restlichen sichtbaren Spieler können derweil nicht völlig ignoriert werden, da ihre Aktionen nun einmal sichtbar sind und der Vollständigkeitshalber ebenfalls dargestellt werden müssen. Weil diese Aktionen allerdings nicht so sehr vom Spieler beachtet werden, kann an dieser Stelle eine höher auftretende Ungenauigkeit eher toleriert werden als bei den Spielern im *interest set* die sich im Fokus des Spielers befinden. Zur Erzielung einer weiteren Reduzierung der ausgetauschten Menge an Spielinformationen und damit die sichtbaren Spieler die nicht im *interest set* liegen ein für den Spieler konsistentes und flüssiges Verhalten aufweisen, werden leichtgewichtige Bots, sogenannte Doppelgängers, eingesetzt. Dabei wird mit Hilfe der bisherigen Informationen über das Verhalten des zu simulierenden Spielers, die wahrscheinlich als nächstes vom ihm auszuführenden Aktionen soweit bestimmt und stellvertretend ausgeführt, bis wieder neue Informationen über das tatsächliche Verhalten des Spielers vorliegen, die diese stellvertretend ausgeführten Aktionen entweder bestätigen oder revidieren, sodass etwaige Korrekturen vorgenommen werden müssen. Sei es zu Evaluationszwecken oder zur Verhaltenssimulierung in Anwesenheit unvollständiger Spielerinformationen, beiden Ansätze zur Bot-Benutzung haben eines gemeinsam: Die Realisierung möglichst realistischen menschlichen Verhaltens.

3.2 Halo's AI-Architektur

Halo ist eine von *Bungie* entwickelte und von *Microsoft* vertriebene überaus erfolgreiche FPS-Computerspieleserie. Der erste Teil erschien 2002 für die Videospielkonsole Xbox und 2003 für den PC. Dem ersten Teil folgten sehr bald zwei weitere offizielle Nachfolger und einige weitere Ableger, sowie eine Film-Adaption. Wie es in FPS-Spielen üblich ist, muss der Spieler in *Halo* eine Vielzahl unterschied-

licher Gegner bekämpfen, die sich in Verhalten, Ausrüstung und Aussehen voneinander unterscheiden. Ein wesentliches Spielelement von *Halo* ist die Benutzung unterschiedlicher Fahrzeugtypen im Kampf. Die Fahrzeug-Benutzung ist jedoch nicht Spieler exklusiv. Sowohl KI-Mitspieler, als auch die KI-Gegner beherrschen die Benutzung. Die Fahrzeuge können dabei auch gemeinsam bedient werden. Ein Beispiel hierfür wäre ein vom Spieler gesteuerter Buggy auf dessen Ladefläche ein KI-Kamerad am befestigten MG die gegnerischen Truppen ins Visier nimmt.

Halo und *Halo 2* liegt zur Bewältigung dieser Aufgaben dieselbe in Abbildung 6 dargestellte *Game AI*-Architektur zugrunde. Mit der vorgenommenen übergeordneten Vierteilung in *World*, *Perception*, *Motion* und *Actor* lässt sich leicht das in Abschnitt 2.3 vorgestellte Konzept eines rationalen Agenten wiedererkennen. Demnach nimmt der Agent bestimmte ungefilterte Wahrnehmungen (*Perceptions*) der Welt um sich herum wahr, stellt daraufhin einige Berechnungen zur Bestimmung der besten nächsten Aktion an (*Decision Making*) und führt abschließend dann diese über sein Motion-System bzw. seine Aktuatoren aus. Im Agenten selbst werden im ersten Schritt die rohen Wahrnehmungen über die Welt auf ihre Relevanz hin überprüft und in eine, dem Weltmodel des rationalen Agenten ähnliche, *Memory*-Komponente zwischengespeichert. Die *Memory*-Komponente enthält demnach für die anschließende Situationsanalyse nicht nur Informationen über den aktuellen Weltzustand (aktuelle Wahrnehmungen), sondern auch ausgewählte vergangene Informationen früherer Wahrnehmungen oder andere für den Kontext der Situationsanalyse benötigter Informationen. Hauptaufgabe der Situationsanalyse ist es gemäß der ihr zur Verfügung stehenden Informationen über die Welt und des aktuellen Emotionszustandes, welches hauptsächlich die Auswahl der verwendeten Dialognachrichten eines Bots beeinflusst, sogenannte Stimuli zu erzeugen. Stimuli stellen vereinfachend ausgedrückt eine Art Event dar, wie z.B. „Find Enemy“, „Search a friend“, „Damaged“ oder „Player fired!“, welche an die *Decision Making*-Komponente weitergereicht wird, damit diese eine zum Stimuli und zum Bot-Typ passende Aktion bestimmen kann, die als nächstes über das Motion-System ausgeführt werden soll [22, 9].

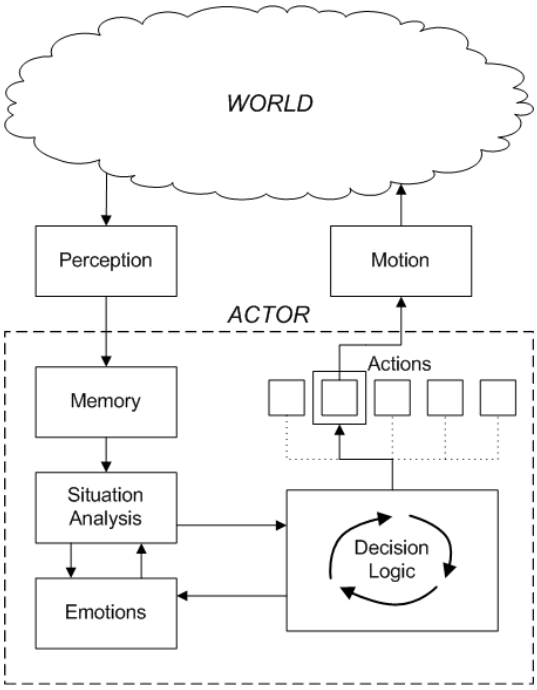

Abbildung 6: Die Game AI-Architektur von Halo. Entnommen aus [9].

Die konkrete *Decision Making*-Komponente ist der Hauptunterscheidungspunkt zwischen *Halo* und *Halo 2*. Während *Halo* auf einen Mix aus *Rule-Based System* und *Finite-State Machine* setzt, verwendet *Halo 2* eine erweiterte Form eines *Behavior Trees*. Ein *Rule-Based System* oder auch Expertensystem ist ein

auf einer Menge von Regeln, bestehend aus *if/then*-Statements, und einer Datenbank mit Informationen über die aktuelle Welt basierendes System, das über einen Matching-Algorithmus bestimmt welche der Regeln gemäß der der Datenbankinformationen aktuell am besten zutrifft. Jede Regel ist wiederum mit einer entsprechenden Aktion verknüpft die beim erfolgreichen *matchen* einer Regel ausgeführt werden soll. Ein FSM bzw. endlicher Automat ist im Grunde ein gerichteter Graph mit einer endlichen Anzahl an Zuständen und Transitionen. Er verhält sich somit ähnlich einem Baum, in dem Kanten in alle Richtungen und in beliebiger, aber endlicher, Zahl erlaubt sind. Ähnlich zum Wurzelknoten eines Baumes muss jeder Endlicher Automat einen Startzustand und, analog zu den Blättern eines Baumes, mindestens einen Endzustand auszeichnen. Zustände im Allgemeinen beschreiben auf abstrakte Weise das Verhalten des Bots. Die Transitionen ermöglichen Zustandsübergänge bzw. im Kontext des Bots Verhaltensänderungen über das Ausführen von mit dem jeweiligen Zustand assoziierten Aktionen. Das von *Halo 2* eingesetzte Konzept der *Behavior Trees* findest ebenfalls im eigenen Ansatz Verwendung, welcher im Rahmen dieser Master-Thesis entstanden ist, und wird dementsprechend in Kapitel 4 ausführlich behandelt.

3.3 GOAP - Goal-Oriented Action Planning

GOAP ist ein von Jeff Orking für den *First-Person Shooter F.E.A.R.* (First Encounter Assault Recon) aus dem Jahre 2005 entwickelter Planer. Die für Computerspiele damals fortschrittliche KI war einer der Gründe, warum *F.E.A.R.* u.a. von *Gamespy* als eins der Top-10 PC-Spiele des Jahres 2005 ausgezeichnet wurde [5]. Der GOAP-Planer ist zwar eine konkrete Implementierung die auf *F.E.A.R.* zugeschnitten wurde, doch veröffentliche Orking zahlreiche Papers zum allgemeinen Ansatz seines GOAP-Planers, sodass seitdem auf den GOAP-Prinzipien beruhende Planer erfolgreich in mehreren modernen Computerspielen wie *Fallout 3*, *Empire: Total War* oder *Silent Hill: Homecoming* eingesetzt werden konnten [37, S. 1].

GOAP ist eine vereinfachte, allerdings an manchen Stellen auf erweiterte [39, S. 10-13], Version des *STRIPS*[6] *Planning*-Systems der Stanford University aus dem Jahr 1970 [39, S. 4f]. Die Aufgabe eines STRIPS-Planers oder generell irgendeines Planers ist nach Russel [46, S. 465f] definiert als die Suche nach einer Aktionsfolge von einer Ausgangssituation zu einer gewünschten Zielsituation. Jeder Zustandsübergang repräsentiert dabei einen der möglichen Folgezustände, welcher durch Ausführung einer entsprechenden Aktion des Agenten in der virtuellen Welt erreicht werden kann. Der Pfad zum Zielknoten stellt dementsprechend die gesuchte Aktionsfolge vom Start- zum Zielzustand dar. Ruft man sich das in 2.3 vorgestellte Konzept eines rationalen Agenten wieder in Gedächtnis zurück, lässt sich GOAP bzw. irgendein *Planning*-System als möglicher Bestandteil der *Decision Making*-Komponente betrachten. Stellt es doch eine Möglichkeit der Auswahl der besten als nächstes auszuführenden Aktion dar. Mit einer auf der C4++-Architektur des MIT [22, 21] basierenden Erweiterung, die Orking in [38] vorstellt, findet der GOAP-Planer als wichtigster Bestandteil der *Decision Making*-Komponente eben jene Verwendung in einem zum Konzept eines rationalen Agenten vergleichbaren übergeordneten Ansatz.

Das Herzstück eines Planers ist dabei die verwendete formale Repräsentation zur Zustands- und Aktionsbeschreibung. Sie bestimmt Maßgeblich die Mächtigkeit des Planers, weil diese Art und Umfang der dem Planer zur Verfügung stehenden Informationen bestimmt, welche wiederum zur dynamischen Reduzierung des Suchraums genutzt werden können. Da diese eine Überprüfung irrelevanter Aktionen ermöglichen, deren Folgezustände mit hoher Wahrscheinlichkeit nicht zum Ziel führen und somit verworfen werden können. Eine Repräsentation von Planungsproblemen sollte die logische Struktur des Problems erfassen, damit weitestgehend auf zusätzliches Expertenwissen zur Suchraumverkleinerung verzichtet werden kann. Das Ziel ist es dabei eine Sprache zu finden die Ausdrucksstark genug ist, um

[5] Siehe für die Top-10 PC-Spiele Liste aus dem Jahr 2005: `http://goty.gamespy.com/2005/pc/index6.html`

[6] STRIPS steht für *Stanford Research Institute Problem Solver* und bezeichnete 1970 zunächst den Planer. Später jedoch etablierte sich der Begriff STRIPS als Bezeichnung für die im STRIPS-Planer eingesetzte Repräsentationsbeschreibung der Problemdomäne.

eine Vielzahl von Problemen abzudecken, aber dabei gleichzeitig einschränkend genug bleibt, um Lösungen effizient bestimmen zu können. Es existiert eine Menge an unterschiedlichen Repräsentationen, die sich anhand dieser beiden Dimensionen unterscheiden lassen. Die von GOAP verwendete Repräsentationsbeschreibung ist dabei eine abgewandelte Form des *STRIPS*-Planers und besteht nach Russel [46, S. 467f] aus folgenden Elementen:

- **Zustand:** Beschreibung eines Weltzustands anhand logischer Bedingungen in Form einer Konjunktion funktionsfreier Grundliterale. Es gilt weiter die Annahme der geschlossenen Welt, d.h. alle Bedingungen, die in einem Zustand nicht erwähnt werden, werden als falsch angenommen. Beispiel: $Arm \wedge Krank$ könnte den Zustand eines vom Glück verlassenen Agenten repräsentieren.

- **Ziel:** Ein partiell spezifizierter Zustand, der als Konjunktion positiver Grundliterale repräsentiert wird. Beispiel: Der Zustand: $Reich \wedge Berühmt \wedge Unglücklich$ erfüllt das Ziel: $Reich \wedge Berühmt$.

- **Aktion:** Eine Aktion oder Aktionsschema besteht aus folgenden drei Bestandteilen:
  - **Bezeichner:** Aktionsname und Parameterliste zur Identifizierung. Beispiel: *Fliegen(p, von, nach)*
  - **Vorbedingung(en):** Konjunktion funktionsfreier positiver Grundliterale, die angeben, was in einem Zustand gelten muss, damit die Aktion ausgeführt werden kann. Alle Literale müssen auch in der Parameterliste erscheinen.
  - **Effekt(e):** Konjunktion funktionsfreier Grundliterale, die angeben, wie sich Zustände verändern, nachdem die Aktion ausgeführt wurde. Alle Literale müssen auch in der Parameterliste erscheinen.

  Eine Aktion ist damit in jedem Zustand genau dann anwendbar, wenn alle Vorbedingungen dieser erfüllt sind. Durch die Anwendung einer Aktion wird der Zustand in einen Folgezustand unter Anwendung der Effekte überführt, wobei nicht erwähnte Literale unverändert bleiben (*STRIPS*-Annahme). Wenn die Vorbedingungen einer Aktion nicht erfüllt sind, kann diese auch nicht angewendet werden und somit auch keine Überführung in einen Folgezustand bewirken.

Mit der eben spezifizierten STRIPS-Repräsentation ist es nun möglich das Ausgangsproblem, das Finden einer Aktionsfolge von einem Startzustand zu einem Zielzustand, angemessen zu beschreiben und zu lösen. Dabei handelt es sich beim Zielzustand nicht ausdrücklich um ein konkretes Ziel, sondern um einen Weltzustand der das jeweils aktuelle Ziel erfüllt. Beim GOAP-Planer handelt es sich um einen sogenannten regressiven Zustandsraumplaner. Vereinfachend ausgedrückt kann man sich Planen im Zustandsraum als ein Wege- bzw. Navigationsproblem in einem Graphen vorstellen, bei dem jeder Knoten einen vom Startzustand tatsächlich erreichbaren Weltzustand darstellt und jede Kante die entsprechende Aktion zur Zustandsüberführung. Aufgrund der bereitgestellten Informationen über das Planungsproblem ist es nun möglich in jede Richtung zu suchen bzw. zu planen. Die Rückwärts-Zustandssuche oder Regressionsplanung sucht dabei in die entgegengesetzte Richtung, d.h. vom einem zielerfüllenden Endzustand aus den aktuellen Startzustand.

Der dabei verwendete Suchalgorithmus ist eine A*-Suche, die in Computerspielen häufig im Bereich des Pathfindings eingesetzt wird. Aufgrund der gleichartigen Problemlösungsdomäne kann die A*-Suche jedoch hier ebenfalls verwendet werden. Die Kostenfunktion entspricht dabei nicht der Streckendistanz von einem Knoten zum anderen, sondern berechnet sich aus den Kosten der Ausführung einer Aktion. Da sowohl Ziele durch einfache Literale beschrieben werden, als auch die durch einen Knoten repräsentierten Weltzustände, berechnet die Heuristik-Funktion einfach die Menge der unerfüllten Literale von einem Zustand zum Ziel aus. Zusammengefasst sucht der GOAP-Planer mit der A*-Suche die Aktionsfolge die ausgehend vom Zielzustand den aktuellen Startzustand erreicht und dabei die geringsten Aktionskosten aufweist [37, S. 4-8].

Ein Beispiel soll den eben vorgestellten *Planning*-Prozess von GOAP zusammenfassend verdeutlichen. Dazu stelle man sich folgendes Szenario mit dem Ziel *KillTargetEnemy* vor, bestehend aus einer einzigen Bedingung: *kTargetIsDead = true* und den aktuellen Weltzustand mit den drei Bedingungen: *kCoverIsUsed=false* $\wedge$ *kTargetIsDead=false* $\wedge$ *kWeaponIsLoaded=false* vor. Erster vor der eigentlichen Planung auszuführender Schritt ist die Auswahl des zu verfolgenden Ziels. In GOAP werden Ziele nach einer nummerischen Priorität sortiert. Da in dem Beispiel nur ein Ziel existiert wird dieses als aktuelles Ziel ausgewählt. Nächster Schritt ist es einen möglichen Weltzustand zu finden der das ausgewählte Ziel erfüllt. In Abbildung 7 ist der oberste Zustand, beschrieben durch die Bedingungen *kCoverIsUsed=false* $\wedge$ *kTargetIsDead=true* $\wedge$ *kWeaponIsLoaded=true* ein solch möglicher Zustand. Ausgehend von diesem Zielzustand, der nun den Startzustand der Suche darstellt, wird dann im nächsten Schritt regressiv nach einer Aktionsfolge gesucht die zum aktuellen Weltzustand führt, dem Zielzustand der Suche. Dazu werden alle Aktionen betrachtet die zum aktuellen Suchzustand, im ersten Schritt der Zustand der das Ziel erfüllt, führen. Existiert keine direkte Aktion die zum aktuellen Weltzustand, dem Zielzustand, führt, müssen nach und nach die Suchzustände untersucht werden die eine Mindest-Annäherung von jeweils einer übereinstimmenden Bedingung an die Bedingungsmenge des Zielzustandes bedeuten. Dabei dürfen im jeweiligen Suchzustand nur relevante und konsistente Aktionen berücksichtigt werden, was einen viel kleineren Verzweigungsfaktor als bei der Vorwärtssuche ergibt. Eine Aktion ist genau dann relevant, wenn sie mindestens eine Bedingung des Ziels “wahr“ macht, die nicht bereits im Vorgängerzustand als “wahr“ galt. Eine Aktion ist genau dann konsistent, wenn durch ihre Ausführung keine anderen Ziele rückgängig gemacht werden. In der Abbildung ist die auf diese Weise gefundene Aktionskette, erst Waffe laden dann schießen, rot hervorgehoben und der aktuelle Weltzustand dementsprechend der Rechte der beiden unteren Zustände.

Nach Orkin [37, S. 4ff] [39, S. 7-10] würden sich durch die in GOAP vorgenommene Trennung von Aktionen und Zielen in Verbindung mit dem dynamischen Lösungsprozess der Planung eine Menge an Vorteilen gegenüber anderen damals etablierten *Decision Making*-Konzepten ergeben. So würde eine Erleichterung der Integration neuer Spielelemente existieren, ohne der sofortigen Gefahr eines Zusammenbrechen des Systems, da neue Aktions- oder Zieldefinitionen parallel zu den bisherigen Elementen hinzugefügt werden können. Weiter könnten durch den Planungsprozess neue, vom Designer nicht berücksichtigte, Lösungen gefunden werden. Dies führt zu einem gesenkten Bedarf an benötigter Spezifikation, zu einer besseren Wartbarkeit, Modularität und Skalierbarkeit. Ein weiterer Vorteil ist die gestiegene Variationsfreiheit, die u.a. dazu eingesetzt werden kann um leicht unterschiedliches Verhalten zu definieren. Orkin schlägt dazu die Variation der für einen NPC zur Verfügung stehenden Aktionen vor und/oder die Variation der jeweils zugewiesenen Zielmenge.

Die Nachteile beim Einsatz von GOAP sind auf der anderen Seite die Beschränktheit oder Komplexität der Repräsentationssprache, die Komplexität bzw. der Berechnungsaufwand des Planungsprozesses und die mangelnde Kontrolle über das Verhalten der NPCs.

Der GOAP-Planer verwendet mit STRIPS eine sehr einfache Repräsentationssprache. Dies macht den Planer entsprechend effizient, allerdings können damit nur sehr einfache Problemlösungsdomänen beschrieben werden. Komplexere Weltbeschreibungen die nicht alleine oder nur sehr umständlich mittels Literale beschrieben werden können, führen entweder zu Performance-Einbrüchen, weil die Anzahl an Literale explodiert, oder die Problemdomäne einfach nicht abgebildet werden kann. Zwei typische Probleme von STRIPS sind z.B. das Ramifikationsproblem und das Qualifikationsproblem. Das Ramifikationsproblem bezeichnet die Schwierigkeit der Repräsentation impliziter Effekte wie z.B. dem Effekt, dass sich Passagiere innerhalb des Flugzeuges, welches von A nach B unterwegs ist, ebenfalls mit diesem (indirekt) fortbewegen. Das Qualifikationsproblem bezeichnet die problematische Erfassung aller Umstände die zum Scheitern einer Aktion führen können. Beide Probleme können unter Anwendung der *STRIPS*-Repräsentation nur unzureichend oder gar nicht gelöst werden. Das Problem von anderen mäch-

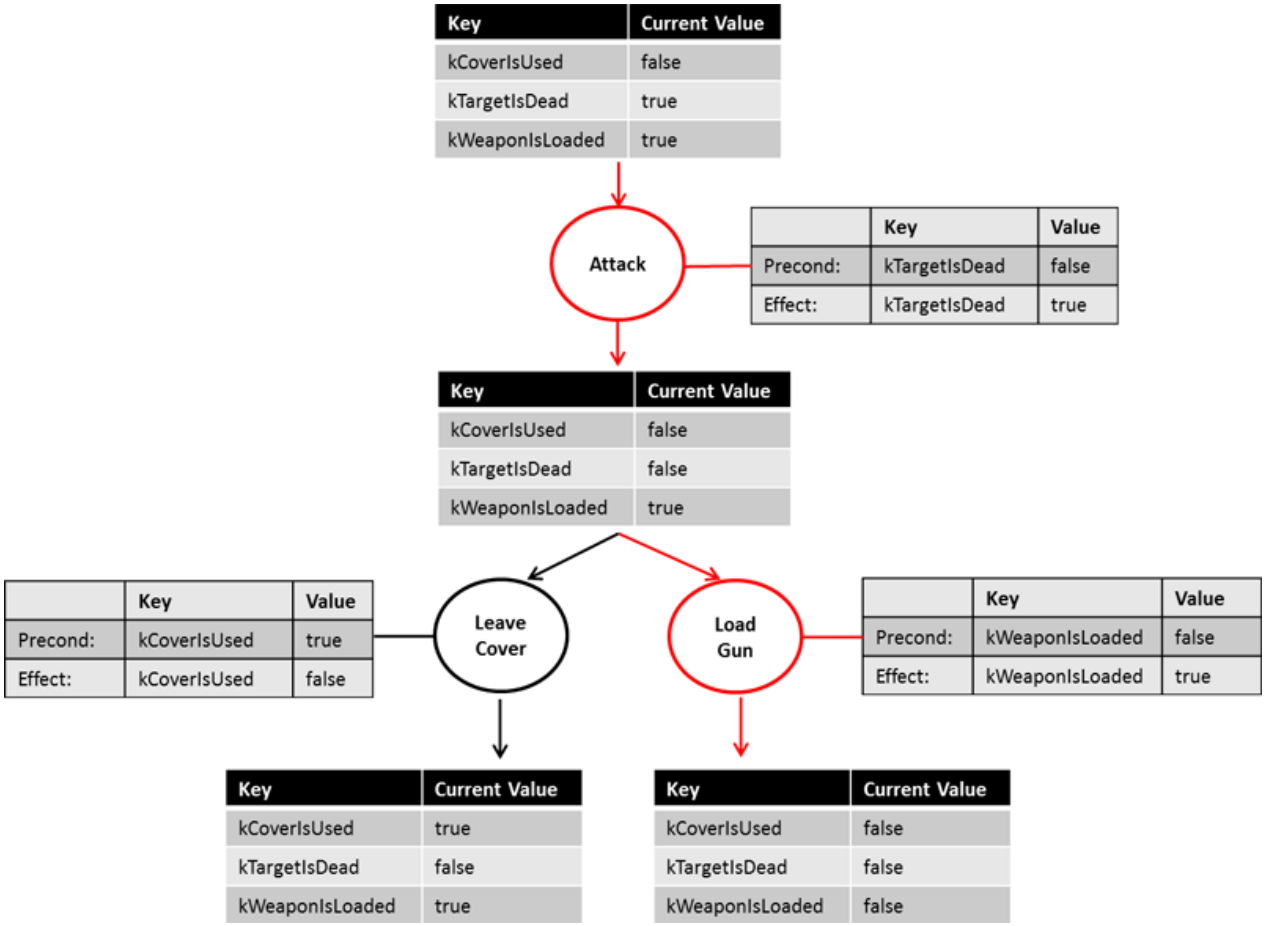

Abbildung 7: Ein Beispiel für den von GOAP realisierten Entscheidungsprozess zur Auswahl der nächsten Aktion. Beachtet werden sollte das die Planungsrichtung (Roter Pfeil) entgegengesetzt zur Ausführungsrichtung ist. Angelehnt an Beispiel und eine Grafik aus [37].

tigeren Repräsentationssprachen ist, dass sie sehr viel komplexer sind und dies direkten Einfluss auf die Performance des Planungsprozesses hat.

Der Planungsprozess als solches ist ebenfalls als komplex anzusehen, was sich negativ auf die Laufzeit auswirkt. Selbst mit der von Millington in [35, S. 418-425] vorgeschlagenen Verbesserung, die sich durch den Austausch des A*-Algorithmus durch den IDA*-Algorithmus ergibt, liegt die Rechenlaufzeit von GOAP in $O(mn^d)$, wobei m die Anzahl der Ziele, n die Anzahl der möglichen Aktionen und d die maximale Suchtiefe bezeichnet. In stark dynamischen Welten in denen schnelle Reaktionen aufgrund sich stetig und rapide ändernder Situationen erforderlich sind, wie Beispielsweise in *Planet PI4*, könnten die anfallenden Kosten für Planung bzw. ständiger Neu-Planung fatale Folgen für die Performance und Skalierbarkeit haben.

Die mangelnde Kontrolle über das generierte Verhalten der NPCs mag zwar zu den bereits erwähnten Vorteilen wie Wartbarkeit, Skalierbarkeit oder generell zu einer gestiegenen Mächtigkeit führen, allerdings stellt es im Kontext von reproduzierbaren und konfigurierbaren Verhaltensweisen einen entscheidenden Nachteil dar. Denn man könnte allenfalls einen indirekten Einfluss auf den Aktions-Auswahlprozess ausüben und sich somit wieder ein Stück weit vom Glück abhängig machen, ob am Ende wirklich das gewünschte bzw. aktuell zu testende Verhalten tatsächlich realisiert wird oder eben doch vom Planer ein anderes Verhalten als der Situation angemessener angesehen wird.

3.4 Verfahren zur Kombinierung von Steering Behaviors

Ein wichtiger Punkt in der *Steering*-Ebene ist die Tatsache das *Steering Behaviors* in erster Linie nur Bewegungsvorschläge darstellen und erst die Evaluation, Auswahl oder Kombinierung dieser zum konkreten *Steering*-Ergebnis führt. Ein Problem dabei ist das bisherige Ansätze zur Lösung dieses Problems vereinzelt zu suboptimalen, ungewünschten oder schlichtweg katastrophalen Verhaltensweisen des Agenten führen können, wie beispielsweise das Vorhandensein eines stabilen Equilibriums bzw. Gleichgewichts, was zum völligen Stillstand des Agenten führt und nicht vom Agenten alleine wieder aufgelöst werden kann. In Abbildung 9 ist ein Beispiel für ein solches stabiles Equilibrium dargestellt, auf das allerdings in Kürze näher eingegangen wird.

Nach Millington und Funge in [35, S. 95-108] lassen sich die vorhandenen Ansätze zur Kombinierung von *Steering Behaviors* in *Blending* und *Arbitration*, sowie in Mischformen dieser unterscheiden. *Blending*-Ansätze führen alle relevanten *Steering Behaviors* separat aus und kombinieren in Anschluss daran die Ergebnisse unter Einbeziehung von Gewichten, Prioritäten oder anderer Bewertungskriterien zu einem Endergebnis. Die einfachste Form eines *Blending*-Ansatzes ist es die Ergebnisse anhand vorher definierter Gewichte für jedes *Steering Behavior* zusammenzurechnen. Dies bedeutet im Umkehrschluss, dass in der Anwesenheit mehrerer *Steering Behaviors* niemals ein *Steering Behavior* vollständig alleine ausgeführt wird, sondern immer ein Kompromiss aus mehreren *Steering Behaviors* eingegangen wird. Das größte Problem von reinen *Blending*-Ansätzen ist das zuvor schon erwähnte Problem des Eintretens stabiler oder instabiler Equilibrien oder generell das Problem des Hängenbleibens, vor allem in komplexeren oder einengenderen (Innen-)Arealen.

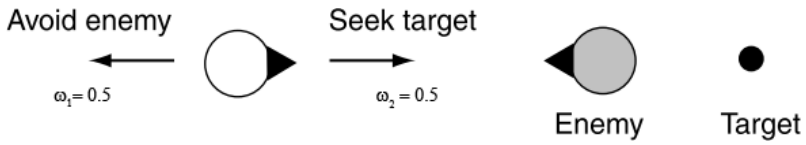

Abbildung 8: Ein instabiles Equilibrium. Angelehnt an eine Grafik aus [35, S. 100].

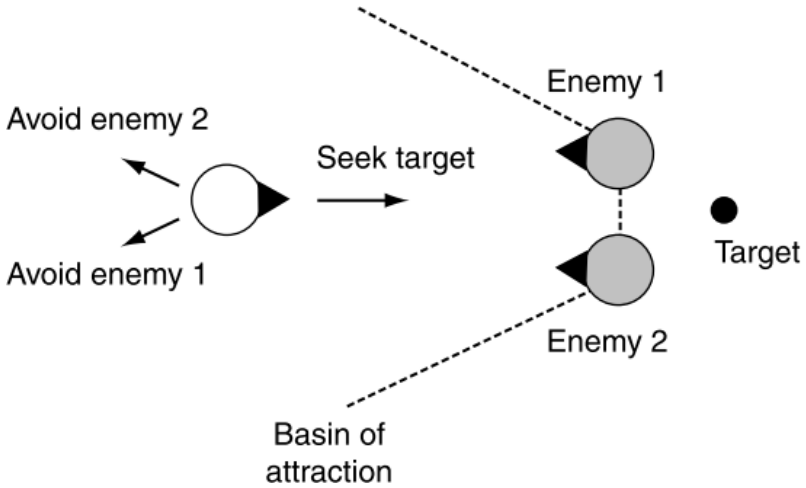

Abbildung 9: Ein stabiles Equilibrium. Entnommen aus [35, S. 101].

Abbildung 8 zeigt ein Beispiel für das Problem eines instabilen und Abbildung 9 für das eines stabilen Equilibriums. Beide treten genau dann auf, wenn *Steering Behaviors* in Konflikt zueinanderander stehende *Steering*-Ziele vorschlagen. In Abbildung 8 wird dieser Konflikt dargestellt, indem der Vorschlag zum Erreichen des Ziels dem genauen Gegenteil bzw. dem invertierten *Steering*-Vektor des vom Flucht-*Behavior* präsentierten Vorschlags entspricht. Wenn wie im Beispiel dargestellt die Gewichte w_i in diesem

Fall noch gleich sind, ist die Folge das die beiden *Steering*-Vektoren sich gegenseitig aufheben und der Agent somit zum Stillstand gelangt. Diesen Fall nennt man deswegen instabil, weil durch nummerische Ungenauigkeiten oder andere Faktoren der Agent selbst dazu in der Lage ist aus diesem Gleichgewicht zu entkommen und somit doch noch ein annehmbares Verhalten zu präsentieren.

Ein stabiles Equilibrium hingegen ist in bestimmten Situation, wie beispielsweise der in Abbildung 9 dargestellen, ein ernsthafteres Problem, weil der Agent sich eben nicht selbst aus dem Gleichgewicht befreien kann. Im abgebildeten Beispiel würde der Agent zwar vlt. ebenfalls aufgrund nummerischer Ungenauigkeiten sich langsam innerhalb des Sichtradius der beiden Feinde bewegen können, doch würde er sich aufgrund der sich quasi gegenseitig aufhebenden Vorschläge nur sehr langsam und nicht konsequent genug in eine Richtung fortbewegen um solch einem großen Flucht-Radius der Feinde effektiv entkommen zu können. Das Ergebnis ist ein Festsitzen des Agenten der sich nicht zwischen Zielerreichung und Flucht entscheiden kann.

Das *Inverse Steering Behavior* [3, 4] Verfahren der Universität Koblenz-Landau ist ein Vertreter dieses *Blending*-Ansatzes, beweist allerdings mit der Beispiel-Implementierung eines dribbelnden Fußball spielenden Roboter Agenten, dass mit der Verwendung kostenbasierender Metriken, statt einfacher experimentell bestimmter Gewichte die meisten zuvor erwähnten Nachteile aus den Weg geräumt werden können. Wie die Bezeichnung „Inverse“ im Namen bereits vermuten lässt, wird dabei eine im Vergleich zum *Steering*-Ansatz von Reynolds invertierte Herangehensweise bei der Berechnung des *Steerings*-Outputs angewandt. Statt einer separaten Ausführung aller *Steering Behaviors* im ersten Schritt und anschließender Kompromissbildung aus den Ergebnissen, wird zu aller Erst je nach Situation eine Menge an möglichen *Steerings* bestimmt, um anschließend von den *Steering Behaviors* separat anhand einer individuellen Kostenmetrik bewertet zu werden. Die *Steering*-Vorgabe welches die geringsten Kosten über die Menge aller relevanten Bewertungen der *Steering Behaviors* aufweist, stellt nach dem *Inverse Steering Behavior*-Verfahren die ideale Kompromisslösung zur Bewältigung der aktuellen Situation dar. Dabei ist die Kompromisslösung nicht ein Berechnungskompromiss wie bei einfachen *Blending*-Ansätzen, wobei die Kompromisslösung im schlimmsten Fall keines der *Steering Behavior*-Ziele erreicht, sondern ein Auswahlkompromiss zwischen mehreren plausiblen Lösungen.

Stellt man sich die in Abbildung 10 dargestellte Situation vor, in der ein Ziel von zwei nahen statischen Objekten verdeckt wird, so wird im *Inverse Steering Behavior*-Verfahren in etwa wie folgt vorgegangen: Im ersten Schritt wird realisiert, dass in naher Zukunft bei Beibehaltung der aktuellen Bewegungsrichtung eine Kollision mit einem statischen Hindernis stattfinden würde. Gesucht wird nun eine Richtungsänderung bzw. ein neuer *Steering*-Vektor, dessen Ausführung die zukünftige Kollision noch abwenden kann. Dazu werden je nach Situation oder Umgebung, auf eine nicht näher bestimmte Weise, eine unterschiedliche Menge an möglichen Lösungsvorschlägen bzw. *Steering*-Vektoren ermittelt. Im zweiten Schritt werden diese *Steering*-Vorschläge von allen relevanten *Steering Behaviors* separat untersucht. Bei der Untersuchung wird jeweils eine eigene für das *Steering Behavior* repräsentative Kostenfunktion als Bewertungsgrundlage herangezogen. Die Idee dahinter ist, dass die einzelnen *Steering Behaviors* nicht mehr für die Berechnung eines *Steering*-Vektors zuständig sind die ihre Absicht, wie z.B. das Erreichen eines Ziels oder das Ausweichen vor statischen Hindernissen, am besten erfüllt, sondern eine Kostenbewertung vornehmen, die diejenigen *Steering*-Vektoren mit den geringsten Kosten bewertet die ihre Absicht am besten zu erfüllen scheint.

Im Beispiel könnte das *Seek Behavior* nach [3] eine Bewertung anhand der graduellen Abweichung zur Luftlinie zum Ziel vornehmen. Mit dieser Funktion würden somit die Vorschläge θ_1 und θ_4 mit 80° höhere Kosten verursachen als θ_2 und θ_3 mit 40°. Das *Obstacle Avoidance Behavior* würde hingegen das Vorhandensein von statischen Hindernissen entlang eines *Steering*-Vektors untersuchen. Auf das Bespiel in der Abbildung übertragen würde das bedeuten, dass θ_2 Kosten von 1 und der Restkosten

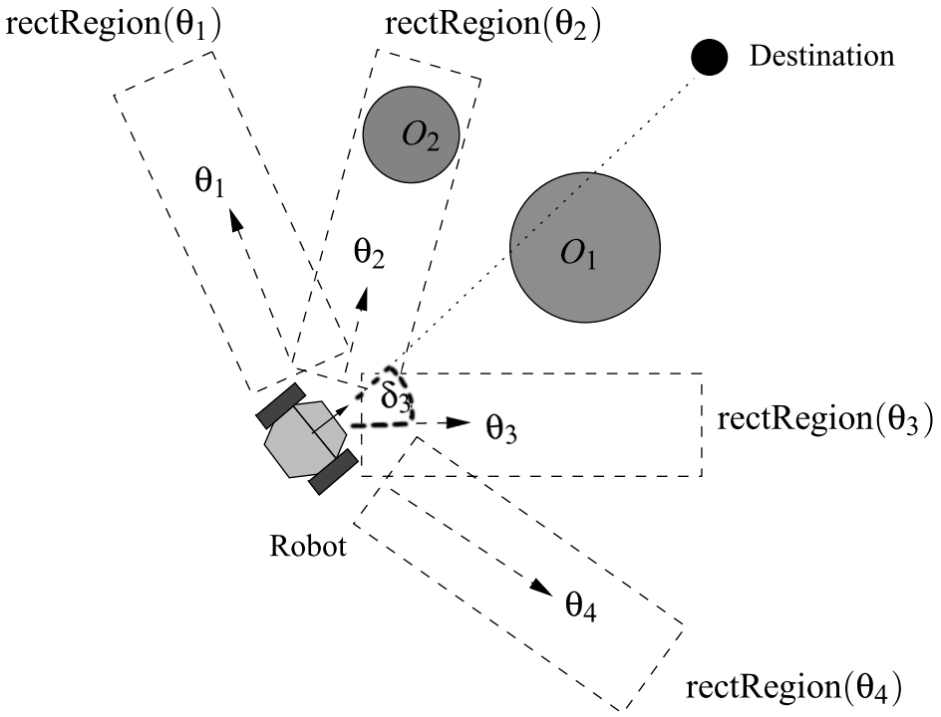

Abbildung 10: Die Evaluierung der möglichen Ausweichbewegungen zur Verhinderung der Kollision mit Hindernissen. Angelehnt an zwei Grafiken aus [3].

von 0 verursachen würden. Haben sämtliche *Steering Behaviors* alle *Steering*-Vorschläge aus Schritt eins bewertet, erfolgt im nächsten Schritt die Berechnung aller Kosten eines Vorschlag mittels eines *Blending*-Verfahrens, wobei die einfachste Form die direkte Aufsummierung ohne Gewichte wäre. Im finalen Schritt würde dann der *Steering*-Vorschlag ausgewählt werden, der über die geringsten Kosten verfügt. Mit der direkten Aufsummierung und der Normierung aller Kostenfunktionen auf ein Intervall von [0,1] würde im Beispiel θ_3 mit 0.5 die geringsten Kosten aufweisen, da es mit θ_2 die geringste Winkelabweichung aufweist und somit die geringsten *seek*-Kosten von 0.5, aber im Gegensatz zu θ_2 ebenfalls die geringste Kostenbewertung nach *obstacle avoidance* mit Kosten = 0 erhält. Tabelle 2 enthält nochmals alle Kostenbewertungen, sowie die abschließende Kostenaufsummierung pro *Steering*-Vorschlag.

	θ_1	θ_2	θ_3	θ_4
$\mathbf{C}^{Seek}$	80	40	40	80
$\mathbf{C}^{OA}$	0	1	0	0
$\mathbf{h}(\mathbf{C}^{Seek}, \mathbf{C}^{OA})$	1	1.5	0.5	1

Tabelle 2: Kostenbewertung der *Steering*-Vorschläge sowie ihre Kostensumme.

Der größte Vorteil bei der Verwendung der Kostenfunktion ist die relativ einfache Einbeziehung von zusätzlichen *high level*-Wissen bei der Bewertung der *Steering*-Vorschläge. Die *Steering*-Ebene ist somit nicht wie im traditionellen Ansatz von Reynolds ausschließlich auf lokale Informationen über die Umgebung beschränkt. Als mögliches *high level*-Beispiel fürs *Inverse Steering Behavior*-Verfahren wird in [4, S. 226f.] die Einbeziehung des Wissen einer Influence Map aufgeführt. Eine *Influence Map* ist eine Datenstruktur die regionale Informationen über das aktuelle Gegner- und Verbündeten Verhältnis führt. Auf das eben aufgeführte Beispiel übertragen würde das bedeuten, dass es eine dritte Kostenfunktion geben könnte, welche die *Steering*-Vorschläge in Hinblick darauf bewertet, ob diese zu mehr oder weniger stark vom Gegner kontrollierten Gebieten führen oder nicht. Wie die in [3, 4] vorgestellte Umsetzung eines dribbelnden Fußball spielenden Roboter-Agenten zeigt, scheint das *Inverse Steering Behavior* Verfahren

fähig zu sein auch schwierige Situationen erfolgreich zu meistern ohne dabei in bekannte Fallen zu tappen, wie z.B. das Hängenbleiben aufgrund stabiler Gleichgewichte.

Der Nachteil der gewonnenen Mächtigkeit ist allerdings auf der anderen Seite der gestiegene Berechnungsaufwand. Es müssen einerseits mehrere situationsabhängige und allen voran plausible *Steering*-Vorschläge bestimmt werden, andererseits beliebig komplexe und beliebig viele Kostenberechnungsfunktionen für jeden einzelnen Vorschlag ausgeführt werden. Gerade wenn man nach Reynolds die *Steering*-Ebene als die „non-deliberative"-Ebene auffasst, die in der Regel in jedem Entscheidungszyklus eines Agenten ausgeführt werden muss, und dementsprechend performant zu sein hat, stellt der erhöhte Berechnungsaufwand einen nicht zu vernachlässigten Umstand dar.

