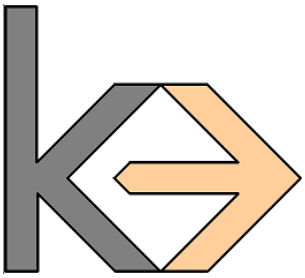
Bachelorarbeit
in der Knowledge Engineering Group

Wintersemester 2007/2008



Knowledge Engineering

Fachbereich Informatik

TU Darmstadt

gAImes-Engine

Ein KI Framework für Computerspiele

Peter Kaufmann, George-Petru Ciordas-Fanghäuser

Betreuer: Prof. Dr. Johannes Fürnkranz

28.03.2008

Ehrenwörtliche Erklärung

Hiermit versichern wir, die vorliegende Bachelorarbeit 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 gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 28.03.2008

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Entstehung der Computerspiele 1
1.2 Technische Errungenschaften 2
1.3 Motivation 3
1.4 Dokumentübersicht 4

2 Marktverhältnisse **5**
2.1 Herstellerspezifische KIs 9
2.2 Herstellerunspezifische KIs 10
2.3 Einordnen der gAImes-Engine 11

3 Fragestellung **12**
3.1 Ziele der gAImes-Engine 12
3.2 Herausforderungen 13
3.3 Schnittstelle: Kommunikation vom Spiel zu gAImes-Engine . . 14
3.4 Einbinden der Spieleinheiten 14
3.5 Schnittstelle: Kommunikation von der gAImes-Engine zum Spiel 15
3.6 Ausführung 16
3.7 Kommunikation der Logikkomponenten 17
3.8 Speichern von Informationen 17

4 Der KI Entwickler **19**
4.1 KI-Skript 19
4.2 Spieleinheit 20
4.3 ExecuteGameUnit 20
4.4 Zugriff auf die gAImes-Engine 21
4.4.1 Zugriff auf die Karte 21
4.4.2 Zugriff auf Einheiten 22
4.4.3 Zugriff auf das Gedächtnis 23
4.4.4 Zugriff auf die Befehle 23
4.5 Schichtenmodell 24
4.5.1 Erweiterte Befehle 25
4.6 Entwicklungsschritte für KI-Skripte 26
4.6.1 KI-Skripte erstellen 26
4.6.2 Datenstrukturen erweitern 26
4.6.3 Erster Schritt: Befehle abrufen 26

4.6.4 Zweiter Schritt: Spielumfeld analysieren 27
4.6.5 Dritter Schritt: Befehle absetzen 27

5 Der Spiel Entwickler 28
5.1 Einbinden der gAImes-Engine 28
5.1.1 Einbinden der jar Datei 28
5.1.2 Erstellen des Ordners AIDevelopment 29
5.1.3 AI Skripte in Ordner kopieren 29
5.2 Kommunikation mit der gAImes-Engine 29
5.2.1 Implementieren der AIListener Schnittstelle 29
5.2.2 Erstellen der Spieleinheiten 30
5.2.3 Erstellen des Spielfelds 33
5.2.4 Erstellen der Spielereignisse 34
5.3 Erste Anwendungsschritte vor dem Spielablauf 35
5.3.1 Instanzen der gAImes-Engine 35
5.3.2 Erster Schritt: Spieler registrieren 35
5.3.3 Zweiter Schritt: Startzustand 37
5.3.4 Dritter Schritt: Spielbeginn 37
5.4 Spielablauf 38
5.4.1 Übertragung der KI-Aktionen 38
5.4.2 Aktualisieren der gAImes-Engine 39

6 Die gAImes-Engine 40
6.1 Die Architektur 40
6.1.1 Kommunikation 40
6.1.2 Verwaltung 41
6.1.3 Logik 43
6.2 Verwaltung der Datenstrukturen 43
6.2.1 Spieleinheiten 43
6.2.2 Karte 44
6.2.3 Spielereignisse 45
6.2.4 Befehle 46
6.2.5 Gedächtniseinträge 46
6.3 Verarbeitung der Datenstrukturen 47
6.3.1 Spielereignisse 47
6.3.2 Ausführungskontrolle 48
6.3.3 Kontrollfluss 49
6.3.4 Laden der KI-Skripte 49

7 2D-RTS Spiel als Testumgebung 51
7.1 Technologie . . . 52
7.2 Architektur . . . 54
7.2.1 Antz3D . . . 54
7.2.2 EnvironmentalCreation . . . 54
7.2.3 Initialzustand . . . 55
7.2.4 Antz3DGraphics . . . 55
7.2.5 Initalisierung . . . 55
7.3 Implementierung der Spieleinheiten . . . 58
7.3.1 DataUnitGraphic . . . 58
7.3.2 DataUnitBasics . . . 59
7.3.3 DataUnitGraphicHarvesterSphere . . . 60
7.3.4 DataUnitL3Harvester . . . 63
7.4 Implementierung der Ereignisschnittstelle . . . 63
7.4.1 Was sendet die gAImes-Engine . . . 65
7.4.2 Wie reagiert man auf die Ereignisse . . . 65
7.4.3 Verarbeiten der Events . . . 67

8 Implementierte KI-Skripte 68
8.1 Zivilist . . . 68
8.2 Ressourcensammler . . . 70
8.2.1 Kommandant der Ressourcensammler . . . 70
8.2.2 Ressourcensammler . . . 72
8.3 Soldaten . . . 75
8.3.1 Ein einfacher Soldat . . . 76
8.3.2 Kommandant der Soldaten: Strategie Direktangriff . . . 77
8.3.3 Ein Soldat mit Gruppenzugehörigkeit . . . 78
8.3.4 Kommandant der Soldaten: Strategie eigene Soldaten gruppieren . . . 81
8.3.5 Ein Beispiel . . . 85

9 Ausbau der gAImes-Engine 88
9.1 Wegfindung . . . 88
9.1.1 A*-Algorithmus . . . 88
9.1.2 Barrieren auf dem Spielfeld . . . 89
9.1.3 Kollisionen mit anderen Spieleinheiten . . . 90
9.1.4 Kopplung der Wegfindungsklassen . . . 91
9.2 Finden von Mustern in Zeichenketten . . . 92
9.3 Zugriff auf DataUnits . . . 92
9.4 Der Sortieralgorithmus Quicksort . . . 93
9.5 Gruppieren mit k-means und k-means++ . . . 93

9.6 Eine Gruppe von Spieleinheiten 95
9.7 Spieleinheitensortierer 96

10 Zusammenfassung **97**
10.1 Vorschläge zur Weiterentwicklung 98
10.1.1 Evaluierungsumgebung für KI-Skripte 98
10.1.2 KI-Skripte als Thread starten 98
10.1.3 Kommunikation zwischen gAImes-Engines 99
10.1.4 Aufzeichnen des Spielverlaufs 99
10.1.5 KI-Skripte-Datenbank 99
10.1.6 Die gAImes-Engine für andere Programmiersprachen . 100

1 Einleitung

1.1 Entstehung der Computerspiele

Eines der ersten Computerspiele wurde 1972 entwickelt. Dabei handelte es sich um zwei senkrechte Rechtecke, die als Schläger dienten und ein Quadrat, welches als Ball zwischen den Schlägern hin und her gespielt werden konnte. Bekannt wurde dieses Konzept als Pong. Später wurde es zu Tennis, Ping-Pong oder Eishockey weiterentwickelt. Das Spielprinzip änderte sich dabei kaum. Eine Person versuchte durch geschickte Positionieren ihres Schlägers den von, alleine fliegenden Ball, auf die Seite des Gegenspielers zu lenken. Trifft ein Spieler den Ball nicht, so bekommt der Gegenspieler einen Punkt. Dieses Programm war allerdings noch lange nicht als kommerzielles Computerspiel gedacht und lief lediglich bei einigen Universitäten auf Großrechnern. Erst in den 1980er Jahren wurden die Computersysteme klein und komplex genug, um den Weg in die Wohnzimmer zu finden. In den kommenden Jahren entwickelten sich die Systeme, sowie die Nachfrage stetig weiter und wurden zu einer Industrie. 1983 läutete die Firma Nintendo in Japan eine neue Ära der Heimcomputerspiele ein. Die Welle schwappte zwei Jahre später nach Nord-Amerika und Europa über.