Eine Alternative zu *Blending*-Ansätzen stellen *Arbitration*-Verfahren dar, die im Grunde die Gewichte durch Prioritäten ersetzen. Die Vorgabe einer Reihenfolge bei der Ausführung von *Steering Behaviors* verhindert die Problematik der Kombination widersprüchlicher *Steering Behavior*-Ergebnisse, was das größte Problem von *Blending*-Verfahren darstellt. Umgangen wird das Problem dadurch, dass immer nur das Ergebnis eines Einzigen oder einer sehr kleinen Menge von *Steering Behaviors* zu einem bestimmten Zeitpunkt Verwendung findet. Ermöglicht wird dies indem *Steering Behaviors* der Reihe nach ausgeführt werden bis das Erste sich signifikant von „Null" unterscheidende Ergebnis ermittelt wird, wobei ein *Null*-Ergebnis eines *Steering Behaviors* etwas lapidar ausgedrückt bedeutet: Aktuellen Kurs halten! Eine Alternative dazu ist es die Ausführung solange Fortzusetzen bis der erste Konflikt auftritt. Es wäre beispielsweise denkbar, dass ein *Steering Behavior* nur den Torso eines Körpers steuern möchte und die momentane Bewegungsrichtung unangetastet lässt, sodass diese im Anschluss daran noch von einem Nachfolger gesetzt werden könnte. Ein mögliches *Arbitration*-Verfahren ist das in Abschnitt 4 vorgestellte Konzept einer *Steering Pipeline*.

Der Nachteil des Einsatzes einer solchen Priorisierung ist auf der anderen Seite, dass durch eine zu strikte oder starre Vorgabe der Reihenfolge, nicht unbedingt alle aktuell zu Berücksichtigten *Steering Behaviors* auch tatsächlich ausgeführt und im Endergebnis beachtet werden. Stelle man sich beispielsweise die Situation vor in der nicht nur statischen Hindernissen, sondern auch beweglichen Hindernissen ausgewichen werden muss, so würde die ausschließliche Beachtung einer Hindernisart unter Umständen nicht die Kollision mit der anderen Art verhindern können. Aus diesem Grund wird in [35, S. 95-108] die These aufgestellt, dass nur eine Kombination aus *Blending* und *Arbitration* zu einem wirklich zufriedenstellenden Ergebnis führt. Als Beispiel einer solchen Kombination wird ein System vorgestellt, welches *Steering Behaviors* in mehrere Gruppen einteilt, wobei die Ausführung der Gruppen einer festen Reihenfolge folgt (*Arbitration*). Innerhalb der Gruppe werden alle *Steering Behaviors* ausgeführt und das Ergebnis mittels *Blending* zusammengeführt. Eine mögliche Aufteilung der *Steering Behaviors* wäre in eine *collision avoidance*-, *group behavior*- und *single behavior*-Gruppe. Die erste Gruppe wäre für das Ausweichen vor statischen und beweglichen Hindernissen zuständig. Die zweite Gruppe für die Einhaltung von Gruppenbewegungen, wie z.B. die Verhinderung des Zusammenpralls (*Seperation*) oder des Auseinanderdriftens (*Cohesion*) einzelner Vögel inmitten eines Schwarmflugs. Letzt genannte Gruppe würde alle *Steering Behaviors* umfassen die sich um die Verfolgung separater Ziele des Agenten kümmern, z.B. das Erreichen eines bestimmten Punktes. Ein weiteres Beispiel für ein kombiniertes System wäre das von Reynolds getaufte *prioritized acceleration allocation system* [45]. Hier werden alle *Steering Behaviors* in eine feste und durch die *condition checking*-Ausführung bedingte variable Reihenfolge gebracht. Anschließend werden ähnlich dem *Arbitration*-Ansatz so lange Ergebnisse gesammelt, bis eine Grenze überschritten wurde. Die Grenze wird hierbei nicht durch das Auftreten konkurrierender Ergebnisse bestimmt, sondern stellt eine für jeden Agenten festlegbare Beschleunigungsgrenze dar, die durch das akkumulieren der einzelnen *Steering Behaviors* erreicht werden kann.

Mit kombinierten Ansätzen können die Vorteile beider Ansätze ausgenutzt werden und gleichzeitig die Nachteile entsprechend abgeschwächt werden, auch wenn diese nie ganz oder nur mit erheblichen Mehraufwand wie beim *Inverse Steering Behavior*-Verfahren ausgemerzt werden können. So bleibt festzuhalten, dass es in diesem Gebiet nicht die eine richtige Lösung gibt, sondern je nach konkretem Einsatzgebiet und Anforderungen ein bestimmter Ansatz ausgewählt werden sollte, der am besten für die jeweilige Situation zu passen scheint. Mit dem in Kapitel 4 vorgestellten Konzept einer *Steering Pipeline* wird zwar ein vorrangig dem *Arbitration*-Ansatz zugeordnetes Verfahren vorgestellt, das allerdings im Gegensatz zu anderen Ansätzen einen erheblichen Vorteil bietet, auch gegenüber kombinierten Ansätzen. Es ermöglicht, dass *Steering Behaviors* im Konfliktfall ihre für sich gewonnenen Informationen gegenseitig austauschen können, um auf dieser Grundlage weitere Berechnungen ausführen zu können.

3.5 Vorgänger-Implementierung: DragonBot

Der Ansatz der kontext-sensitiven AI-Spieler wurde bereits in *Planet PI4* in einer ersten Version umgesetzt. Dimitri Wulffert hat im Rahmen seiner Bachelorarbeit [59] zwei unterschiedliche Bot-Implementierungen entwickelt, welche bereits in ersten Tests erfolgreich eingesetzt werden konnten. Es gelang u.a. das Erreichen einer verbesserten Genauigkeit bzw. des Realitätsgrads der generierten Netzwerklast in Hinblick auf die Anzahl der Nachbarn. Die Anzahl der Nachbarn ist eine von mehreren, der noch in Entwicklung befindlichen, Metriken (zweite Komponente) der Benchmarking-Methodik [30]. Erste größer angelegte Tests mit bis zu 500 Spielern, zeugen auch von einer entsprechend guten Performance als wichtiger Faktor des Skalierbarkeits-Kriteriums.

Allerdings bestünde keine Notwendigkeit für die vorliegende Master-Thesis, wenn die bisherigen zwei Bot-Implementierungen alle eben vorgestellten Kriterien in ausreichender Form erfüllen würden. Nach eingehender Analyse beider Bot-Implementierungen wurden hinsichtlich der Erfüllung der aufgestellten vier Kriterien aus 1.2 folgende Mängel aufgedeckt bzw. Verbesserungsmöglichkeiten ausgemacht:

- **Reproduzierbarkeit:** Beide Implementierungen erzeugen weitesgehend deterministisches und somit reproduzierbares Verhalten. Jedoch weisen einige Testdurchläufe Diskrepanzen auf, die auf eine fehlerhafte oder fehlende Verwendung des zur Verfügung gestellten Zufallszahlengenerators hindeuten.

- **Skalierbarkeit:** Die Performance der Bot-Implementierungen ist gut. Es fehlen allerdings sämtliche Konfigurationsmöglichkeiten, d.h. die Bot-Implementierungen erzeugen in etwa immer dieselbe Last bei stets gleichbleibendem Realismus-Niveau.

- **Realitätsgrad:** Erste Tests lassen zwar bereits einen guten Realitätsgrad vermuten, doch scheint gerade in diesem Bereich noch ein großer Verbesserungsspielraum zu bestehen. So werden Aspekte wie *Shoot-*, *Hit-* oder *Death*-Rate noch nicht genau genug abgedeckt, siehe dazu auch Abschnitt 6. Die Komplexität beider Bot-Implementierungen hält sich dabei ebenfalls sehr in Grenzen und konzentriert sich dementsprechend ausschließlich auf einige Kernprozesse der Entscheidungsfindung (Decision Making). Rollenverhaltens- oder Gruppenverhaltensweisen werden nur sehr rudimentär oder gar nicht umgesetzt. Ersteres kann beispielsweise zur Auslösung unterschiedlicher Verhaltensweisen unter gleichen Bedingungen eingesetzt werden und letzteres stellt gerade in MMOGs einen wichtigen Aspekt dar und bedarf einer weiter gehenden Berücksichtigung.

- **Konfigurierbarkeit:** Bei der Lasterzeugung ist das jeweilige Spielerverhalten der ausschlaggebende Faktor. Es existieren zwar beim Realitätsgrad, anders als bei der Skalierbarkeit, einige Möglichkeiten der Variierung des Bot-Verhaltens, doch ermöglichen diese nur eine indirekte und eine auf das Verhalten und somit letzen Endes auf die zu generierende Netzwerklast kaum nachvollziehbare Konfigurationsmöglichkeit.

Wie bereits am Ende von Abschnitt 1.2 angedeutet, ist ein Hauptkritikpunkt der geleisteten Vorarbeit das Fehlen einer zugrundeliegenden *Game AI*-Architektur, welche sich Besonders an der Umsetzung beider Bot-Implementierungen in einer jeweils völlig separaten Klassenhierarchie ausmachen lässt. Wie später in Abschnitt 3 behandelt, gibt es bei der Entwicklung einer *Game AI* zahlreiche unterschiedliche Lösungsansätze, die ebenfalls ein breites Spektrum an unterschiedlichen Implementierungsmöglichkeiten zulassen. Dennoch gibt es gewisse Aspekte die für jede *Game AI* bzw. Bot-Implementierung eines Computerspiels gelten, siehe dazu Abschnitt 2, insbesondere wenn es um denselben konkreten Einsatzzweck geht.

Die teilweise nicht ausreichende Erfüllung der aufgestellten vier Kriterien, sowie der Umstand einer fehlenden *Game AI*-Architektur, die eine effektive und erfolgversprechende Weiterentwicklung entscheidend erschwert, legt eine vollständige Neu-Konzeptionierung, Entwicklung und Umsetzung des Ansatzes der kontext-sensitiven AI-Spieler für *Planet PI4* nahe und soll im Rahmen dieser Master-Thesis entsprechend realisiert werden.

4 Konzept und Rahmenbedingungen der Game AI

Das Kapitel beschreibt das entwickelte allgemeine *Game AI*-Konzept zur Lösung der in der Einleitung definierten Zielvorgaben zur synthetischen Generierung einer realistischen Netzwerklast unter Einsatz kontext-sensitiver AI-Spieler.

Abschnitt 4.1 „Einsatzumgebung Planet PI4“ beschreibt dabei zuerst die konkrete Einsatzumgebung der *Game AI*. Die Beschreibungen zum Spiel sind für das weitere Verständnis der in Abschnitt 4.2 „Funktionale Anforderungen der Game AI“ präsentierten, in Hinblick auf die Fähigkeit zum Spielen des Spiels, konkretisierten Anforderungen und des in Abschnitt 4.3 „Konzept der Game AI“ vorgestellten groben Lösungsansatzes notwendig.

Der abschließende Abschnitt 4.4 „Behavior Tree“ stellt den allgemeinen Ansatz des verwendeten *Decision Making*-Konzepts der *Game AI* vor. Der *Behavior Tree*-Ansatz ist noch recht jung und erfreut sich einer stetig ansteigenden Beliebtheit. Der Ansatz entspringt, anders als viele andere Konzepte aus diesem Bereich, nicht direkt akademischen Gefilden, sondern ist eher ein Produkt der Games-Industrie.

4.1 Einsatzumgebung Planet PI4

Planet PI4 ist ein *Third-Person Shooter* für *Online Multiplayer*-Partien über ein *Peer-to-Peer*-Netzwerk. Der Spieler übernimmt im Spiel die Kontrolle über ein Raumschiff in einer frei erkundbaren 3D-Weltraumwelt die im Orbit des namensgebenden Planeten *PI4* angesiedelt ist. Der Orbit des Planeten besteht zwar größtenteils aus Asteroiden, beherbergt jedoch darüber hinaus noch sogenannte *Upgrade Points* und Schildgeneratoren - die *Points of Interests* des Spiels. *Upgrade Points* sind großflächige Areale die vom Spieler erobert werden können und diesem und seinem Team daraufhin bestimmte Boni verleihen. Schildgeneratoren hingegen regenerieren die Lebens- und Schildenergie in der Nähe befindlicher Spieler und sind im Gegensatz zu *Upgrade Points* etwas kleiner und können nicht erobert werden. Das Spiel wird über eine Tastatur/Maus-Steuerung gespielt. Zur Fortbewegung im 3D-Raum kommt die übliche WASD-Steuerung zum Einsatz, wobei das Drücken und Festhalten der Leertaste einen Geschwindigkeitsschub bewirkt - den *Boost*. Gezielt wird mit der Maus, wobei erst durch das Auslösen der linken Maustaste gefeuert wird.

Abbildung 11 zeigt einen Screenshot des Spiels aus Sicht eines Spielers, wobei *Upgrade Points* mit der Bildmarkierung 1 und Schildgeneratoren mit der Bildmarkierung 2 gekennzeichnet sind. Anders als die in der Abbildung dargestellte grafische Benutzeroberfläche (GUI) des Spiels am unteren Rand mit ihrem Anzeige-Tupel vermuten lässt, ist der Zustand eines Raumschiffes durch das folgende Wertetupel definiert:

$$SpaceshipState = \{Health, Shield, Energy\} \tag{1}$$

Jedoch ist der visuell hervorgehobene *Upgrade Point*-Counter am unteren Rand der GUI ebenfalls für den Zustand eines Raumschiffes von großer Bedeutung. Was sich hinter den Elementen des Wertetupels verbirgt oder welche Bedeutung der *Upgrade Point*-Counter besitzt, wird in der folgenden Aufzählung näher erläutert:

- **Upgrade Point-Counter [Bildmarkierung 3]:** Zählt die Anzahl der aktuell vom eigenen Team kontrollierten *Upgrade Points*. Jeder zusätzlich eroberte Upgrades Point bietet folgende drei Team-Boni, deren einzelne Effekte akkumulierbar sind:
  - Die maximale Energiekapazität wird um 5% erhöht
  - Die maximale Endgeschwindigkeit eines Raumschiffes wird leicht erhöht

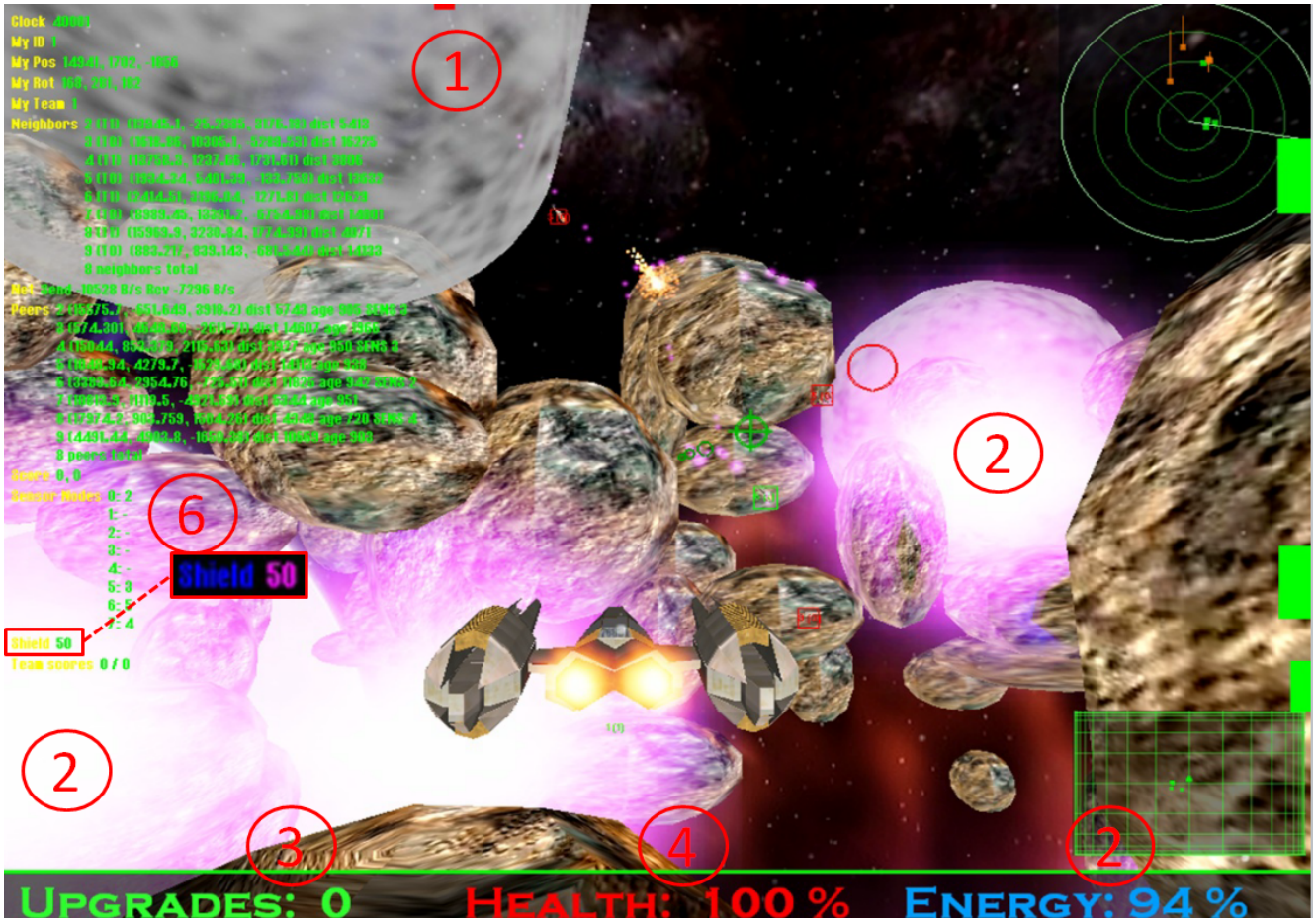

Abbildung 11: Die grafische Benutzeroberfläche von *Planet PI4*. Angelehnt an eine Grafik aus [30].

  - Die maximale Schildenergie wird um 5% erhöht

  Durch Änderungen des *Upgrade Point*-Counters gewährte Team-Boni wirken sich erst nach der Zerstörung des eigenen Raumschiffes und anschließendem Respawn aus.

- **Health [Bildmarkierung 4]:** Gibt die Lebensenergie des Schiffes in einem Prozent-Intervall von [0-100] an. Die Lebensenergie eines Schiffes kann entweder durch Treffer gegnerischer oder eigener Schiffe, sowie durch Kollisionen mit Asteroiden oder anderen Schiffen sinken. Sinkt die Lebensenergie auf 0% explodiert das Raumschiff in einer netten Animation und wird mit voller Lebensenergie an einem anderen Punkt auf der Karte respawned.

- **Energy [Bildmarkierung 5]:** Dient zum Antrieb des *Boosts* und zur Versorgung der Waffensysteme. Die Benutzung lässt das Energielevel, welches zu Beginn einer Partie in einem Prozent-Intervall von [0-100] liegt, kontinuierlich sinken. Durch Eroberung von *Upgrade Points* kann das Intervall entsprechend vergrößert werden. Anders als die Lebensenergie regeneriert sich die Energie bei Nichtbenutzung des *Boosts* oder der Waffensysteme langsam aber sicher von selbst.

- **Shield [Bildmarkierung 6]:** Dient zur Abmilderung des erlittenen Schadens durch die Waffensysteme anderer Schiffe. Die Schildenergie lädt sich nicht von selbst auf, kann allerdings durch den Besuch von Schildgeneratoren regeneriert werden. Der Anfangswert des Schildes beträgt 50 und kann durch die Eroberung von *Upgrade Points* weiter gesteigert werden.

Das Spiel *Planet PI4* ist ein reiner Multiplayer-Titel. Es enthält somit keine Geschichte, keine Skriptsequenzen oder sonstige für Singleplayer Spiele üblichen Spielelemente. Im Grunde gibt es auch nur ein einziges Level, welches im Orbit des Planeten spielt. Doch ist dieses Level weder in der Levelgröße, noch in der Team- oder Spieleranzahl fest vorgegeben oder begrenzt. Durch einen Levelgenerator lassen sich die Anzahl und Verteilung der Asteroiden und der POIs, sowie zahlreiche weitere Level-Parameter einstellen und demnach unterschiedliche Variationen diesen einen Levels generieren.

Weiter gibt es auch kein wirkliches Spielende und auch kein fest vorgegebenes Spielziel. Man kann sich das Spiel viel eher als eine Art Spielwiese vorstellen, in der die Spieler auf ihre eigene Art und Weise spielen und dabei ihre eigenen festgelegten Ziele verfolgen können. Möglich wäre es z.B., dass jeder Spieler in einer Jeder-gegen-Jeden-Manier alleine ein Team vertritt und am Ende derjenige Spieler gewinnt, der die meisten Raumschiffe abschießen konnte. Genauso gut könnte man dieses Prinzip auf zwei oder mehrere konkurrierende Teams übertragen, sodass das Team gewinnt, welches zusammen betrachtet die meisten Abschüsse erzielt hat. Eine andere Variante wäre es vorher festgelegtes Zeitfenster zu wählen und das Team zum Sieger zu ernennen, welches nach Ablauf der Zeit die meisten *Upgrade Points* beherrscht. In Kapitel 6 werden diese und andere unterschiedliche Spielarten zur Evaluation des eigenen Ansatzes eingesetzt. Die zu entwickelnde *Game AI* sollte idealerweise flexibel genug sein um je nach Spielmodi ihr Verhalten entsprechend anpassen zu können.

4.2 Funktionale Anforderungen der Game AI

In Abschnitt 1.2 wurden die übergeordneten Ziele der Master-Thesis vorgestellt. Diese Ziele gilt es zu erfüllen, damit die durch den Ansatz der computergesteuerten Spieler (Bots) generierte Netzwerklast erfolgreich zur Evaluation von P2P-Overlay-Netzwerken eingesetzt werden kann. Damit diese Lasterzeugung realisiert werden kann, müssen die eingesetzten Bots die Fähigkeit entwickeln das Spiel *Planet PI4* erfolgreich spielen zu können. Folgende Auflistung an funktionalen Anforderungen definiert dazu die von den Bots zu erbringende minimale Leistung, unterteilt in die Kategorien Bewegung, Kampf und weitere Interaktionen:

Bewegung:

- Das Raumschiff im 3D-Weltraum bewegen (vorwärts, seitlich und rückwärts fliegen)
- Einen beliebigen Punkt auf der Karte ansteuern
- Während des Fluges den *Boost* benutzen
- Bewegliche Spieler verfolgen und abfangen
- Die Karte eigenständig erkunden
- Statischen Hindernissen bzw. Asteroiden ausweichen
- Beweglichen Objekten bzw. anderen Raumschiffen ausweichen

Kampf:

- Ein stehendes oder bewegliches Ziel anvisieren
- Das Feuer auf stehende oder bewegliche Ziele eröffnen
- Im Stillstand oder während der Bewegung anvisieren und schießen
- Eine Position bzw. POI (Upgrade Point oder Schildgenerator) verteidigen oder angreifen
- Vor anderen Raumschiffen fliehen

Weitere Interaktionen:

- POIs lokalisieren
- *Upgrade Points* erobern

- Schildgeneratoren benutzen
- Informationen (z.B. gefundene Gegner, POIs) an Mitspieler weitergeben
- Unterstützung anfordern und leisten

4.3 Konzept der Game AI

Das entwickelte *Game AI*-Konzept der Master-Thesis stellt eine Erweiterung des reynoldschen Modells aus Abschnitt 2.4 dar und sieht eine Aufteilung der Verantwortlichkeiten der *Game AI* in eine strategische, taktische und operative Ebene vor. Die dabei vorgenommene Einteilung ist namentlich und in ihrer Wirkungsdauer (lang-mittel-kurz) der in der Unternehmensplanung gängigen Art der Unterscheidung bei der Auslegung und Reichweite von Planungszielen nachempfunden [18, S. 88ff.]. Die Erweiterung des reynoldschen Modells beschränkt sich dabei nicht nur auf die Umbenennung der Ebenen, sondern äußert sich auch in der Umdeutung und erweiterten Auslegung des Verantwortungsbereichs der einzelnen Ebenen, die in der folgenden Aufzählung näher erläutert werden:

- **Strategische Ebene (langfristig):** Diese Ebene ist für die Aufstellung und nachhaltige Verfolgung längerfristiger bzw. übergeordneter Zielsetzungen zuständig. Dafür müssen immer wieder neue Impulse zu dessen Nachverfolgung gesetzt werden. Hier wird bestimmt auf welche Art und Weise oder genauer gesagt, unter welchen Parameterbedingungen die Ziele im Weiteren verfolgt werden sollen. Dies hat vor allem Auswirkungen auf den Aktionsauswahlprozess der taktischen Ebene. Ebenfalls sind hier längerfristige Berechnungen oder Analysewerkzeuge angesiedelt, dessen permanente Neuberechnung entweder nicht nötig oder zu aufwendig erscheint.

  Der Verantwortungsbereich dieser Ebene würde im Agentenmodell nach Reynolds als ein eher untergeordneter Bestandteil der *Action Selection*-Ebene interpretiert werden. Aufgrund der Größe und Bedeutung der strategischen Aspekte für die *Game AI* von *Planet PI4*, wird im erweiterten Modell dieser Verantwortungsbereich aus der *Action Selection*-Ebene herausgezogen und in einer eigenen strategischen Ebene vereint. Da die strategische Ebene Vorgaben für den Aktionsauswahlprozess des Agenten definiert, kann diese Ebene des Weiteren als eine der *Action Selection*-Ebene übergeordnete Ebene betrachtet werden.

- **Taktische Ebene (mittelfristig):** Die Aufgabe der Ebene ist die Konkretisierung und Erfüllung der Ziele der strategischen Ebene unter Beachtung der einzuhaltenden strategischen Rahmenbedingungen oder Vorgaben. Dazu ist die Bestimmung und Ausführung der zur Erfüllung notwendigen Aktionen notwendig. Zur Realisierung dieser Aufgabe ist hier als wichtigste Komponente das Herzstück des rationalen Agenten angesiedelt - das *Decision Making*.

  Die taktische Ebene entspricht somit weitestgehend der *Action Selection*-Ebene nach Reynolds, ohne dessen strategische Aspekte.

- **Operative Ebene (kurzfristig):** In dieser Ebene geht es um die direkte und kurzfristige Realisierung der Ziele bzw. um die Umsetzung der ausgewählten Aktionen der taktischen Ebene. Dabei können bei Bedarf zur direkten Reaktion auf plötzliche Situationsänderungen weitere unmittelbar wirksame Aktionen eingestreut werden, wie z.B. das Ausweichen vor Hindernissen.

  Diese Ebene ist vergleichbar mit der *Steering*-Ebene nach Reynolds, die der übergeordneten Aktionsauswahl-Ebene einen Zugriff auf eine Menge von abstrakten *Behaviors* zur Erfüllung der Ziele ermöglicht, statt der direkten Übergabe der Aktuator-Steuerung. Im Unterschied zur *Steering*-Ebene, beschränken sich die von operativen Ebene definierten *Behaviors* nicht ausschließlich auf *Steering Behaviors*, sondern bieten zusätzlich auch andere Spielrelevante *Behaviors* an, wie

die Bekämpfung eines Feindes, der Eroberung oder Verteidigung eines *Upgrade Points* oder die Wiederherstellung der Lebensenergie mittels Schildgenerator.

Dieses so erweiterte Modell nach Reynolds kann in einem zweiten Schritt in das Konzept eines rationalen (nutzenbasierten) Agenten aus Abschnitt 2.3 integriert werden. Eine mögliche Integration könnte wie in Abbildung 12 dargestellt aussehen und stellt gleichzeitig das eigentliche *Game AI*-Konzept dieser Master-Thesis dar.

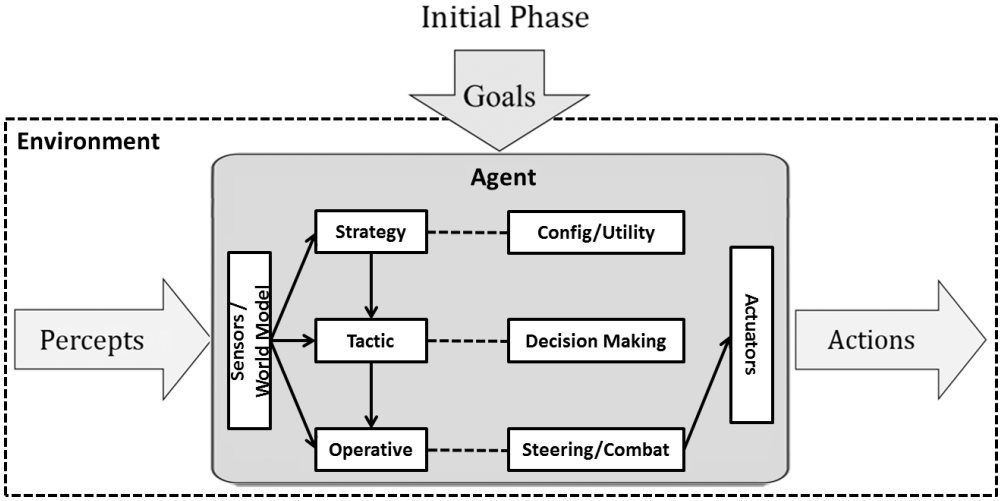

Abbildung 12: Das *Game AI*-Konzept als Erweiterung des Agentenmodells nach Reynolds und der Integration dessen in das Konzept eines rationalen (nutzenbasierten) Agenten.

Ein Vergleich des *Game AI*-Konzepts mit dem nutzenbasierten Agenten aus Abbildung 2 des Abschnitts 2.3 zeigt, dass alle Elemente des nutzenbasierten Agenten auch im *Game AI*-Konzept berücksichtigt werden, nur dass die Anordnung der Ebenenzugehörigkeit der Elemente entsprechend angepasst und um Aspekte der reynoldschen Modellerweiterung ergänzt worden ist.

Das *World Model* und die Sensoren des Agenten bieten in diesem Sinne nicht mehr ausschließlich der *Decision Making*-Komponente Zugriff auf Welt- und Zustandsinformationen bzw. auf aktuelle Wahrnehmungen an, sondern weiten den Zugriff auf alle Ebene des Agenten aus. Weiter wird der *Actuators*-Zugriff, wie bereits in der Beschreibung der drei Ebenen erwähnt, der *Decision Making*-Komponente entzogen und die operative Ebene dazwischen geschaltet. Die Aufgabe der Bewertung der Aktionsausführung (*Utility Function*) wird dabei der strategischen Ebene zugeordnet, da diese die Ausführung der taktischen Ebene überwacht, die wiederum für den Aktionsauswahlprozess bzw. für das *Decision Making* zuständig ist. Durch die Zuordnung der *Utility Function* zur strategischen Ebene können die durch die Bewertung gewonnenen Erkenntnisse direkt in Form von neuen Vorgaben und angepassten Konfigurationen an die taktische Ebene weitergeleitet werden und somit das Verhalten des Agenten Stück für Stück den langfristigen Zielvorgaben entsprechend angepasst werden.

4.4 Behavior Tree

Behavior Trees stellen eine Modellierungsart zur Beschreibung von komplexen Verhaltensweisen dar, welche bereits in einigen modernen Computerspielen wie *Halo 2* & *Halo 3*, *Crysis* & *Crysis 2* oder *Spore* zum Einsatz kam. Am ehesten lässt sich ein *Behavior Tree* mit einem *Hierarchical Finite State Maschine* (HFSM) [35, S. 318-331] vergleichen, welcher anstelle von Zuständen als elementare Konstruktionseinheit „Tasks“ verwendet und um Aspekte wie *Scheduling*, reaktives Planen und Aktionsausführung erweitert. Die Bezeichnung „Tree“ ist dabei allerdings als irreführend anzusehen, da es sich in Wahrheit

um einen gerichteten azyklischen Graphen (DAG) handelt. Ein Knoten kann nämlich zwecks Wiederverwendbarkeit von Verhaltensbeschreibungen innerhalb des Graphen mehrere Eltern haben und nicht wie ein Baum nur einen Einzigen. Jedoch wird im weiteren Verlauf, auf Grund der etablierten Verwendung des Begriffs „Behavior Tree“ an dieser nicht ganz korrekten Bezeichnung festgehalten [10].

Alle Tasks besitzen zur gleichartigen Handhabung ein gemeinsames Interface, welches die Rückgabe eines Statuscodes über die erfolgreiche oder fehlgeschlagene Ausführung der Tasks vorschreibt. Tasks können dabei wie Zustände eines HFSM aus Sub-Tasks bestehen, was eine hierarchische Anordnung von Verhaltensweisen ermöglicht. Im Allgemeinen werden drei Arten von Tasks unterschieden:

- **Condtion:** Bedingungen überprüfen Eigenschaften des aktuellen Weltzustands. Häufig findet nur eine simple Überprüfung von Zustandswerten statt, allerdings können Bedingungen auch komplexere Berechnungen darstellen, wie z.B. die Berechnung und anschließende Überprüfung des Sichtradius eines Charakters.

- **Action:** Aktionen führen zu Änderungen des Weltzustands oder des internen Zustandes des Charakters. Beispielsweise das Ausführen einer Animation oder einer Pathfinding-Routine.

- **Composite:** Ein Kompositum ist ein Container für andere Tasks (*Condition*, *Action* und *Composite*) und ermöglicht somit die Hierarchische Anordnung von Tasks. Die einzelnen *Composite*-Tasks unterscheiden sich allerdings nicht durch ihre Menge an Sub-Tasks, sondern auch anhand ihrer Ausführungssemantik der Sub-Tasks. Ein *Behavior Tree* verfügt meistens über mindestens folgende zwei grundlegende *Composite*-Arten:
  - **Selector:** Ein Selektor bestimmt die Ausführungsreihenfolge der Sub-Tasks, wo jedoch bei der ersten erfolgreich ausführbaren Task die *Composite*-Tasks ebenfalls als Erfolgreich gilt. Beispiele für Ausführungsreihenfolgen wären eine Links-nach-Rechts Traversierung oder eine Zufallsauswahl.
  - **Sequence:** Eine Sequenz bestimmt ebenfalls eine Ausführungsreihenfolge, wobei jedoch die *Composite*-Tasks erst dann als Erfüllt angesehen wird, wenn jede Sub-Tasks erfolgreich ausgeführt werden kann.

Conditions und *Actions* befinden sich dabei ausschließlich an Blättern des Baums, wohingegen innere Knoten aus *Composite*-Tasks bestehen [35, S. 334-340].

Der komplete *Behavior Tree* repräsentiert somit alle möglichen Aktionen, die ein Charakter ausführen kann. Der Pfad von der Wurzel zu einem Blattknoten stellt dabei den Entscheidungsprozess der Auswahl der nächsten auszuführenden Aktion des Charakters in der jeweiligen Situation dar. Gewöhnlich wird dabei eine Tiefensuche durchgeführt, die allerdings durch entsprechende *Composite*-Tasks Implementierungen, z.B. *Random Selectors*, für jeden Teilbaum separat angepasst werden kann. Werden mehrere Game-Zyklen für die Ausführung einer Task gebraucht, so kann sich entweder die letzte gewählte Task gemerkt und diese im nächsten Schritt weiter ausgeführt werden oder die Traversierung des Baums fängt wieder von der nächst höheren Task-Ebene bzw. sogar von der Wurzel aus erneut an [35, S. 334-340].

Die von Alex Champamdard [10, 40] suggerierte Zielorientierung von *Behavior Trees* ergibt sich, wenn man die *Composite*-Tasks als zu erfüllende Ziele und die Sub-Tasks als die Menge der möglichen Wege diese Ziele zu erreichen betrachtet. *Composite*-Tasks ermöglichen darüber hinaus auch die Priorisierung der Ziele untereinander, indem Ziele entweder in eine hierarchische Beziehung zueinander gestellt werden oder die Ausführungsreihenfolge einer *Composite*-Task die Zielauswahl direkt bestimmt. Abbildung 13 zeigt einen exemplarischen *Behavior Tree* für das Eintreten in einen Raum. Ein Knoten mit einem Fragezeichen bezeichnet dabei einen Selektor und ein Knoten mit einem Pfeil eine Sequenz. Alle Blätter die nicht mit einem Fragezeichen am Ende des Bezeichners enden sind *Actions*, ansonsten handelt es

sich um eine *Condition*. Der Wurzel-Selektor wählt je nach verwendeter Ausführungsreihenfolge einen der beiden Sequenzen aus. In diesem Beispiel kann man von einer links-nach-rechts Priorisierung ausgehen. Wird demnach die linke Sequenz ausgewählt, prüft die erste *Condition*, ob die Tür offen ist. Sollte diese offen sein, wird mit einer erfolgreichen Ausführung der „Move (into room)"-Aktion die Sequenz ebenfalls erfolgreich abgeschlossen. Da für den Erfolg eines Selektors nur eines seiner Kinder erfolgreich sein muss, ist in diesem konstruierten Beispiel die Wurzel ebenfalls Erfolgreich ausgeführt worden. Würde allerdings eine Task der Sequenz im linken Teilbaum scheitern, so würde die ganze Sequenz scheitern und nach dem zurückspringen zur Wurzel würde der linke Teilbaum ausprobiert werden.

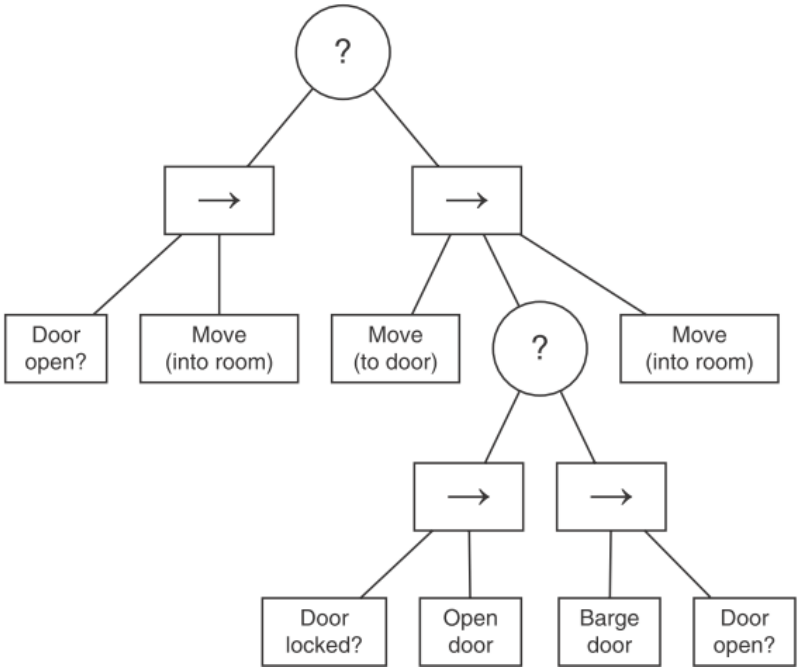

Abbildung 13: Beispiel für einen *Behavior Tree* fürs Eintreten in einen Raum. Entnommen aus [35, S. 339].

Die Vorteile eines *Behaivor Trees* sind die Verwendung eines restriktiven Vokabulars und der Fokussierung auf Tasks als zentrale Konstruktionseinheit, was das Verständnis bei der Modellierung des Verhaltens von NPCs deutlich erleichtert, da man im Gegensatz zu HFSMs nicht mehr in Zuständen denken muss. Die Trennung von Zielen und Verhalten, welche diese Erfüllen, die Wiederverwendbarkeit von Verhaltensweisen und die Modifizierbarkeit bzw. Parametrisierbarkeit der Ausführungsreihenfolgen ermöglichen einen flexiblen und vor allem leicht erweiterbaren Ansatz zur Beschreibung von komplexen Verhalten. Mögliche Erweiterungen sind u.a. die Verwendung von Selektoren die eine parallele Ausführung ihrer Sub-Tasks unterstützen, die Anwendung des *Decorator Patterns* auf Tasks (siehe dazu auch Abschnitt 4) oder die Integration von Team-Tasks, wie sie u.a. in Crysis [41] Verwendung findet.

Ein Nachteil von BTs kommt dann zum Tragen, wenn man tatsächlich Zustände statt Verhalten modellieren will, was immer dann der Fall sein kann, wenn auf externe Ereignisse reagiert werden soll. Ein Beispiel wäre das Abbrechen einer Patrouille bei plötzlichem Feindkontakt. Dies ist prinzipiell ebenfalls mit einem BT modellierbar, doch um einiges umständlicher als mit einem HFSM. So wurde z.B. für Halo 2 die Erweiterung der *Behavior Stimuli* entworfen, die eine Event-Mechanik in den BT einfügt. Würde in dem eben aufgeführten Patrouillen-Beispiel ein Feindkontakt gemeldet, so hätte dies eine *Stimuli*-Erzeugung zur Folge, welche an einer oder mehreren definierten Stellen entsprechende spezielle Verhaltensweisen zum Umgang mit Feinden in den Baum einfügen würde. Diese wären zeitlich befristet und somit nach einer gewissen Zeit wieder aus dem Baum entfernt werden [20].

Ein weiterer Nachteil von BTs ist, dass die Art und Weise wie die Aktionsauswahl ermittelt wird, eine Form des reaktiven Planens darstellt - einer sehr simplen Form der Planung. Selektoren ermöglichen das Ausprobieren bestimmter Verhaltensweisen. Scheitern diese werden alternative Verhaltensweisen ausprobiert. Ein Charakter kann in diesem Sinne nicht vorausschauend agieren, da er um eine Aktion zu evaluieren diese erst Ausführen muss. Außerdem ist die Auswahl der Aktion oder der Aktionsketten durch die Traversierung des BTs fest vorgegeben ist. Mit anderen Worten, auch wenn einem BT alle möglichen Aktionen bekannt sind, verfügt dieser nicht über die Möglichkeit selbst zu bestimmen, welche Aktionskette am besten auszuführen ist. Er kann nur die Aktionskette finden, die vom Designer für die jeweilige Situation als die beste erdacht wurde. Auf der anderen Seite wird somit im Gegensatz zum im Abschnitt 3.3 vorgestellten *Decision Making*-Konzept GOAP eine dank der deterministischen Ausführungssemantik der *Composite*-Tasks viel stärkere und direktere Ausführungskontrolle gewährleistet. Eine wichtige Voraussetzung für die Reproduzierbarkeit und Konfigurierbarkeit der zu erzeugenden Netzwerklast.

5 Design und Implementierungsdetails der Game AI

Dieses Kapitel beschreibt die genaue Umsetzung des *Game AI*-Konzepts und der aufgestellten funktionalen Anforderungen an diese aus Kapitel 4. Dazu wird das konkrete Design des Systems und seiner Subsysteme vorgestellt, sowie auf zahlreiche wichtige Implementierungsdetails eingegangen.

Im ersten Abschnitt 5.1 „Einbindung der Game AI in Planet PI4“ wird die Integration der *Game AI* in das Computerspiel *Planet PI4* beschrieben. Dazu werden die Abhängigkeiten zum System des Spiels und die Einbindung der *Game AI* in den Spielablauf erläutert.

Der Abschnitt 5.2 „Game AI-Architektur“ beschreibt die Realisierung des *Game AI*-Konzeptes auf Klassenebene. Dabei wird besonders der nicht-lineare Informationsfluss im System beschrieben und welche Konsequenzen daraus entstehen.