Die meisten Spiele zu dieser Zeit wurden alleine gespielt, in einem so genannten Einzelspielermodus. Das Spielgeschehen wurde dabei hauptsächlich durch die Eingaben des Spielers beeinflusst. Der Computer übernahm lediglich die Spielmechanik, zum Beispiel den Ballflug bei Pong. Die Reaktionen des Computers waren oft ausschlaggebend für die subjektiv Empfundene Qualität des Spiels. Dem Spieler sollte suggeriert werden, er sei nicht alleine in der Computerspielewelt. Um solche Illusionen möglichst realistisch wirken zu lassen, sind komplexe künstliche Intelligenzen notwendig. Die Computersysteme aus den 1980er hatten jedoch eine stark eingeschränkte Leistungsfähigkeit auf Grund der damals technischen Möglichkeiten. Realistisch wirkende künstliche Intelligenz erfordert die Verarbeitung von großen Datenmengen. Im vordergründigen Interesse der Spielergemeinschaft stand zudem die grafische Qualität.
Im Laufe der Jahre, wurde Hardware immer leistungsfähiger. Die Spieleindustrie entwickelte grafisch aufwändigere Spiele. Der größte Markt waren die Konsolen von Nintendo und Sega. Da die Hardware der Konsolen nicht modifiziert und den neuen Standards angepasst werden konnte, versuchten die Spielehersteller der wachsenden Fangemeinde mit immer aufwändiger entwickelten Titeln gerecht zu werden, die die technischen Möglichkeiten der Konsolen immer mehr ausreizte. Eine Möglichkeit dies zu erreichen war erneut der Versuch die Spiele lebendiger wirken zu lassen. Da dies auf Grund der

Konsolenhardware nicht auf grafischem Wege möglich war, versuchte man die inhaltliche Komplexität der Spielszenen zu erhöhen. Plötzlich wurde die Interaktion mit Computergesteuerten Charakteren (NPC) interessanter. Man bekam als Spieler schnippige Antworten, wie zum Beispiel: „Hast du deine Aufgabe immer noch nicht verstanden!“, wenn man eine Spielfigur mehrmals ansprach, die dem Spieler eine Aufgabe gegeben hatte. In der Spielergemeinde kam das sehr gut an und es wurde zum Standard.
Umso größer die Leistungsfähigkeit von Computersystemen und Konsolen wurde, umso mehr boten Spiele Realismus an. Die Grafik war dabei am meisten gefragt. Jedoch konnte kein Titel erfolgreich sein, ohne die Spielewelt zum Leben zu erwecken. Mit der Entwicklung der dreidimensionalen Computergrafiken kam das Wort „Spielphysik“auf. Es bezeichnete den realistischen Verlauf einer Spielszene. Wurde zum Beispiel in einer 3D Spielwelt eine Holzkiste eine 45° Schräge hinunter geschoben, passte sich die Orientierung der Kiste nicht der Schräge an, sonder sie ‚schwebte‘in der Luft, als würde sie ebenerdig stehen. Dies wirkte sehr unrealistisch. So wurden Algorithmen entwickelt, die das ‚korrekte‘Verhalten von Körpern auf ihre Umwelt simulieren sollten. Zunächst waren diese sehr einfach und konnten zum Beispiel Kisten korrekt auf einer Schräge orientieren. Mit der Zeit wurde ein Markt daraus, so dass es Hardware gab, die zur Berechnung korrekter Physik in 3D Spielszenen war. Diese setzte sich allerdings nicht durch, aufgrund der hohen Zusatzkosten. Die Leistungssteigerung der Hardware schritt schneller voran, was den Gebrauch von Zusatzkarten relativierte.
Etwa zeitgleich mit Einführung von Spielphysik wurden die NPCs in Spielen immer schlauer. Vorreiter war das so genannte 3D Shooter Genre. 3D Shooter waren und sind sehr beliebt, hauptsächlich bei jüngeren Spielern bis 16 Jahren. Meist geht es darum den Gegner mit Handfeuerwaffen zu eliminieren. Diese Spiele werden fast immer aus der Ego-Perspektive gespielt. Dies bedeutet, dass der Spieler seinem virtuellen Ich aus den Augen schaut. Da diese den Eindruck vermitteln mitten im Geschehen zu sein, sind diese Spiele oft Gegenstand der Presse, auf Grund von Gewaltverherrlichung, gewesen. Abgesehen von den negativen Schlagzeilen wurde in diesem Genre besonderen Wert auf intelligent agierende computergesteuerte Gegenspieler gelegt. Aktuell sind Gegenspieler mit taktischem Verständnis und der Möglichkeit Situationen einzuschätzen möglich.