Die restlichen drei Abschnitte des Kapitels beschreiben die Realisierung der Subsysteme der operativen (Abschnitt 5.3), taktischen (Abschnitt 5.4) und strategischen Ebene (Abschnitt 5.5).

5.1 Einbindung der Game AI in Planet PI4

Planet PI4 verwendet als zugrundeliegende 3D-Engine die Irrlicht Engine [1]. Irrlicht ist eine „open source high performance realtime 3D engine“, die *Cross Platform*-Entwicklungen und zahlreiche Features gängiger kommerzieller 3D-Engines unterstützt. *Planet PI4* und Irrlicht sind jeweils in C++ entwickelt worden, weswegen auch die *Game AI* in dieser Sprache geschrieben wurde. Die *Game AI* ist ein wesentlicher Bestandteil der *Planet PI4*-Architektur, ist jedoch als eigenständige Komponente insoweit abgekapselt, dass sich das Wissen der *Game AI* auf einige wenige Irrlicht-Datentypen und auf eine überschaubare Menge an *Planet PI4*-Schnittstellen oder Klassen beschränkt. Aus diesem Grund wird in den weiteren Ausführungen nur auf Aspekte von *Planet PI4* und Irrlicht näher eingegangen die für das Verständnis der *Game AI*-Entwicklung unerlässlich sind. Für genauere Ausführungen zur *Planet PI4*-Architektur oder *Planet PI4* im Allgemeinen sei an dieser Stelle auf [27] und [25] verwiesen.

Das Klassendiagramm aus Abbildung 14 zeigt, wie die *Game AI* in *Planet PI4* eingebunden wird. Um die *Game AI* mit dem Spiel zu verbinden, muss die *Game AI* die zwei Schnittstellen *IBot* und *ITask* implementieren. *IBot* gibt dabei die Implementierung folgender Initialisierungsmethode vor:

```
public virtual void init(IGameStateView* state, IShipControl* shipControl,
                         ITaskEngine* taskEngine, IRandom* random)
```

Diese Methode wird von der *Game AI* implementiert und vom Spiel automatisch einmal in der Initialisierungsphase aufgerufen. Somit übergibt das Spiel vor dem eigentlichen Spielstart die wichtigsten Informations- und Steuerungsobjekte an die *Game AI*, die hinter folgenden Schnittstellen versteckt sind:

- **IGameStateView:** Bietet Zugang zum aktuell partiell beobachtbaren Spielzustand. Partiell deswegen, weil in einem P2P-Spiel niemals sämtliche (globale) Informationen einem Peer und somit einem Spieler zur Verfügung stehen, sondern nur ausgewählte (partielle) Informationen die maßgeblich von der aktuellen AOI abhängen. Bereitgestellt werden u.a. Informationen zu in der Nähe befindlichen Raumschiffen und statischen Objekten wie Asteroiden, *Upgrade Points* und Schildgeneratoren. Dabei wird auf andere Raumschiffe über *RemoteShip*-Klasse zugegriffen, wobei bei Teamkameraden erst der Umweg über die *PlayerId*-Klasse genommen werden muss. Die statischen Objekte verfügen mit der *IStaticObject*-Schnittstelle über einen einheitlichen Zugriff. Die

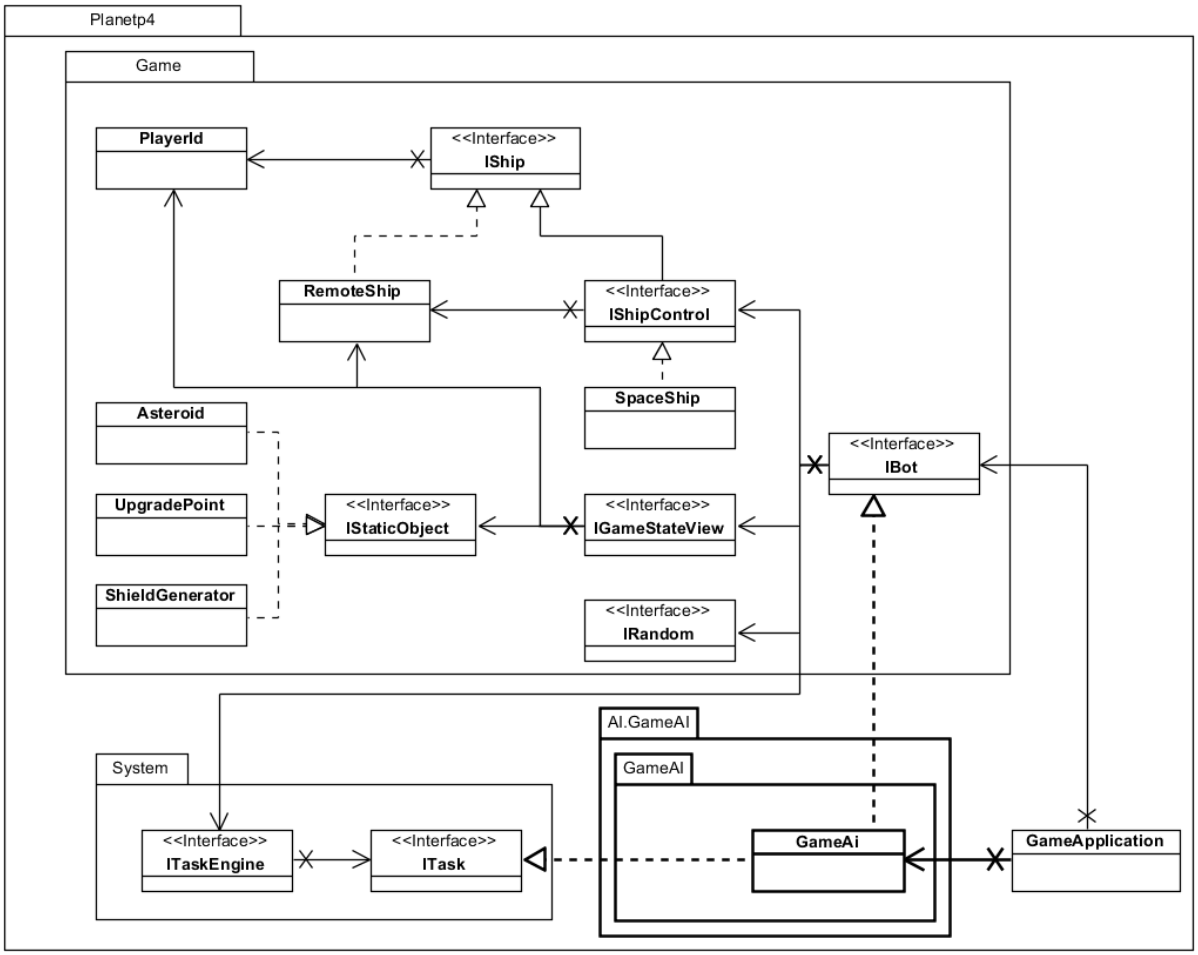

Abbildung 14: Klassendiagramm zur Einbindung einer *Game AI* in *Planet PI4*.

IGameStateView-Schnittstelle bietet darüber hinaus noch Informationen über den eigenen aktuellen *Health*-Zustand, sowie den maximal erlaubten *Health*-Wert.

- **IShipControl:** Bietet Zugriff auf die über die *IShip*-Schnittstelle zur Verfügung gestellten Informationen über den aktuellen Zustand des Schiffes. *IShip* enthält u.a. Positions-, Orientierungs-, Geschwindigkeits- und Rotationsangaben, sowie Informationen über Teamzugehörigkeit und die eigene *PlayerId*. Diese Informationen von IShip werden um weitere Schiffsinformationen, wie etwa aktueller Schildzustand oder Energielevel ergänzt. Die zweite Aufgabe der Schnittstelle ist es der *Game AI*-Steuerungsmöglichkeiten des eigenen Raumschiffs zur Verfügung zu stellen. Die Steuerungsmöglichkeiten sind dabei dieselben, die auch dem Spieler über die Tastatur/Maus Steuerung angeboten werden, d.h. Methoden zur Bewegung im Raum, eine *Boost*-Aktivierungsmethode, sowie eine *fire()*-Methode.

- **ITaskEngine:** Bietet einerseits Zugriff auf die aktuelle Spielzeit, andererseits meldet sich die *Game AI* über diese Schnittstelle mit folgender Methode bei der *TaskEngíne* von *Planet PI4* an:

  ```
  public virtual void addTask(ITask* task, int ms, int ofs = 0)
  ```

  Die *TaskEngine* ist für die korrekte Ausführung von Subprozessen zuständig, wie z.B. der *Game AI*-Ausführung. Der zweite Parameter gibt an in welchem Millisekunden-Intervall die Task bzw. die *Game AI* ausgeführt werden soll.

- **IRandom:** Bietet mehrere Methoden zur deterministischen Erzeugung von Zufallszahlen, die über alle Game-Instanzen hinweg dieselben Zahlenreihen erzeugen. Dies ist notwendig um reproduzierbare Ergebnisse im Sinn der Lasterzeugung zu garantieren.

Damit die *IBot.init()*-Methode überhaupt ausgeführt werden kann, muss die *GameAI*-Klasse der *GameApplication*-Klasse bekannt gemacht werden. Dies ist leider momentan nur durch direkte Änderung am Code möglich. Die *GameApplication*-Klasse verwaltet welche konkrete *Game AI*-Implementierung eingesetzt werden soll. Eine neue Implementierung muss entsprechend diesem Auswahlprozess hinzugefügt werden.

Die alleinige *GameApplication*-Bekanntmachung ist allerdings nicht ausreichend. Sie deckt nur die Initialisierungsphase ab und muss somit um die in der Beschreibung zur *ITaskEngine*-Schnittstelle erwähnte Anmeldung der *Game AI* bei der *TaksEngine* von *Planet PI4* ergänzt werden. Damit diese gelingt, muss die *GameAI*-Klasse die *ITask*-Schnittstelle implementieren, sodass sie sich als *ITask*-Objekt der *TaskEngine* mittels eines Aufrufs der *addTask()*-Methode selbst übergeben kann. Jede *ITask*-Realisierung muss dabei folgende Methode implementieren:

```
protected virtual Result executeTask()
```

Die *executeTask()*-Methode wird nach der Initialisierungsphase von der *TaskEngine* wiederholt aufgerufen. Wenn möglich entspricht das Intervall zwischen den einzelnen Methoden-Aufrufen dabei dem übergebenen Intervall-Parameter bei der Anmeldung. Mit diesem Callback-Mechanismus wird mit jedem *executeTask()*-Aufruf, der *Game AI* entsprechende Zeit zur Verfügung gestellt, um eigene Berechnungen zur Aktionsauswahl und anschließender Aktionsausführung mittels der *IShipControl*-Schnittstelle vorzunehmen. Der Rückgabetyp *Result* dient dabei als Statusmeldung die von der *TaskEngine* entsprechend interpretiert werden kann.

5.2 Game AI-Architektur

Wie die Abbildung 14 aus dem vorherigen Abschnitt bereits zeigte, ist die *GameAI*-Klasse die Schnittstelle zwischen dem Spiel *Planet PI4* und der *Game AI* bzw. dem dadurch repräsentierten Bot. Die *GameAI*-Klasse hat somit die Kenntnis über die vorgestellten Informations- und Steuerungsobjekte und implementiert zur Integration in den Spielablauf die beiden wichtigen Schnittstellen *IBot* und *ITask*. Die konkrete Implementierung der *GameAI*-Klasse ist dabei leichtgewichtig, was hauptsächlich daran liegt, dass sie alle Verantwortlichkeiten zur Bestimmung und Ausführung der nächsten Aktion bzw. den dazu gehörigen Berechnungen an andere Klassen weiterleitet. Die Aufgaben der *GameAI*-Klasse beschränken sich somit auf folgende drei Aspekte:

- Realisiert die Anbindung an *Planet PI4* und den Spielablauf. Dient der *GameApplication*-Klasse als parametrisierbare *Facade*-Klasse [16, S. 212-222] für den Bot.

- Ist für die Erstellung und Parametrisierung der konkret zu verwendeten Implementierungen zuständig, an die die Verantwortlichkeiten zur Bestimmung der nächsten Aktion weitergereicht werden. Ein wichtiger Punkt ist hier die Weiterleitung oder Bereitstellung der Informations- und Steuerungsobjekte an die verantwortlichen Objekte.

- Leitet nach Aufruf der *executeTask()*-Methode die Bestimmung und Ausführung der nächsten Aktion an dafür zuständige Klassen bzw. Objekte weiter.

Die angesprochene Weiterleitung der Verantwortlichkeiten wird dabei nicht anhand einiger weniger direkt der *GameAI*-Klasse bekannter Klassen realisiert. Das Klassendiagramm aus Abbildung 15 zeigt, dass

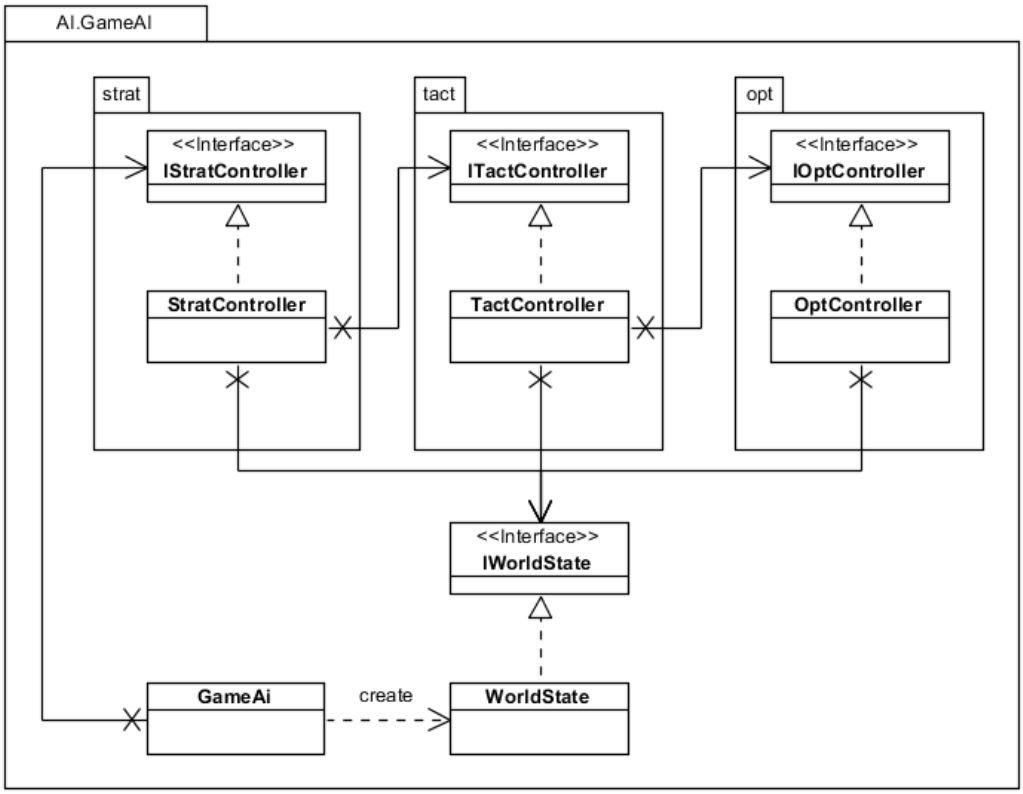

Abbildung 15: Die Game AI-Architektur mit dem Zusammenspiel der wichtigsten Komponenten und Subsysteme.

stattdessen für die Aufgabe der Aktionsbestimmung und -ausführung ein Geflecht aus drei Subsystemen eingesetzt wird. Die Bezeichnung und die Aufgabenbereiche dieser Subsysteme entspricht weitestgehend der konzeptionellen Beschreibung der drei Ebenen der Gama AI aus Abschnitt 4.3.

Abbildung 15 zeigt das hierarchische Zusammenspiel der Hauptkomponenten der drei Subsysteme mit der *GameAI*-Klasse. Die *GameAI*-Klasse stößt dazu den Aktionsauswahl-Prozess mit dem Aufruf des *Facade*-Objekts der strategischen Ebene bzw. des strategischen Subsystems an. In diesem Subsystem befinden sich, wie in den anderen auch, weitere Klassen und Komponenten auf die später noch in den einzelnen Abschnitten zum jeweiligen Subsystem näher eingegangen wird.

Nachdem das strategische Subsystem seine Berechnungen beendet hat, gibt es die Informationen an die taktische Ebene weiter, dazu definiert das taktische Subsystem selber auf welche Art und Weise und zu welchem Zweck zusätzliche Informationen, Entscheidungen oder Zielvorgaben ihm hinzugefügt werden können. Nachdem alles Notwendige weitergereicht worden ist, leitet das strategische Subsystem den *Decision Making*-Auswahlprozess des taktischen Subsystems ein, an dessen Ende immer die getroffene Entscheidung über die als nächstes auszuführende Aktion steht. Das taktische Subsystem hat dabei allerdings keinen direkten Zugriff zur Steuerung des Schiffs, wie dieser z.B. über *IShipControl*-Schnittstelle möglich wäre, um damit die gewünschte Aktion selber auszuführen. Stattdessen erhält das taktische Subsystem Zugriff auf die von der operativen Ebene angebotenen Aktionen zur Steuerung des Schiffs über das entsprechende *Facade*-Objekt des operativen Subsystems.

Das operative Subsystem kapselt somit alle vom Agenten ausführbaren Aktionen in einer eigenen Schnittstelle. Bei den angebotenen Aktionen handelt es sich nicht nur um die direkte Weiterleitung primitiver Befehle an die dafür zuständigen Schnittstellen wie z.B. der *IShipControl*-Schnittstelle. Die

operative Ebene bietet vielmehr wie die *Steering*-Ebene von Reynolds bestimmte komplexere *Behaviors* an, die bereits die Kombination bestimmter Aktionsausführungen auf *Low Level*-Ebene einschließt. Im Gegensatz zur reynoldschen *Steering*-Ebene beschränken sich die dabei angebotenen *Behaviors* nicht nur auf die Bewegung des Agenten, sondern Erweitern das Konzept auf *Behaviors* zur Bekämpfung von Gegnern, der Eroberung von *Upgrade Points* und der Wiederherstellung der Lebensenergie mit Hilfe von Schildgeneratoren. Ein Beispiel soll dieses Zusammenspiel verdeutlichen.

Man stelle sich vor, die taktische Ebene unterstütze einen *Decision Making*-Prozess, der die unterschiedlichen Charakterzüge eines Bot-Agenten berücksichtigt. Die strategische Ebene könnte solche Charaktereigenschaften setzen und so z.B. bestimmen, dass der momentane Bot sehr aggressiv spielen soll. Mit dieser strategischen Vorgabe würde der *Decision Making*-Prozess der taktischen Ebene eventuell andere Aktionen auswählen, als er es für nicht aggressive Bots tun würde. Diese so getroffene Auswahl einer eher aggressiven Aktion würde dann von der taktischen Ebene zur Ausführung an die operative Ebene weitergereicht werden. Nehme man weiter an es handelt sich dabei um die Aktion „FightEnemy XY“, dann würde die operative Ebene versuchen diesen Befehl auszuführen, könnte aber davor vlt. feststellen, dass der Feind sich hinter einem Asteroiden befindet und somit nicht direkt unter Beschuss genommen werden kann. Die operative Ebene würde in so einer Situation keine Fehlermeldung an die taktische Ebene zurückgeben und somit nicht anschließend den Dienst verweigern. Sie würde eher die Ausführung der eigentlichen *Fight*-Aktion ignorieren und stattdessen versuchen einen Weg zu finden dem Asteroiden auszuweichen, damit im Anschluss daran der Feind schließlich erfolgreich angegriffen werden kann. Wenn die taktische Ebene in darauffolgenden Spielzyklen irgendwann eine andere nächste Aktion als die *Fight*-Aktion bestimmen würde und der Agent noch mit dem Ausweichen des Asteroiden beschäftigt wäre oder soeben das Ausweichen beendet würde, so kann es durchaus vorkommen, dass die *Fight*-Aktion, obwohl von der taktischen Ebene über mehrere Spielzyklen zur Ausführung bestimmt, nicht einmal tatsächlich ausgeführt worden ist.

Ein wichtiger Aspekt beim Zusammenspiel der Ebenen ist, dass die Ausführungsfrequenz jedes einzelnen Subsystems unabhängig voneinander regulierbar ist. Dazu wird ein zur Initialisierungszeit einstellbarer *Prozess*-Parameter des entsprechenden *Facade*-Objekts gesetzt. Der *Prozess*-Parameter ist dem Intervall-Parameter der *addTask()*-Methode der *TaskEngine* von *Planet PI4* nachempfunden und soll die Zeit zwischen zwei aufeinanderfolgenden Ausführungen bestimmen. Weil die Auslassung von Aktionen nicht ratsam erscheint, sollte zumindest das operative Subsystem mit dem gleichen Intervall betrieben werden, wie die *GameAI*-Klasse. Somit wird sichergestellt, dass immer eine Aktion ausgeführt wird, auch wenn es sich dabei nur um die Wiederholung der letzten Aktion handelt, weil die Neuberechnungen der taktischen Ebene in einem Zyklus ausgelassen wurden.

Auch wenn eines oder mehrere der Subsysteme im aktuellen Spielzyklus ausgelassen werden sollen, so werden doch immer alle drei *Facade*-Objekte in jedem Zyklus aufgerufen und ausgeführt, da diese für die Entscheidung der (Nicht-)Ausführung zuständig sind. Es wird demnach unabhängig der Auslassung einiger Subsysteme immer an der in Abbildung 16 dargestellten Ausführungsreihenfolge festgehalten, die jeweils von der *TaskEngine* von *Planet PI4* ausgehend beginnt.

Damit das *Facade*-Objekt der operativen Ebene immer eine Aktion zur Ausführung bestimmen kann, speichert sich das taktische *Facade*-Objekt seine jeweils letzte, für die operative Ebene relevante, Aktionsbestimmung und reicht diese im Falle eines auslassenden Berechnungszyklus der taktischen Ebene an die operative Ebene weiter. Das strategische *Facade*-Objekt muss keine letzte Aktion zwischenspeichern, da diese nur Konfigurationen an der taktischen Ebene vornimmt, die von dieser sowieso gespeichert werden müssen.

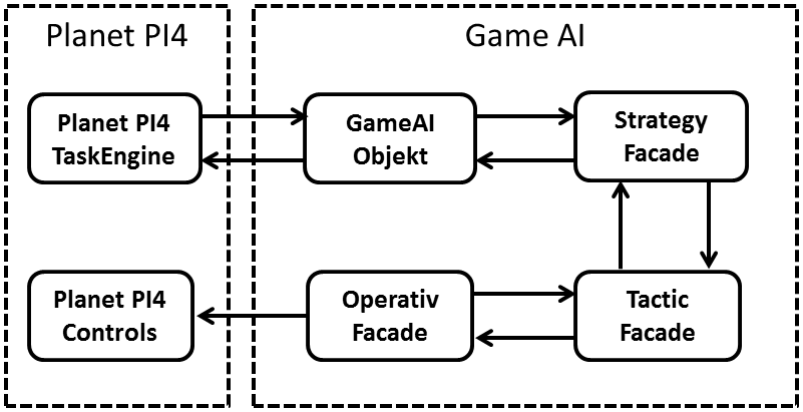

Abbildung 16: Die feste Ausführungsreihenfolge der Komponenten der Game AI und der von Planet PI4 innerhalb eines Spielzyklus.

Damit die drei Ebenen ihrer Arbeit nachgehen können, müssen sie vor allem irgendwie einen Zugriff auf die von *Planet PI4* bereitgestellten Informationen und Steuerungsmöglichkeiten erhalten. Dabei werden die von einer Ebene benötigten Schnittstellen in der Regel an das jeweilige *Facade*-Objekt einer Ebene zur Initialisierungszeit direkt weitergereicht. So wird z.B. die *IShipControl*-Schnittstelle an das *Facade*-Objekt der operativen Ebene weitergereicht, da diese für die tatsächliche Aktionsrealisierung zuständig ist. Beim Zugriff auf Informationen die keine *Listener*-Anmeldung (siehe Abschnitt 5.4) voraussetzen wird jedoch mit der Bereitstellung einer eigenen *IWorlState*-Schnittstelle ein anderer Weg des gemeinsamen Zugriffs gewählt. Der Grund hierfür ist, dass die der *Game AI* bereitgestellten Informationen sich im Grunde auf alle vier im vorherigen Abschnitt vorgestellten Schnittstellen verteilen und darüber hinaus noch zusätzlich zu reinen Informationen andere Funktionen bereitstellen, wie z.B. die Steuerungsfunktionen von *IShipControl*. Zur Vermeidung der Unterscheidung der einzelnen Schnittstellen bei der Bot-Implementierung und zur Vermeidung der unnötigen Gewährung nicht zwingend erforderlicher Zugriffsrechte auf *Planet PI4* (z.B. Steuerungsmöglichkeiten). Werden alle reinen Spielinformationen die für den Bot in irgendeiner Weise relevant sind in der *IWorldSate*-Schnittstelle gebündelt und allen drei Ebenen ein direkter Zugriff darauf gewährt. Mit dieser Bündelung der Informationen in einer Schnittstelle wird außerdem ganz im Sinne des **I**nterface **S**egregation **P**rinciple (ISP) [32] erreicht, dass die drei Ebenen zumindest im wichtigen Informationsbeschaffungskontext nicht von anderen für sie irrelevanten Änderungen, wie z.B. Änderungen an der Raumschiffsteuerung, betroffen werden. Somit sind die drei Ebenen der *Game AI* ein Stück mehr gegenüber zukünftigen Refactoring-Maßnahmen zur Säuberung der Schnittstellen vorbereitet, deren im Zuge der Weiterentwicklung des Spiels durchaus als realistisch anzunehmen sind.

Das Anbieten einer einheitlichen Informationsschnittstelle die vom konkreten Informationsbezug abstrahiert, ermöglicht auch zukünftige Performance-Optimierungen. Können an dieser Stelle doch leicht leistungskritische Aufrufe, wie die Abfrage aller in der näher befindlicher Schiffe zwischengespeichert werden und somit der Zugriff darauf beschleunigt werden. Ein Zwischenspeichern und Cachen ist deswegen auch eine erfolgsversprechende Optimierung, weil die Aufteilung der *Game AI* in drei voneinander abgekapselten Hierarchieebenen keinen direkten Informationsfluss erlaubt. Abfrageergebnisse wie die Nachbarschaftsliste können nicht ohne weiteres zwischen den Ebenen weitergereicht werden, sondern erfolgen voneinander unabhängig und können sich somit innerhalb eines Zyklus wiederholen.

5.3 Operative Ebene

Die operative Ebene ist die unterste der drei Ebenen der vorgestellten *Game AI*-Hierarchie aus Abschnitt 5.2. Ihre Aufgabe ist die Umsetzung der Ziele der taktischen Ebene. Dazu wird eine Menge an Aktionen definiert und der taktischen Ebene über die *IOptController*-Schnittstelle zur Verfügung gestellt. Die angebotenen Aktionen sind dabei keine primitiven Aktionen, wie das Abfeuern eines Schusses oder der Ausführung einer einfachen Steuerungsbewegung, sondern komplexe Aktionen die aus der kombinierten Ausführung oder Mehrfachausführung primitiver Aktionen bestehen können. Welche konkreten komplexen Aktionen genau bereitgestellt werden, listet und erläutert kurz die folgende Aufzählung:

- **Arrive Position:** Eine beliebige Position soll erreicht werden ohne dabei über das eigentliche Positionsziel hinauszuschießen.
- **Capture Upgrade Point:** Ein *Upgrade Point* soll angesteuert und erobert werden.
- **Evade Enemy:** Ermöglicht die Flucht vor beweglichen Zielen, d.h. vorrangig vor gegnerischen Raumschiffen.
- **Fight Enemy:** Ermöglicht das Abfangen und Bekämpfen (gegnerischer) Raumschiffe.
- **Flee:** Es soll sich so schnell wie möglich von einem Punkt entfernt werden.
- **Pursue Enemy:** Ermöglicht die Verfolgung bzw. das Abfangen (gegnerischer) Raumschiffe.
- **Reset:** Setz den internen Zustand der operativen Ebene zurück. Davon sind allen voran Zustandsbehaftete Berechnungen betroffen.
- **Restore Energy:** Ein Schildgenerator soll angesteuert werden und die Lebenenergie aufgeladen werden.
- **Wander:** Es soll eine bestimmte Region der Spielwelt erkundet werden.

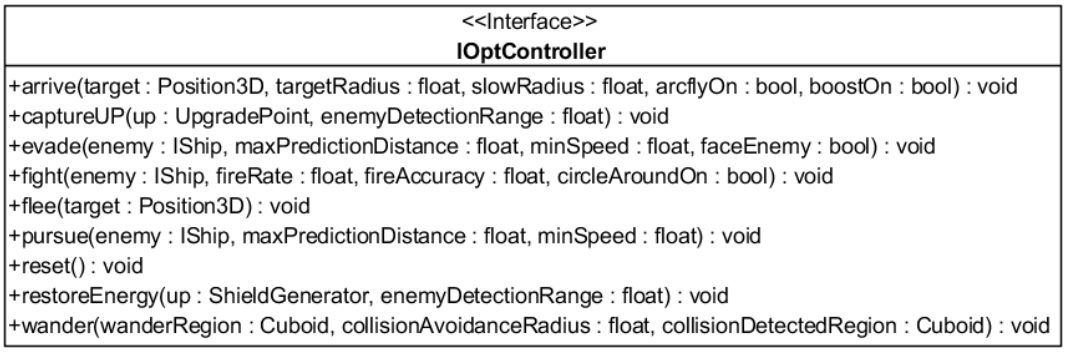

Abbildung 17: Die *IOptController*-Schnittstelle die den Leistungskatalog der operativen Ebene definiert.

Jede dieser sieben Aktionen verfügt dabei über mehrere Parameter die die genaue Aktionsausführung maßgeblich beeinflussen. Abbildung 17 zeigt die Methodensignatur der Aktionen, welche die jeweiligen Parameter einer Aktion definiert. Aus Gründen der Übersicht wurden einige wenige Parameter und genaue Typspezifizierer wie const-, Pointer- oder Referenzangaben ausgelassen. Die genauen Parameterbedeutungen oder mögliche Restriktionen wie Intervall-Beschränkungen können direkt den Code-Kommentaren entnommen werden.

Die angebotenen Aktionen werden zwar wie in Abbildung 15 dargestellt von der *OptController*-Klasse, welche die *IOptController*-Schnittstelle implementiert, bereitgestellt, doch wird die tatsächliche Aktionsumsetzung an die zwei Subsysteme *Steering Pipeline* und *Combat System* weiterreicht. Das Strukturdiagramm aus Abbildung 18 verdeutlicht diese Weiterleitung der Verantwortlichkeiten, wobei die *Steering Pipeline* für die Umsetzung der Bewegungsaktionen *Arrive*, *Evade*, *Flee*, *Pursue* und *Wander* und das *Combat System* für die Kampfaktionen *FightEnemy*, *CaptureUP* und *RestoreEnergy* zuständig ist. Neben der Bereitstellung der Aktionen nach außen ist die Organisation, Verwaltung und Konfiguration der Aktionsausführung mittels ausgewählter Subsysteme eine weitere wichtige Aufgabe der *OptController*-Klasse.

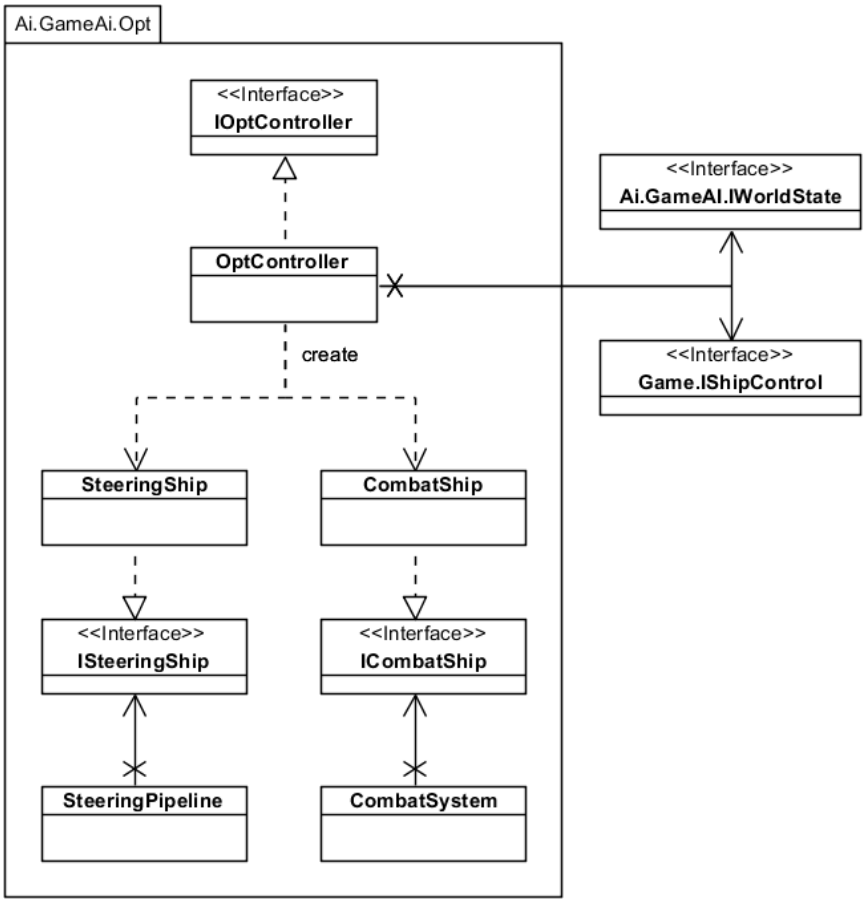

Abbildung 18: Die Struktur der operativen Ebene und die Abhängigkeiten zu anderen Systemebenen.

Zwischen der konkreten *OptController*-Realisierung und den Subsystemen befinden sich keine weiteren Zugriffs-Schnittstellen. Aus diesem Grund ist auf den ersten Blick die Kopplung zwischen diesen als hoch anzusehen. Darüber hinaus stellt es gleichzeitig einen Verstoß gegen das **D**ependency **I**nversion **P**rinciple (DIP) [33] dar, welches die Wiederverwendung von höheren Ebenen mittels gemeinsamer Abstraktionen forciert. Möchte man andere Subsysteme verwenden oder bestehende Implementierungen austauschen ist es unter Umständen ratsamer eine neue *IOptController*-Realisierung zu implementieren. Dieser Nachteil wird allerdings an dieser Stelle bewusst in Kauf genommen. Denn anders als beim Standard-DIP-Fall ist die Logik der höheren Ebene, d.h. die von der *OptController*-Klasse realisierte Logik, vergleichsweise simpel und eine Wiederverwendung entsprechend von geringem Nutzen. Auf der anderen Seite hingegen ist die Wiederverwendung der Subsysteme, die die eigentliche Logik zur Aktionsrealisierung enthalten von bedeutend größerem Nutzen.

Um die Wiederverwendung der Subsysteme zu erleichtern, definieren die Subsysteme mit den Schnittstellen *ISteeringShip* und *ICombatShip* die Informationen oder Steuerungsoperationen die sie benöti-

gen um ihrer Aufgabe nachgehen zu können. Diese Schnittstellen werden von den beiden Klassen *SteeringShip* und *CombatShip* implementiert, die wiederum von der *OptController*-Klasse erstellt werden. Damit ist die *OptController*-Klasse über die Implementierung der Subsystem-Schnittstellen dafür zuständig die ausgewählten Informationen und Steuerungsmöglichkeiten, die ihr über den direkten Zugriff auf die *IWorldState*-Schnittstelle und die *IShipControll*-Schnittstelle zur Verfügungen stehen, an die Subsysteme weiterzuleiten. Diese Designentscheidung entkoppelt die Subsysteme vollständig von *Planet PI4*-Schnittstellen wie *IShipControl* und was im Zuge der Wiederverwendbarkeit noch viel wichtiger erscheint von eigenen Schnittstellen-Kreationen der *Game AI* wie die *IWorldState*-Schnittstelle. Diese Entkopplung erleichtert somit erheblich den Einsatz der Subsysteme in völlig unterschiedlichen *IOptController*-Implementierungen oder gar in vollständig anderen Bot-Implementierung wie z.B. der in Abschnitt 3.5 vorgestellten Vorgänger-Implementierung von Dimitri Wulffert oder zukünftigen Bot-Implementierungen.

5.3.1 Steering Pipeline

Die Aufgabe der *Steering Pipeline* als Subsystem der operativen Ebene ist die Implementierung und Verwaltung von Bewegungsaktionen bzw. von *Steering Behaviors*, die von der *IOptController*-Schnittstelle dem restlichen System zur Verfügung gestellt werden. *Steering Behaviors* wurden in Abschnitt 2.4 ausführlich vorgestellt. Dabei wurden mehrere Möglichkeiten der Realisierung einer solchen *Steering*-Komponente präsentiert, sowie auf etwaige Vor- und Nachteile dieser eingegangen. Der Ansatz der *Steering Pipeline* lässt sich in diesem Kontext der *Arbitration*-Fraktion zuordnen, wobei das größte Unterscheidungskriterium zu anderen *Arbitration* oder auch *Steering*-Ansätzen der kooperative Charakter der einzelnen *Steering Behaviors* untereinander darstellt. Entwickelt wurde der Ansatz erstmals von Marcin Chady und u.a. von Millington und Funge in [35, S. 108-120] ausführlich vorgestellt. Obwohl der Ansatz einer *Steering Pipeline* viele Vorteile bietet, findet er doch in kommerziellen Computerspielen bisher keine nennenswerte Verwendung.

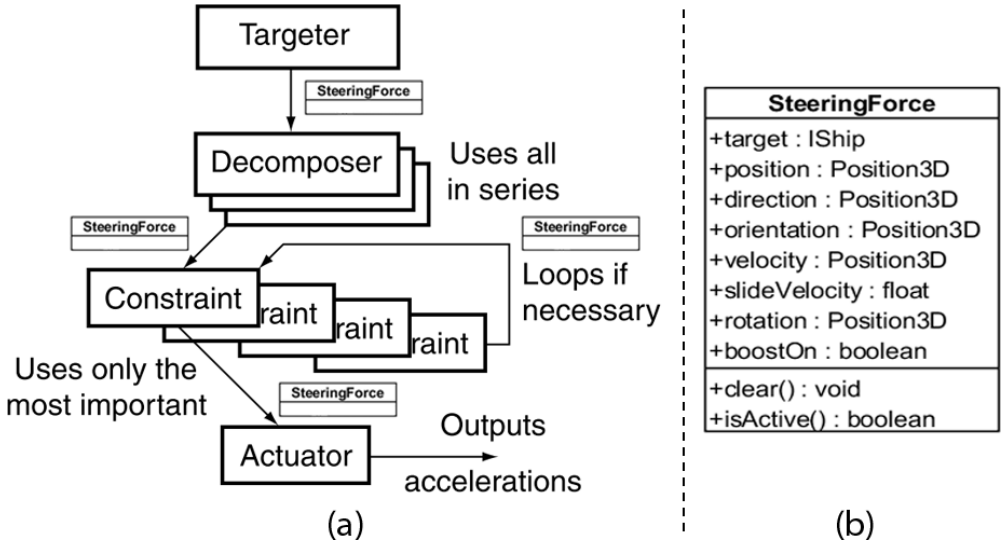

Abbildung 19: (a) Die generelle Struktur der *Steering Pipeline*. Angelehnt an [35, S. 108]. (b) Der Aufbau und die Elemente der *SteeringForce*-Datenstruktur, die von der hier vorgestellten *Steering Pipeline*-Implementierung verwendet wird.

Abbildung 19 (a) zeigt den generellen Aufbau einer *Steering Pipeline* nach Millington und Funge. Demnach besteht die *Steering Pipeline* mit *Targeter*, *Decomposer*, *Constraint* und *Actuator* aus vier wichtigen Komponentenarten und mit *SteeringForce* aus einer zentralen Datenstruktur. Mit Ausnahme der *Actuator*-Komponentenart verbergen sich hinter den anderen Arten die einzelnen *Steering Behaviors*. Die *Steering*

Behaviors werden den einzelnen Komponentenarten zugeordnet und somit der dargestellten hierarchischen Komponenten-Struktur unterstellt. Die Verwendung einer Hierarchie bedeutet nichts anderes als eine Gruppen-Priorisierung der *Steering Behaviors*, wie es für *Arbitration*-Ansätze üblich ist. Das Besondere an der *Steering Pipeline* ist nun, das mit der Verwendung einer gemeinsamen Ergebnis-Datenstruktur (*SteeringForce*) eine Möglichkeit geschaffen wird wie ausgewählte *Steering Behaviors* miteinander kommunizieren können, um somit ihre Ergebnisse untereinander austauschen zu können. Aus diesem Grund beschreiben Millington und Funge den *Steering Pipeline*-Ansatz als „cooperative arbitration approach", dessen einzelne Komponenten und konkrete Realisierungen im Rahmen dieser Master-Thesis im Folgenden näher erläutert werden sollen.

Targeters spezifizieren die konkreten *Steering Behaviors* die von außen aus aufgerufen werden können. Sie sind voneinander unabhängig und pro Spielzyklus sollte jeweils nur ein bestimmter *Targeter* ausgeführt werden, weswegen *Targeters* auch nicht untereinander zu kommunizieren brauchen. *Targeters* definieren sozusagen das Bewegungsziel des Agenten. Abbildung 20 zeigt die von der *Game AI* implementierte *Targeter*-Menge. Ein Vergleich mit der Methodenspezifikation der *IOptController*-Schnittstelle aus Abbildung 17 zeigt, dass hier die durch Methoden repräsentierten Aktionen beinahe 1:1 abgebildet worden sind. Einzig die Existenz des *Face Targeters* mag hier vielleicht auffallend erscheinen, dessen fehlen allerdings insoweit erklärt werden kann, dass der *Face Targeter* ein *Steering Behavior* implementiert, welches als zu rudimentär bzw. *low level* mäßig für die einmalige Ausführung von der taktische Ebene aus angesehen werden kann und deswegen dieser auch nicht angeboten werden muss.

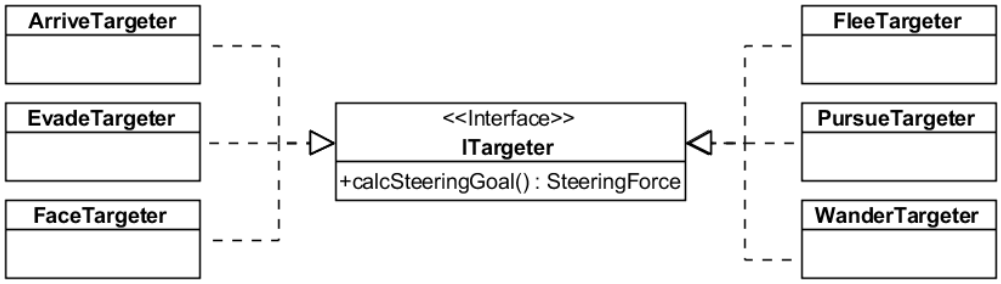

Abbildung 20: Vererbungshierarchie der *Targeter*-Menge der *Steering Pipeline*.

Für die Funktionsweise der *Steering Pipeline* ist die Existenz des *Face Targeters* hingegen von elementarer Bedeutung. Denn obwohl die *Targeter*-Ausführung von der taktischen Ebene aus betrachtet, die über *IOptController* einen indirekten bzw. gekapselten Zugang zur *Targeter*-Ausführung erlangt, als voneinander unabhängig betrachtet wird und pro Spielzyklus sich für jeweils eines der von einem *Targeter* repräsentierten *Steering Behavior* entschieden werden muss, so können die *Targeter* zur Realisierung ihres *Steering Behaviors* jedoch auf beliebige Weise untereinander zugreifen. Dies wird im konkreten Design dadurch realisiert, dass den *Targeters* Zugang zur *SteeringPipeline*-Klasse gewährt wird, welches die Menge der *Targeters* verwaltet. Somit können *Targeters*, wie im ursprünglichen *Steering*-Modell nach Reynolds auf den Ergebnissen untereinander aufbauen und somit durch die Kombination und Erweiterung der Ergebnisse einzelner *Targeters* komplexe *Steering Behaviors* realisieren. Der *Face Targeter* implementiert mit der Ausrichtung des Raumschiffes auf eine bestimmte Position oder ein bestimmtes Objekt ein *Steering Behavior*, welches vom *Arrive Targeter* verwendet wird, das wiederum von fast allen anderen *Targeters* für die eigenen Berechnungen benutzt wird.

Damit die einzelnen *Targeters* auf den Ergebnissen der anderen *Targeters* aufbauen können und damit ebenfalls die *Targeter*-Ergebnisse weiter entlang der *Steering Pipeline* gereicht werden können, wird mit der **SteeringForce**-Datenstruktur fortwährend an einem gemeinsamen Ergebnis gearbeitet. Die *Targeters* führen wie die *Steering Behaviors* von Reynolds nicht die Aktionen direkt aus, sondern sie halten ihre Ergebnisse in einer Instanz der *SteeringForce*-Datenstruktur fest. Aus diesen Grund implementieren alle *Targeters* die in Abbildung 20 dargestellte *ITargeter*-Schnittstelle, die aus folgender Methode besteht:

```
SteeringForce calcSteeringGoal()
```

Anders als bei Reynolds umfasst die *SteeringForce*-Datenstuktur nicht ausschließlich einen einzelnen Beschleunigungs-Vektor, sondern gleich mehrere für die Steuerung des Raumschiffes wichtige Steuerungs-Variablen. Welche das konkret sind kann der Abbbildung 19 (b) entnommen werden. Einige Elemente dienen dabei nicht der Steuerung des Schiffes, sondern eher dem Informationsaustausch zwischen den einzelnen *Targeters*. Ein Biespiel hierfür ist die Berechnung des aktuellen Richtungsvektors zum Ziel (direction), dieser enthält keine Steuerungsanweisung, wird aber von vielen Targerts zur weiteren Berechnung benötigt.

Die Ergebnis-Struktur kann von den einzelnen *Targeters* um jeweils weitere Teilergebnisse erweitert werden. Damit *Targeters* nicht unbewusst bereits gespeicherte Ergebnisse überschreiben, wurde der Datenstruktur ein Mechanismus hinzugefügt, womit überprüft werden kann, ob bestehende Ergebniselemente bereits von anderen *Targeters* gesetzt wurden. Dies wurde realisiert indem jedem Datenelement eine boolsche *isDataElementSet*-Variable hinzugefügt wurde, die allerdings Übersichthalber in der Abbildung weggelassen wurden. Die *isActive*()-Methode überprüft, ob überhaupt eins der Datenelemente gesetzt wurde und somit das Ergebnis gültig ist. Die *clear()*-Methode setzt die boolschen *isDataElementSet*-Variablen wieder auf „false“ zurück.

Decomposers nehmen die Zielvorgabe, repräsentiert durch die Berechnungsergebnisse der *SteeringForce*-Datenstruktur, eines *Targeters* entgegen und versuchen durch Zerlegung in weitere Unterziele die verbesserte und effektivere Umsetzung dieses Ziels. Zum Beispiel könnte ein *Pathfinding Decomposer* zur Routenbestimmung für weit entfernte Punkte auf der Karte eingesetzt werden. Dafür könnte ein *path planning*-Algorithmus oder ein Kürzeste-Wege-Algorithmus eingesetzt werden. Ein weiterer *Decomposer* könnte hingegen für die Routenverfolgung zuständig sein und eventuell auftretende Abweichungen durch das Hinzufügen neuer Zwischenziele korrigieren. Dies ist ebenfalls ein Beispiel dafür, dass die *Decomposer*, wie in Abbildung 19 angedeutet, in einer festen Reihenfolge angeordnet sind und nacheinander ihre Arbeit erledigen.

In der aktuellen *Steering Pipeline*-Implementierung der *Game AI* ist zwar eine *IDecomposer*-Schnittstelle definiert und somit der *Decomposer*-Einsatz vorgesehen, jedoch existieren momentan noch keine konkreten Schnittstellen-Realisierungen. Dieser Umstand ist in erster Line dem bewussten Verzicht auf den Einsatz einer *Pathfinding*-Routine geschuldet. *Pathfinding*-Algorithmen finden zugegebenermaßen in fast jedem (größeren) Spiel Verwendung, dennoch gibt es in *Planet PI4* drei wichtige Gründe die gegen einen Einsatz sprechen:

- Wegen des dezentralen Charakters des Spiels existieren keine globalen Informationen über das aktuelle Level und müssten somit von der *Game AI* selbst erkundet und verwaltet werden.

- Aus dem ersten Grund ergibt sich ein nicht zu unterschätzender Berechnungs- und Speicheraufwand, der zu einer unmittelbaren Verschlechterung der Skalierbarkeit der *Game AI* und somit von *Planet PI4* führen würde. Die in Abschnitt 1.2 formulierte Zielsetzung der Master-Thesis hebt die Skalierbarkeit als ein wichtiges zu erfüllendes Kriterium hervor.

- *Pathfinding* ist immer dann von großen Nutzen, wenn oft statischen Hindernissen ausgewichen werden muss oder sich innerhalb enger verschlungener Innenareale bewegt wird, wo ein reaktives statt vorrauschauendes Ausweichen vor Hindernissen zu neuen Hindernissen oder sogar in Sackgassen führt. In *Planet PI4* ist man hingegen in großflächigen Außenarealen unterwegs, wo vereinzelt Asteroiden den Weg versperren und somit reaktives Ausweichen als ausreichend erscheinen mag.

Vor allem die beiden Letzt genannten Gründe lassen davon ausgehen das der Einsatz von *Pathfinding* in *Planet PI4* in keinem ausreichenden Verhältnis von Aufwand und Ertrag steht, der den Einsatz rechtfertigen würde.

Constraints überprüfen ob die Ziele der *Targeter* bzw. die Unterziele der *Decomposer* realisiert werden können. Jedes *Constraint* untersucht dabei einen bestimmten Aspekt, welcher vom gewünschten Verhalten zur Zielrealisierung nicht verletzt werden darf. Wird von einem *Constraint* solch eine Verletzung erkannt, berechnet sie selbst eine mögliche Lösung. Der Lösungsbildungsprozess beachtet bei der Verhinderung der *Constraint*-Verletzung die übergeordnete Zielvorgabe, versucht demnach die bestmögliche Zielrealisierung ohne dabei den vom *Constraint* repräsentierten Aspekt zu verletzen. Abbildung 19 zeigt das *Constraints* wie *Decomposers* in einer festen Reihenfolge abgearbeitet werden. Somit wird sichergestellt, dass die vorgeschlagenen Lösungen der *Constraints* dahingehend überprüft werden können, ob diese nicht wiederum andere *Constraints* verletzen. Der Unterschied zu *Decomposers* besteht allerdings darin, dass *Constraints* in einer Schleife angeordnet sein können, da solange die Lösung überprüft werden muss bis wirklich alle *Constraints* mit dieser einverstanden sind. Hierbei besteht natürlich die Gefahr einer Endlosschleife, sodass nach einer gewissen Zeit der Schleifendurchlauf abgebrochen werden muss und eine Lösung der *Constraints* oder ein Default-Verhalten ausgewählt wird. Die operative Ebene der *Game AI* implementiert die in Abbildung 21 dargestellten zwei *Constraints*:

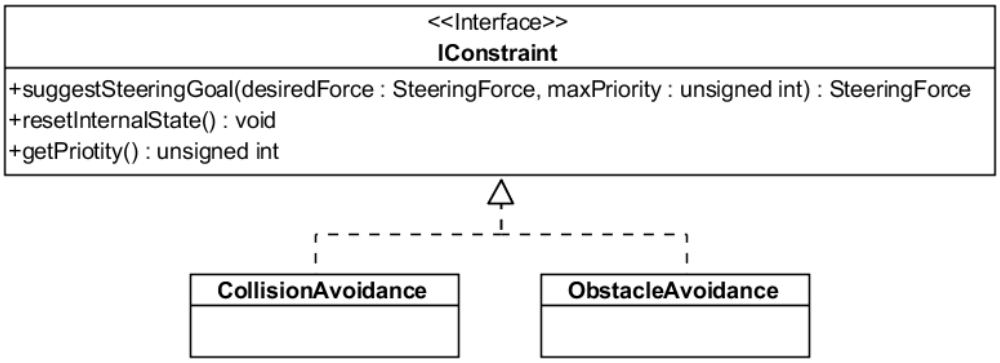

Abbildung 21: Vererbungshierarchie der *Constraint*-Menge der *Steering Pipeline*.

- **Obstacle Avoidance:** Untersucht ob sich in naher Zukunft eine Kollision mit Asteroiden ereignen könnte. Neben Asteroiden existieren mit *Upgrade Points* und Schildgeneratoren noch zwei weitere statische Objektarten. Diese können allerdings ignoriert werden, weil sie jeweils ein masseloses Gebiet bezeichnen, mit dem das Raumschiff nicht kollidieren kann.

  In der aktuellen Implementierung wird zum Ausweichen vor Asteroiden im ersten Schritt die Menge der nahen Asteroiden auf die Menge der Asteroiden in Bewegungsrichtung reduziert. Anschließend wird ermittelt ob bei Beibehaltung der gewünschten Richtung in den nächsten paar Spielzyklen einem dieser Asteroiden zu nahe gekommen wird bzw. eine Kollision geschehen würde. Dem ersten Asteroiden auf denn diese Annahme zutritt wird dann versucht auszuweichen. Dies geschieht in dem nach einem festen Muster sechs mögliche Ausweichrouten um den Asteroiden herum bestimmt werden. Die Ausweichrouten werden dann mit dem ursprünglich gewünschten Positionsziel verglichen. Dazu wird die Entfernung vom Ende der Ausweichroute zum Positionsziel ermittelt und die Ausweichrouten in aufsteigender Entfernungsweite sortiert. Abschließend wird nacheinander geprüft, ob die Ausweichroute auf ihrem Weg eine unmittelbare, d.h. nicht in mehreren Spielzyklen mögliche, Kollision mit einen anderen Asteroiden der zuvor reduzierten Menge verursachen würde. Wenn nicht, wird diese Ausweichroute als neues Bewegungsziel definiert und als Ergebnis zurückgegeben. Wird eine Kollision gefunden wird die nächste Ausweichroute unter-

sucht. Scheitern alle wird eine Default-Bewegung (ein nach links oder rechts driften) eingeleitet, um die Ausgangslage zu verändern und somit vielleicht in naher Zukunft eine gültige Ausweichroute zu bestimmen.

- **Collsion Avoidance:** Ermittelt ob sich in naher Zukunft eine Kollision mit einem anderen Raumschiff ereignen könnte. Dazu wird die Menge der nahen Schiffe untersucht. Dabei wird für alle Schiffe ermittelt ob und, wenn ja, wann diese in naher Zukunft mit dem eigenen Schiff kollidieren könnten. Es wird anschließend dem Schiff versucht auszuweichen, welches an der nächstwahrscheinlichen Kollision beteiligt ist. Wichtig ist an dieser Stelle, dass nicht dem in Bezug auf Entfernung am nächsten kommenden Schiff ausgewichen wird, sondern unter Berücksichtigung der Eigenen Richtung und Geschwindigkeit, sowie der des anderen Raumschiffes, ermittelt wird wann die erste Kollision mit einem Raumschiff sich ereignen könnte und dem entsprechend Gegenmaßnahmen eingeleitet werden. Zur Bestimmung der Ausweichbewegung wird auf die Berechnungen des *Evade Targeters* zurückgegriffen, welche eine Bewegung bestimmt die entgegengesetzt zu der des zu fliehenden Schiffes einleitet. Dadurch das die *Collison Avoidance*-Ausführung nur von kurzer Dauer ist, wird der eigentliche Kurs nur für kurze Zeit verlassen. Anschließend wird der ursprüngliche Kurs weiterverfolgt.

Die beiden *Constraints* implementieren die *IConstraint*-Schnittstelle. Die *suggestSteeringGoal()*-Methode ist für die Untersuchung des aktuellen Ziels, repräsentiert durch den Parameter desiredForce, auf *Constraint*-Verletzungen verantwortlich. Jedes *Constraint* und jeder *suggestSteeringGoal()*-Methodenaufruf ist weiter mit einer Priorität versehen. Damit wird der *Steering Pipeline* ein Mechanismus bereitgestellt um im Konfliktfall der *Constraints* die Berechnung weniger priorisierter *Constraints* zu unterbinden.

Der **Actuator** ist für die endgültige Ausführung der in der *Steering Pipeline* unternommenen Bewegungs-Berechnungen zuständig. Ähnlich der *Locomotion*-Ebene des Modells nach Reynolds aus Abschnitt 2.4 ist die Aufgabe des *Actuators* die Ergebnisse der *SteeringForce*-Datenstruktur zu interpretieren und in Befehle die das Spiel versteht umzuwandeln. Anders als bei den anderen Komponentenarten gibt es pro Charakter nur eine einzige *Actuator*-Instanz. Unterschiedliche *Actuator*-Instanzen können allerdings die Andersartigkeit verschiedener Bot-Typen charakterisieren, da es in *Planet PI4* allerdings nur den Raumschiff-Agententypen gibt, ist dies hier nicht erforderlich. Es wäre aber denkbar das ein Mutterschiff-*Actuator* über erweiterte Steuerungsmöglichkeiten verfügen könnte und deswegen eine andere Umsetzung der *Steering Behavior* in Spielbefehle erfordern würde. Abbildung 22 zeigt die *IActuator*-Schnittstelle und ihre einzige Implementierung *ShipActuator* zur Umsetzung der *Steering Behaviors* in Steuerbefehle eines Raumschiffes. Der einzigen Methode der *IActuator*-Schnittstelle werden dabei beide Ergebnisse übergeben, d.h. das gewünschte Ziel der *Targeter/Decomposer*-Ebene, sowie die eventuell vorhandenen Korrekturmaßnahmen der *Constraint*-Ebene. Es ist die Aufgabe des *Actuators* zu entscheiden welche davon auszuführen ist und in welchem Ausmaß.

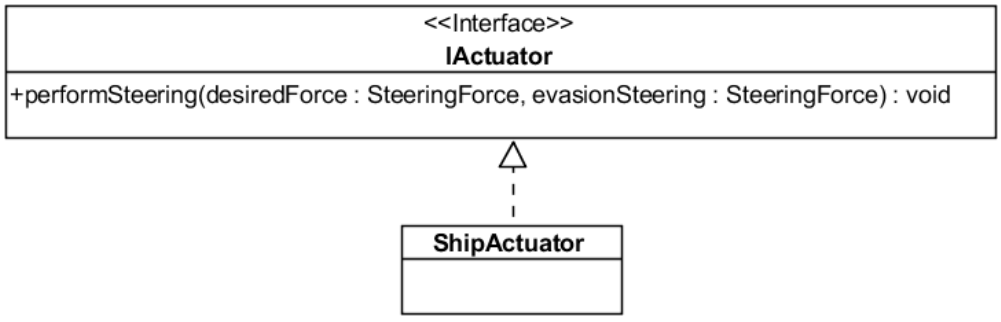

Abbildung 22: Vererbungshierarchie der *ShipActuator*-Klasse der *Steering Pipeline*.

5.3.2 Combat System

Das in Abschnitt 2.4 vorgestellte Modell von Reynolds ordnete zwischen die Aktionsauswahlebene und der Ausführungsebene die *Steering*-Ebene. Die *Steering*-Ebene umfasst dabei ausschließlich Bewegungsaspekte. Die eben beschriebene *Steering Pipeline* kann dabei als eine mögliche Komponente der *Steering*-Ebene nach Reynolds betrachtet werden. In *Planet PI4* existieren jedoch nicht ausschließlich primitive Aktionen zur Bewegungssteuerung. Mit der *fire()*-Methode wird ebenfalls vom Spiel eine Möglichkeit zur Steuerung des Kampfverhaltens angeboten. Mit dem Einsatz des *Combat Systems* wird der Grundgedanke des *Steering*-Modells nach Reynolds, nämlich die Aufteilung des Aktionsbestimmungsprozesses auf drei hierarchisch angeordnete Ebenen, auf andere als zur Bewegungssteuerung beschränkte primitive Aktionen erweitert. Dies ist mit einer der Gründe, weshalb die operative Ebene nicht einfach *Steering*-Ebene genannt werden kann. Denn dafür umfasst sie neben der Berechnung von *Steering*-Verhalten (*Targeters* und *Constraints* der *Steering Pipeline*) und deren (*Locomotion*-)Ausführung (*Actuator* der *Steering Pipeline*) noch den Aspekt der Kampfsteuerung (Combat System).

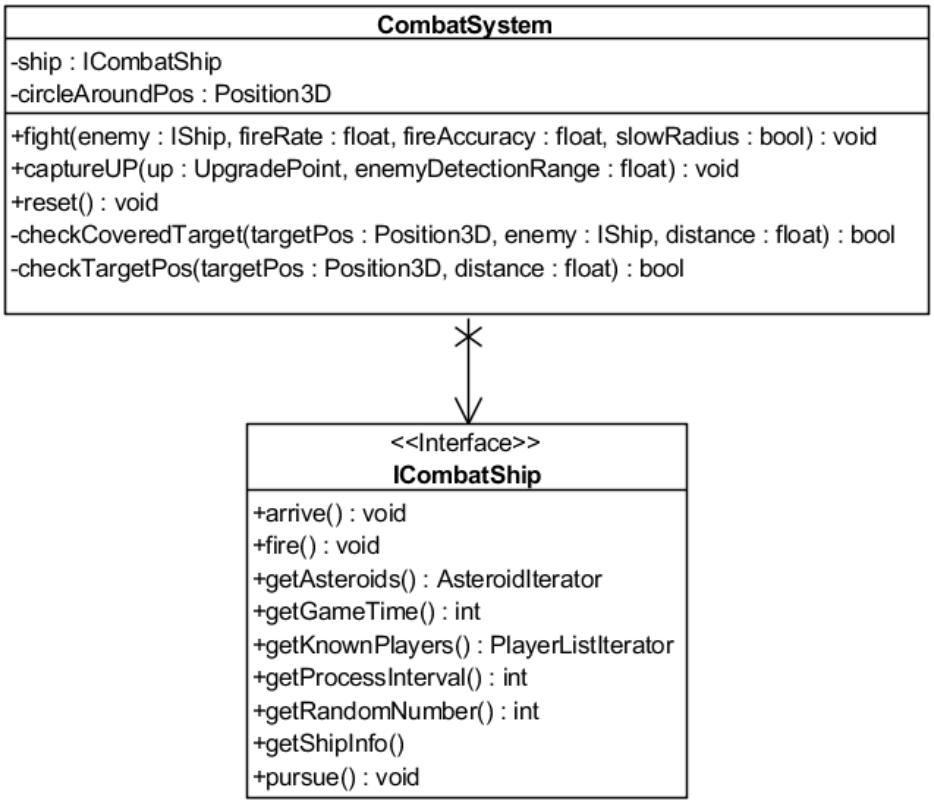

Abbildung 23: Die *CombatSystem*-Klasse und die *ICombatShip*-Schnittstelle, welche die vom *Combat System* benötigten Informationen und Steuerungsbefehle zur Aktionsrealisierung definiert. Die Parameter der *ICombatShip*-Schnittstelle wurden zur besseren Übersicht weggelassen.

Abbildung 23 zeigt das aktuelle Version des *Combat Systems* der *Game AI* folgende zwei Kampfaktionen anbietet, die basierend auf der *fire()*-Methode des Spiels und der Verwendung ausgewählter *Steering Behaviors* der *Steering Pipeline* ein jeweils abstrakteres Kampverhalten spezifizieren als die bloße Ausführung der *fire()*-Methode es zu lassen würde:

- **Capture Upgrade Point:** Ein *Upgrade Point* soll angesteuert und erobert werden. Gegner im unmittelbaren Sichtradius werden entlang des Fluges zum *Upgrade Point* beschossen aber nicht weiter verfolgt. Stattdessen wird der *Upgrade Point* angesteuert. Am *Upgrade Point* angekommen werden solange gegnerische Schiffe innerhalb oder in der Nähe des *Upgrade Points* angegriffen bis dieser

erobert wurde. Wie groß der Kampfradius dabei die Größe des *Upgrade Points* überschreitet ist abhängig vom *enemyDetectionRange*-Parameter der Methode. Der Parameter gibt an um welchen Faktor die Größe des *Upgrade Points* erweitert werden soll. Übergibt man z.B. *enemyDetectionRange* = 1.0 wird nur innerhalb des *Upgrade Points* auf Gegner reagiert. Gibt man einen Wert größer 1.0 an, wird auch außerhalb gekämpft. Gibt man einen Wert nahe oder gleich 0.0 an, wird weder außerhalb noch innerhalb gekämpft. Sollten keine Gegner vorhanden sein wird solange am Mittelpunkt des *Upgrade Points* ausgeharrt bis dieser erobert oder erfolgreich verteidigt wurde. Wichtig ist, dass an dieser Stelle nicht überprüft wird, welchem Team der Upgrade Point angehört oder wann dieser erobert wurde. Die Aktionsauswahlebene muss demnach bestimmen, welcher *Upgrade Point* zu welchem Zweck (Eroberung/Verteidigung) angesteuert wird und die Aktionsausführung in dem Moment unterlassen, wenn dieser Zweck entweder erfüllt oder gescheitert ist. Ein Grund für das Scheitern könnte beispielsweise der eigene Tod und ein weit weg gelegener respawned Punkt sein.

- **Fight Enemy:** Ein gegnerisches Schiff soll bekämpft werden. Ähnlich dem *Capture Upgrade Point*-Verhalten wird erstmals der Gegner angesteuert, falls dieser sich außer Reichweite befindet. Anders als beim *Upgrade Point* handelt es sich hierbei nicht um einen statischen Punkt auf der Karte, welchen man mittels *arrive*-Ausführung ansteuern kann, sondern um ein bewegliches Ziel, dass mittels *pursue*-Anwendung abgefangen werden soll (siehe dazu Abbildung 23).

  Befindet sich der Gegner in Reichweite des Waffensystems wird dieser in Abhängigkeit der Parameterwerte *FireRate* und *FireAccuracy* beschossen. FireRate gibt dabei die Feuerrate an, die definiert in welchem Spielzyklus-Intervall geschossen werden soll. Ein Wert von 1.0 gibt an das man in jedem Spielzyklus feuern soll, wohingegen ein Wert von 0.5 bedeutet, dass man nur durchschnittlich in jedem zweiten Spielzyklus feuert. *FireAccuracy* definiert ab welchem Schusswinkel gefeuert werden soll. Bei einem Wert von 1.0 wird der Gegner beschossen wenn dieser sich gerade am Rand des Sichtfeldes befindet, wohingegen ein Wert nahe 0 bedeutet, dass man nur schießt wenn dieser sich in der Mitte des Sichtfeldes befindet und sozusagen nicht verfehlt werden kann. Die Hilfsmethode *checkCoveredTarget()* überprüft ob das Ziel von einem Asteroiden verdeckt wird und dementsprechend gefeuert werden soll oder nicht. Der Parameter *circleAroundOn* definiert ob am Standard-Verhalten, dem Stoppen des Schiffes und Feuern aus dem Stand, falls die Entfernung zum Ziel zu gering wird, festgehalten wird oder ob stattdessen ein Verhalten aktiviert werden soll, welches den Gegner mit voller Geschwindigkeit trotz nächster Nähe ansteuert und auf die *Collision Avoidance* der *Steering Pipeline* zur Kollisionsverhinderung vertrauend einen Kreis fliegt, um im Anschluss sich wieder auf den Feind zu stürzen.

- **Restore Energy:** Spezifiziert genau das gleiche Verhalten wie *Capture Upgrade Point* nur das nicht ein *Upgrade Point*, sondern ein Schildgenerator angeflogen wird.

5.4 Taktische Ebene

Die taktische Ebene ist die mittlere der drei Hierarchieebenen der *Game AI*-Architektur aus Abschnitt 5.2. Ihre zentrale Aufgabe ist der Aktionsauswahlprozess bzw. das *Decision Making*. Wie der Einleitungstext zum Related Works Kapitel und insbesondere die Auflistung in Tabelle 1 andeuteten existieren zahlreiche *Decision Making*-Techniken. Im Rahmen der Master-Thesis wurde sich für den Einsatz von *Behavior Trees* (BTs) entschieden. Das allgemeine Konzept, sowie Vor- und Nachteile von BTs wurden bereits in Abschnitt 4.4 behandelt.

Abbildung 24 zeigt den allgemeinen Aufbau der taktischen Ebene und die Abhängigkeiten zu anderen Teilen des Systems. Die Struktur der Ebene, wie auch das Aussehen der *ITactController*-Schnittstelle sind

voll und ganz auf den Einsatz von BTs als *Decision Making*-System ausgerichtet. So bietet die *ITactController*-Schnittstelle mit der *performNextAction()*-Methode die Möglichkeit den *Decision Making*-Prozess anzustoßen. Ob dieser erfolgreich verlaufen ist oder nicht gibt das BT spezifische Ergebnis-Enum *BEHAVIOR_STATUS* an, auf welches etwas später noch genauer eingegangen wird. Die beiden Methoden *resetBTStatus()* und *getBlackboard()* geben der strategischen Ebene die Möglichkeit den BT-Status zurückzusetzen und über den direkten Zugriff aufs *Blackboard* gezielt Informationen oder Vorgaben an die *Decision Making*-Komponente weiter zugeben. Mit der *setBehaviorTag()*-Methode wird eine Möglichkeit zum selektiven Ausschluss bestimmter Teilbäume des BTs vom Aktionsauswahlprozess nach außen hin angeboten. Dies ermöglicht eine Erweiterung des allgemeinen BT-Konzeptes auf das ebenfalls später noch näher eingegangen wird. Der so vorgenommene Entwurf der *ITactControler*-Schnittstelle ist zwar einerseits nicht ganz so elegant gelöst wie auf der operativen Ebene, wo sich hinter der *IOptController*-Schnittstelle unterschiedliche Realisierungsmöglichkeiten verbergen könnten, andererseits ist die Definition einer allgemeinen *Decision Making*-Schnittstelle um einiges schwieriger. Die einzelnen *Decision Making*-Techniken unterscheiden sich teilweise sehr stark voneinander und die enge Verzahnung zwischen der strategischen und taktischen Ebene erfordert einen gewissen *White-Box*-Zugriff. Nichtsdestotrotz wäre die weitere Entkopplung und die Verallgemeinerung der *ITactController*-Schnittstelle ein möglicher Ansatzpunkt für zukünftige Refactoring-Maßnahmen.

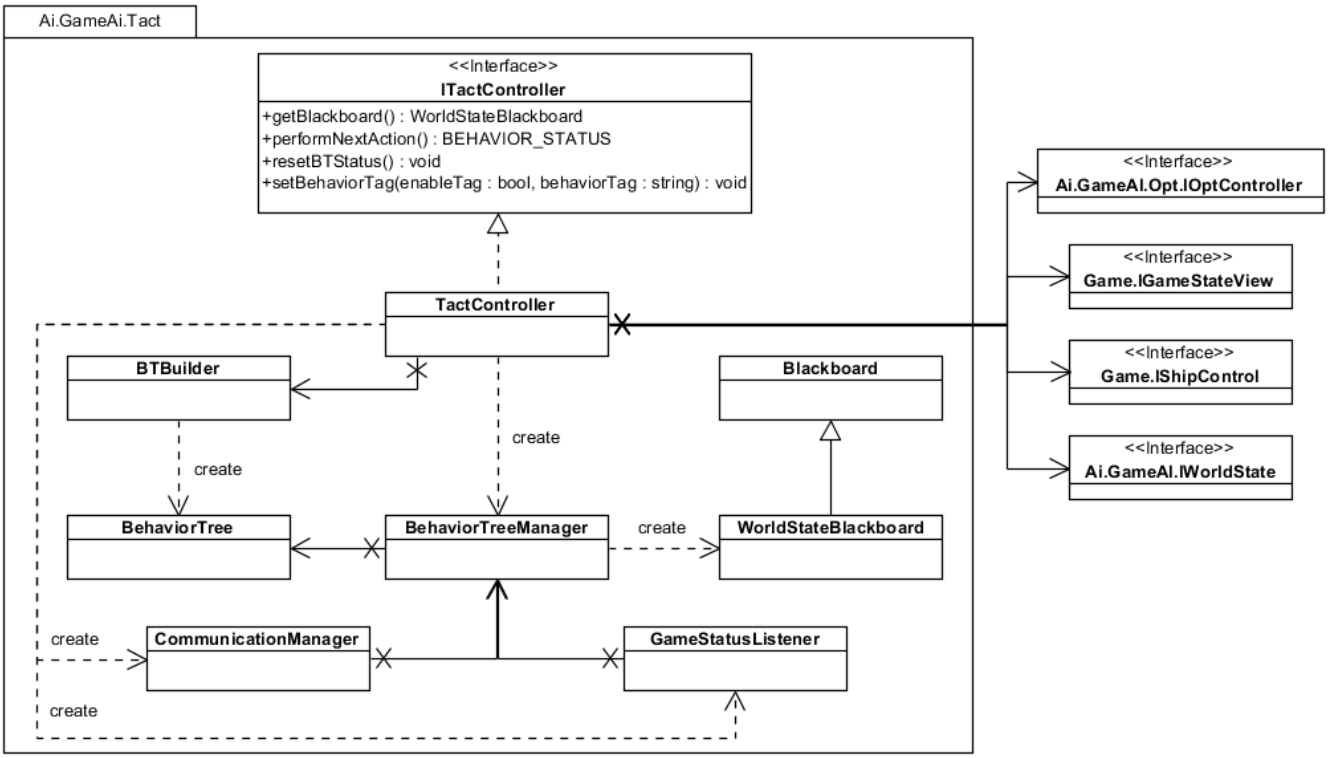

Abbildung 24: Die Struktur der taktischen Ebene und die Abhängigkeiten zu anderen Systemebenen.

Die eigentliche BT-Implementierung besteht aus einem Geflecht aus mehreren Klassen und wird in Abbildung 24 stellvertretend durch die *BehaviorTree*-Klasse repräsentiert. Die tatsächliche Realisierung wird in Abschnitt 5.4.2 ausführlich vorgestellt. Für die folgenden Ausführungen zur allgemeinen Struktur ist jedoch die Vorstellung der BT-Implementierung als ein eigenständiges Subsystem völlig ausreichend.

Für den Bau der konkreten BT-Instanz ist die *BTBuilder*-Klasse zuständig. Dies bedeutet sie definiert und instanziiert den BT-Baum, der letzten Endes von der *Game AI* zur Aktionsbestimmung verwendet wird. Dieser wird in Abschnitt 5.4.4 vorgestellt. Die *BTBuilder*-Klasse wird von der *TactController*-Klasse mit den benötigten Informationen und Schnittstellen die zum Bau des BTs notwendig sind versorgt.

Die *BehaviorTreeManager*-Klasse ist für die Verwaltung und den Funktionszugriff einer BT-Instanz zuständig. Sie sorgt ebenfalls dafür, dass die BT-Instanz Zugriff zu aktuellen Informationen des Spiels erhält und über ein *Blackboard*-Informationen innerhalb des Baums austauschen kann. Erläuterungen zum *Blackboard* sind im nachfolgenden Abschnitt 5.4.1 zu finden. Realisiert wird der Informations- und *Blackboard*-Zugriff dadurch, dass eine Instanz der *WorldStateBlackboard*-Klasse erstellt und diese an die BT-Instanz weitergebenen wird. Die *WorldStateBlackbord*-Klasse verbindet den Zugriff auf die *IWorldState*-Schnittstelle aus dem *AI.GameAI*-Package mit dem Zugriff auf eine vom konkreten Verwendungszweck unabhängige *Blackboard*-Realisierung.

Wie bereits bei der Beschreibung zur *IWorldState*-Schnittstelle erwähnt, enthält die Schnittstelle nicht alle vom Spiel bereitgestellte Informationen. Die *GameStatusListener*-Klasse ist für die Integration der fehlenden Informationen in das *Blackboard* zuständig. Dazu meldet sie sich nach dem *Listener*-Prinzip bei der *IShipControll*- und *IGameStateView*-Schnittstelle für entsprechende eventbasierte Informationen an und wird bei Eintreffen eines Events darüber informiert. Ein Beispiel für ein solches Event ist die Zerstörung des eigenen Schiffes oder die Menge an regenerierten *Shield*-Punkten bei der Benutzung von Schildgeneratoren.

Ein Sonderfall stellt dabei die Kommunikation mit anderen Raumschiffen, insbesondere mit Teamkameraden da. Hier wird ebenfalls ein *Listener*-Konzept zum Empfang von Informationen angeboten. Darüber hinaus gibt es allerdings auch die Möglichkeit selber Nachrichten an andere Spieler zu versenden. Der Kommunikationsaspekt mit anderen Schiffen wurde im aktuellen Stand der *Game AI* noch nicht umgesetzt. Eine mögliche Verwendung wurde allerdings an dieser Stelle vorgesehen. In Kapitel 7 wird auch kurz auf eine mögliche Realisierung und Integration in das Konzept von BTs eingegangen.

5.4.1 Blackboard Architektur

Ein wichtiger Aspekt bei der Realisierung des BT-Ansatzes ist die Beantwortung der Frage, wie der Informationsaustausch zwischen den einzelnen Knoten im Baum realisiert werden soll. Dazu gibt es mehrere Möglichkeiten. Eine Möglichkeit wäre die einzelnen Knoten des Baums zu parametrisieren und die benötigten Daten entlang des Baumes weiterzureichen. Dies ist allerdings nicht ohne weiteres möglich, wenn man eine der Stärken des BT-Ansatzes - die Wiederverwendbarkeit der Teilbäume - nicht einbüßen will. Denn für die Weiterreichung der Daten entlang des Baumes muss explizites Wissen darüber vorliegen, welche konkreten Informationen von den darunterliegenden Ebenen tatsächlich benötigt werden, was wiederum die genaue Kenntnis der eingesetzten Teilbäume voraussetzt. Ein Austauschen der unteren Ebene führt somit unweigerlich zu einer Code-Änderung der oberen Ebene. In [50] wird eine Möglichkeit der BT-Parametrisierung vorgestellt, die mittels Annotationen und einer *Lookup*-Mechanik versucht diesem Problem entgegen zu wirken. Dazu werden beliebig große Teilbäume in unterschiedliche, evtl. verschachtelte, *Scope*-Schichten unterteilt. Innerhalb einer *Scope*-Schicht werden die Parameter der einzelnen BT-Knoten automatisch mit den aktuell bekannten Parameterwerten befüllt. Existiert in der aktuellen *Scope*-Schicht kein passender Wert wird in der nächst höheren *Scope*-Schicht gesucht. Dieses Konzept stellt eine Erweiterung des *Blackboard*-Ansatzes für BTs dar, der den Datenaustausch im Baum über eine externe Schicht ermöglicht. Im Konzept eines Parametrisierbaren BTs könnten *Blackboards* dazu eingesetzt werden um den Daten-*Lookup* der aktuellen Parameterwerte innerhalb einer *Scope*-Schicht zu realisieren.

Das *Blackboard* kann sich als die Informationszentrale oder als den Datenspeicher der BT-Realisierung vorgestellt werden. *Blackboard*-Architekturen werden eigentlich primär zur Koordination des Verhaltens mehrerer autonomer Agenten eingesetzt [21]. Dies wird realisiert in dem alle Agenten Nachrichten über eine gemeinsame Informationseinheit - das *Blackboard* - austauschen. Das *Blackboard* ist neben der Sammlung und Bereitstellung von Informationen auch für deren Synchronisierung zuständig. Ein *Black-*

board kann sich auch als eine Tafel aus dem Klassenzimmer vorgestellt werden, wo die ganze Klasse zusammen an einer gemeinsamen Lösung arbeitet. Diese Tafel ist für jeden zugreifbar und jeder Schüler kann durch Lösen von Teilproblemen einen gewissen eigenen Anteil zum Lösungsprozess beisteuern, der mit der restlichen Klasse geteilt wird. Dieser Verwendungsgedanke des gemeinsamen Lösungsprozesses lässt sich ebenfalls auf BTs übertragen. Stellt man sich die Wurzel eines BT-Baumes als die zu lösende Aufgabe an einer Tafel vor, so zerlegen die einzelnen Teilbäume das Problem in viele kleinere einfacher zu lösende Teilprobleme. Die dabei erzielten Ergebnisse werden im Anschluss daran auf der *Blackboard*-Tafel veröffentlicht, damit an anderer Stelle im Baum anhand der aktuellen Ergebnisse weitere Berechnungen angestellt werden können die zur Lösung des übergeordneten Problems beitragen.

Tatsächlich ist der *Blackboard*-Ansatz die am weitesten verbreitete Art der Integration eines Datenaustauschmechanismus innerhalb von BTs [35, S. 361-365]. Zwar stellt das Konzept der Parametrisierten BTs mit unterschiedlichen *Scope*-Schichten eine Erweiterung des einfachen *Blackboard*-Ansatzes dar, doch wurde sich bei der BT-Realisierung der *Game AI* für den Einsatz des ursprünglichen *Blackboard*-Einsatzes entschieden, ergänzt um ein paar simple Parametrisierungs-Features. Der Grund hierfür ist die Simplizität des allgemeinen *Blackboard*-Ansatzes und des damit verbundenen geringeren Performance-Verlustes der u.a. durch die Verwaltung unterschiedlicher *Scope*-Bereiche anfallen könnte.

In der *Blackboard*-Realisierung der taktischen Ebene werden die Daten anhand des folgenden Musters im *Blackboard* hinterlegt:

```
BlackboardEntry<ValueType>: key:String || value:ValueType || timeStamp:unsigned int
```

Der key-Parameter dient dabei der eindeutigen Identifizierung des *Blackboard*-Eintrags und muss entsprechend vom Typ *unique* sein. Der *timeStamp*-Wert dient zur Ermittlung der Aktualität des entsprechenden Eintrags. Mithilfe der *any*-Datenstruktur aus dem *boost*-Package existieren keine Beschränkungen was den *ValueType* des *Blackboard*-Eintrags anbelangt. Auch die Verwendung von Typspezifizierer wie Pointer, Referenz oder const sind gestattet. Die Unterstützung von const impliziert allerdings das *Blackboard*-Einträge nicht ihren unter value gespeicherten Wert ändern können. Um dennoch eine Unterstützung von sich ändernden *Blackboard*-Einträgen zu gestatten, wird der gesamte Eintrag vom *Blackboard* selbstständig gelöscht und durch einen neuen Eintrag mit dem veränderten value-Wert ersetzt. Die *Blackboard*-Klasse bietet im Großen und Ganzen eine kleine Menge an unterschiedlichen öffentlichen *setData()*- und *getData()*-Methoden, die den Zugriff auf das *Blackboard* steuern.

Diese simple Realisierung eines *Blackboards* weist auf der anderen Seite zwei Nachteile auf. Der größte Nachteil ist das allgemeine Problem des *Blackboard*-Ansatzes, dass durch die Trennung der Datenebene von der Strukturebene des BTs nicht ohne weiteres ersichtlich ist, wie und wo die Daten innerhalb des Baums verwendet werden. Mit anderen Worten der Informationsfluss im BT ist nur schwer zu erkennen. Der zweite Nachteil ist der simplen Realisierung und der Verwendung eines einzigen *Blackboards* im ganzen Baum geschuldet. Denn ohne Kontrollmechanismen oder der Einführung von *Scope*-Schichten können Einträge ohne Kenntnisnahme betroffener Knoten beliebig überschrieben werden. So könnte beispielsweise ein *nextEnemy*-Eintrag von zwei unterschiedlichen Stellen im Baum gesetzt werden und von einem Ausführungsknoten gelesen werden. Dies könnte durchaus ein gewolltes Verhalten im Baum sein, doch ebenso gut könnte dies auch ein ungewollter Fall sein, der zu nicht beabsichtigten und schwer nachvollziehbaren Verhaltensstörungen führen könnte.

5.4.2 Behavior Tree Implementierung

Abbildung 25 zeigt wie die in Abschnitt 4.4 vorgestellten Komponenten eines BTs im System umgesetzt werden. Die gemeinsame Oberklasse aller Komponenten ist die abstrakte *TaskNode*-Klasse. Sie gibt die Implementierung einer *execute()*- und einer *reset()*-Methode vor. Die *execute()*-Methode dient dazu die

Knotenausführung von der aktuellen Knotenart zu abstrahieren und damit einen einheitlichen Zugriff nach außen anzubieten. Mit dem *WorldStateBlackboard*-Parameter wird den einzelnen Knoten der Zugriff auf den aktuellen Spielkontext gewährt, sowie mit dem Zugriff auf ein *Blackboard* der gemeinsame Datenaustausch im Baum ermöglicht. Die Übergabe zur Ausführungszeit, statt zur Initialisierungszeit ermöglicht den Austausch der tatsächlich verwendeten *Blackboard*-Instanz zur Laufzeit. Aktuell wird nur eine einzige *Blackboard*-Instanz für denselben Baum benutzt. Eine spätere Erweiterung zur Verwendung mehrerer oder unterschiedlicher *Blackboard*-Instanzen um z.B. ein *Scoping* wie in Abschnitt 5.4.1 vorgestellt zu realisieren, wäre damit zumindest erheblich erleichtert. Nach der Ausführung der *execute()*-Methode wird mit der Rückgabe eines Elements des *BEHAVIOR_STATUS*-Enums die Knotenausführung bewertet. Die jeweiligen Enum-Elemente haben dabei folgende Bedeutung für die Traversierung des Baums:

- **BS_SUCCESS:** Die Ausführung war erfolgreich. Fahre je nach aktueller Ausführungssemantik im Baum weiter fort.