1.2 Technische Errungenschaften

Mit der Entwicklung von hoch qualitativer Computergrafik wurden viele Werkzeuge und Bibliotheken entwickelt. Zwei der bekanntesten freien Bibliotheken für unter Anderem grafische Darstellung sind DirectX und OpenGL.

Spieleentwicklern wird eine umfangreiche API(application programming interface) bereitgestellt, welche alle wichtigen Grundschritte für die Grafikprogrammierung bereit hält. DirectX und OpenGL enthalten zum Beispiel alles mathematische Wissen in linearer Algebra, welches zur Darstellung und Berechnung von 3D Objekten im 3D Raum notwendig ist. Es gibt Firmen, die sich auf die Entwicklung so genannter Spieleengines spezialisiert haben und diese unter einer Lizenz vertreiben. Die Firma ID Software entwickelt beispielsweise derartige Engines. Sie tragen den Titel eines bekannten Spiels in diesem Genre „Quake“. Solche Engines vereinen die Prinzipien von den oben genannten Techniken, wie DirectX oder OpenGL und erweitern sie um viele technische, meist hoch komplexe Details. Zum Beispiel effizientere Algorithmen oder neue Algorithmen für noch realistischere Darstellungen und Spezialeffekte.
Analog zur grafischen Darstellung gibt es Frameworks und Bibliotheken für Physik. Eine bekannte Physik-Engine hat den Namen PhysiX. Zu diesem gibt es die bereits erwähnte Zusatzhardware, die zur Berechnung hoch komplexer Szenen dient.
Während Grafik und Physik in Computerspielen stätig vorangetrieben wurden, stagnierte die Entwicklung künstlicher Intelligenz in Computerspielen geradezu(siehe Kapitel 2).

1.3 Motivation

Computerspiele entwickeln sich stetig weiter und werden immer komplexer in nahezu jeder Hinsicht. Sie sind optisch immer anspruchsvoller und sollen den Konsumenten in Staunen versetzen. Die Spielwelt wird größer und aufwändiger gestaltet und die Interaktionsmöglichkeiten werden realistischer. Bei all der Liebe zum Detail wird künstliche Intelligenz nur als Nebenprodukt behandelt(siehe Kapitel 2). Es existiert in diesem Bereich der Computerspieleindustrie ein großer Nachholbedarf. Der Inhalt dieser Bachelorarbeit soll diesen Bedarf decken. Es soll die Grundlage zur Entwicklung einer Komponente gelegt werden, mit der die Entwicklung und Implementierung künstlicher Intelligenz in Computerspielen leichter und schneller möglich ist. Dabei geht dieses Dokument auf die Grundidee, die Planung und die Umsetzung dieser Technik, ein. KIs sollen unabhängig von Computerspielen entwickelt werden können, um so den Entwicklungsaufwand aus der Spielentwicklung zu entfernen. Der Grundgedanke ist der gleiche, wie bei der Grafikentwicklung. Würde ein Spieleentwickler zu jedem Produkt die komplexe Mathematik hinter der 3D-Grafik jedes mal neu implementieren müssen, würden heutige Spiele noch wie vor zehn Jahren aussehen. Die Programmierung wird in der Sprache Java vorgenommen. Das Endprodukt soll eine herstellerunspezifische

KI-Engine(siehe Kapitel 2.2) sein auf OpenSource Basis. Der erhoffte Erfolg soll eine Community sein, die die KI Engine entwickelt und KI-Skripte erstellt. Dadurch, dass die Software, sowie die Skripte OpenSource sind können viele Entwickler die KI-Skripte verbessern. Möglicherweise ergibt sich daraus ein Standard, so dass Hardwarehersteller gezielt Produkte dafür entwickeln können und der Einsatz in Computerspielen effizienter werden kann.(siehe Kapitel2)