- **BS_FAILURE:** Die Ausführung ist fehlgeschlagen. Fahre je nach aktueller Ausführungssemantik im Baum weiter fort.

- **BS_ERROR:** Während der Ausführung ist ein schwerwiegender Fehler aufgetreten. Die BT-Ausführung wird teilweise oder komplett eingestellt.

- **BS_RUNNING:** Der Knoten konnte seine Ausführung innerhalb des aktuellen Ausführungszyklus nicht beenden. Die BT-Traversierung wird angehalten und gegebenenfalls im nächsten Zyklus an derselben Stelle fortgesetzt. Dies kann passieren, wenn die Ausführung mehr als einer *IOptController*-Aktion versucht wird. Dabei handelt es sich um eine inhärente Condition des Baums und muss nicht im Baum via *Condition*-Knoten extra modelliert werden.

- **BS_IGNORE:** Bedeutet das die Ausführung des aktuellen Knoten unterbunden oder ausgelassen wird. Wird verwendet in Kombinationen mit *Behavior Tags*, die eine Erweiterung des BT-Konzeptes darstellen und in Abschnitt 5.4.3 näher erläutert werden.

CompositeTaskNode

Wichtig ist im Kontext der Traversierung von BTs zu verstehen, dass die Statusmeldung *BS_FAILURE* keine Fehlermeldung darstellt. Eine *BS_FAILURE*-Statusmeldung bezeichnet fehlgeschlagene Ausführungen, wie z.B. die Nicht-Erfüllung einer *Condition* im Baum, die je nach aktiver Traversierungssemantik unterschiedliche Auswirkungen nach sich ziehen kann. Die Traversierungssemantik wird dabei von den *CompositeTaskNode*-Knoten oder kurz *Composite*-Knoten bestimmt. *Composite*-Knoten können andere *TaskNode*-Knoten über die *addChild()*-Methode als Kinder aufnehmen, was die hierarchische Anordnung der Knoten im Baum ermöglicht. Weil ein *Composite*-Knoten andere Knoten als Kinder aufnimmt, bestimmt diese auch die Ausführungsreihenfolge der Kinder und wie die entsprechenden Statusmeldungen dieser interpretiert werden sollen.

Eine allgemeine Handhabung über alle *Composite*-Knoten hinweg, erfolgt beim Umgang mit der *BS_RUNNING*-Statusmeldung. Dazu wird mit dem *currentPos*-Attribut der Index des aktuell aktiven Kindes abgespeichert, damit im Falle einer Fortführung im Nächsten Zyklus der Kindknoten direkt ausgeführt werden kann.

Die Art der Traversierung ist der Punkt indem sich die unterschiedlichen Ableitungen der abstrakten *CompositeTaskNode*-Klasse voneinander unterscheiden. Neben der Realisierung der bereits in Abschnitt 4.4 vorgestellten Traversierungsarten eines *Selectors* und einer *Sequenz* wurde zusätzlich noch ein *Personality Selector* entwickelt. Ein *Personality Selector* sortiert die Kinder nach einer *Personality*-Evaluierung

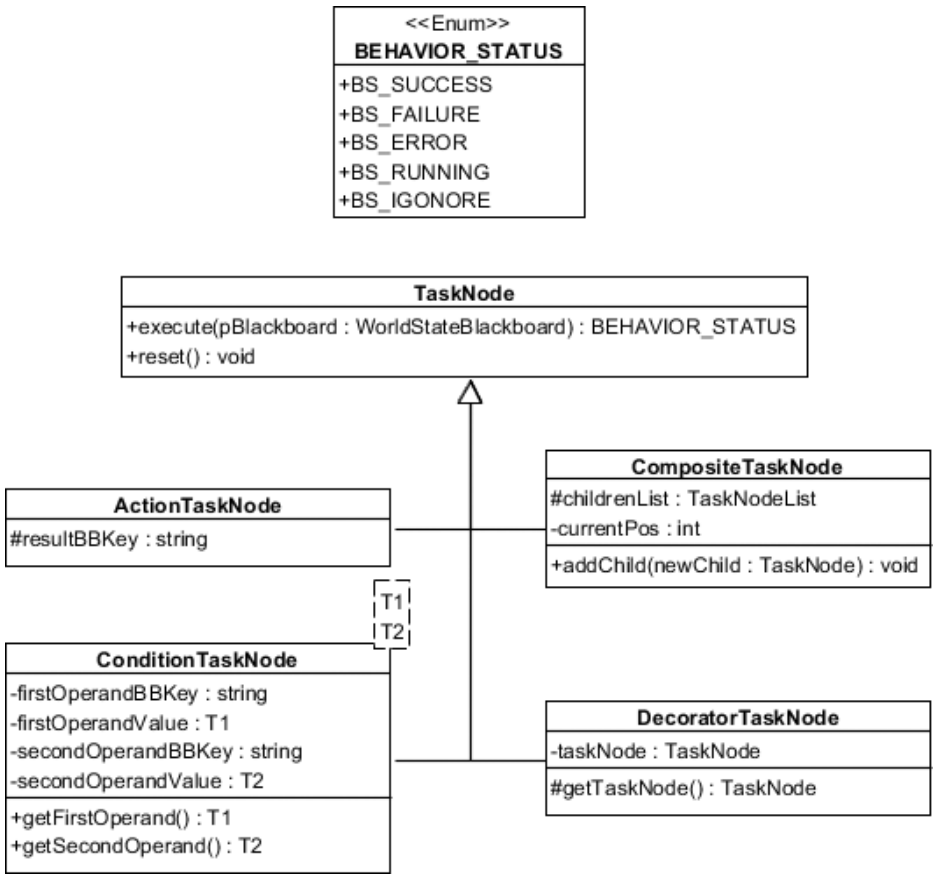

Abbildung 25: Klassendiagramm der wichtigsten übergeordneten Komponenten der BT-Implementierung.

und führt diese entsprechend dieser Reihenfolge aus. Der *Personality Selector* ist nur ein Teil einer Erweiterung des allgemeinen BT-Konzeptes und wird in Abschnitt 5.4.3 ausführlicher erläutert.

ActionTaskNode

Die Ausführung von Aktionsknoten führt entweder zu Änderungen des Weltzustands bzw. des Agentenzustands oder andererseits zu Änderungen des internen Zustands des BTs. Alle Aktionen erben von der abstrakten Klasse *ActionTaskNode*. Die abstrakte Klasse gibt die Verwendung eines *resultBBKeys*-Strings vor, welche mit der Ausführung der *storeResult()*-Methode innerhalb der *execute()*-Methodenausführung die Speicherung des Aktionsergebnisses als Eintrag im *Blackboard* realisiert. Das Ergebnis einer Aktion hängt von der Aktionsart ab, die wiederum durch die Art der Zustandsänderung bestimmt wird.

Die Aktionen, die den Weltzustand bzw. den Zustand des Agenten ändern, werden innerhalb ihrer Ausführung durch die *IOptController*-Schnittstelle bereitgestellten Aktionen der operativen Ebene ausgeführt. In diesem Sinne definiert die taktische Ebene folgende fünf Aktionsgegenstücke zur operativen Ebene: *CaptureUPAction*, *FightEnemyAction*, *EvadeAction*, *WanderAction* und *RestoreShieldAction*. Die Aktionen verfügen über eine Reihe von *Setter*-Methoden, die die einzelnen Parameter der operativen Aktionsgegenstücke entgegen nehmen. Dabei können Parameter-Werte entweder die Werte direkt enthalten oder stattdessen ein *Blackboard*-Key spezifizieren. Alle Parameter die als *Blackboard*-Key angegeben worden sind, werden vor der Ausführung des *IOptController*-Aktionsgegenstücks in der *execute()*-Methode neu vom *Blackboard* geladen. Da diese Form von Aktionen tatsächliches Agenten-Verhalten produziert und somit nicht im *Blackboard* als Aktionsresultat hinterlegt werden kann, speichert diese Art von Aktionen *Blackboard*-Einträge die das Aktionsverhalten dokumentieren. Beispielsweise speichert die *FightEnemyAction* die *IShip*-Instanz des zuletzt angegriffenen Schiffes ab, welche wiederum an einer anderen Stelle im BT abgefragt werden kann. Tatsächlich geschieht genau dies in der aktuellen BT-Instanz die in

Abschnitt 5.4.4 vorgestellt wird. Dort wird im jeweils nächsten Aktionszyklus möglicherweise geprüft, ob das letzte angegriffene Schiff noch in Reichweite ist und dementsprechend weiter angegriffen oder stattdessen ein neues Ziel ausgewählt werden soll.

Aktionen die den internen Zustand des BTs ändern rufen keine Aktionen der *IOptController*-Schnittstelle auf. Momentan sind folgende drei *Determine*-Aktionen implementiert, die die Spielwelt auf bestimmte Objekte untersuchen und das Ergebnis im *Blackboard* abspeichern Sollte kein Ergebnis gefunden werden wird ein *BS_FAILURE* zurückgegeben:

- **DetermineNearestShipInRangeAction:** Untersucht die nahe Nachbarschaft auf das nächste feindliche und/oder befreundete Schiff und speichert eine Referenz der *IShip*-Instanz im *Blackboard* ab. Ein Enum-Parameter gibt dabei die gewünschte Schiffsart an und ein *Range*-Parameter den Suchraum.

- **DetermineNearestUPInRegionAction:** Untersucht die Region auf das nächste eroberte und/oder nicht eroberte UP-Vorkommen und speichert eine Referenz der *Upgrade Point*-Instanz im *Blackboard* ab. Ein Enum-Parameter gibt dabei den gesuchten Eroberungsstatus an und ein anderer die Suchregion.

- **DetermineNearestSGInRangeAction:** Untersucht die nahe Nachbarschaft auf das nächste SG-Vorkommen und speichert eine Referenz der Schildgenerator-Instanz im *Blackboard* ab. Ein *Range*-Parameter gibt dabei den gewünschten Suchraum an.

ConditionTaskNode

Aktionsausführungen können Zustandsänderungen der Welt, des Agenten oder des BT herbeiführen. *Constraints* stellen hingegen Abfragen und Bedingungsformulierungen auf genau dieser Zustandsmenge dar. Jeder *ConditionTaskNode*-Knoten oder kurz *Condition*-Knoten erbt dabei von der abstrakten Klasse *ConstraintTaskNode*, die wie die anderen abstrakten Klassen der BT-Implementierung die Realisierung der *TaskNode.execute()*-Methode an ihre Spezialisierungen weitergibt. Zusätzlich definiert sie allerdings das jedes *Contraint* aus min. zwei Operanden besteht, deren Verwaltung in der abstrakten Klasse selber realisiert ist. Wie bei der Parametrisierung von Aktionen können die *Contraint*-Operanden entweder jeweils direkt den Wert eines beliebigen Typs T1 bzw. T2 annehmen oder einen *Blackboard*-Key enthalten, der bei jedem neuen Aufruf der *Getter*-Methode für ein neu Laden des im *Blackboard* hinterlegten Wertes genutzt wird. Abbildung 26 zeigt welche *ConstraintTaskNode*-Ableitungen im System existieren:

- **ArithmeticCondition:** Hilfsklasse die zwei Zahlen vom gleichen arithmetischen Typ übernimmt und verschiedene Vergleichsoperatoren (<,>,>=,<=,==,!=) auf diese anwendet.

- **ShipStatusCondition:** Prüft entweder die aktuelle Schiffsenergie, den *Health*- oder den *Shield*-Wert des Schiffs gegen einen arithmetischen Zahlenwert unter Anwendung eines arithmetischen Vergleichoperators.

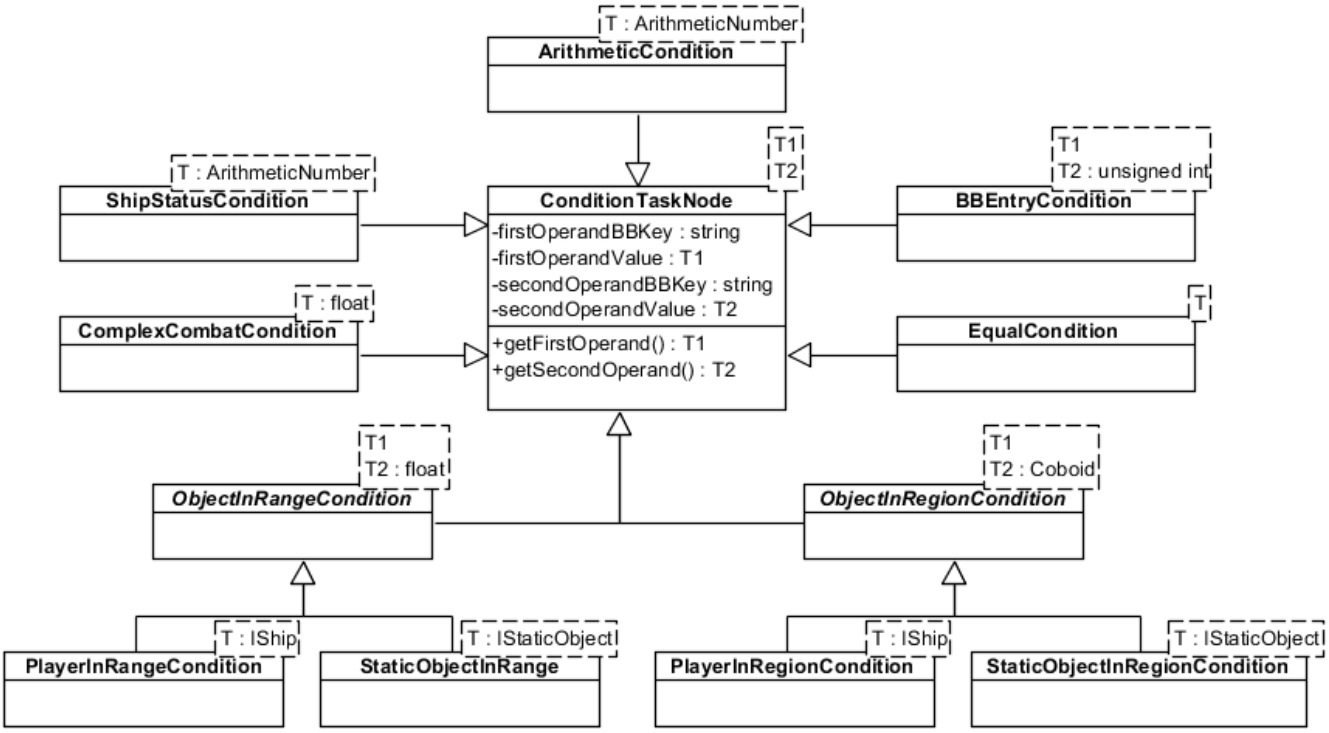

Abbildung 26: Vererbungshierarchie und Typspezifikation der *Conditions* der BT-Implementierung.

- **ComplexCombatCondition:** Prüft den *Complex Combat*-Value, der die aktuelle Schiffsenergie, den *Health*-Wert und ein *Team/Enemy*-Verhältnis umfasst, gegen einen arithmetischen Zahlenwert unter Anwendung eines arithmetischen Vergleichsoperators. Durch die Angabe von Gewichten ist die Priorisierung der drei Faktoren möglich. Die genaue Berechnungsformel des *Complex Combat*-Values lautet dabei wie folgt:

$$CC\ Value = \frac{\frac{CHV}{MHV} * HW + \frac{CEV}{MEV} * EW + \mu(NTC, NEC) * TERW}{HW + EW + TERW}$$

$$\mu(a,b) \quad \begin{cases} > 0.7 \text{ , falls a} > \text{b} \\ = 0.7 \text{ , falls a} = \text{b} \\ < 0.7 \text{ , falls a} < \text{b} \end{cases}$$

wobei folgendes gilt,

$$\begin{aligned} CHV &= Current\ Health\ Value \\ MHV &= Max\ Health\ Value \\ CEV &= Current\ Energy\ Value \\ MEV &= Max\ Energy\ Value \\ NTC &= Near\ Team\ Member\ Count \\ NEC &= Near\ Enemy\ Count \\ HW &= Health\ Weight \\ EW &= Energy\ Weight \\ TERW &= Team/Enemy\text{-}Ratio\ Weight \end{aligned}$$

- **BBEntryCondition:** Prüft auf Existenz oder Aktualität eines *Blackboard*-Eintrags. T1 spezifiziert dabei den Typ des Eintrags und T2 ist vom *unsigned int*-Typ des *timeStamp*-Parameters. Die Überprüfung ist notwendig da *Determine*-Aktionen nicht zwingend ein Ergebnis produzieren müssen oder das letzte Ergebnis entsprechend lange zurück liegen kann. Beispielsweise kann die *DetermineNearestShipInRange*-Aktion keine anderen Schiffe in der Nähe finden und legt dementsprechend keinen neuen Ergebnis-Eintrag im *Blackboard* ab.

- **EqualCondition:** Prüft beliebige Objekte vom nicht arithmetischen Typ T auf Gleichheit.

- **ObjectInRangeCondition:** Abstrakte Klasse die Objekte vom beliebigen Typ T1 entgegen nimmt, ihre Entfernung zum eigenen Raumschiff bestimmt und gegen einen *float*-Zahlenwert prüft. Dabei wird in der *execute()*-Methode der *Condition* das *Template Method-Pattern* [16, S. 366-372] angewendet, welche die Berechnung und den Vergleich selber definiert aber die dabei zu verwendete Positionsangabe des Objektes an seine Unterklassen weiterleitet. Die dabei zu implementierende Methode ist folgende:

  ```
  Position3d getObjectPos(WorldStateBlackboard* pBlackboard);
  ```

  Die zwei Spezialisierungen *PlayerInRangeCondition* nimmt ein beliebiges Schiff entgegen und *IStaticObjectInRangeCondition* ein *IStaticObjekt*-Objekt, welches entweder ein Asteroid, *Upgrade Point* oder *ShieldGenerator*-Objekt sein darf.

- **ObjectInRegionCondition:** Gleiches Prinzip wie bei der *ObjectInRangeCondition* nur, dass nicht gegen einen *float*-Zahlenwert geprüft wird, sondern ob die Position des Objektes sich innerhalb oder außerhalb der übergebenen Region befindet.

DecoratorTaskNode

DecoratorTaskNode-Knoten bzw. kurz *Decorator*-Knoten sind nach dem *Decorator*-Patten [16, S. 199-211] entworfen, welches eine Alternative zur Unterklassenbildung darstellt, um die Funktionalität einer Klasse zu erweitern, ohne dabei Modifikation am bestehenden Code vorzunehmen. Dazu wird die zu erweiternde Klasse mit der neuen Klasse umwickelt, indem diese dasselbe Interface implementiert und die zu erweiternde Klasse als Klassenelement entgegen nimmt. Von außen sind beide Klassen nicht voneinander zu unterscheiden. Ein beliebtes Beispiel ist das dekorieren von GUI-Fenstern. Eine GUI-Fenster Klasse wird beispielsweise mit einem Rahmen-*Decorator* versehen. Immer wenn das Fenster gezeichnet werden soll, fügt der vorgeschaltete *Decorator* einen Rahmen um das Fenster hinzu. Zur eigentlichen Darstellung des Fensterinhaltes ruft der Rahmen-*Decorator*, nach dem Zeichnen des Rahmens, die entsprechende *draw()*-Methode der Fenster-Klasse auf. Eine ähnliche Verwendung schlagen Champandard in [40] oder Millington und Funge in [35, S. 345-351] für BTs vor.

Ein *Decorator*-Knoten nimmt in diesem Sinne genau einen anderen *TaskNode*-Knoten auf, implementiert selber die *TaskNode*-Schnittstelle des zu dekorierenden Objektes und fügt dieser zusätzliche Funktionalität hinzu. Ein beliebtes Beispiel für einen *Decorator* ist ein *Limit Decorator* der die Knoten-Aufrufe des zu dekorierenden Knotens zählt und ab einem bestimmen Limit die weitere Ausführung verhindert. Mit dem *Behavior Tag Decorator* und dem *Personality Decorator* existieren im *Game AI*-System genau zwei unterschiedliche *Decorators*. Beide *Decorators* sind allerdings Bestandteil jeweils einer eigens entwickelten Erweiterung des allgemeinen BT-Prinzips, auf das im nachfolgenden Abschnitt gesondert eingegangen wird.

5.4.3 Behavior Tags und Personality Selection

Behavior Tags sind ein nicht näher spezifizierter *Prunning*-Mechanismus in Halo 2 [20], der dazu angewandt wird bestimmte Bereiche des Baumes ein- und auszuschalten. Im einen Baum könnte beispielswei-

se die Verhaltenslogik mehrerer Bot-Typen kodiert sein, wie z.B. Fahrer, Beifahrer oder Fußgänger. Ein Fußgänger benötigt allerdings die im Baum enthaltene Logik zum Steuern eines Fahrzeuges nicht, sodass diese entsprechend weggelassen werden kann. Was die Größe des Baumes reduziert und gleichzeitig die Traversierungsgeschwindigkeit erhöht. In Halo 2 sind *Behavior Tags* als Bit-Vektoren kodiert die eine partielle Zustandsbeschreibung des Agenten angeben, die einem Knoten zugewiesen und zur Laufzeit mit dem aktuellen Agentenzustand verglichen werden kann. Wird die partielle Zustandsbeschreibung vom *Behavior Tag* vom aktuellen Agentenzustand erfüllt bleibt der Teilbaum aktiv, ansonsten wird er deaktiviert bzw. geprunnt.

Diese Art von *Behavior Tags* wird nicht 1:1 in der *Game AI* auf der taktischen Ebene umgesetzt, der Grundgedanke des *Prunnings* von bestimmten Teilbäumen bleibt allerdings erhalten. Dazu wird in erster Linie mithilfe des *Behavior Tag Decorators* einem beliebigen Knoten im BT ein String-Tag zugewiesen, wie dies beispielsweise in Abbildung 27 bei den Knoten *FightInRegion*, *FightNotInRegion*, *DaredavelTactic*, *TactismTactic* und *ScardyCatTactic* der Fall ist. Der Tag steht zwar in der Abbildung jeweils am Knoten, jedoch ist dies nur eine vereinfachte visualisierte Darstellung dafür, dass an dieser Stelle eigentlich zwei Knoten stehen müssten. So besteht beispielsweise der *FightInRegion*-Knoten in Wirklichkeit aus dem vorgeschalteten Dekorierer-Knoten mit dem Tag „Tag.FIR“. Erst darunter würde als Kindknoten der zu dekorierende *FightInRegion*-Knoten kommen.

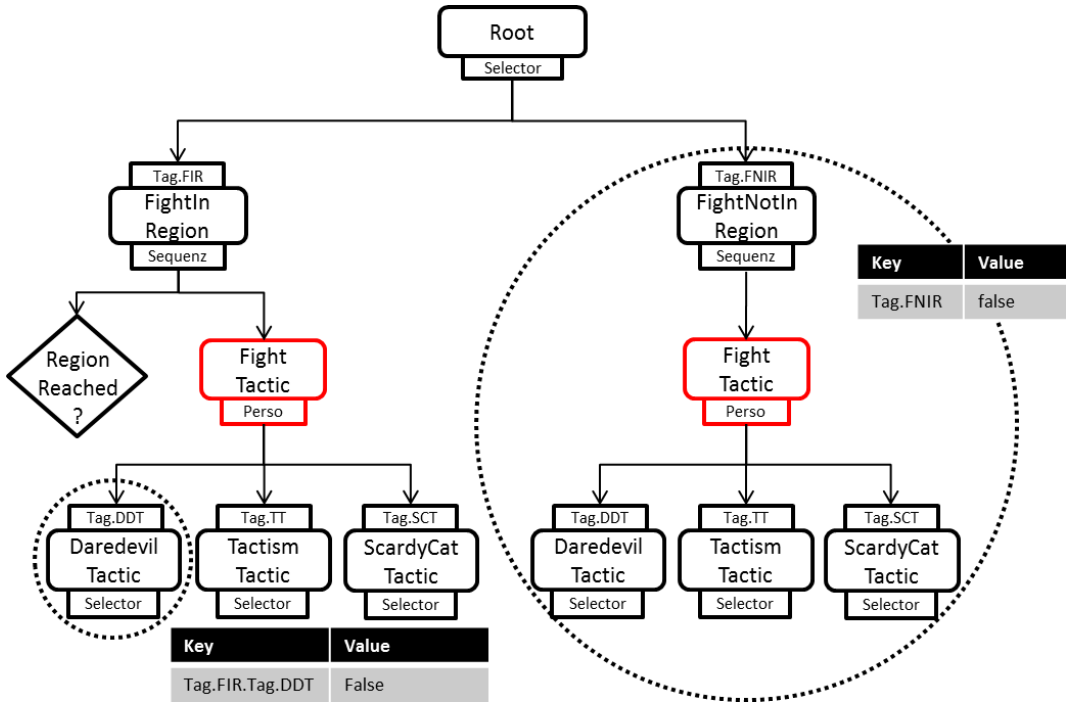

Abbildung 27: Beispielhafter Einsatz von *Behavior Tags*.

Die Aufgabe des *Decorator*-Knoten zur Ausführungszeit ist es vor der Ausführung seines Kindes im *Blackboard* nachzuschauen, ob es einen Eintrag mit dem String-Tag als Key gibt. Existiert solch ein Key und entspricht der Value-Eintrag dem boolschen „false“-Wert, so wird ein *BS_IGNORE* an den Elternknoten des *Decorators* zurückgegeben und die aktuelle Ausführung des Unterbaums unterbunden. Wird kein Eintrag oder einer mit dem boolschen „true“-Wert gefunden, so wird der aktuelle Tag-Pfad im *Blackboard* abgespeichert und der Unterbaum entsprechend ausgeführt. Die Speicherung des aktuellen Pfades ermöglicht das *Prunning* bestimmter Teilbäume in Abhängigkeit der durch den Baum und *Behavior Tag Decorator* aufgebauten Tag-Hierarchie. Möchte man in Abbildung 27 beispielsweise den linken *Daredevil*-Knoten entfernen, nicht jedoch den rechten im *FightNotInRegion*-Teilbaum, so kann dies mit folgendem *Blackboard*-Eintrag realisiert werden:

```
BlackboardEntry<bool> = {"‘Tag.FIR.Tag.DDT"’, false}
```

Möchte man hingegen einen ganzen Unterbaum prunnen, z.B. den in Abbildung 27 rechten *FightNotInRegion*- Unterbaum, so reicht folgender *Blackboard*-Eintrag:

```
BlackboardEntry<bool> = {"‘Tag.FNIR"’, false}
```

Ein Sonderfall existiert wenn ein Knoten überall im Baum entfernt werden soll, unabhängig der aktuellen Hierarchie. Realisiert wird dies dadurch, dass ein jeder *Decorator*-Knoten neben dem aktuellen Pfad + Knoten-Tag zuerst einmal nur nach einem *Blackboard*-Eintrag mit seinem Tag sucht. So würde beispielweise der folgende *Blackboard*-Eintrag alle beide *DaredevilTactic*-Knoten aus dem Baum entfernen:

```
BlackboardEntry<bool> = {"‘Tag.DDT"’, false}
```

Das Setzen von *Behavior Tag*-Einträgen ins *Blackboard* wird über die *ITactController*-Schnittstelle bzw. über deren *setBehaviorTag()*-Methode gesteuert und somit auch der Strategischen Ebene zur Verfügung gestellt.

Die Motivation für die **Personality**-Erweiterung ist der Umstand, dass Spieler unterschiedlich auf ein und dieselbe Spielsituation reagieren können. Während beispielsweise Spieler A bei einer Übermacht feindlicher Spieler direkt die Flucht ergreifen würde, könnte Spieler B eine Herausforderung in der Übermacht sehen und sich blindlings in den Kampf stürzen. In BTs können solch unterschiedliche Verhaltensweisen für ein und dieselbe Situation modelliert und in den Baum integriert werden. Das Problem dabei ist jedoch, dass im allgemeinen Modell bei Alternativen die Verwendung von Selektoren vorgesehen ist. Selektoren gehen solange die Menge der Alternativen von links nach rechts durch bis die erste erfolgreich war. Diese Vorgehensweise führt dazu, dass weiter rechts modellierte Alternativen evtl. niemals ausgeführt werden, weil die weiter links stehenden immer erfolgreich sind und somit stattdessen ausgeführt werden. Ein weiterer Nachteil ist die immer gleiche Traversierungsreihenfolge derselben BT-Instanz. Es wird eben immer von links nach rechts gegangen ganz egal welche Art von Bot-Spieler simuliert wird. Um diesen Nachteilen entgegenzuwirken wurde mit dem *Personality Selector* ein *CompositeTaskNode* definiert der eine neue Art der Traversierung im Baum ermöglicht.

Der *Personality Selector* berechnet den *Personality Value* all seiner Kinder und sortiert diese danach in absteigender Ordnung. Anschließend werden nacheinander solange die sortierten Kinder ausgeführt bis das erste erfolgreich war oder bis alle fehlgeschlagen sind. Damit der *Personality Value* der Kinder berechnet werden kann, müssen die Kinder zunächst mit dem *Personality Decorator* dekoriert worden sein. Abbildung 28 zeigt die wichtigsten Klassenelemente beider *Personality*-Komponenten.

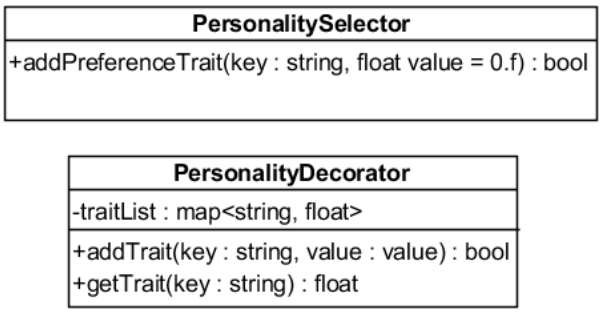

Abbildung 28: Die Elemente der *Personality*-Erweiterung.

Der *Personality Decorator* fügt einem Knoten *Personality Traits* hinzu. Ein *Personality Trai*t ist dabei ein (String,Value)-Paar welches eine Charaktereigenschaft oder einen Gemütszustand des Agenten quantifiziert. Es können einem Knoten beliebig viele, aber durch den String-Key eindeutig identifizierbare, *Personality Traits* über die *addTrait()*-Methode hinzugefügt werden. Die Werte sind dabei vom Typ *float*

und müssen im Intervall [0,1] liegen. In dem Moment, wo ein *Personlity Trait* einem Knoten über den Dekorator hinzufügt wird, findet eine Bewertung des modellierten Verhaltens des zu dekorierenden Knotens statt. Stellt man sich beispielsweise einen Aggressivitätswert vor der einem Knoten mit dem Wert 1.0 zugewiesen wird. Dies würde bedeuten das Verhalten das durch den Knoten realisiert wird ist als aggressiv zu bewerten und sollte von aggressiven Charakteren bevorzugt werden.

Dem *Personality Selector* können ebenfalls *Personality Traits* über die *addPreferenceTrait()*-Methode hinzugefügt werden. Jedoch besitzen diese *Traits* hierbei eine andere Bedeutung. *Personality Traits* bewerten nicht das vom Unterbaum des *Personality Selectors* modellierte Verhalten, wie dies bei *Personality Decorators* der Fall ist, sondern sie definieren welche *Personality Traits* bei der Verhaltensauswahl berücksichtigt werden sollen und wenn ja mit welchem Gewicht dies getan werden soll. Das Besondere an dieser Stelle ist, dass zwar die zu beachtenden *Personality Traits* und ihre Gewichte zur Initialisierungszeit feststehen, jedoch nicht ihre genaue Ausprägung. Aus diesem Grund sollte der *float*-Paramter der *addPreferenceTrait()*-Methode, der das Gewicht des *Personality Traits* definiert, nicht mit der Ausprägung verwechselt werden, die bei der *Decorator*-Methode übergeben wird. Zur Ausführungszeit kommt dem String-Key des *Personality Traits* auf *Personality Selector*-Ebene zwei Aufgaben zu. Erstens die Identifizierung der *Personality Trait*-Verwendung auf der Unterebene, d.h. auf *Decorator*-Ebene und zweites die Identifizierung mit dem String-Key korrespondierenden *Blackboard*-Eintrages, der die aktuelle Ausprägung des *Personality Traits* auf *Personality Selector*-Ebene enthält. Ein Beispiel soll dies Zusammenspiel verdeutlichen.

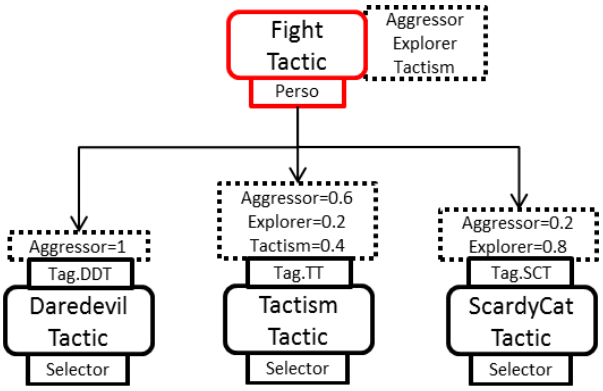

Abbildung 29: Beispielhafter Einsatz der Personality-Erweiterung.

In Abbildunng 29 wird ein *Pesonality Selector FightTactic* als *CompositeTaskNode*-Knoten verwendet. Diesem werden die drei *Personality Traits Aggressor*, *Explorer* und *Tactism* hinzugefügt. Den drei alternativen Verhaltensweisen, zwischen denen es auszuwählen gilt, werden ebenfalls *Personality Traits* hinzugefügt. Die Menge der *Traits* muss dabei nicht der des *Selectors* entsprechen. Beispielsweise wird der *DaredevilTactic* nur der *Aggressor Trait* mit der Ausprägung 1.0 zugewiesen. Während der Ausführung sucht der *Personality Selector* im *Blackboard* nach den Ausprägungen für seine drei *Traits* und bekommt z.B. dabei folgendes Ergebnis heraus:

```
Aggressor = 0.8, Explorer = 0.6 , Tactism = 0
```

Im nächsten Schritt wird der *Personality Value* eines jeden Kindes nach folgender Formel berechnet:

$$Personality\ Value_n = \frac{\sum_{i=1}^{j}(SPTV_i * \sum_{k=1}^{nl}(\mu(SPTK_i,\ CPTK_{nk}) * CPTV_{nk})}{\sum_{i=1}^{j} SPTV_i}$$

$$\mu(a,b) = \begin{cases} 1, \text{ falls a = b} \\ 0, \text{ sonst} \end{cases}$$

wobei folgendes gilt,

n = *Anzahl der Kinder*

j = *Anzahl der Personality Traits des Selectors*

l = *Anzahl der Personality Traits des n-ten Kindes*

$SPTV_i$ = *Ausprägung des i-ten Personality Traits des Selectors*

$CPTV_{nk}$ = *Ausprägung des k-ten Personality Traits des n-ten Kindes*

$SPTK_i$ = *String-Key des i-ten Personality Traits des Selectors*

$CPTK_{nk}$ = *String-Key des k-ten Personality Traits des n-ten Kindes*

Was in Kombination mit den Ausprägungen des *Personality Selectors* und den in Abbildung 29 dargestellten Werten der drei *Decorators* zu folgender *Personality Value*-Sortierung führt:

```
Personality Value(DaredevilTactic) = 62%
Personality Value(ScardyCatTactic) = 49%
Personality Value(TactismTactic) = 46%
```

Diese Liste wird im letzten Schritt dann von oben nach unten bis zum Ende abgearbeitet oder beim ersten Aufkommen eines *BS_SUCCESS*, *BS_RUNNING* oder *BS_ERROR* beendet. Dadurch das die Ausprägungen im *Blackboard* hinterlegt sind und die strategische Ebene jederzeit die Möglichkeit besitzt über die *ITact-Controller*-Schnittstelle diese zur Laufzeit zu ändern können für ein und dieselbe BT-Instanz mehrere Traversierungsreihenfolgen existieren die sich zur Laufzeit ändern können.

5.4.4 Der Behavior Tree der Game AI

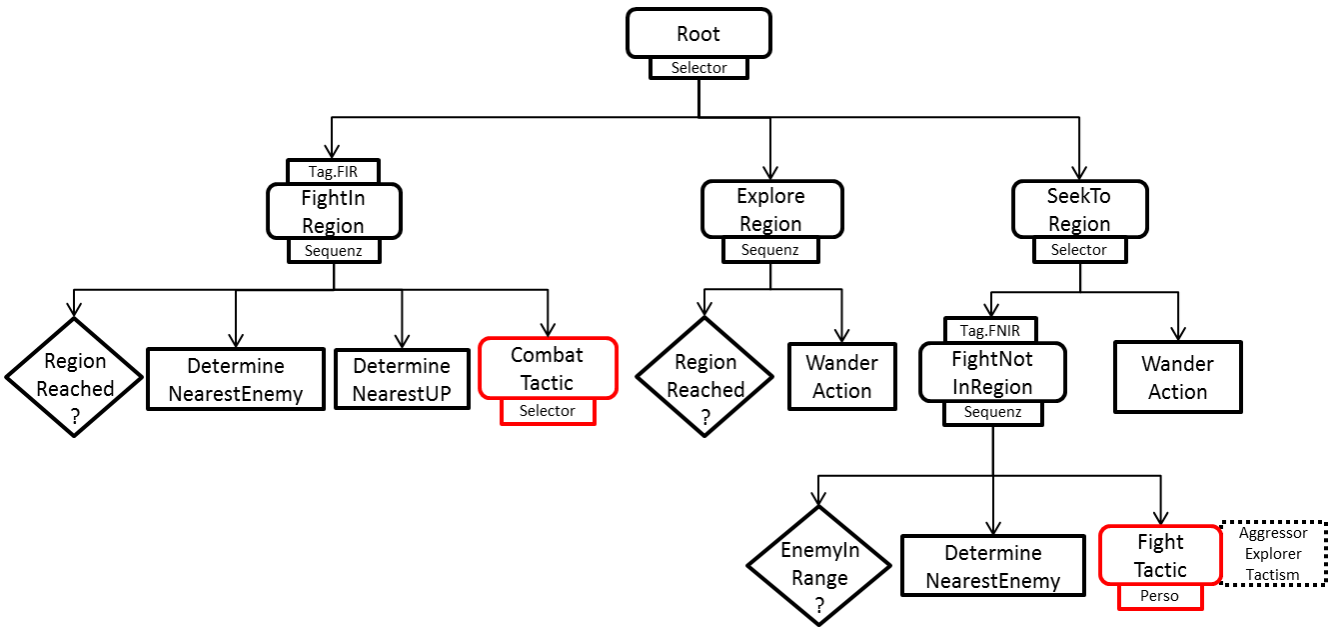

Abbildung 30: Oberste Ebene des konkreten *Behavior Trees* der *Game AI*.

Abbildung 30 zeigt die erste Ebene des konkreten BT-Baums welcher von der *BTBuilder*-Klasse realisiert wird. In der aktuellen Version des *BTBuilders* wird der Bau direkt im Code definiert. Der Root-Knoten

wird zu Beginn jeder Iteration als erstes angesteuert. Wurde der Aktionsauswahlprozess in der vorherigen Iteration nicht beendet und der BT-Status von außen nicht zurückgesetzt, wird das jeweils als letzte besuchte Kind direkt angesteuert und die Traversierung an dieser Stelle fortgesetzt. Ist dies nicht der Fall werden die drei Alternativen von links nach rechts durchgegangen.

Die erste Alternative *FightInRegion* definiert wie innerhalb einer Ziel-Region gekämpft werden soll, falls diese erreicht worden ist wird die nahe Umgebung auf Gegner und auf nicht eingenommene *Upgrade Points* hin untersucht. Eine Region ist ein durch die *Cuboid*-Klasse definierter 3D-Raum der zur Orientierung im Level verwendet und von der strategischen Ebene bereitgestellt und definiert wird. Im *CombatTactic*-Knoten erfolgt dann die Entscheidung welcher der beiden Objekt-Typen angesteuert werden soll. Falls die Ziel-Region erreicht wurde aber weder Feinde noch *Upgrade Points* in der Nähe sind, scheitert die erste Alternative und die zweite Alternative wird ausprobiert. Ein solches Scheitern ist in der Abbildung nicht direkt ersichtlich, weil diese sich erst im Unterbaum von *CombatTactic* ereignen könnte. Aus diesem Grund ist auch der Knoten rot-markiert, um zu verdeutlichen dass an dieser Stelle noch ein größerer Unterbaum existiert, auf den später noch genauer eingegangen wird. Die zweite Alternative ist bereits in der Abbildung komplett dargestellt und würde eine Erkundung der Region durchführen. Wurde hingegen die Ziel-Region noch nicht erreicht wird zum *SeekToRegion*-Knoten gesprungen und versucht diese Alternative erfolgreich auszuführen. *SeekToRegion* untersucht dabei als erstes ob Feine in der Nähe sind und wenn ja wird im *FightTactic*-Knoten bestimmt auf welche Weise gekämpft werden soll. Wurden keine Feinde gesichtet wird über die *WanderAction* ein beliebiger Punkt in der Ziel-Region angesteuert um sich der Region weiter zu nähern.

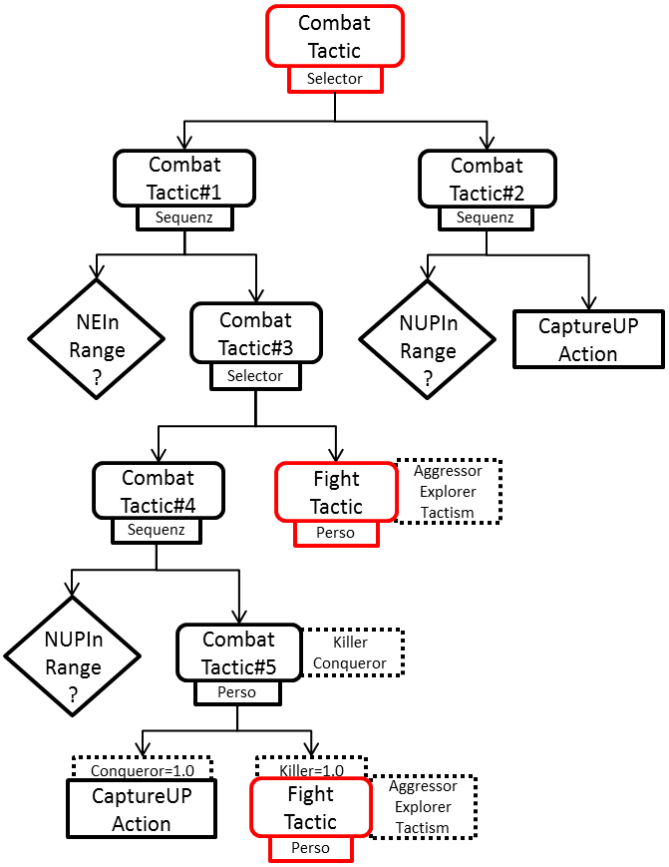

Abbildung 31: Der modellierte Auswahlprozess des *CombatTactic*-Unterbaums. Es wird entweder keine Aktion, die *CaptureUP*-Aktion oder der *FightTactic*-Unterbaum zur näheren Auswahl des konkreten Kampfverhaltens ausgewählt.