1.4 Dokumentübersicht

Diese Arbeit beschreibt den Entwicklungsprozess der gAImes-Engine von der Idee bis zum Einsatz in einem Computerspiel. Beginnend mit der Einstufung in aktuelle Marktverhältnisse, folgt darauf eine Definition aller notwendigen Funktionen, die die gAImes-Engine unterstützen muss. Dabei wird explizit auf Design- und Programmierprobleme eingegangen, die bei der Umsetzung aufgetreten sind. Nachdem alle Grundlagen feststehen, werden die unterschiedlichen Vorgehensweisen für Spiel- und KI-Entwickler beleuchtet. Dabei gehen die jeweiligen Abschnitte sowohl auf konzeptionelle Probleme ein, die bei der Planung eines Spiels entstehen, als auch auf Probleme bei der Erstellung des Programmcodes. Alle Schnittstellen für Spiel- und KI-Entwickler der gAImes-Engine sind nun beschrieben. Es folgt die Umsetzung der gAImes-Engine. Die Funktions- und Arbeitsweise der Schnittstellen werden erläutert und die Transportwege aller Nachrichten innerhalb der Engine beschrieben. Verdeutlichen sollen dies Skizzen und Bilder an entsprechender Stelle. Zum Ende der Arbeit wird die Implementierung einiger Funktionen der gAImes-Engine in ein Computerspiel gezeigt, das extra für diese Demonstration angefertigt wurde. Der dargestellte Code enthält nützliche Tipps und soll exemplarisch den Umgang mit der gAImes-Engine verdeutlichen. Abschließend werden Hinweise zur Weiter- und Neuentwicklung einiger Teile der gAImes-Engine gegeben. Die Arbeit endet mit einer Zusammenfassung aller Vorkommnisse.

Die Verantwortung für diese Ausarbeitung und die Implementierung liegt bei beiden Autoren zu gleichen Teilen. Im Detail können dem Autor Kaufmann jedoch die Kapitel 1, 2, 5, 6.3, 7 und 10 sowie dem Autor Ciordas-Fanghäuser die Kapitel 3, 4, 6.1, 6.2, 8 und 9 größtenteils zugeordnet werden.

2 Marktverhältnisse

Künstliche Intelligenz wird in Computerspielen schon seit jeher benötigt. Es existiert praktisch kein Spiel ohne eine, wenn auch sehr geringe, Intelligenz, die einen Gegenspieler simuliert. Mit Voranschreiten der Technik wurden Computerspiele immer komplexer, die KI stagnierte jedoch.

1996: Ein für heutige Verhältnisse extrem pixeliger Ernter sucht im Strategie-Primus Command & Conquer nach dem Rohstoff Tiberium, rollt dabei jedoch einfältig mitten in das Hauptquartier des Feindes und zerplatzt in einer hässlichen Mini-Explosion.

2007: Ein Ernter auf höchstem grafischen Niveau sucht im Strategie-Hit Command & Conquer 3 nach dem Rohstoff Tiberium, rollt dabei jedoch einfältig mitten in das Hauptquartier des Feindes und wird von gleißenden Laserstrahlen eindrucksvoll in seine Einzelteile zerrissen, die physikalisch korrekt in alle Himmelsrichtungen fliegen. Elf Jahre liegen zwischen den oben beschriebenen Szenen. Elf Jahre, in denen sich die Optik enorm weiter entwickelt hat: 3D-Grafik, Spezialeffekte, Physik-Simulation. Elf Jahre, in denen sich aber auch ein wichtiges Detail nicht verändert hat: Der Ernter verhält sich 2007 noch genauso unklug wie 1996.

Das Spiel Command & Conquer steht dabei nur exemplarisch für ein grundlegendes Phänomen: Die Grafik von Computerspielen wird fortdauernd verbessert, doch die Simulation von künstlicher Intelligenz steht auf niedrigem Niveau still. Die Ursachen für diese immer breiter werdende Kluft zwischen realistischen Welten und realistischem Verhalten ist vielschichtig.