Abbildung 31 zeigt wie der Unterbaum des in Abbildung 30 abgebildeten *CombatTactic*-Knotens konkret aussieht. Der Kontext des Knotens ist das die Ziel-Region bereits erreicht wurde und die Nahe Umgebung auf Feinde und auf nicht eingenommene *Upgrade Points* untersucht wurde. Der Baum realisiert dabei eine Art *if-then*-Konstrukt welches folgendes besagt: Wenn nur Feinde in der Nähe sind oder nur nicht vom eigenen Team besetzt *Upgrade Points* gefunden worden sind, dann führe für den ersten Fall den *FightTactic*-Knoten und im zweiten Fall die *CaptureAction* aus. Treffen beide Umstände zu lass die Wahl von einem *Personality Selector* erledigen der zwischen beiden auswählt. Trift keiner der Umstände zu gilt der Unterbaum als gescheitert und weil *CombatTactic* der letzte Knoten in einer *Sequenzt* der ersten Alternative auf oberster Ebene ist, scheitert somit auch diese komplett und die Nächste Alternative wird ausgewählt.

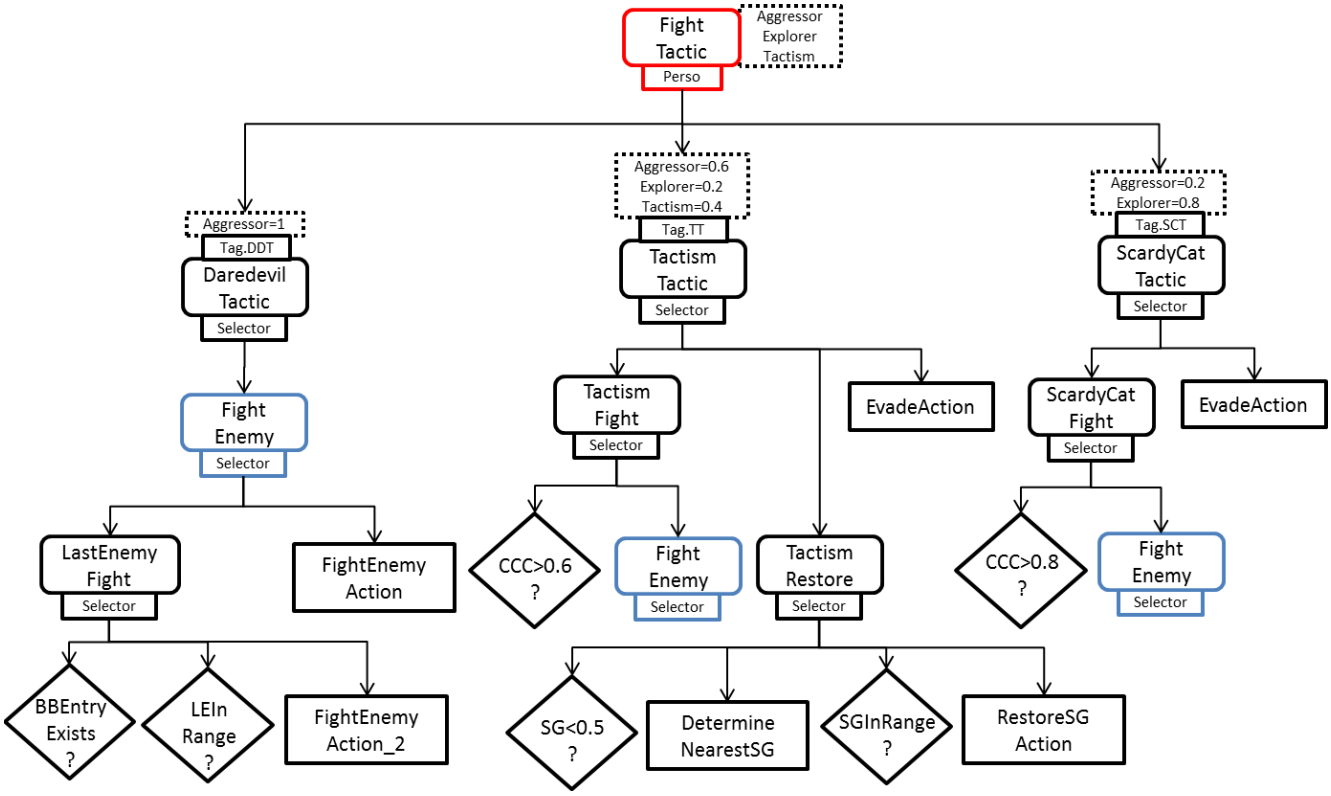

Abbildung 32: Der modellierte Auswahlprozess des *FightTactic*-Unterbaums zur Wahl der gewünschten Kampftaktik von gegnerischen Schiffen im freien Raum. Zur Wahl stehen drei unterschiedliche Taktiken.

Abbildung 32 präsentiert den *FightTactic*-Unterbaum, der bestimmt auf welche Art gegnerische Schiffe bekämpft werden sollen. Der Unterbaum wird an verschiedenen Stellen des Baumes benutzt. So z.B. vermehrt innerhalb des *CombatTactic*-Unterbaums, der ersten Alternative im übergeordneten BT-Baum, als auch in der *SeekToRegion*-Alternative. Wie in der Abbildung zur erleichterten Darstellung nicht jedes Mal der komplette Unterbaum des *FightTactic*-Knotens dargestellt wird, so muss der *FightTactic*-Knoten bzw. andere BT-Knoten im Baum nicht jedes Mal neu definiert werden. Einmal definiert können an unterschiedlichen Stellen im Baum Referenzen auf diese wiederverwendet werden.

Der *FightTactic*-Knoten ist genau genommen ein *Personality Selector*-Knoten und wählt wie bereits in Abschnitt 5.4.3 als Beispiel aufgeführt zwischen den drei unterschiedlichen Kampftaktiken *DaredevilTactic*, *TactismTactic* und *ScardyCatTactic*. Jede der drei Taktiken wird mit*Personality Decorators* bzw. mit *Personality Traits* die in gestrichelten Kästchen über den Knoten in der Abbildung visualisiert sind versehen. Sie beschreiben abstrakt das modellierte Verhalten. Die aggressive Verhaltensweise der *DaredevilTactic*-Variante bevorzugt immer den direkten Kampf, wohingegen die *ScardyCatTactic* erst die Situation mittels der *ComplexCombatCondition* evaluiert und nur wenn dieser Wert und somit der Glaube an einen Sieg

hoch genug ist wird dann tatsächlich gekämpft, ansonsten die immer die Flucht angetreten. Die *Tactism-Tactic*-Variante verhält sich wie die *ScardyCatTactic*, nur mit dem Unterschied das sie vor der Fluchtantritt noch die Möglichkeit der Benutzung eines nahen Schildgenerators in Betracht zieht.

5.5 Strategische Ebene

Die strategische Ebene ist die oberste oder abstrakteste der *Game AI*-Hierarchie. Ihre Aufgabe ist die übergeordnete langfristige Zielvorgabe und die Unterstützung der Zielerfüllung durch die unteren Ebenen. Abbildung 33 zeigt den generellen Aufbau der Ebene und die Operationen die von der *IStratController*-Schnittstelle angeboten werden.

Die strategische Ebene ist die Ebene die direkt mit der *GameAI*-Klasse verbunden ist und somit die Ebene die von dieser die Aufgabe zur Aktionsauswahl weitergereicht bekommt. Dazu ruft die *GameAI*-Klasse die *performNextAction()*-Methode der *IStratController*-Schnittstelle in jedem Spielzyklus neu auf. Neben der Aufgabe der Weiterleitung des Aktionsauswahlprozesses an die taktische Ebene, ist diese Ebene für die Konfigurierung der taktischen Ebene, sowie für die Regionsvorgabe zuständig. Ersteres wird direkt von der *StratController*-Klasse im Zusammenspiel mit der *PersonalityManager*-Klasse umgesetzt. Letzteres wird von der *RegionMapManager*-Klasse implementiert. Die Nachfolgenden Unterabschnitte gehen auf genau diese zwei Aspekte näher ein. Die restlichen Methoden der *IStratController*-Schnittstelle dienen ebenfalls dazu die Realisierung dieser Aufgaben zu konfigurieren und werden deshalb entsprechend ihres Kontextes erläutert.

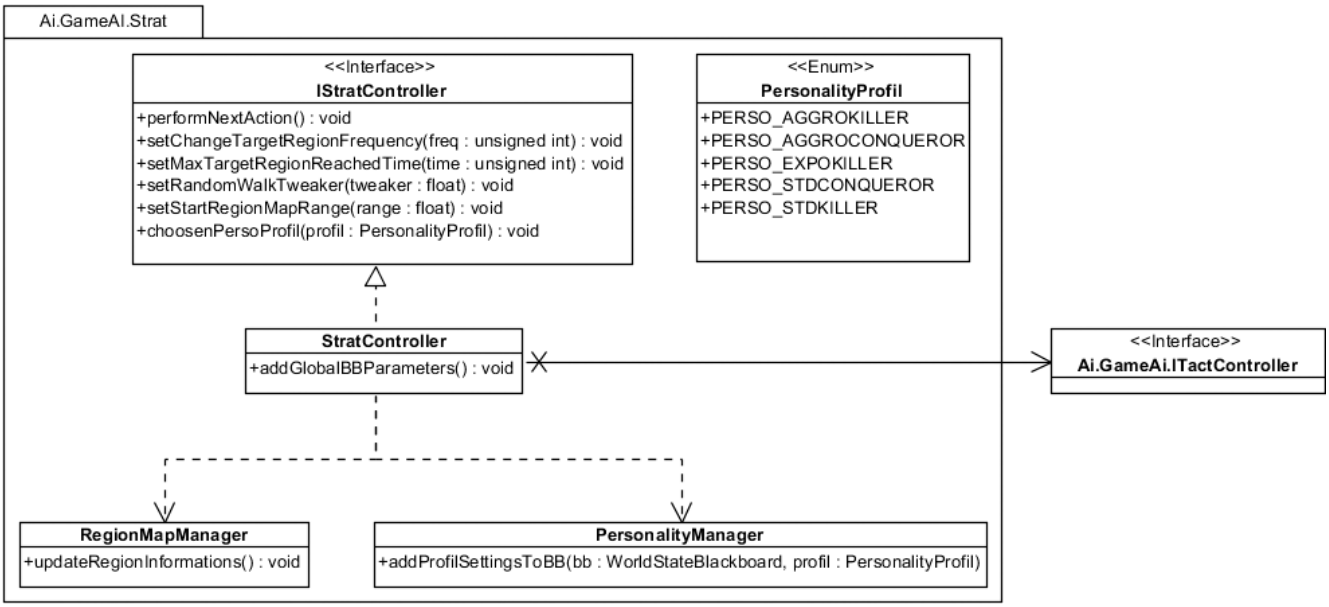

Abbildung 33: Die Struktur der strategischen Ebene und die Abhängigkeiten zu anderen Systemebenen.

5.5.1 Konfigurationsmöglichkeiten

Die Konfigurierungsmaßnahmen beschränken sich zurzeit auf zwei wesentliche Punkte. Der erste ist die Definition und Quantifizierung globaler *Blackboard*-Parameter, die im Aktionsauswahlprozess des BTs verwendet werden. Eine Aktion wie *FightEnemyAction* beispielsweise ermöglicht die Angabe mehrerer Parameter wie *FireRate* oder *FireAccuracy*. Diese Parameter können entweder während des Baus in der *BTBuilder*-Klasse definiert werden oder zur Laufzeit jeweils aus dem *Blackboard* gelesen werden. Mit der Definition globaler Parameter können Default-Werte für die einzelnen Aktionen definiert werden.

Dies hat bei einer breiten Verwendung der Default-Werte den Vorteil, dass an einer zentralen Stelle im Code die Parameter mehrerer Aktionen gleichzeitig angepasst werden können. Mit der Änderung oder Anpassung der globalen *Blackboard*-Parameter kann somit das Bot-Verhalten maßgeblich und einfach beeinflusst werden. Die globalen *Blackboard*-Parameter sind nicht nur auf Aktionsparameter beschränkt. Die Anpassung des *EnemyDetectionRange*-Parameters beispielsweise definiert den Bereich indem Gegner erkannt werden, worauf u.a. *LastEnemyInRangeCondition* prüft. Die kontext-abhängige Bestimmung und Verwaltung unterschiedlicher Parameter für jede Aktion ist ebenfalls möglich und würde eine noch genauere Steuerung des Bot-Verhaltens ermöglichen. Der Aufwand und die Pflege der unterschiedlichen Parameter ist allerdings dann entsprechend größer als bei der Verwendung einiger weniger Default-Parameter.

Die Definition der globalen *Blackboard*-Parameter findet zurzeit in der *addGlobalBBParameters()*-Methode der *StratController*-Klasse hart-codiert statt. Dazu wird über die *ITactController*-Schnittstelle auf eine Instanz der *WorldStateBlackboard*-Klasse zugegriffen und über deren *setData()*-Methoden die globalen Parameter in das *Blackboard* eingefügt.

Die zweite Form der Konfigurationsmaßnahmen der strategischen Ebene betrifft den Einsatz von Persönlichkeits-Eigenschafen innerhalb des BTs zur Steuerung des Bot-Verhaltens. Wie bereits in Abschnitt 5.4.3 erwähnt, ist die *Personality*-Erweiterung dazu gedacht, eine neue Traversierungsreihenfolge in den Baum zu integrieren, die in Abhängigkeit zur aktuellen Persönlichkeit des Bots erfolgt. Die strategische Ebene ist der Ort, in der genau diese Persönlichkeit für einen Bot definiert, ausgewählt und evtl. im Laufe des Spiels angepasst wird. Dazu bietet die *IStratController*-Schnittstelle nicht die Möglichkeit der eigenen Persönlichkeitsdefinition anhand einer Zusammenstellung von *Personality Traits* an, sondern ermöglicht über die *choosePersoProfil()*-Methode die Auswahl eines von mehreren vorgefertigten Persönlichkeits-Profilen. Wie die Abbildung 33 zeigt, existieren aktuell folgende fünf Profile, die jeweils von einem Element des *Personality Profile*-Enums identifiziert werden:

- **PERSO_AGGROKILLER:** Zieht die Bekämpfung und Verfolgung gegnerischer Schiffe der Eroberung und der Verteidigung von *Upgrade Points* vor. Im Kampf wird unabhängig der Spielsituation bis zum letzten *Health Point* gekämpft.

- **PERSO_AGGROCONQUEROR:** Zieht die Eroberung und die Verteidigung von *Upgrade Points* der Bekämpfung und Verfolgung gegnerischer Schiffe vor. Im Kampf wird unabhängig der Spielsituation bis zum letzten *Health Point* gekämpft.

- **PERSO_EXPOKILLER:** Zieht die Bekämpfung und Verfolgung gegnerischer Schiffe der Eroberung und der Verteidigung von *Upgrade Points* vor. Im Kampf wird die Flucht ergriffen, falls der Sieg nicht sehr wahrscheinlich erscheint.

- **PERSO_STDCONQUEROR:** Zieht die Eroberung und die Verteidigung von *Upgrade Points* der Bekämpfung und Verfolgung gegnerischer Schiffe vor. Im Kampf wird je nach Situation die direkte Konfrontation gesucht, die Flucht ergriffen oder ein nahegelegener Schildgenerator angesteuert.

- **PERSO_STDKILLER:** Zieht die Bekämpfung und Verfolgung gegnerischer Schiffe der Eroberung und der Verteidigung von *Upgrade Points* vor. Im Kampf wird je nach Situation die direkte Konfrontation gesucht, die Flucht ergriffen oder ein nahegelegener Schildgenerator angesteuert.

Die *PersonalityManager*-Klasse ist für die Spezifizierung der Profile und die Einpflege des aktuell ausgewählten Profils in das *Blackboard* zuständig. Die Einpflege erfolgt dabei durch den Aufruf der *addProfilSettingsToBB()*-Methode, welches die entsprechenden *Personality Traits* eines Profils ins *Blackboard*

abspeichert. Das Persönlichkeits-Profil *PERSO_AGGROKILLER* besteht beispielsweise aus folgender *Personality Trait*-Menge:

```
Killer = 1.f
Aggressor = 1.f
```

Mit dieser Menge wird das in der *PERSO_AGGROKILLER*-Beschreibung spezifizierte Verhalten im BT realisiert. Die Wahl eines Profils hat allerdings nicht nur Auswirkungen auf die Traversierungsreihenfolge der *Personality Selectors* im Baum. Die Profilauswahl beeinflusst darüber hinaus auch den Einsatz von *Behavior Tags*. *Behavior Tags* ermöglichen, wie in Abschnitt 5.4.3 vorgestellt, die temporäre Aktivierung bzw. Deaktivierung bestimmter Teile des Baums. Das Persönlichkeits-Profil *PERSO_AGGROKILLER* beispielsweise verwendet folgende *Behavior Tags*:

```
BlackboardEntry<bool> = {"'Tag.FNIR.Tag.TT"', false}
BlackboardEntry<bool> = {"'Tag.FNIR.Tag.SCT"', false}
```

Mit diesen beiden *Behaviors Tags* wird die Ausführung der in Abbildung 32 modellierten Kampftaktiken *TactismTactic* und *ScardyCatTactic* deaktiviert. Das bedeutet selbst wenn die übriggebliebene *DaredevilTactic* fehlschlägt, wird keine der anderen Alternativen ausprobiert. Außerdem kann sich in diesem Unterbaum auch bei der Anpassung der *Personality Trait*-Ausprägungen des Profils niemals eine andere Traversierungsreihenfolge im Unterbaum ergeben, weil solange diese beiden *Behaviors Tags* aktiv sind nur eine einzige Alternative zur Auswahl steht. Zu beachten ist, dass die Deaktivierung der beiden Kampftaktiken nur die Instanzen im Unterbaum von *FightNotInRegion* betrifft, nicht jedoch die der anderen Stellen im Baum, da diese außerhalb der FNIR spezifizierten *Behavior Tag*-Hierarchie liegen. So wird beispielsweise im *FightInRegion*-Unterbaum nach wie vor alle drei Kampftaktiken evaluiert und gegebenenfalls nacheinander ausgeführt.

5.5.2 Region Map und die strategische Zielbestimmung

Die Aufgaben der *RegionMapManager*-Klasse der strategischen Ebene sind die Erstellung und Verwaltung der *Region Map*, die Erfassung der aktuellen Region, die Bestimmung eines jeweils neuen Regionsziels und die Propagation der Regionsinformationen an die taktische Ebene. Nachfolgend werden diese unterschiedlichen Aufgaben kurz erläutert.

Erstellung und Verwaltung der Region Map
Die *Region Map* nimmt eine räumliche Aufteilung des aktuellen Levels in 3D-Quadrate mit Hilfe der *Cuboid*-Klasse vor. Dazu wird in Abhängigkeit des über die *setStartRegionMapRange()*-Methode der *IStratController*-Schnittstelle übergebenen Parameters die Größe des 3D-Quadrats zur Initialisierungszeit festgelegt. Dieses 3D-Quadrat wird daraufhin in acht gleichgroße Teil-Quadrate zerlegt, welche zur Definition der Regionen der *Region Map* eingesetzt werden. Der Mittelpunkt des übergeordneten 3D-Quadrats entspricht dabei immer der Null-Position des Raums, sodass die Teil-Quadrate um die Null-Position des Raums herum gebildet werden. Abbildung 34 veranschaulicht den Aufbau eines 3D-Quadrats und Abbildung 35 den eben beschriebenen Aufteilungsprozess.

Bei jedem von der *RegionMapManager*-Klasse erkannten Regionswechsel, wird einmal auf die globale Regions-Zugehörigkeit aller in Sichtweite befindlichen *Upgrade Points* geprüft. Wird bei der Überprüfung festgestellt, dass sich *Upgrade Points* außerhalb der *Region Map* befinden, so erfolgt eine *Region Map*-Erweiterung um den Faktor der Initialisierungsgröße. Bei jeder Erweiterung wird demnach zusätzlich eine neue Unterebene mit 8 Regionen erstellt. Die Anzahl der zu verwalteten Regionen entwickelt sich exponentiell mit steigender Ebenen-Tiefe. Um die dadurch evtl. entstehende Speicherlast zu reduzieren, werden nur die Unterregionen erstellt und im Speicher gehalten die aktuell relevant sind. Alle nicht

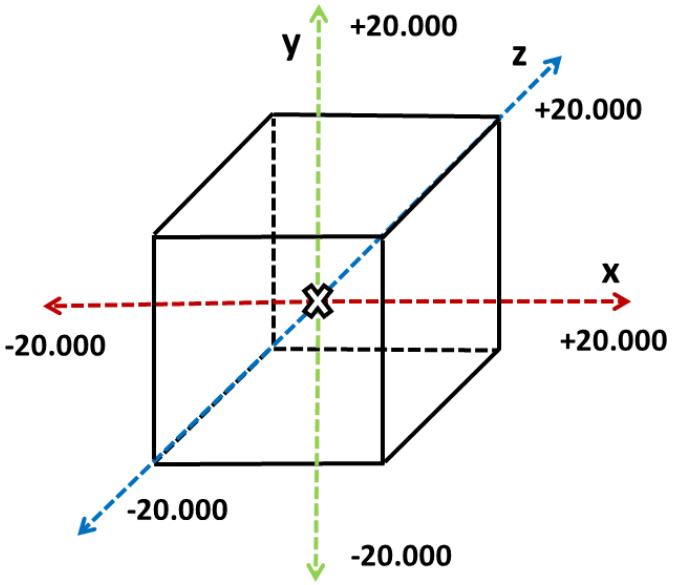

Abbildung 34: Aufbau des 3D-Quadrats einer Region mit einer Initialisierungsgröße von 40000.

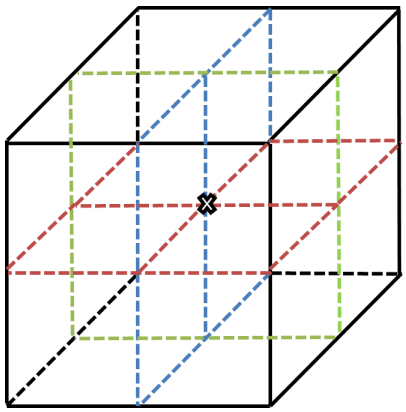

Abbildung 35: Aufteilung des 3D-Quadrats einer Region in seine 8 Unterregionen.

benötigte Regionen werden entweder nicht erstellt oder wieder gelöscht, sodass immer nur eine überschaubare Anzahl an Regionen im Speicher gleichzeitig gehalten werden muss.

Erfassung der aktuellen Region innerhalb der Region Map

Die strategische Ebene ruft in jedem ihrer Zyklen die *RegionMapManager*-Klasse zur Überprüfung der aktuellen Region auf. Dazu wird im ersten Schritt die aktuelle Position des Schiffes auf die Regionszugehörigkeit zur aktuellen Region des vergangenen Zyklus überprüft. Falls die Position zur alten aktuellen Region gehört, ist die Überprüfung an dieser Stelle bereits beendet, falls nicht muss die neue Region ermittelt werden. Dazu wird beginnend von der obersten Ebene der *Region Map* sich von Ebene zu Ebene gesucht. Diese *Top Down*-Suche ist deswegen möglich, weil die Größe der übergeordneten Region genau der Größe aller Unterregionen entspricht. Wurde die aktuelle Region gefunden, die sich immer auf der untersten Ebene der *Region Map* befindet, ersetzt die neue Region die alte aktuelle Region in der *RegionMapManager*-Klasse für zukünftige Regionszugehörigkeits-Vergleiche. Darüber hinaus werden die 3D-Quadrat-Ausmaße der neuen aktuellen Region, die über die *Cuboid*-Klasse definiert werden, an die taktische Ebene als aktuelle Region weitergegeben.

Bestimmung eines Regionsziels

Die *RegionMapManager*-Klasse ist ebenfalls für die Bestimmung der Region zuständig, die als nächstes

von der taktischen Ebene angesteuert werden soll. Durch die Vorgabe einer Zielregion wird die taktische Ebene nicht sofort dazu aufgefordert diese Region zu erreichen, sondern es dient als langfristige strategische Zielvorgabe. Es ist also durchaus eine gültige und gewünschte taktische Entscheidung die Erreichung der Zielregion solange aufzuschieben, bis nahe Gegner erfolgreich bekämpft wurden.

Aktuell wird zur Zielbestimmung entweder ein *Random Walk* zwischen beliebigen Regionen oder einem zwischen Regionen mit bekannten *Upgrade Point*-Vorkommen eingesetzt. Der *Random Walk* zwischen beliebigen Regionen dient einerseits der Entdeckung neuer Regionen mit *Upgrade Points* und anderseits der Überprüfung auf eine notwendige Map-Erweiterung. Die andere *Random Walk*-Variante soll die bevorzugte Ansteuerung von POIs simulieren. Anders als bei der ersten *Random Walk*-Art wird die Zielregion, nach der Untersuchung auf *Upgrade Points*, nicht sofort wieder verlassen, sondern eine Zeit lang als Zielregion beibehalten. Somit wird der taktischen Ebene eine gewisse Zeit gelassen evtl. Eroberungen oder Verteidigungen von *Upgrade Points* in der Region vorzunehmen. Ein einstellbarer *Tweaker* gibt dabei das gewünschte Verhältnis zwischen der den Random Walk-Varianten an.

Die *Region Map* bietet zusammenfassend dem Bot die Fähigkeit zur Erfassung des Levels. Das Potenzial dieser Fähigkeit wird zurzeit noch nicht völlig ausgenutzt. Es kann davon ausgegangen werden, dass erst mit der Integration koordinierten Gruppenverhaltens das wahre Potenzial der *Region Map* ausgeschöpft wird. Denn die *Region Map* bietet allen voran eine lückenlose Erfassung des Levels an, welche zum Informationsaustausch zwischen den Bots genutzt werden kann. Dabei kann die Granularität der Regionsgröße beliebig angepasst werden.

6 Evaluation

Dieses Kapitel beschreibt den Ansatz zur Evaluation der in Kapitel 4 präsentierten Bot-Implementierung zur Generierung einer realistischen Netzwerklast. Abschnitt 6.1 „Ziele und Rahmenbedingungen“ definiert die Zielbestimmung und Zieleingrenzung der Evaluation und beschreibt des Weiteren die Testplattform und den Einsatz des *Discrete Event Game Simulators* zur Evaluation.

Die Evaluation umfasst 9 verschiedene Testdurchläufe, die zu jeweils unterschiedlichen Bedingungen ablaufen. Abschnitt 6.2 „Eingesetzte Metriken“ stellt die für die Evaluation eingesetzten Metriken vor, die zur Erfassung der in Abschnitt 6.4 „Durchgeführte Testszenarien“ präsentierten Evaluationsergebnisse der neun verschiedenen Testdurchläufe. Abschnitt 6.3 „Allgemeiner Aufbau und Ablauf der Testszenarien“ beschreibt zuvor den allgemeinen Aufbau und Ablauf der durchgeführten Testszenarien. Dazu wird vor allem auf die Rolle des *Discrete Event Game Simulators* eingegangen, welcher maßgeblich den Ablauf und die Konfiguration der Testszenarien bestimmt.

Der abschließende Abschnitt 6.5 „Auswertung der Ergebnisse“ der Evaluation wertet die gewonnenen Ergebnisse anhand der Metriken aus und versucht einige Schlüsse und Aussagen betreffend der Eignung zur Erzeugung der Netzwerklast zutreffen.

6.1 Ziele und Rahmenbedingungen

Das Hauptziel der Evaluation ist die Überprüfung ob und in welchem Ausmaß die in Abschnitt 1.2 definierten vier Kriterien **Reproduzierbarkeit**, **Skalierbarkeit**, **Realitätsgrad** und **Konfigurierbarkeit** vom implementierten Ansatz aus Kapitel 4 erfüllt werden. Anhand der Überprüfung dieser Kriterien soll die konkrete Bot-Implementierung daraufhin untersucht werden, ob diese zur synthetischen Generierung einer realistischen Netzwerklast zu Evaluationszwecken verschiedenartiger P2P-Overlays geeignet erscheint. Darüber hinaus soll anhand der Evaluationsergebnisse ebenfalls ein erster Eindruck davon gewonnen werden, ob der allgemeine Ansatz der Bot-Implementierung, nämlich **die Verwendung kontext-sensitiver AI-Spieler zur Lasterzeugung**, eine ernst zu nehmende Alternative im Vergleich zu den anderen, in Abschnitt 3.1 vorgestellten, Ansätzen der Lasterzeugung darstellt.

Bei der Untersuchung wird nicht der Anspruch auf die abschließende Beantwortung der Frage nach der Geeignetheit der Bot-Implementierung bzw. des allgemeinen Ansatzes zur Lasterzeugung erhoben. Solch eine umfassende Evaluation würde ein breites Spektrum an verschiedenen Testreihen unter Berücksichtigung aussagekräftiger und evaluierter Netzwerk-Metriken zur Messung der Güte der generierten Netzwerklast erfordern. Dies würde insbesondere eine genaue Untersuchung der dabei tatsächlich generierten Netzwerklast und dem Vergleich dieser mit Referenzdaten echter Netzwerklasten oder der generierten Last alternativer Ansätze mit einschließen. Die ausschöpfende und allen voran Netzwerk technisch orientierte Beantwortung dieser Frage wird somit nachfolgenden Arbeiten im Rahmen des QuaP2P-Forschungsprojektes überlassen, welches das übergeordnete Projekt dieser Master-Thesis darstellt. Aus diesem Grund ist das Ziel der Evaluation dieser Master-Thesis eine erste (grobe) Untersuchung der generellen Geeignetheit der Bot-Implementierung bzw. des Ansatzes zur Lasterzeugung als Grundlage für eventuell weiterführende, exaktere oder aufwendigere Evaluationen.

Zur Evaluation wird ein *Discrete Event Game Simulator* des DVS-Fachbereichs [31, 30] eingesetzt. Der Simulator ist in der Lage das Spiel mit mehreren Peers, statt in einem realen Netzwerk, in einer vom Simulator kontrollierten und reproduzierbaren Umgebung auszuführen. Echtzeit Game-Events, die über das Netzwerk weitergleitet werden müssten, werden vom Simulator abgefangen und an die betreffenden

Peers unter Berücksichtigung der simulierten Netzwerk-Charakteristik weitergeleitet. Die aktuell unterstützen Overlay-Netzwerke sind *pSense* [49], *BubbleStorm* [55] und eine *Client/Server*-Version namens CUSP. Die Simulierung des Netzwerks und die mögliche Abschaltung der grafischen Spieldarstellung ermöglichen, unter der Voraussetzung der Verfügbarkeit ausreichender Systemleistung und -ressourcen, eine Beschleunigung der tatsächlichen Spielausführung. Die Evaluation wurde auf einem Computer mit folgenden technischen Eckdaten ausgeführt:

- **CPU:** Intel Core 2 Duo T9600 (2,8 GHz)
- **RAM:** 4 GB
- **OS:** Ubuntu 12.04.1 LTS (32-Bit)

6.2 Eingesetzte Metriken

Für die Untersuchung der generellen Eignung zur Lasterzeugung der Bot-Implementierung werden dabei die in der Tabelle 3 aufgeführten Metriken zur Beurteilung des Bot-Verhaltens definiert und in den durchgeführten Testdurchläufen verwendet. Es handelt sich dabei um relativ simple Metriken, deren Bezeichnungen größtenteils selbsterklärend sind. Beispielsweise erfasst die *Shot Accuracy*-Metrik die Schussgenauigkeit in Prozent. Die meisten Metriken können sich dabei entweder auf eine Bot-Instanz im Schnitt oder auf alle Bot-Instanzen eines gesamten Testdurchlaufs beziehen. Die Bezeichner-Ergänzung „(Team)“ weist zusätzlich auf eine mögliche Auswertung über die aggregierte Menge der Bot-Instanzen eines Teams hin.

Bezeichnung	**Abkürzung**	**Wertebereich**	**Relevante Kriterien**
Number of Kills	Kills#	$\mathbb{Z}$ in [0,∞[	Konfigurierbarkeit
Number of Deaths	Deaths#	$\mathbb{Z}$ in [0,∞[	Konfigurierbarkeit
(Team) Kill/Death-Ratio	(T)KD-Ratio	% in [0,∞[	Realitätsgrad
Number of Shots	Shots#	$\mathbb{Z}$ in [0,∞[	Konfigurierbarkeit
Number of Hits	Hits#	$\mathbb{Z}$ in [0,∞[	Konfigurierbarkeit
Shots per Player per Min.	PShots	$\mathbb{Q}$ in [0,∞[	Konfigurierbarkeit
Hits per Player per Min.	PHits	$\mathbb{Q}$ in [0,∞[	Konfigurierbarkeit
Shot Accuracy	Accuracy	% in [0,100]	Konfigurierbarkeit, Realitätsgrad
(Team) Upgrade Point Score	(T)UP-Score	$\mathbb{Q}$ in [0,∞[	Konfigurierbarkeit, Realitätsgrad
Simulation Time	Time	Minutes in [0,∞[	Konfigurierbarkeit, Skalierbarkeit
Simulation Memory VIRT	Memory VIRT	Mega Bytes in [0,∞[	Skalierbarkeit
Simulation Memory RES	Memory RES	Mega Bytes in [0,∞[	Skalierbarkeit
Measurement Bias	Bias	% in [0,100]	Reproduzierbarkeit

Tabelle 3: Auflistung der in der Evaluation verwendeten Metriken mit Wertebereichsangabe und der Zuordnung zu relevanten Kriterien.

In der Tabelle 3 wird weiterhin eine Zuordnung von Metriken zu relevanten Kriterien vorgenommen. Diese Zuordnung soll die Bedeutung der Metriken für bestimmte Kriterien hervorheben, auf die in der Auswertung der einzelnen Testdurchläufe noch näher eingegangen wird. Die Anzahl der abgefeuerten Schüsse beispielsweise, welche von der *Shots*-Metrik erfasst wird, bestimmt direkt die Menge der übertragenden Nachrichten im Netzwerk. Anders als auf *Client/Server*-Systemen basierenden Spielen, müssen bei *P2P*-Spielen Spielinformationen in der Regel im Netzwerk mittels Nachrichten weitergereicht werden. In *Planet PI4* ist das Abfeuern von Schüssen eine solch weiterzureichende Spielinformation und

demnach ihr Aufkommen eine wichtige Stellschraube zur Kalibrierung der Netzwerklast.

Im Folgenden werden kurz die Metriken oder der Kriterienbezug erläutert, deren Bedeutung oder dessen Beziehung evtl. nicht sofort ersichtlich erscheinen:

- **(T)KD-Ratio:** Misst das Verhältnis zwischen *Kills* und *Deaths* eines Bots. Die effektive Vernichtung der gegnerischen Schiffe ist neben der Eroberung von *Upgrade Points* eines der möglichen zentralen Spielziele von *Planet PI4*. Aus diesem Grund ist das *Team Kill/Death*-Verhältnis ein wichtiger Indikator für die Spielstärke eines Bots und ermöglicht den Fähigkeiten-Vergleich zum Spielen des Spiels zwischen verschiedenen Bot-Implementierungen, Bot-Konfigurationen oder menschlichen Spielern.

  Anders als die Anzahl der Schüsse eine direkte Kalibrierung der Netzwerklast ermöglicht, ist die Auswirkung der Anpassung der *KD-Ratio* für die Netzwerklast nicht direkt ersichtlich. Allerdings ermöglicht die *KD-Ratio* die Bewertung des Bot-Verhaltens in Bezug auf den Realitätsgrad der erzeugten Last. Im Spiel können beispielsweise *Deaths* nicht nur durch feindliche Abschüsse erfolgen, sondern ebenfalls durch *Friendly Fire*, der Kollision mit Team-Bots oder dem Zusammenprall mit Asteroiden. Diese *Deaths* werden jedoch den Verursachern nicht als reguläre *Kills* zugeschrieben, sodass im Laufe eines Spiels sich mehr *Deaths* ereignen können als *Kills*. Unterstellt man weiter menschlichen Spielern die erfolgreiche Vermeidung solcher „negativen“ *Deaths*, so sollte eine Bot-Implementierung eine *KD-Ratio*, über alle Bot-Instanzen eines Spiels betrachtet, von ≈ 1 anstreben. Ein Wert von genau 1 würde nämlich bedeuten, dass alle *Deaths* von regulären *Kills* verursacht worden sind und somit keine „negativen“ *Deaths* vorkamen. Mit so einem Ergebnis würde gezeigt werden, dass die erzeugte Netzwerklast als realistisch anzunehmen sei, dass sie durch menschenähnliches Verhalten erzeugt worden ist.

- **Time:** Gibt die benötigte Zeit eines Testdurchlaufs oder den gemittelten Wert mehrerer Testdurchläufe im Simulator an. Weil die Simulation je nach verfügbaren Ressourcenkapazitäten, die Berechnung des Spielgeschehens beschleunigen bzw. bei Belastung verlangsamen kann, ist die benötigte Berechnungszeit des Simulators ein grober Indikator für die Performance der Bot-Implementierung. Um einen Richtwert zur Beurteilung der Performance zu haben, wird die Vorgänger-Implementierung von Dimitri Wulffert (siehe Abschnitt 3.5) als Referenz herangezogen.

- **Memory:** Gibt den durchschnittlichen Speicherverbrauch des Simulatorprozesses während der Ausführung eines Testdurchlaufes oder mehrerer Testdurchläufe im Schnitt an. Wie bei der *Time*-Metrik dient dies zur ersten groben Einschätzung der Speicherauslastung, indem gegen die Vorgängerimplementierung geprüft wird.

  „VIRT“ bezeichnet dabei die virtuelle Größe eines Prozesses und ist die Summe seines aktuell belegten Speichers.

  „RES“ bezeichnet dabei die tatsächliche bzw. physikalische Größe eines Prozesses und ist die Summe seines aktuell belegten Speichers.

- **Bias:** Misst die Metrik-Abweichung für wiederholte Testdurchläufe zur Überprüfung der Reproduzierbarkeit der Ergebnisse. Der Bias wird ermittelt, indem die proztentuelle Abweichung aller Metriken zwischen den einzelnen Simulationswiederholungen berechnet und im Anschluss daran gemittelt wird. Eine Angabe von 0% bedeutet, dass es keine Abweichung zwischen den Ergebnissen ermittelt werden konnte.

Der Bias dient somit der Überprüfung auf Nicht-Determinismus der Simulationsabläufe. Die Reproduzierbarkeit wird dabei größtenteils vom Simulator gewährleistet, der die Kontrolle über die simulierte Umgebung besitzt und diese zur Generierung deterministischer Prozessabläufe einsetzt. Nichtsdestotrotz werden in der Bot-Implementierung stochastische Prozesse verwendet, z.B. zur Bestimmung des *Random Walks* zwischen Regionen der strategischen Ebene (siehe Abschnitt 5.5.2). Damit solche Prozesse nicht die deterministische Eigenschaft der Simulation verletzen, bietet das Spiel die Benutzung eines vom Simulator kontrollierbaren Zufallszahlengenerators an, der für stochastische Prozesse reproduzierbare pseudo-Zufallszahlenreihen produziert.

6.3 Allgemeiner Aufbau und Ablauf der Testszenarien

Ein Testdurchlauf bezeichnet eine durch eine Menge von Ausführungsparametern konfigurierte Spielausführung mit einer festgelegten Spieleranzahl und Spieldauer. Damit die in Abschnitt 6.2 definierten Metriken reproduzierbare Ergebnisse produzieren, erfolgt die Spielausführung unter Einsatz des *Discrete Event Game Simulators*. Abbildung 36 zeigt dabei den generellen Testaufbau.

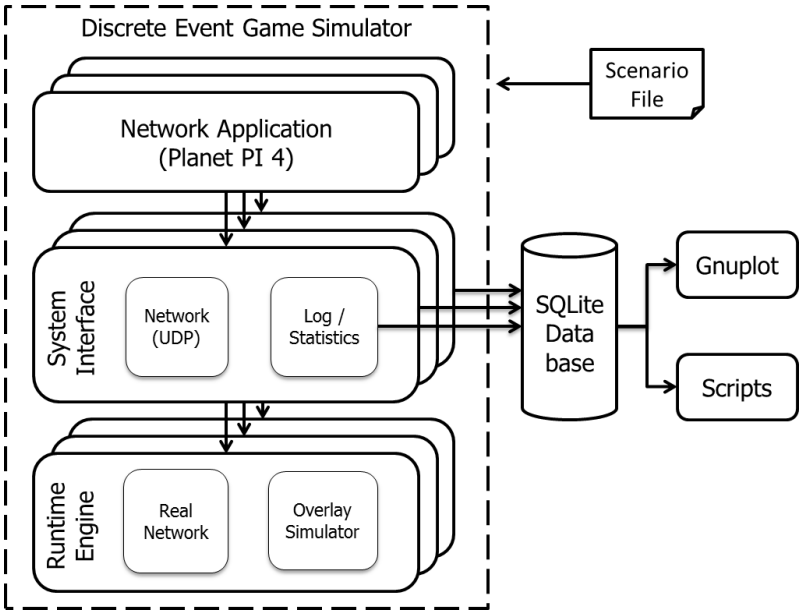

Abbildung 36: Übersicht des allgemeinen Testaufbaus. Angelehnt an [31].

Mit der Definition des *Scenario-Files* werden unterschiedliche Spielparameter für die Simulation oder Spielausführung definiert, wie beispielsweise Spieleranzahl, Spieldauer, *Seed*-Parameter des Zufallszahlengenerators oder die Auswahl des zu simulierenden *P2P*-Overlays. Das in der *Scenario-File* definierte Spielszenario wird vom Simulator daraufhin zur Ausführungszeit kontrolliert umgesetzt, d.h. der Simulator kümmert sich um die Initialisierung, Verwaltung und Terminierung der verschiedenen Spielinstanzen, sowie um die pseudo Netzwerk-Kommunikation dieser untereinander zur Spielzeit. Somit kann jede Spielinstanz im Simulator als ein zu simulierender Peer betrachtet werden.

In der Abbildung wird ein Peer als ein drei schichtiges System dargestellt. Die oberste Ebene ist die *Network Application*-Ebene, welche die eigentliche Logik der zu simulierenden Anwendung enthält. Im Rahmen dieser Evaluation geht es dabei um das *MMOG*-Spiel *Planet PI4*, für welches die Bot-Implementierung entwickelt wurde. Es könnte aber auch eine beliebig andere verteilte Anwendung sein, wie beispielsweise das *Search Overlay* von *BubbleStorm* [55].

Alle *Network Applications* abstrahieren dabei durch die Verwendung einer gemeinsamen System-Schicht in gewisser Weise von der eigentlichen System-Ausführung in einem verteilten Netzwerk. Die System-Schicht bzw. das *System Interface* bietet den Anwendungen Zugriff auf gemeinsame Funktionalitäten, wie die eigentliche Abwicklung der Netzwerk-Kommunikation, das *Scheduling* oder die Ausführung bestimmter Anwendungsereignisse (Events) auf Netzwerkebene. Eine für die Evaluation wichtige gemeinsame Funktionalität ist die Bereitstellung von Statistik- und Logging-Mechanismen zur Erfassung und Speicherung gewünschter Anwendungs- und Netzwerkdaten. Die somit erfassten Daten werden in einer Datenbank hinterlegt und können anschließend ausgewertet werden. Es existieren bereits einige Berechnungs-Skripte oder Visualisierungsmöglichkeiten über *Gnuplot* zur Aufbereitung der Daten. Zur Bestimmung der Metrik-Daten aus Abschnitt 6.2 und zur Ergebniserfassung und -auswertung in Abschnitt 6.4 wurden größtenteils auf bereits vorhandenes zurückgegriffen.

Die unterste der drei Ebenen eines Peers ist die *Runtime Engine*-Ebene. Sie spezifiziert welches konkrete Netzwerk über den *Overlay Simulator* simuliert werden sollen, oder ob die Testausführung über ein reales Netzwerk ausgeführt werden soll.

Der Ablauf einer jeden Simulation erfolgt dabei nach demselben Muster. Der Simulator startet das Spiel zum Simulationsstartzeitpunkt und lässt danach in einem bestimmten Intervall nacheinander weitere Peers das Spiel beitreten. Die Erfassung der globalen Statistiken erfolgt dabei zum Startzeitpunkt der Simulation und wird jeweils zum Eintrittszeitpunkt eines Peers, durch dessen individuelle Statistiken erweitert. Das Spiel wird dann die gewünschte Zeit lang gespielt, während die Statistik-Daten in der Datenbank hinterlegt werden. Terminiert wird das Spiel zum Endzeitpunkt der Simulation ohne zuvor einen Prozess zum geordneten Austritt der einzelnen Peers auszuführen. Im Anschluss auf einen Testdurchlauf werden die Ergebnisse über Skripte und *Gnuplot* aufgearbeitet.

Um die *Bias*-Abweichung zu ermitteln, wird zusätzlich jeder Testdurchlauf zwei Mal ausgeführt. Mit dem Bias soll getestet werden, ob die Ergebnisse der Testdurchläufe auch reproduzierbar sind. Der Bias bezieht sich dabei nicht auf die Ergebnisse der Laufzeit- und Speichermessung, da diese nur grob erfasst werden können und aufgrund externer Faktoren schwanken können. Grafische Ergebnisverläufe wie z.B. der *UP Score*-Verlauf werden ebenfalls nicht berücksichtigt. Die in den Testszenarien aufgeführten Metrik-Ergebnisse sind immer die Ergebnisse der ersten Testausführung, die Ergebnisse der zwei zusätzlichen Wiederholungen zur Ermittlung des Bias werden nicht extra aufgeführt, sondern werden nur anhand einer möglichen Bias-Abweichung angezeigt.

Für die Zeit- und Speichermessung wird Ubuntus Systemüberwachungsprogramm „top“ verwendet.

6.4 Durchgeführte Testszenarien

In diesem Abschnitt werden die Ergebnisse der einzelnen Testszenarien präsentiert. Die Präsentation folgt dabei folgendem Schema: Als erstes gibt eine kurze Beschreibung Auskunft über die Besonderheit des Testszenarios, was sich vor allem auf abweichende Parametereinstellungen konzentriert. Danach folgt die tabellarische Darstellung der Simulationsparameter. Die Parameterdarstellung ist aufgeteilt in Simulator- und Spieleinstellungen in einer gemeinsamen und den Bot-Einstellungen in einer separaten Tabelle. Falls die Bot-Parameter zwischen den Teams unterschiedlich sind, erfolgt die Darstellung der Bot-Parameter in jeweils einer eigenen Tabelle. Im Anschluss daran werden die Metrik-Ergebnisse in tabellarischer und grafischer Form präsentiert.

Abschließend noch einige allgemeine Erläuterungen zum besseren Verständnis der Simulationsparameter. Der Simulationsparameter *Join-Verhalten* gibt das Eintrittsverhalten der Peers zum Simulationsstart an. *Linear in 00:01* bezeichnet dabei einen linearen Eintrittsprozess in einem 1-Sekunden Takt. Der

Seed-Parameter gibt den gleichnamigen Wert für den Zufallszahlengenerator des Simulators an. Die Abkürzen UP und SG stehen für *Upgrade Point* und *Shield Generator*. Die *Personality Profile*-Einstellungen entsprechen dabei den in Abschnitt 5.5.1 präsentierten Profilen und Erläuterungen.

6.4.1 1. Testszenario

Im ersten Testszenario treten zwei gleich konfigurierte Teams gegeneinander an. Allen Bots wird somit das *PERSO_AGGROKILLER*-Profil zugewiesen, was die Bekämpfung gegnerischer Schiffe der UP-Eroberung deutlich bevorzugt.

Simulationsparameter:

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 4: Simulator- und Spielparameter für das 1. Testszenario.

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Personality Profile:	PERSO_AGGROKILLER	Fire Accuracy:	100%.
Enemy Detection Range:	25000	Fire Rate:	70%.
UP Detection Range:	50000		

Tabelle 5: Bot-Parameter von Team Alpha & Bravo für das 1. Testszenario.

Metrik-Ergebnisse:

Metrik	**Ausprägung**	**Metrik**	**Ausprägung**	**Metrik**	**Ausprägung**
Shots#:	41048	Accuracy:	61.22%	UP-Score:	2.38
PShots:	128.28	Kills#:	261	Bias:	0%
Hits#:	25128	Deaths#:	289	Time:	11:33
PHits:	78.53	KD-Ratio:	90.03%	Memory VIRT:	184 MB
				Memory RES:	110.5 MB

Tabelle 6: Metrik-Ergebnisse von Team Alpha & Bravo für das 1. Testszenario.

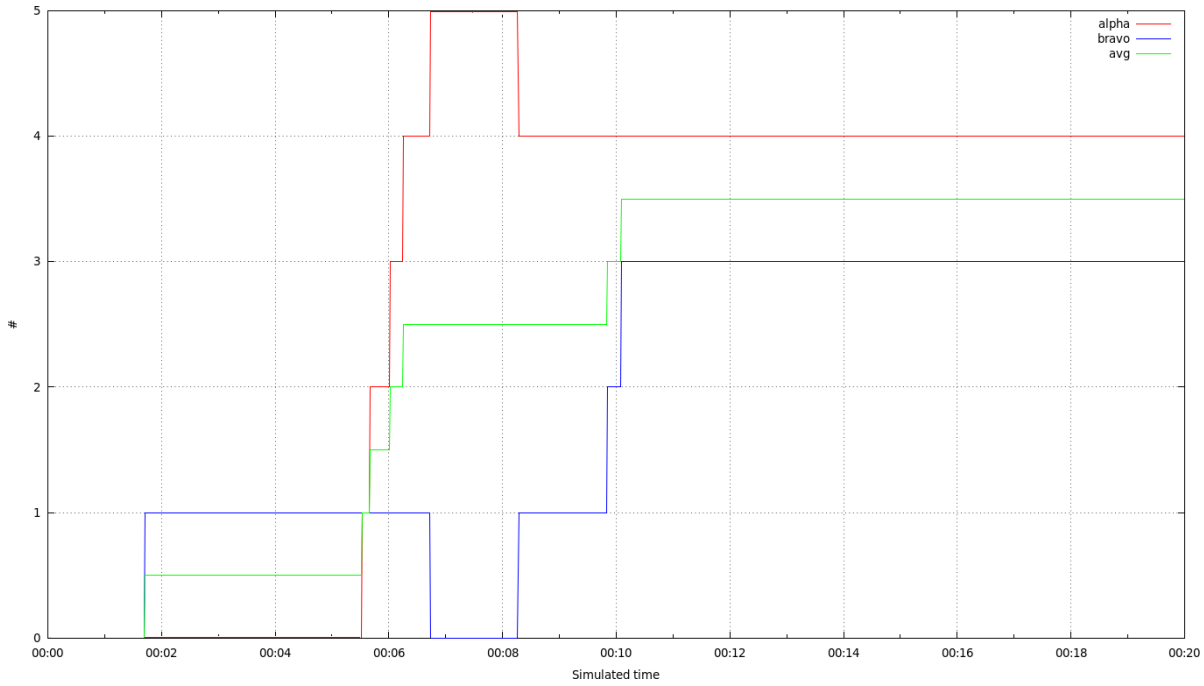

Abbildung 37: Upgrade Point Score Verlauf für das 1. Testszenario.

6.4.2 2. Testszenario

Im zweiten Testszenario wird im Grunde das erste Testszenario erneut ausgeführt, nur die *Fire Accuracy*-Einstellung der Bots wird deutlich reduziert. Damit soll untersucht werden, ob dadurch die Schusshäufigkeit, Schussgenauigkeit und *Kills*-Anzahl beeinflusst werden.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 7: Simulator- und Spielparameter für das 2. Testszenario.

Parameter	Ausprägung	Parameter	Ausprägung
Personality Profile:	PERSO_AGGROKILLER	Fire Accuracy:	10%.
Enemy Detection Range:	25000	Fire Rate:	70%.
UP Detection Range:	50000		

Tabelle 8: Bot-Parameter von Team Alpha & Bravo für das 2. Testszenario.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	59622	Accuracy:	41.90%	UP-Score:	3.6
PShots:	186.32	Kills#:	276	Bias:	0%
Hits#:	24979	Deaths#:	296	Time:	12:22
PHits:	78.06	KD-Ratio:	93.24%	Memory VIRT:	189.5 MB
				Memory RES:	116.5 MB

Tabelle 9: Metrik-Ergebnisse von Team Alpha & Bravo für das 2. Testszenario.

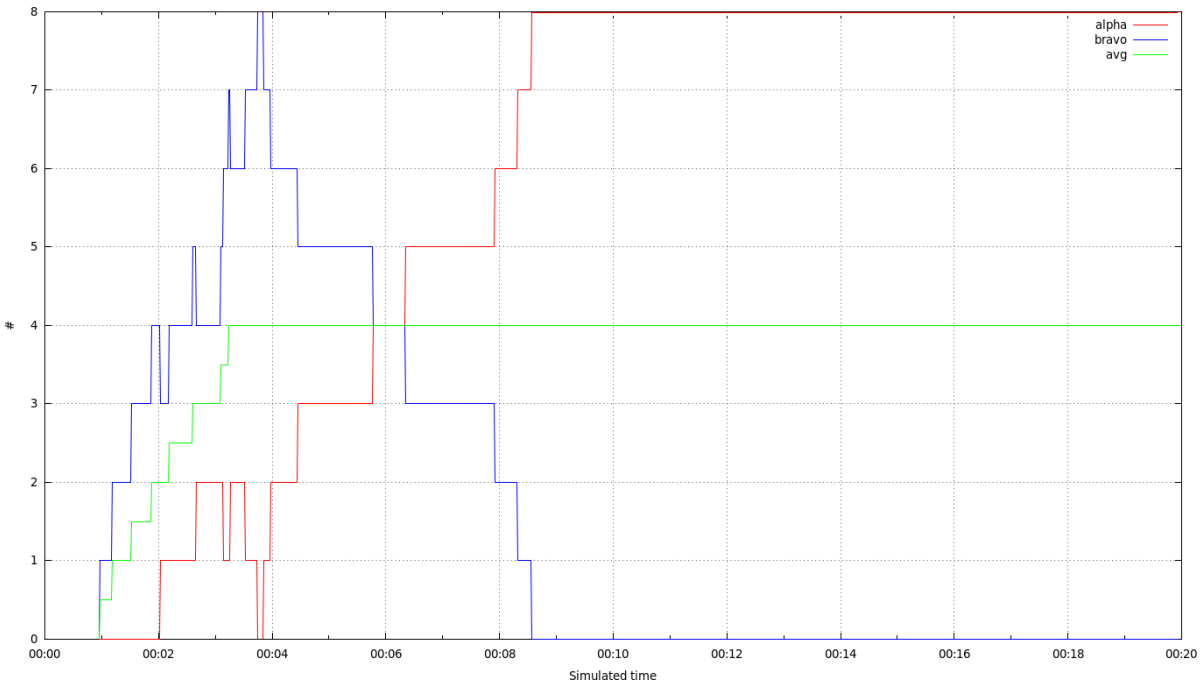

Abbildung 38: Upgrade Point Score Verlauf für das 2. Testszenario.

6.4.3 3. Testszenario

Im dritten Testszenario wird im Grunde das erste Testszenario erneut ausgeführt, nur die *Fire Rate*-Einstellung der Bots wird deutlich reduziert. Damit soll untersucht werden, ob dadurch die Schusshäufigkeit, Schussgenauigkeit und *Kills*-Anzahl beeinflusst werden.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 10: Simulator- und Spielparameter für das 3. Testszenario.

Parameter	Ausprägung	Parameter	Ausprägung
Personality Profile:	PERSO_AGGROKILLER	Fire Accuracy:	100%
Enemy Detection Range:	25000	Fire Rate:	40%
UP Detection Range:	50000		

Tabelle 11: Bot-Parameter von Team Alpha & Bravo für das 3. Testszenario.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	20915	Accuracy:	59.87%	UP-Score:	3.29
PShots:	65.36	Kills#:	146	Bias:	0%
Hits#:	12521	Deaths#:	158		
PHits:	39.13	KD-Ratio:	92.41%		

Tabelle 12: Metrik-Ergebnisse von Team Alpha & Bravo für das 3. Testszenario.

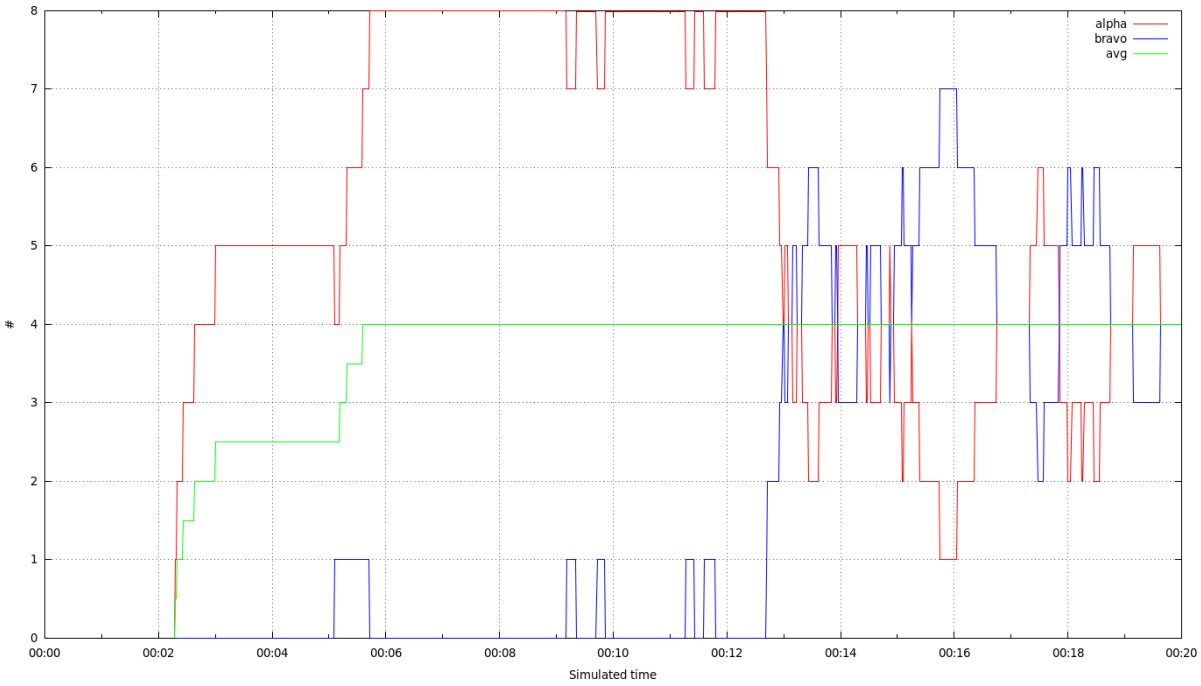

Abbildung 39: Upgrade Point Score Verlauf für das 3. Testszenario.

6.4.4 4. Testszenario

Im vierten Testszenario wird das *Personality Profile* des Bots geändert. Mit der Wahl des *PERSO_AGGROCONQUEROR*-Profils wird die Eroberung von *Upgrade Points* jetzt der Bekämpfung gegnerische Schiffe im freien Raum vorgezogen. In den ersten drei Testszenarien war dies genau anderes herum der Fall. Hier sind besonders die Auswirkungen auf den *UP Score*, die *Kills*-Anzahl und den *UP Score*-Verlauf, sowie auf die Laufzeit- und Speichermessung interessant.

Simulationsparameter:

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 13: Simulator- und Spielparameter für das 4. Testszenario.

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Personality Profile:	PERSO_AGGROCONQUEROR	Fire Accuracy:	100%
Enemy Detection Range:	25000	Fire Rate:	70%
UP Detection Range:	50000		

Tabelle 14: Bot-Parameter von Team Alpha & Bravo für das 4. Testszenario.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	27345	Accuracy:	50.52%	UP-Score:	3.87
PShots:	85.45	Kills#:	161	Bias:	0%
Hits#:	13815	Deaths#:	168	Time:	3:15
PHits:	43.17	KD-Ratio:	95.83%	Memory VIRT:	147.5 MB
				Memory RES:	74.5 MB

Tabelle 15: Metrik-Ergebnisse von Team Alpha & Bravo für das 4. Testszenario.

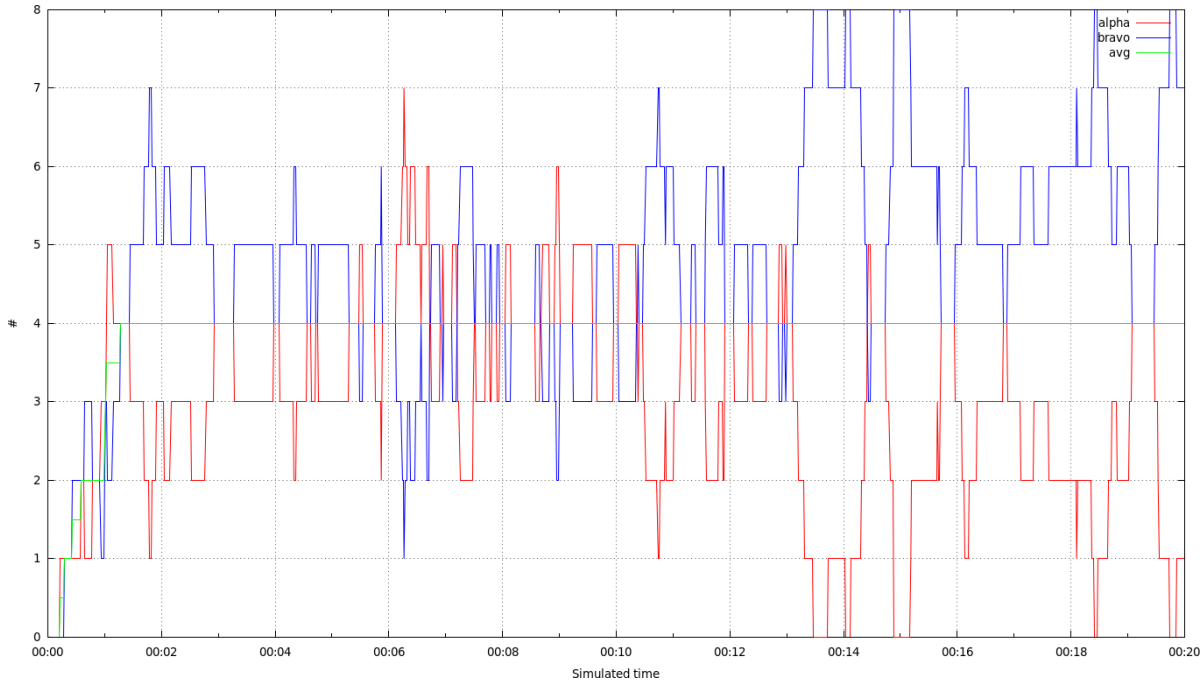

Abbildung 40: Upgrade Point Score Verlauf für das 4. Testszenario.

6.4.5 5. Testszenario

Im fünften Testszenario wird wieder das *Personality Profile* des Bots geändert. Diesmal wird aber nicht wieder ein anderes Profil gewählt, sondern zwei Unterschiedliche innerhalb eines Teams verwendet. Die dabei verwendeten Profile sind die des 1. und 4. Testszenarios. Aus diesem Grund sollten die Ergebnisse irgendwo dazwischen liegen.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	41

Tabelle 16: Simulator- und Spielparameter für das 5. Testszenario.

Parameter	Ausprägung	Parameter	Ausprägung
1. Personality Profile:	4 x PERSO_AGGROKILLER	Fire Accuracy:	100%
2. Personality Profile:	4 x PERSO_AGGROCONQUEROR	Fire Rate:	70%
Enemy Detection Range:	25000	UP Detection Range:	50000

Tabelle 17: Bot-Parameter von Team Alpha & Bravo für das 5. Testszenario.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	44541	Accuracy:	68.84%	UP-Score:	3.8
PShots:	139.19	Kills#:	291	Bias:	0%
Hits#:	30664	Deaths#:	308		
PHits:	95.83	KD-Ratio:	94.48%		

Tabelle 18: Metrik-Ergebnisse von Team Alpha & Bravo für das 5. Testszenario.

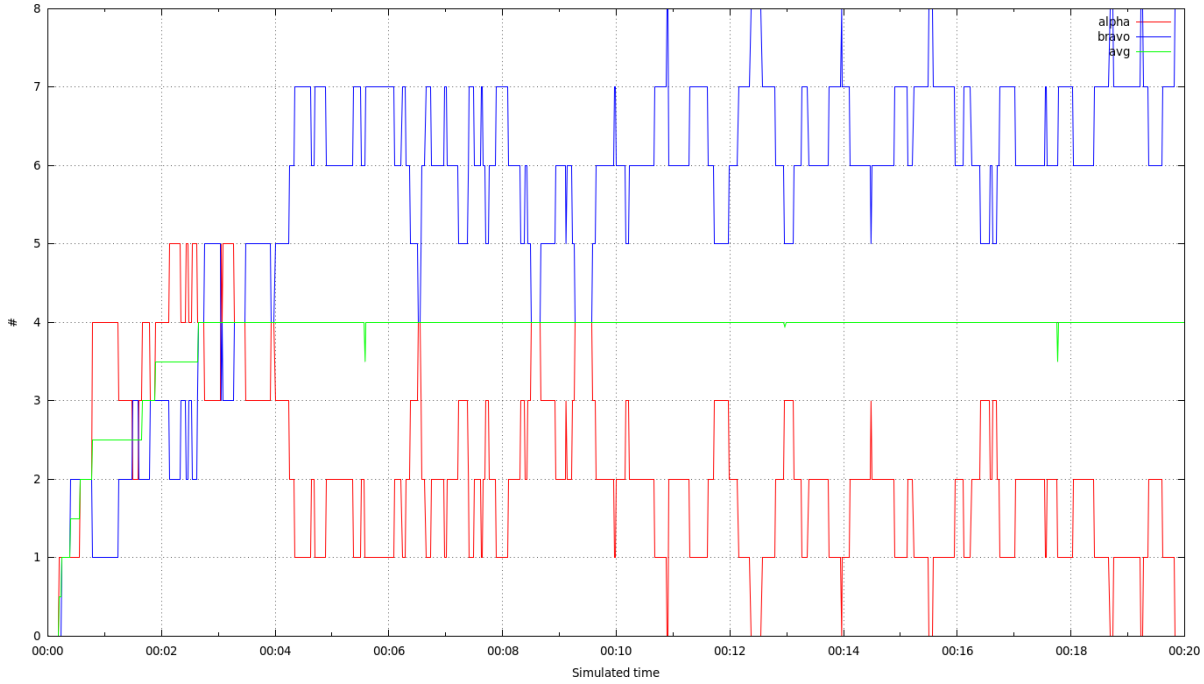

Abbildung 41: Upgrade Point Score Verlauf für das 5. Testszenario.

6.4.6 6. Testszenario

In diesem Testszenario wird die Asteroiden-, Spieler- und UP-Anzahl erhöht und die Teamanzahl auf eins reduziert. Es soll getestet werden, ob der Bot trotz der erschwerten Bedingungen dazu in der Lage ist alle UPs zu finden. Die andere interessante Frage ist, wie realistisch geht der Bot dabei vor? Ist der Bot dazu in der Lage unbeschadet die UPs zu erobert oder kollidiert er währenddessen immer wieder mit den vielen statischen und beweglichen Hindernissen? Die *KD-Ratio* sollte eine Antwort auf diese Frage liefern können.

Simulationsparameter:

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Spieleranzahl:	31	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	1	Netzwerk-Overlay:	CUSP
Verteilung:	31 vs. 0	UP-Anzahl:	18
Spieldauer:	20 Minuten	SG-Anzahl:	0
Asteroiden-Anzahl:	270	Seed:	42

Tabelle 19: Simulator- und Spielparameter für das 6. Testszenario.

Parameter	**Ausprägung**	**Parameter**	**Ausprägung**
Personality Profile:	PERSO_AGGROCONQUEROR	Fire Accuracy:	100%
Enemy Detection Range:	25000	Fire Rate:	70%
UP Detection Range:	50000		

Tabelle 20: Bot-Parameter von Team Alpha für das 6. Testszenario.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	0	Kills#:	0
Hits#:	0	Deaths#:	56
UP-Score:	13.92	Bias:	0%

Tabelle 21: Metrik-Ergebnisse von Team Alpha für das 6. Testszenario.

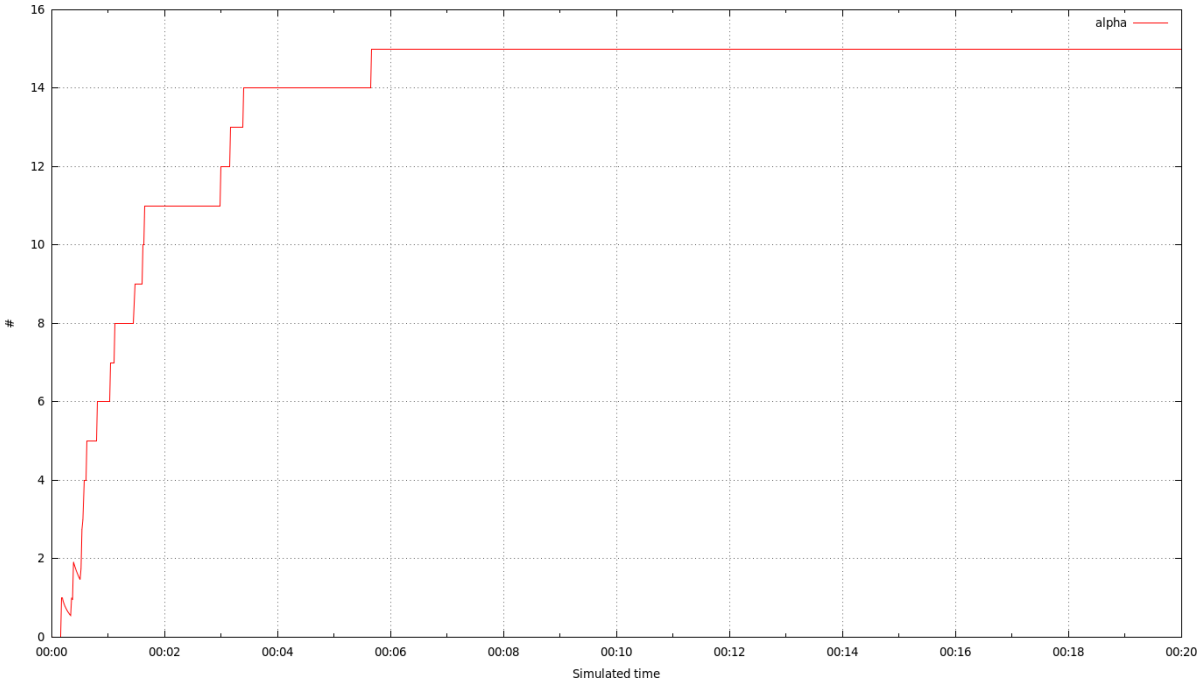

Abbildung 42: Upgrade Point Score Verlauf für das 6. Testszenario.

6.4.7 7. Testszenario

Dieses Szenario entspricht genau dem 6. Testszenario, nur mit dem Unterschied, dass hier die alte Implementierung zu Vergleichszwecken getestet wird.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	31	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	1	Netzwerk-Overlay:	CUSP
Verteilung:	31 vs. 0	UP-Anzahl:	18
Spieldauer:	20 Minuten	SG-Anzahl:	0
Asteroiden-Anzahl:	270	Seed:	42

Tabelle 22: Simulator- und Spielparameter für das 7. Testszenario.

Team Alpha:

- Der *DragonBot* (Vorgänger-Implementierung) in seiner Standard-Konfiguration.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	0	Kills#:	0
Hits#:	0	Deaths#:	144
UP-Score:	11.62	Bias:	0%

Tabelle 23: Metrik-Ergebnisse von Team Alpha für das 7. Testszenario.

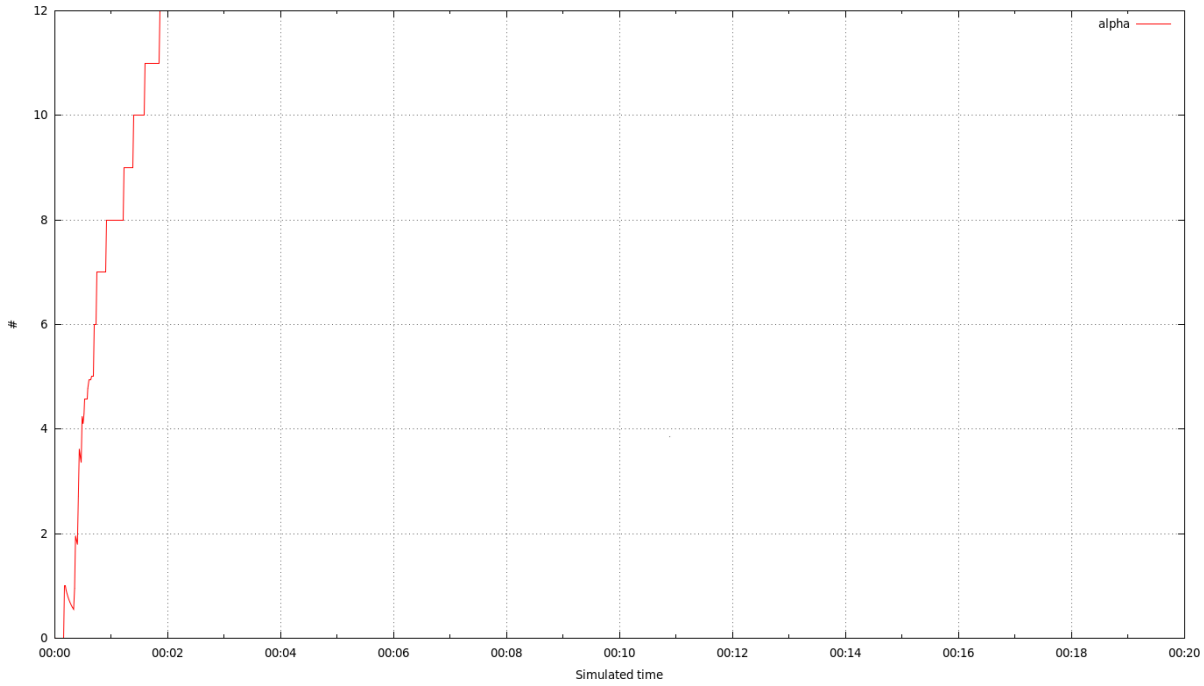

Abbildung 43: Upgrade Point Score Verlauf für das 7. Testszenario.

6.4.8 8. Testszenario

In diesem Testszenario werden die Simulationsparameter der ersten Testszenarien wiederverwendet. Allerdings wird hier nicht die neue Bot-Implementierung, sondern die alte Bot-Implementierung getestet. Dieses Szenario soll die entsprechenden Vergleichswerte produzieren, allen voran für einen Vergleich der Laufzeit und des Speicherverbrauchs.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	16	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	2	Netzwerk-Overlay:	CUSP
Verteilung:	8 vs. 8	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 24: Simulator- und Spielparameter für das 8. Testszenario.

Team Alpha & Bravo:

- Der *DragonBot* (Vorgänger-Implementierung) in seiner Standard-Konfiguration.

Metrik-Ergebnisse:

Metrik	Ausprägung	Metrik	Ausprägung	Metrik	Ausprägung
Shots#:	53819	Accuracy:	51.66%	UP-Score:	3.73
PShots:	168.18	Kills#:	260	Bias:	11.3%
Hits#:	27806	Deaths#:	327	Time:	8:38
PHits:	86.89	KD-Ratio:	79.51%	Memory VIRT:	180 MB
				Memory RES:	92 MB

Tabelle 25: Metrik-Ergebnisse von Team Alpha & Bravo für das 8. Testszenario.

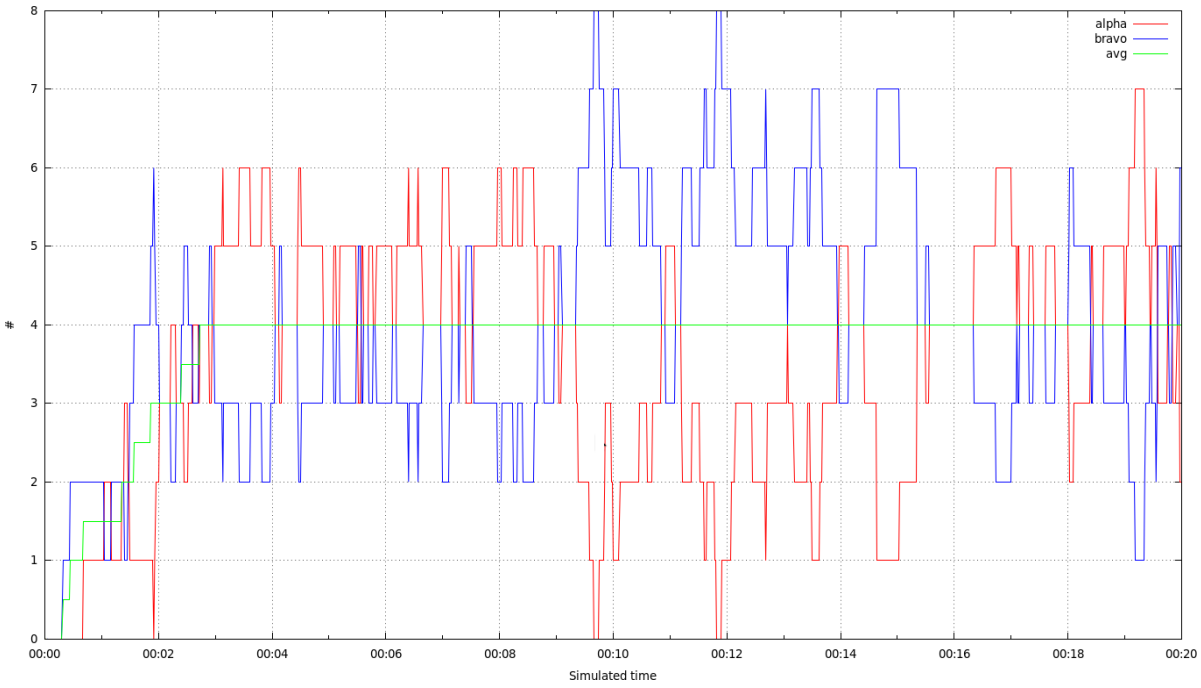

Abbildung 44: Upgrade Point Score Verlauf für das 8. Testszenario.

6.4.9 9. Testszenario

Im letzten Szenario tritt die neue gegen die alte Implementierung zur Messung der allgemeinen Spielstärke an. Des Weiteren soll überpüft werden, ob die erhöhte Team-Anzahl irgendwelche Auswirkungen auf die Daten ausübt.

Simulationsparameter:

Parameter	Ausprägung	Parameter	Ausprägung
Spieleranzahl:	15	Join-Verhalten:	Linear in 00:01
Team-Anzahl:	3	Netzwerk-Overlay:	CUSP
Verteilung:	5 vs. 5 vs. 5	UP-Anzahl:	9
Spieldauer:	20 Minuten	SG-Anzahl:	9
Asteroiden-Anzahl:	225	Seed:	42

Tabelle 26: Simulator- und Spielparameter für das 9. Testszenario.

Parameter	Ausprägung	Parameter	Ausprägung
Personality Profile:	PERSO_AGGROKILLER.	Fire Accuracy:	100%
Enemy Detection Range:	35000	Fire Rate:	100%
UP Detection Range:	50000		

Tabelle 27: Bot-Parameter von Team Alpha für das 9. Testszenario.

Parameter	Ausprägung	Parameter	Ausprägung
Personality Profile:	PERSO_STDCONQUEROR.	Fire Accuracy:	100%
Enemy Detection Range:	35000	Fire Rate:	100%
UP Detection Range:	50000		

Tabelle 28: Bot-Parameter von Team Bravo für das 9. Testszenario.

Team Charlie:

- Der *DragonBot* (Vorgänger-Implementierung) in seiner Standard-Konfiguration.

Metrik-Ergebnisse:

Team	Kills	Deaths	KD-Ratio	UP-Score	Bias
Alpha:	72	96	75%	1.63	6.12%
Bravo:	70	135	51,85%	2.90	4.45%
Charlie:	119	60	198.33%	3.11	12.4%
Alle:	261	291	89.69%	2.55	7.66%

Tabelle 29: Metrik-Ergebnisse von Teams für das 9. Testszenario.

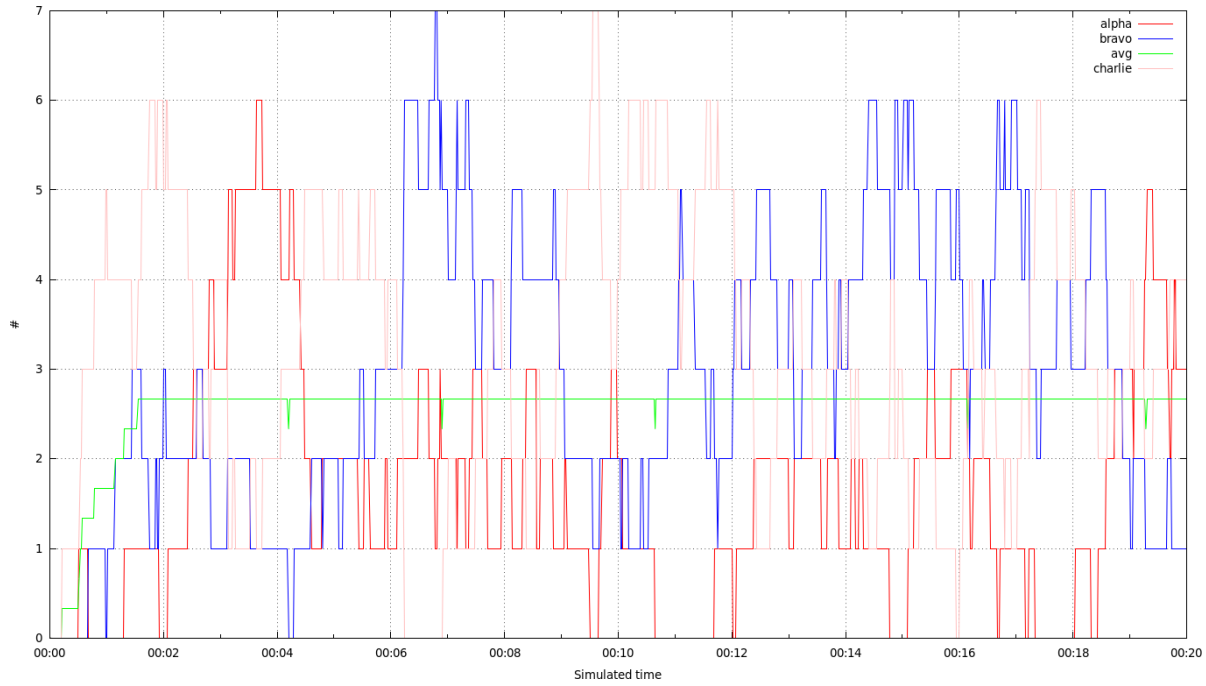

Abbildung 45: Upgrade Point Score Verlauf für das 9. Testszenario.

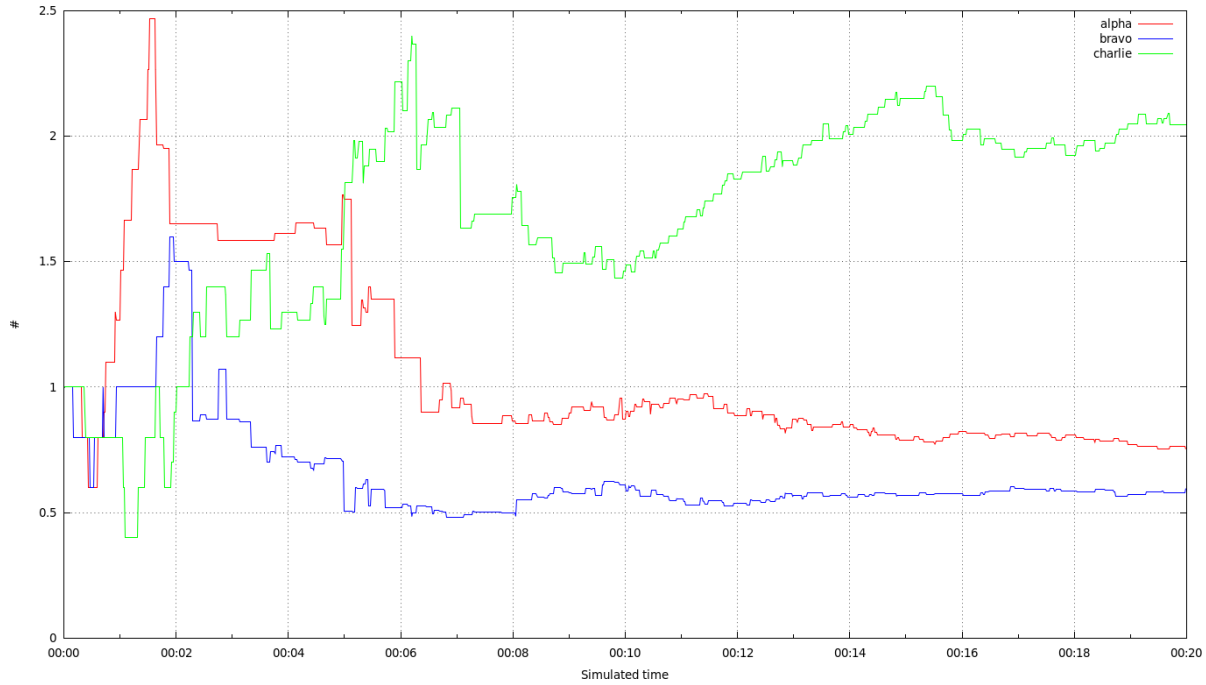

Abbildung 46: Kill/Death-Ratio Verlauf der einzelnen Teams für das 9. Testszenario.