Computerspiele haben häufig sehr komplexe Regeln. Würden diese überschaubar bleiben, könnten KIs erstaunliche Leistungen vollbringen. Eindrucksvoll verdeutlichen das Schach-Simulationen wie Fritz oder Shredder. Sie schlagen seit den 1990er-Jahren regelmäßig menschliche Weltmeister. Die Komplexität in aktuellen Computerspielen hebt sich jedoch sehr stark von Spielen wie Schach ab. Schach hat ein Spielfeld mit 64 Feldern und sechs unterschiedlichen Spielfiguren. Zum Vergleich, das Spiel Anno 1701, eine Taktiksimulation mit wirtschaftlichen Aspekten, wie Produktionsketten, Siedlungsplanung und Schlachten, hat ca. 1000 mal 1000 Spielfelder, sowie Hunderte Gebäude, Warenketten und Einheiten. Der notwendige Arbeitsaufwand zur Berechnung von komplexen Spielzügen, wie in Schach wäre von keinem Computer und schon gar nicht von Heimcomputern in vertretbarem Aufwand zu berechnen.

Es ergeben sich nahezu unendlich viele Spielkombinationen. Computerspiele werden weiterhin immer komplexer und die Anforderung an die künstliche Intelligenz steigt stetig an.

Damit eine KI heutzutage dennoch intelligent wirkt und das Spiel für den Konsumenten attraktiver macht, bauen die Programmierer Tricks ein. Nahezu jede KI schummelt! Im Beispiel Anno 1701 baut die KI Häuser und Wege nicht Stück für Stück wie der Spieler, sondern es existieren vordefinierte Muster, die eine effiziente Produktionskette darstellen. Bei kriegerischen Aktivitäten, wie Einheitenbewegungen und Kämpfen, werden Einheiten nicht einzeln gesteuert, sondern in Gruppen. So entfällt der Rechenaufwand zur Zusammenstellung einer Gruppe. Es wird so getan, als seien z.B. zwölf Einheiten nur eine. Auch in anderen Genres, wie Beispielsweise **Egoshootern**, wo aus der Ich-Perspektive der Spieler eine Einheit steuert, haben kleine Tricks für die KI gesteuerten Einheiten. Es existieren so genannte Waypoints(Wegepunkte), die für den Spieler unsichtbar sind, jedoch die Wege für die KI markieren. So laufen Einheiten immer zwischen diesen Markierungen hin und her. An einigen Stellen, wo der Weg mehrere Möglichkeiten bietet, wird dann eine Richtung berechnet. Diese Markierungen haben zudem unterschiedliche Prioritäten. In Spielen wie Unreal Tournament kann der Spieler und auch die KI Fahrzeuge besteigen. Nicht alle Wege sind für Fahrzeuge geeignet. Die KI müsste berechnen, welcher Weg passierbar ist. Das wird umgangen, da die KI die Steuerung eines Fahrzeuges erkennt und nur vorher ausgewählte Wege benutzt.

Eine weitere Hürde für den Einsatz einer KI sind Stories im Spiel. Das Herzstück der meisten Computerspiele ist eine Kampagne. Diese erzählt eine Geschichte, in der sich der Spieler während des Spiels bewegt. Die Kampagne soll bei jedem Spieler immer gleich ablaufen. Aus diesem Grund wird die KI mit Absicht von den Entwicklern eingeschränkt. Hätte man eine völlig autonom und extrem intelligent agierende KI, würde das Spiel, abhängig von der Handlung des Spielers, immer unterschiedlich ablaufen. Es würde dadurch nahezu unmöglich eine Storyline(Spielgeschichte) vorauszuplanen. Ein ähnliches Problem ergibt sich bei einer sehr intelligenten KI, wenn man das Spiel für den Spieler spielbar halten möchte. Gerade zu Beginn eines Spiels, ist der Spieler noch nicht mit allen Interaktionmöglichkeiten vertraut. Die KI verfügte jedoch schon über alle Informationen des Spiels und der Spieler hätte große Probleme den Gegner zu schlagen. Der Spieler ist schnell frustriert und das Spiel wird subjektiv negativ bewertet.

Aus diesen Gründen verwenden Entwickler häufig **Skripts**. Skripts sind fest-

gelegte Aktionen, die beim Eintreffen eines bestimmten Ereignisses ausgelöst werden. In Anno 1602 dem Vorvorgänger von 1701 wurde der Schwierigkeitsgrad über Skripts angepasst. Der Computergegner wartet mit dem Ausbau seiner Siedlung immer auf den Spieler. Hat dieser eine bestimmte Technologie erforscht oder ein Gebäude gebaut, dass ihm einen Vorteil, z.B. bei der Truppenproduktion ermöglicht, hat der Computergegner dieses mit einer kleinen Verzögerung auch gebaut. So blieb das Spiel zu jedem Zeitpunkt fordernd und zugleich konnte der Computerspieler nicht übermächtig werden. Der Preis von Skripten aus der Sicht von KIs ist hoch. Es geht viel Dynamik im Spiel verloren und auf lange Sicht, wenn der Spieler hinter diesen Trick durchschaut hat, wirkt die KI berechenbar und dumm.