6.5 Auswertung der Ergebnisse

Die Ergebnisse der ersten zwei Testszenarien zeigen, dass der Bot-Parameter *Fire Accuracy* zur Justierung der Schussgenauigkeit eingesetzt werden kann. Diese ist besonders für die realistische Abbildung menschlichen Kampfverhaltens wichtig. Zieht man auch die anderen Testszenarien heran, kann die Schussgenauigkeit in einem Bereich zwischen 40-70% eingestellt werden, je nachdem welche anderen Parameter sich noch zusätzlich auf die Schussgenauigkeit auswirken. Besonders auffällig ist in diesem Zusammenhang der Anstieg der Schussgenauigkeit im 5. Testszenario gegenüber dem 1. und 3. Testsze-

nario um ca. 10% bei gleichbleibender maximaler *Fire Accuracy*. Eine mögliche Erklärung hierfür ist die Mischung der Bot-Teams mit unterschiedlichen *Personality Profiles*. Die *PERSO_AGGROCONQUEROR*-Bots bevorzugen ein Ansteuern nicht eroberter UPs gegenüber der Bekämpfung gegnerischer Schiffe im freien Weltall, auch wenn diese sie leicht abschießen können, weil sie z.B. direkt hinter Ihnen fliegen. Scheinbar sind sie in dieser Anflugphase für *PERSO_AGGROKILLER*-Bots besonders leichte und treffsichere Beute.

Der Vergleich zwischen 1. und 3. Testszenario zeigt, dass der Bot-Parameter *Fire Rate* die Schusshäufigkeit direkt beeinflusst. Anders als bei der Justierung der Schussgenauigkeit über den *Fire Accuracy*-Parameter, wird hierbei allerdings die *Kills*-Anzahl, als Nebeneffekt ebenfalls deutlich mit beeinflusst. Es scheint wohl eine stärkere Abhängigkeit zwischen Schusshäufigkeit und *Kill*-Anzahl als zwischen Schussgenauigkeit und *Kill*-Anzahl in der Bot-Implementierung zu geben. Ob diese Abhängigkeiten realistisch sind, kann erst ein Vergleich mit echten Referenzdaten klären.

Mit der Auswahl des *Personality Profiles* lässt sich klar das Bot-Verhalten dahingehend steuern, ob bevorzugt gegnerische Schiffe bekämpft (*PERSO_AGGROKILLER*) oder eher *Upgrade Points* erobert werden sollen (*PERSO_AGGROCONQUEROR*). Ein Beleg für diese Aussage ist die unterschiedliche *Kills*-Anzahl, Schusshäufigkeit und die Anzahl und Art der UP-Eroberungen. Die abweichende Schusshäufigkeit und die dadurch bedingte geringere *Kills*-Anzahl ergeben sich wohl aus der Tatsache, dass mehr Zeit mit der Ansteuerung von UPs als mit der Bekämpfung der Gegner verbracht wird. Weiter wird mit der Profilauswahl maßgeblich die Geschwindigkeit der Ersteroberung der *Upgrade Points* und die Intensität der Eroberungswechsel bestimmt. Ersteres lässt sich anhand der durchschnittlichen Höhe des *UP Scores* und den *Average*-Verläufen der *UP Score*-Grafiken (Abbildungen 37-40) ableiten. Letzteres wird durch die *UP Score*-Verläufe der jeweiligen Teams deutlich. Während in den Abbildungen 37-39 eher wenig Wechsel stattfinden, erfolgen nach Abbildung 40 im 4. Testszenario mehrere UP-Wechsel pro Spielminute. Ein möglicher Grund, warum gerade in der Anfangsphase der ersten 3 Testszenarien sehr wenige Eroberungen stattfinden, ist die Tatsache, dass die Startpunkte der beiden Teams sehr nah beieinanderliegen und somit diese sich direkt in den Kampf stürzen. Würden die Basen der Teams weiter auseinanderliegen, würden die UP-Verläufe aller Wahrscheinlichkeit eine größere Ähnlichkeit zum 4. Testszenario aufweisen.

Die Ergebnisse des 5. Testszenarios zeigen, dass mit der Mischung der Teams zwar nicht die Anzahl der *Kills* zwischen die der Ergebnisse der ersten drei Testszenarien und dem 4. Testszenario justiert werden kann, dafür allerdings die Art und Weise der UP-Eroberungen. Sowohl der Zeitpunkt der Ersteroberung der UPs, als auch die Intensität der Eroberungswechsel liegt genau zwischen beiden Testszenarien-Gruppen.

Das 6. und 7. Testszenario enthalten jeweils Bots die nur zu einem Team gehören. Mit diesen Szenarien soll untersucht werden, wie sich die neue Bot-Implementierung im Vergleich zur Vorgänger-Implementierung bei der Bewegung im Raum und der Eroberung der *Upgrade Points* verhält. Dazu wurden die Map-Parameter bezüglich Asteroiden-, UP und Spieleranzahl erhöht, um somit erschwerte Bedingungen zu simulieren. Wie die Ergebnisse des 6. Testszenarios zeigen, findet die neue Bot-Implementierung 15 von 18 *Upgrade Points* und das ziemlich schnell. Die Vorgänger-Implementierung findet die *Upgrade Points* in einer vergleichbaren Geschwindigkeit, jedoch ist bei 12 von 18 *Upgrade Points* bereits Schluss. Die anderen Testszenarien mit 9 UPs zeigen, dass sowohl vom neuen, als auch vom alten Bot nur max. 8 UPs gefunden und erobert werden. Ein kleiner Anteil der UPs muss somit in relativ weiten Abstand zu den anderen UPs liegen, sodass die Bots Schwierigkeiten haben diese zu finden.

Bei der Erläuterung der *KD-Ratio*-Metrik in Abschnitt 6.2 wurde aufgeführt, dass diese zur Beurteilung der realistischen Bewegung des Bots im Raum herangezogen werden kann. Sie gibt an, ob alle *Deaths* von regulären *Kills* verursacht worden sind und eben nicht durch andere Todesarten, wie dem Zusammenstoß mit Asteroiden oder befreundeten Schiffen. Die ersten 5 Testszenarien zeigen, dass der

Bot mit einer *KD-Ratio* von über 90% sehr gut in der Lage ist sich im Raum zu bewegen. Allerdings zeigt erst der *KD-Ratio*-Vergleich zwischen dem 6. und 7. Testszenario mit erhöhter Hindernisdichte, dass die neue Bot-Implementierung fast 3-mal geschickter darin ist Hindernissen auszuweichen als die Vorgänger-Implementierung. Das diese Erkenntnis nicht nur auf Umgebungen ohne Gegner beschränkt ist, zeigt die *KD-Ratio* der Vorgänger-Implementierung von ca. 80% im 8. Testszenario.

Das letzte Testszenario erfasst die Spielstärke der einzelnen Bots im direkten Vergleich. Während der Kampf um die UPs relativ ausgeglichen erscheint, besonders zwischen der Vorgänger-Implementierung und den *PERSO_STDCONQUEROR*-Bots, dominiert die Vorgänger-Implementierung nach dem ersten Viertel das Spiel deutlich im Hinblick auf das *Kill/Death*-Verhältnis. Anders als im Kampf um die UPs, ist hier jedoch eher das Team mit den *PERSO_AGGROKILLER*-Bots dazu in der Lage mit der Vorgänger-Implementierung mitzuhalten. Im ersten Viertel des Spiels ist sie sogar das klar dominierende Team, verliert diese Dominanz aber wieder. Ein Grund hierfür ist sicherlich der geringe *UP Score* des Teams. Während am Anfang ausgeglichene Verhältnisse gelten, kann sich der Bot sehr gut behaupten. Der Vorgänger-Implementierung gelingt es aber durch die UP-Eroberung sich nach und nach einen Boni-Vorteil bezüglich *Hitpoints* und *Energy* zu verschaffen. Die *PERSO_STDCONQUEROR*-Bots die sich ebenfalls diesen Vorteil erkämpfen, können diesen jedoch nicht ausnutzen. Die ungeschützte Anflugphase zu den *Upgrade Points* oder eine zu geringe *EnemyDetectionRange* um die *Upgrade Points* herum, werden hier aller Wahrscheinlichkeit nach die Gründe für die Unterlegenheit sein. In weiteren Testläufen sollte untersucht werden, wie sich gemischte Teams gegen die Vorgänger-Implementierung behaupten oder was für Auswirkungen die Justierung der *EnemyDetectionRange* zur Folge hätte.

Beim 1., 2., 4. und 8. Testszenario wurden Laufzeit- und Speichermessungen vorgenommen. Die ersten drei Testszenarien wurden dabei mit der neuen Bot-Implementierung ausgeführt und das letzte mit der Vorgänger-Implementierung. Generell lassen die Daten auf eine leicht erhöhte virtuelle Speichernutzung von bis zu 5% und eine höhere physikalische Speicherlast von ca. 20-25% der neuen Implementierung gegenüber der Alten schließen. Die Laufzeit weicht besonders in dem 2. Testszenario, welches die ähnlichsten Eckdaten im Vergleich zum 8. Testszenario aufweist, um ca. 43% nach oben hin ab. Auf der anderen Seite lassen die Ergebnisse des 4.Testszenarios erahnen, dass die neue Bot-Implementierung auch eine bessere Laufleistung und Speicherlast als die Vorgänger-Implementierung vorweisen kann. Im 4. Testszenario wurde eine niedrigere Speicherlast von ca. 23% und eine um mehr als die Hälfte der Zeit beschleunigte Simulationsberechnung ermittelt. Allerdings ist besonders der Vergleich zwischen dem 4. und 8. Testszenario mit Vorsicht zu genießen, da die Eckdaten der Simulationen, wie *Kills*-Anzahl und Schusshäufigkeit, hier besonders stark voneinander abweichen.

Als letzten Punkt lässt sich festhalten, dass nur die Simulation einen Bias von 0% aufweisen, die ausschließlich die neue Bot-Implementierung untersuchen. Das bedeutet, dass jede dieser Simulation zwei Mal exakt dieselben Ergebnisse produziert hat. Damit sind die gewonnen Ergebnisse dieser Testszenarien mit dem Simulator reproduzierbar und das Repoduzierbarkeits-Kriterium aus Abschnitt 1.2 vollständig erfüllt. Die Simulationen hingegen mit Beteiligung der alten Bot-Implementierung weißen eine Ergebnisabweichung von ca. 10% auf, was eindeutig auf einen Nichtdeterminismus der alten Bot-Implementierung schließen lässt. Da aber das 7. Testszenario ebenfalls einen Bias von 0% besitzt, bedeutet dies, dass der Nichtdeterminusmus der Vorgänger-Implementierung sich aus dem Kampfverhalten des Bots zu ergeben scheint.

Zusammenfassend zeigen die Ergebnisse der Evaluation das die neue Bot-Implementierung dazu in der Lage ist das Spiel auf einem hohen Niveau zu spielen, besonders in Hinblick auf die Fähigkeit zur Vermeidung von Kollisionen und der Erkundung des Raums - beides Aspekte, wo gerade die alte Implementierung ihre Schwächen hat. Weiter konnte gezeigt werden, dass viele wichtige Parameter der Lasterzeugung, wie die Bevorzugungseinstellung von *Kills* gegenüber der Eroberung von UPs, sowie

die Anpassung der Schussgenauigkeit oder der Schusshäufigkeit von der Bot-Implementierung gezielt steuerbar sind. Was eine für die Generierung der Netzwerklast wichtige Eigenschaft der neuen Bot-Implementierung darstellt. Auf der anderen Seite geht diese neue Mächtigkeit zu Lasten der Performance. Im Allgemeinen weist die neue Implementierung eine gegenüber der alten Bot-Implementierung schlechtere Performance auf, die sich allerdings in einem noch tolerierbaren Rahmen zu bewegen scheint. Wie groß der Performance-Unterschied tatsächlich ist und welche eventuellen negativen Auswirkungen dieser Umstand mit sich führen kann, müssen weitere und genauere Test zeigen.

7 Ausblick und Fazit

Der in dieser Arbeit vorgestellte Ansatz zur Erzeugung einer realistischen Netzwerklast mittels kontextsensitiver AI-Spieler stellt noch kein vollkommen ausgereiftes Produkt dar. Vielmehr wurde eine solide und funktionsfähige erste Version fertiggestellt, die bereits viele der in der Einleitung definierten Ziele in einem zufriedenstellenden Maße erfüllt und somit als Grundlage für weitere Arbeiten genutzt werden kann. Welche Ziele in welchem Ausmaß weshalb dabei erreicht wurden, fasst der Abschnitt 7.2 „Fazit“ zusammen.

Abschnitt 7.1 „Ausblick“ stellt hingegen zuvor vier wichtige Bereiche für weiterführende Entwicklungen vor und skizziert dabei ebenfalls erste mögliche Lösungsansätze für diese. Die ersten beiden würden zu einer Verbesserung der Verhaltenssteuerung bzw. des zur Verfügung stehenden Verhaltensrepertoires des Bots führen, wohingegen die anderen beiden mögliche Performance-Verbesserungen behandeln.

7.1 Ausblick

Das Konzept der *Personality Traits* aus Abschnitt 5.4.3 ermöglicht die Beeinflussung der Traversierung des BT-Baumes. Dies ist sowohl zur Kompilierzeit, als auch zur Laufzeit möglich. Wie in der Evaluation gezeigt, können somit leicht Teams mit unterschiedlichen Bot-Verhaltensweisen zusammengestellt werden. Momentan hängt die Traversierungsentscheidung der *Personality Selectors* nur von der konfigurierten Persönlichkeitseinstellung des Bots ab. Eine mögliche Verbesserung wäre diese Auswahl durch zusätzliche Spielzustandsinformationen zu erweitern. Zwar können auf diese mittels *Conditions* zugegriffen werden, die sich innerhalb der Unterbäume die von den *Personality Selectors* ausgewählt werden befinden, und somit zum Scheitern des Unterbaums eingesetzt werden, doch wird die eigentliche Traversierungsreihenfolge davon nicht beeinflusst. Eine Einbeziehung von Spielinformationen zur Zeit der Traversierungsauswahl könnte zu einer leichteren Modellierung komplexerer Verhaltensentscheidungen führen.

Das einzig völlig verpasste Teilziel bei der Entwicklung der Bot-Implementierung ist die fehlende Berücksichtigung oder Unterstützung von Gruppenverhalten. Gruppenverhalten stellt einen wichtigen Aspekt eines MMOGs dar und sollte von einer Wert auf Realismus legenden Bot-Implementierung berücksichtigt werden. Außerdem liegt die Vermutung nahe, dass die Steuerung des Gruppenverhaltens ein für die Lasterzeugung wichtiger Faktor darstellt, da diese die die Anzahl der Nachbarn eines *peers* mitbeeinflussen sollte. Die Anzahl der Menge der Nachbarn hat wiederum unmittelbaren Einfluss auf die Menge der auszutauschenden Nachrichten zwischen den *peers* und bestimmt dadurch indirekt die Intensität der Netzwerklast.

Die Integration von Gruppenverhalten in die Bot-Implementierung sollte dabei nicht als vom *Behavior Tree* abgekoppeltes (Sub-)System erfolgen. Vielmehr sollte ähnlich der in Crysis 2 umgesetzten BT-Erweiterung [41] die Integration des Gruppenverhalten als fester Bestandteil des *Behavior Trees* erfolgen. Die Entscheidung zur Partizipation an Gruppenverhalten sollte somit das Ergebnis des Entscheidungsprozesses der BT-Traversierung sein und nicht von außen vorgegeben oder entschieden werden. Die Integration jedes möglichen Gruppenverhaltens als fester Bestandteil des Baums sollte dabei ebenfalls vermieden werden. So eine Integration würde nicht nur die Größe des BT-Baumes unnötig vergrößern, sondern auch negative Auswirkungen auf die Laufzeit haben, da in jedem Entscheidungszyklus die Gruppenverhaltens-Knoten des BTs bei der Traversierung mitberücksichtigt werden könnten, auch wenn diese aktuell gar nicht relevant sind. Denn die Nicht-Relevanz müsste über *Conditions*-Abfragen

jedes Mal aufs Neue erst überprüft werden.

Das Konzept der *Behavior Tags* aus Abschnitt 5.4.3 könnte hingegen zur Integration bestimmter Gruppenverhaltensweisen in den BT zur Laufzeit eingesetzt werden. Die *Behavior Tags* könnten zur Aktivierung und Deaktivierung bestimmter Platzhalter-Knoten im Baum eingesetzt werden, um ein Event-basiertes Einfügen von aktuell zu berücksichtigten Gruppenverhalten in die dafür vorgesehenen Platzhalter zu ermöglichen. Die Platzhalter-Knoten müssten entsprechend noch entwickelt werden, die Mechanik zum Auffinden, sowie der Aktivierung und Deaktivierung der Platzhalter im Baum müsste bereits mit der aktuellen Version der *Behavior Tags* funktionieren. Des Weiteren könnte zur Bestimmung der aktuell zu berücksichtigen Gruppenverhaltensweisen eine Erweiterung des in Abschnitt 5.4.1 präsentierten *Blackbaords* eingesetzt werden. Mit Hilfe des *Blackboards* werden die Daten innerhalb des BT ausgetauscht. Dieses Prinzip ließe sich relativ leicht auf den Daten-Austausch zwischen den einzelnen Bot-Instanzen über die bereits vorhandenen Kommunikations-Mechanismen realisieren. Allerdings müsste auf die Synchronisierung der Daten auf dem *Blackboard* geachtet werden, da durch mögliche Nachrichtenverzögerungen verursachte Inkonsistenzen nicht auszuschließen sind.

Die Laufleistung kann bereits mit dem *Process*-Parameter der drei Ebenen etwas reguliert werden, da dieser bestimmt in welchem Zyklus die einzelnen Ebenen ausgeführt werden. Die Evaluation zeigte allerdings, dass unabhängig davon die allgemeine Performance hinter der Vorgänger-Implementierung liegt, was jedoch aufgrund der höheren Komplexität und der viel größeren Menge an beteiligten Klassen und Objekten nicht weiter verwundert. Nichtsdestotrotz lässt sich die Performance wahrscheinlich durch das Cachen der *IWorldSate*-Informationen deutlich verbessern. Weil es keinen linearen Datenfluss innerhalb des Bots gibt, können bereits abgerufene oder aufbearbeitete Informationen nicht einfach direkt weitergereicht werden. Die *IWorldState*-Schnittstelle jedoch bietet einen zentralen Zugriffspunkt auf viele wichtige Informationen an, der von überall aus im Bot genutzt wird. Würden die Informationen dieser Schnittstelle oder oft benötigte Ergebnisse zwischengespeichert werden, könnte die Ausführung einiger aufwendiger Abfragen vermieden werden. Ein Beispiel hierfür wäre der Zugriff auf die Liste naher Asteroiden, die sowohl von der Hinderniserkennung, als auch vom *Combat System* separat abgerufen und auf sehr ähnliche Art und Weise durchsucht wird.

Wie die Evaluation zeigte liegt nicht nur die Laufleistung hinter der Vorgänger-Implementierung zurück, sondern auch die Speicherauslastung ist im Schnitt etwas höher. Die wahrscheinlich größte Speicherbelastung wird durch die Verwaltung des *Behavior Trees* verursacht. Dieser wird zur Initialisierungszeit einmal vollständig erstellt und nahezu unverändert bis zum Ende der Partie im Speicher gehalten. Eine Verbesserungsmöglichkeit wäre die Entwicklung eines feingranularen Erstellungsprozesses, welcher sowohl die gezielte Erstellung einzelner Unterbäume ermöglicht, als auch die Verschiebung dieser Erstellung zu einem beliebigen Zeitpunkt der Laufzeit. Eine Art *BT-Library* könnte den Erstellungsprozess und den Zugriff auf bereits erstellte Knoten verwalten, sodass die eigentliche Baumstruktur lediglich aus Platzhalterknoten besteht, die erst zur Traversierungszeit die erforderlichen Knoten aus der *BT-Library* anfordern. Die Platzhalterknoten könnten leicht als neue *Decorator*-Knotenart integriert werden und demzufolge den bereits existierenden Erstellungsprozess des BT nutzen.

7.2 Fazit

Mit dieser Arbeit wurde erfolgreich ein Grundstein für die synthetische Generierung einer realistischen Netzwerklast zu Evaluationszwecken verschiedenartiger P2P-Overlays mit dem Einsatz kontext-sensitiver AI-Spieler in *Planet PI4* geschaffen. Unabhängig der in Abschnitt 7.1 ausgemachten Problembereiche und den diesbezüglichen Verbesserungsvorschlägen, konnte anhand der Evaluationsergebnisse aus Kapitel 6 eindeutig eine generelle Eignung der aktuellen Version zur Generierung und vor allem zur Steuerung der Netzwerklast nachgewiesen werden. Weiterhin konnte die Evaluation zeigen, dass alle vier in Abschnitt

1.2 definierten Ziele während der Entwicklung beachtet wurden und von der aktuellen Version des Bots in einer zufriedenstellenden Weise erfüllt werden. Diese Beurteilung ermöglichte insbesondere der direkte Vergleich mit der Vorgänger-Implementierung, die im Hinblick auf die Kriterien Reproduzierbarkeit, Realitätsgrad und der Konfigurierbarkeit der Netzwerklast der neuen Implementierung unterliegt. Einzig in Sachen Skalierbarkeit konnte sich die aktuelle Bot-Implementierung nicht gegenüber der alten Implementierung durchsetzen, der dabei gemessene Performance Unterschied scheint allerdings nicht besorgniserregend groß zu sein, ist aber auf der anderen Seite auch nicht als vernachlässigbar klein zu erachten.

Eines der Gründe für die im Vergleich zur Vorgänger-Implementierung schlechtere Performance ist das komplexere und aufwendigere Design des neuen Bots. Wie in Kapitel 4 aufgeführt sind in der neuen Implementierung zahlreiche Klassen auf verschiedenen Ebenen im Einsatz. Besonders die Aufteilung in verschiedene Ebenen und der dadurch bedingte erhöhte Bedarf an Schnittstellen zur sauberen Entkoppelung, sowie der Verzicht auf einen einfachen linearen Informationsfluss im Bot gehen eindeutig zu Lasten der Performance. Der Lösungsansatz der alten Vorgänger-Implementierung beruht, wie in [59] und teilweise in Abschnitt 3.5 beschrieben, stattdessen eher auf einer simpleren Systemstrukturierung. Das komplexere Systemdesign hat allerdings einen nicht zu unterschätzenden Vorteil gegenüber der alten Implementierung, es erlaubt eine saubere Trennung der Verantwortlichkeiten im Bot. Darüber hinaus stellt die vorgenommene Ebenen Auf- und Einteilung eine Erweiterung des Konzeptes von Reynolds dar (siehe Abschnitt 2.4), welches wiederum ein sehr etabliertes und weit verbreitetes Konzept darstellt.

Ein Beispiel soll diesen Vorteil verdeutlichen: In der Evaluation wurde ermittelt, dass die Bot-Implementierung zu über 90% dazu in der Lage ist Hindernissen auszuweichen. Das Ziel einer möglichen Weiterentwicklung könnte beispielsweise die Ausbesserung der fehlenden 10% sein. Wenn weitere Untersuchungen dazu noch ergeben würden, dass die fehlenden 10% nur aufgrund der vereinzelten Kollision mit Asteroiden und nicht mit befreundeten Schiffen verursacht werden. So wäre der Ort der Ausbesserung bei einer Weiterentwicklung des Bots schnell indentifiziert, da das Design den Ort der statischen Hinderniserkennung mit der *ObstacleAvoidance*-Klasse der *Steering Pipeline* der operativen Ebene an einer genau definierten Stelle vorsieht. Die *ObstacleAvoidance*-Klasse könnte jetzt in dem Beispiel entweder dahingehend verbessert werden oder sogar komplett ausgetauscht werden, ohne dass damit das ganze System bzw. die anderen Ebenen betroffen wären. Das Beispiel zeigt demnach, dass die saubere Trennung besonders im Hinblick auf zukünftige Weiterentwicklungen einen wichtigen Vorteil bietet.

Abbildungsverzeichnis

1 Der schematische Aufbau eines simplen rationalen Agenten. Angelehnt an [46] und [36]. 13
2 Der *Nutzenbasierte Agent* und seine wichtigsten Komponenten als Erweiterung des simplen Agenten aus Abbildung 1. Angelehnt an [46] und [36]. 14
3 Das Modell nach Reynolds zur hierarchischen Unterteilung der Verantwortlichkeiten zur Bewegungssteuerung eines autonomen Agenten. Entnommen aus [44]. 15

4 (a) Nachbildung des Bewegungsprofils der *Quake 2*-Partie aus (b) mit dem RWP-Modell. (b) Bewegungsprofil einer echten *Quake 2*-Partie. Beide Grafiken entnommen aus [53]. . . 19
5 Nachbildung des Bewegungsprofils der *Quake 2*-Partie aus 4 (b) mit dem NGMM-Modell. Grafik entnommen aus [53]. 20
6 Die Game AI-Architektur von Halo. Entnommen aus [9]. 22
7 Ein Beispiel für den von GOAP realisierten Entscheidungsprozess zur Auswahl der nächsten Aktion. Beachtet werden sollte das die Planungsrichtung (Roter Pfeil) entgegengesetzt zur Ausführungsrichtung ist. Angelehnt an Beispiel und eine Grafik aus [37]. 26
8 Ein instabiles Equilibrium. Angelehnt an eine Grafik aus [35, S. 100]. 27
9 Ein stabiles Equilibrium. Entnommen aus [35, S. 101]. 27
10 Die Evaluierung der möglichen Ausweichbewegungen zur Verhinderung der Kollision mit Hindernissen. Angelehnt an zwei Grafiken aus [3]. 29

11 Die grafische Benutzeroberfläche von *Planet PI4*. Angelehnt an eine Grafik aus [30]. 34
12 Das *Game AI*-Konzept als Erweiterung des Agentenmodells nach Reynolds und der Integration dessen in das Konzept eines rationalen (nutzenbasierten) Agenten. 37
13 Beispiel für einen *Behavior Tree* fürs Eintreten in einen Raum. Entnommen aus [35, S. 339]. 39

14 Klassendiagramm zur Einbindung einer *Game AI* in *Planet PI4*. 42
15 Die Game AI-Architektur mit dem Zusammenspiel der wichtigsten Komponenten und Subsysteme. 44
16 Die feste Ausführungsreihenfolge der Komponenten der Game AI und der von Planet PI4 innerhalb eines Spielzyklus. 46
17 Die *IOptController*-Schnittstelle die den Leistungskatalog der operativen Ebene definiert. . 47
18 Die Struktur der operativen Ebene und die Abhängigkeiten zu anderen Systemebenen. . . 48
19 (a) Die generelle Struktur der *Steering Pipeline*. Angelehnt an [35, S. 108]. (b) Der Aufbau und die Elemente der *SteeringForce*-Datenstruktur, die von der hier vorgestellten *Steering Pipeline*-Implementierung verwendet wird. 49
20 Vererbungshierarchie der *Targeter*-Menge der *Steering Pipeline*. 50
21 Vererbungshierarchie der *Constraint*-Menge der *Steering Pipeline*. 52
22 Vererbungshierarchie der *ShipActuator*-Klasse der *Steering Pipeline*. 53
23 Die *CombatSystem*-Klasse und die *ICombatShip*-Schnittstelle, welche die vom *Combat System* benötigten Informationen und Steuerungsbefehle zur Aktionsrealisierung definiert. Die Parameter der *ICombatShip*-Schnittstelle wurden zur besseren Übersicht weggelassen. 54
24 Die Struktur der taktischen Ebene und die Abhängigkeiten zu anderen Systemebenen. . . 56
25 Klassendiagramm der wichtigsten übergeordneten Komponenten der BT-Implementierung. 60
26 Vererbungshierarchie und Typspezifikation der *Conditions* der BT-Implementierung. 62
27 Beispielhafter Einsatz von *Behavior Tags*. 64
28 Die Elemente der *Personality*-Erweiterung. 65
29 Beispielhafter Einsatz der Personality-Erweiterung. 66

30 Oberste Ebene des konkreten *Behavior Trees* der *Game AI*. 67
31 Der modellierte Auswahlprozess des *CombatTactic*-Unterbaums. Es wird entweder keine Aktion, die *CaptureUP*-Aktion oder der *FightTactic*-Unterbaum zur näheren Auswahl des konkreten Kampfverhaltens ausgewählt. 68
32 Der modellierte Auswahlprozess des *FightTactic*-Unterbaums zur Wahl der gewünschten Kampftaktik von gegnerischen Schiffen im freien Raum. Zur Wahl stehen drei unterschiedliche Taktiken. 69
33 Die Struktur der strategischen Ebene und die Abhängigkeiten zu anderen Systemebenen. . 70
34 Aufbau des 3D-Quadrats einer Region mit einer Initialisierungsgröße von 40000. 73
35 Aufteilung des 3D-Quadrats einer Region in seine 8 Unterregionen. 73

36 Übersicht des allgemeinen Testaufbaus. Angelehnt an [31]. 78
37 Upgrade Point Score Verlauf für das 1. Testszenario. 81
38 Upgrade Point Score Verlauf für das 2. Testszenario. 82
39 Upgrade Point Score Verlauf für das 3. Testszenario. 84
40 Upgrade Point Score Verlauf für das 4. Testszenario. 85
41 Upgrade Point Score Verlauf für das 5. Testszenario. 87
42 Upgrade Point Score Verlauf für das 6. Testszenario. 88
43 Upgrade Point Score Verlauf für das 7. Testszenario. 89
44 Upgrade Point Score Verlauf für das 8. Testszenario. 90
45 Upgrade Point Score Verlauf für das 9. Testszenario. 92
46 Kill/Death-Ratio Verlauf der einzelnen Teams für das 9. Testszenario. 92

Tabellenverzeichnis

1 Der Einsatz von *Decision Making*-Verfahren in modernen Computerspielen. 17
2 Kostenbewertung der *Steering*-Vorschläge sowie ihre Kostensumme. 29

3 Auflistung der in der Evaluation verwendeten Metriken mit Wertebereichsangabe und der Zuordnung zu relevanten Kriterien. 76
4 Simulator- und Spielparameter für das 1. Testszenario. 80
5 Bot-Parameter von Team Alpha & Bravo für das 1. Testszenario. 80
6 Metrik-Ergebnisse von Team Alpha & Bravo für das 1. Testszenario. 80
7 Simulator- und Spielparameter für das 2. Testszenario. 81
8 Bot-Parameter von Team Alpha & Bravo für das 2. Testszenario. 81
9 Metrik-Ergebnisse von Team Alpha & Bravo für das 2. Testszenario. 82
10 Simulator- und Spielparameter für das 3. Testszenario. 83
11 Bot-Parameter von Team Alpha & Bravo für das 3. Testszenario. 83
12 Metrik-Ergebnisse von Team Alpha & Bravo für das 3. Testszenario. 83
13 Simulator- und Spielparameter für das 4. Testszenario. 84
14 Bot-Parameter von Team Alpha & Bravo für das 4. Testszenario. 84
15 Metrik-Ergebnisse von Team Alpha & Bravo für das 4. Testszenario. 85
16 Simulator- und Spielparameter für das 5. Testszenario. 86
17 Bot-Parameter von Team Alpha & Bravo für das 5. Testszenario. 86
18 Metrik-Ergebnisse von Team Alpha & Bravo für das 5. Testszenario. 86
19 Simulator- und Spielparameter für das 6. Testszenario. 87
20 Bot-Parameter von Team Alpha für das 6. Testszenario. 87
21 Metrik-Ergebnisse von Team Alpha für das 6. Testszenario. 88
22 Simulator- und Spielparameter für das 7. Testszenario. 88
23 Metrik-Ergebnisse von Team Alpha für das 7. Testszenario. 89
24 Simulator- und Spielparameter für das 8. Testszenario. 90
25 Metrik-Ergebnisse von Team Alpha & Bravo für das 8. Testszenario. 90
26 Simulator- und Spielparameter für das 9. Testszenario. 91
27 Bot-Parameter von Team Alpha für das 9. Testszenario. 91
28 Bot-Parameter von Team Bravo für das 9. Testszenario. 91
29 Metrik-Ergebnisse von Teams für das 9. Testszenario. 91

Literaturverzeichnis

[1] Features. URL http://irrlicht.sourceforge.net/features/. letzter Zugriff am 29.09.2012.

[2] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking*, 22:1–17, 2007.

[3] Heni Ben Amor, Jan Murray, and Oliver Obst. Fast, neat and under control: Inverse steering behaviors for physical autonomous agents, 2003.

[4] Heni Ben Amor, Jan Murray, and Oliver Obst. *AI Game Programming Wisdom 3*, chapter Fast, Neat and Under Control: Inverse Steering Behaviors for Physical Autonomous Agents, pages 221–232. Charles River Media, 2006.

[5] Ashwin Bharambe. Colyseus: A distributed architecture for online multiplayer games. In *In Proc. Symposium on Networked Systems Design and Implementation (NSDI*, pages 3–06, 2006.

[6] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games, 2008.

[7] Michael S. Borella. Source models of network game traffic. *Computer Communications*, 23:403–410, 2000.

[8] Brookhaven National Lab. The First Video Game? before 'pong,' there was 'tennis for two'. Website. http://www.bnl.gov/bnlweb/history/higinbotham.asp, letzter Zugriff am 29.09.2012.

[9] Chris Butcher and Jaime Griesemer. The illusion of intelligence: The integration of ai and level design in halo, 2002.

[10] Alex J. Champandard. Behavior trees for next-gen game ai. Website, Dec 2008. http://aigamedev.com/insider/article/behavior-trees/, letzter Zugriff am 29.09.2012.

[11] D.S. Cohen. Oxo aka Noughts and Crosses - the first video game. Website. http://classicgames.about.com/od/computergames/p/OXOProfile.htm, letzter Zugriff am 29.09.2012.

[12] Johnny Cullen. Modern warfare 3 hits $1 billion in 16 days – info. Website, Dec 2011. http://www.vg247.com/2011/12/12/mw3-hits-1-billion-in-16-days/, letzter Zugriff am 29.09.2012.

[13] Entertainment Software Association. Essential Facts about the Computer and Video Game Industry. Technical report, Entertainment Software Association, 2011. http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf, letzter Zugriff am 29.09.2012.

[14] Lu Fan, Phil Trinder, and Hamish Taylor. Design issues for peer-to-peer massively multiplayer online games, 2009.

[15] J.D. Funge. *Artificial Intelligence for Computer Games: An Introduction*. Ak Peters Series. Peters, 2004.

[16] Erich Gamma. *Entwurfsmuster*. Addison-Wesley Verlag, 1st edition, 2004.

[17] Christian Groß, Max Lehn, Christoph Münker, Alejandro Buchmann, and Ralf Steinmetz. Towards a comparative performance evaluation of overlays for networked virtual environments. In IEEE, editor, *Proceedings of the 11th IEEE International Conference on Peer-to-Peer Computing*, pages 34–43. IEEE, Sep 2011.

[18] Thorsten Hagenloch. *Einführung in die Betriebswirtschaftslehre: Theoretische Grundlagen und Managementlehre*. Books on Demand GmbH, Norderstedt, Germany, 2009.

[19] S.-Y. Hu and G.-M. Liao. Von: A scalable peer-to-peer network for virtual environments. *IEEE Network*, 20 No. 4:22–31, 2006.

[20] Damian Isla. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*, March 2005. URL `http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml`. letzter Zugriff am 29.09.2012.

[21] Damian Isla and Bruce Blumberg. Blackboard architectures. *AI Game Programming Wisdom*, pages 333–344, 2002.

[22] Damian Isla, Robert Burke, Marc Downie, and Bruce Blumberg. A layered brain architecture for synthetic creatures. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 2*, IJCAI'01, pages 1051–1058, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[24] Geraint Johnson. *AI Game Programming Wisdom 3*, chapter Goal Trees, pages 301–310. Charles River Media, 2006.

[25] Denis Lapiner. Gameplay design and implementation for a massively multiplayer online game. Bachelor thesis, Technische Universität Darmstadt, Germany, Mar 2011.

[26] Nachi Lau. *AI Game Programming Wisdom 4*, chapter Knowledge-Based Behavior System: A Decision Tree/Finite State Machine Hybrid, pages 265–274. Charles River Media, 2008.

[27] Max Lehn. Implementation of a peer-to-peer multiplayer game with realtime requirements. Master's thesis, Technische Universität Darmstadt, Germany, Oct 2009.

[28] Max Lehn, Christof Leng, Robert Rehner, Tonio Triebel, and Alejandro Buchmann. An online gaming testbed for peer-to-peer architectures. In *Proceedings of ACM SIGCOMḾ11*. ACM, August 2011. Demo.

[29] Max Lehn, Tonio Triebel, and Wolfgang Effelsberg. Benchmarking p2p gaming overlays. 2011.

[30] Max Lehn, Benjamin Guthier, Robert Rehner, Tonio Triebel, Stephan Kopf, and Wolfgang Effelsberg. Generation of synthetic workloads for peer-to-peer gaming benchmarks. 2012.

[31] Christof Leng, Max Lehn, Robert Rehner, and Alejandro Buchmann. Designing a testbed for large-scale distributed systems. In *Proceedings of ACM SIGCOMḾ11*. ACM, August 2011. Poster.

[32] Robert C. Martin. The interface segregation principle. *The C++ Report*, 8, August 1996. URL `http://www.objectmentor.com/resources/articles/isp.pdf`.

[33] Robert C. Martin. The dependency inversion principle. *The C++ Report*, May 1996. URL `http://www.objectmentor.com/resources/articles/dip.pdf`.

[34] Michael Mateas. Expressive ai: Games and artificial intelligence. In *Proceedings of International DiGRA Conference*, 2003.

[35] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc., 2nd edition, 2009.

[36] Alexander Nareyek. Review: Intelligent agents for computer games. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games, Second International Conference, CG 2000, Hamamatsu, Japan, October 26-28, 2000, Revised Papers*, volume 2063 of *Lecture Notes in Computer Science*, pages 414–422. Springer, 2000.

[37] Jeff Orkin. Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom*, 2: 217–228, 2003.

[38] Jeff Orkin. Agent architecture considerations for real-time planning. In *in Games." Artificial Intelligence & Interactive Digital Entertainment (AIIDE*. AAAI Press, 2005.

[39] Jeff Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Game Developers Conference*, 2006.

[40] Jeff Orkin. Getting started with decision making and control systems. *AI Game Programming Wisdom*, 4:257—-264, 2008.

[41] Ricardo Pillosu. Coordinating agents with behaviour trees, 2009. Presentation.

[42] DFG Forschergruppe QuaP2P. Website. `http://www.quap2p.tu-darmstadt.de/`, Zugriff am 22.12.2011.

[43] Jon Radoff. Anatomy of an Mmorpg. Website, Aug 2008. `http://radoff.com/blog/2008/08/22/anatomy-of-an-mmorpg/`, letzter Zugriff am 29.09.2012.

[44] Craig Reynolds. Steering behaviors for autonomous characters, 1999.

[45] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphics*, pages 25–34, 1987.

[46] S. J. Russell and P. Norvig. *Künstliche Intelligenz: Ein moderner Ansatz*. Pearson Studium, 2nd edition, 2004.

[47] Eric Salen, Katie und Zimmerman. *Rules of play: Game design fundamentals*. MIT Press, 2003.

[48] Mark Schmidt. Die erfolgreichsten Kino-Filme aller Zeiten. Website, Sep 2008. `http://mark-schmidt.suite101.de/kino-die-erfolgreichsten-filme-aller-zeiten-a48386`, letzter Zugriff am 29.09.2012.

[49] Arne Schmieg, Michael Stieler, Sebastian Jeckel, Patric Kabus, Bettina Kemme, and Alejandro Buchmann. psense - maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, P2P '08, pages 247–256, Washington, DC, USA, 2008. IEEE Computer Society.

[50] Alexander Shoulson, Francisco M. Garcia, Matthew Jones, Robert Mead, and Norman I. Badler. Parameterizing behavior trees. In *Proceedings of the 4th international conference on Motion in Games*, MIG'11, pages 144–155, Berlin, Heidelberg, 2011. Springer-Verlag.

[51] Pieter Spronck. *Adaptive Game AI*. Datawyse/Universitaire Pers Masstricht, 2005.

[52] Bjorn Stabell and Ken Ronny Schouten. The story of xpilot. *Crossroads*, 3(2):3–6, November 1996. URL `http://doi.acm.org/10.1145/332132.332134`. `http://www.xpilot.org`, letzter Zugriff am 29.09.2012.

[53] Swee Ann Tan, William Lau, and Allan Loh. Networked game mobility model for first-person-shooter games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, pages 1–9, New York, NY, USA, 2005. ACM.

[54] M. Tanenbaum, A. und van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. Pearson Studium, 2nd edition, 2008.

[55] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 49–60, New York, NY, USA, 2007. ACM.

[56] Jim Waldo. Scaling in games and virtual worlds. *Commun. ACM*, 51(8):38–44, August 2008.

[57] David Winter. Pong-Story: The site of the first video game. Website. `http://www.pong-story.com/intro.htm`, letzter Zugriff am 29.09.2012.

[58] Mark J. P. Wolf. *The Video Game Explosion: A History from Pong to Playstation and Beyond*. Greenwood Press, 2008.

[59] Dimitri Wulffert. Artificial intelligence for a massively multiplayer online game. Bachelor thesis, Technische Universität Darmstadt, Germany, Mar 2011.

[60] A. P. Yu and S. T. Vuong. Mopar: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '05, pages 99—-104, New York, NY, USA, 2005. ACM.