Als Fazit geht hervor, dass die Entwicklung einer lebensnahen KI ein enormer Aufwand ist. „Bei den meisten Entwicklern werden nur 12 bis 24 Mann-Monate für die KI-Programmierung eingeplant. Es müsste aber mindestens das Fünffache sein.“(Zitat: Dr. Andreas Gerber, 33-jähriger promovierter Diplominformatiker vom Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) und Gründer der Firma Xaitment)(ein Mann-Monat = ein Angestellter arbeitet einen Monat)

Die Firma Xaitment entwickelt seit zwei Jahren kommerzielle Software für künstliche Intelligenz in Computerspielen. Es arbeiten knapp 30 Mitarbeiter mit einem Budget von 1,5 Millionen Euro an der Entwicklung einer KI-Engine.

Spieleentwickler müssten viel Zeit und Geld in die Entwicklung ihrer KI stecken, um diese auf dem Stand der Spieltechnik zu halten. Dennoch wird dieser Bereich vernachlässigt, da eine aufwändige KI keinen Verkaufsschlager darstellt, eine spektakuläre Grafik, innovative Spielelemente, interessante Charaktere und Lizenzen jedoch schon. Die kommerzielle Zugkraft von Lizenzen wird bei den aktuellen Spieleumsetzungen von George Lucas deutlich. 2008 wurden mehrere Spiele der Reihe Starwars und Indiana Johnes veröffentlicht. Die Grafik der Produkte war, vom aktuellen Stand der Technik gesehen, mittelmäßig, eine KI war praktisch nicht vorhanden und das Spielprinzip war simpel und linear. Dennoch verkauften sie sich gut.

Aus ihrem Schattendasein kann die KI nur dann hervortreten, wenn man sie zum zentralen Spielelement befördert. Titel wie Assassin's Creed und F.E.A.R. haben dies mit Erfolg vorgemacht. Bei Assassin's Creed sind belebte Städte mit hunderten Passanten von Anfang an ein zentrales Konzept für den Publisher, Marketing und damit auch für den Entwickler. 15 Program-

mierer und Designer haben laut Ubisoft an der KI des Schleichspiels gearbeitet. Bei F.E.A.R. kam der Spieler immer wieder in verwinkelte Räume, die mit Gängen verbunden waren und mehrere Ein- und Ausgänge hatten, z.B. Türen und Fenster. Die dort positionierten Computergegner, eine Eliteeinheit, kommunizierten per Funk miteinander und gaben bei Sichtkontakt die Position des Spielers weiter. Anschließend teilten sie sich auf und versuchten den Spieler zu umzingeln, um ihn anschließend mit koordinierten Attacken auszuschalten. Zusätzlich konnten sie Sperrfeuer geben, Granaten werfen und Deckung hinter den dynamisch verschiebbaren Objekten, wie Kisten, suchen.

Der hohe Preis für diese vergleichsweise intelligente KI ist ein großer Rechenaufwand. Computerspiele sollen jedoch von möglichst vielen Spielern gespielt werden können. Somit muss man einen Kompromiss finden. Nicht jeder Spieler hat einen High-end-PC zu Hause. Zudem ist den Herstellern die Grafik wichtiger, die ebenfalls enorm viel Rechenleistung erfordert.

Geld spielt in der Spieleentwicklung natürlich eine große Rolle. In der Spielbranche ist es schwer ein Budget genau zu kalkulieren. Verschiebt sich die Veröffentlichung eines Spiels nur um wenige Monate, kann es das Aus für ein kleines Entwicklerteam bedeuten. Folglich lassen diese Teams die Finger von größeren KI-Innovationen. Technische Verbesserungen lassen sich bedeutend besser kalkulieren, sowohl bei den Kosten als auch beim Entwicklungsrisiko. Ob eine Spielfigur nun aus hundert Pixeln oder aus fünftausend Polygonen besteht, hat auf die Arbeit der Missions- und Spieldesigner wenig bis gar keinen Einfluss. Die Konsequenz, vor allem bei Spieleserien mit einer hohen Erscheinungsfrequenz ist „Grafik und Physik werden weiterentwickelt, die KI aber nur aus dem Vorgänger übernommen“, so Dr. Andreas Gerber. Deutlich zu sehen ist das bei Produkten wie Fifa und Need for Speed.

Der Trend hin zu phantastischen Grafiken ist auch in der Hardwareunterstützng zu erkennen. So entwickeln Firmen wie Nvidia und ATI etwa eine neue Grafikkartengeneration pro Jahr mit doppeltem Arbeitsspeicher. Diese können komplexe Berechnungen der Spielgrafik auf der Hardware ausführen. Zudem bieten die Treiber der Geräte mittlerweile für viele Effekte, wie z.B. realistisch wirkendes Wasser, Standardfunktionen. Der Implementierungsaufwand für derartige Effekte geht gegen Null.

Für KI-Entwickler macht die komplette Arbeit heute wie zu C64(erschienen 1982) Zeiten die CPU(Central Processing Unit der Hauptprozessor des Computers). Die CPU ist im System für alle Berechnungen zuständig, so muss die KI die Rechenleistung mit allen anderen laufenden Prozessen im Spiel und

dem Betriebssystem teilen. „Die meisten Entwickler sagen: KI darf nur rund fünf Prozent der Rechenleistung belegen. Mindestens das Doppelte wäre aber nötig.“(Zitat: Dr. Andreas Gerber)

Für Grafiker gibt es leistungsfähige Entwicklungssoftware wie 3D Studio Max zur Erstellung komplexer Spielfiguren und für Leveldesigner, die die Spielwelt modellieren, werden sogenannte Leveleditoren erstellt. Lizenznehmer einer Grafikengine, wie der Unreal 3 Engine bekommen diesen sogar mitgeliefert. Für KI-Entwickler gibt es derartige Software nicht. Die Intelligenz wird selbst heute noch per Hand in einen Texteditor eingetippt.

Einen derartigen kommerziellen Editor entwickelt die Firma Xaitment. Man kann bislang Firmen, die solche Software entwickeln an einer Hand abzählen. Ein Grund ist mangelndes Know-How auf dem Markt, ein Weiterer, die bereits erwähnte fehlende Hardwareunterstützung. An viele Universitäten existieren Fachbereiche, die sich mit künstlicher Intelligenz auseinandersetzen. Jedoch “Ein Großteil der Professoren ist sehr konservativ und möchte mit Computerspielen prinzipiell nichts zu tun haben.„(Zitat: Dr. Andreas Gerber) So kommt das Wissen über KI nicht in der Spielbranche an.

(Quelle: Gamestar)

2.1 Herstellerspezifische KIs

Eine herstellerspezifische KI sei definiert, wenn sie von einer Firma entwickelt und nur für interne und eigene Projekte eingesetzt wird. Dies ist in den meisten Firmen, die Spiele entwickeln, der Fall. Bei der Entwicklung einer herstellerspezifischen KI für ein Computerspiel sind die Programmierer durch keine äußeren Vorgaben eingeschränkt. Der KI Entwickler ist zu jeder Zeit im Entstehungsprozess in der Lage die im Spiel eingesetzten Datenstrukturen zu verwenden. Es gibt keine Kompatibilitätsprobleme zwischen dem Spiel und der KI. Die KI wurde spezifisch für ein Spiel entwickelt und abgestimmt. Viele aktuelle Spiele werden mit einer herstellerspezifischen KI entwickelt. Dafür gibt es mehrere Gründe, die bereits im Kapitel Marktverhältnisse erläutert wurden.(siehe Abschnitt 2)
Ein Beispiel für eine herstellerspezifische KI ist das Spiel Warcraft 3 der Firma Blizzard. Es handelt sich um ein Echtzeitstrategie-Spiel, bei dem zwei Völker gegeneinander antreten. Die Darstellung zeigt das Geschehen von schräg oben, so als würde man vor einem Schachbrett sitzen und darauf schauen. Dem Spiel beigelegt ist ein Editor zur Erstellung eigener Szenarien. Über diesen lassen sich auch die gegnerischen KIs anpassen. Es existiert

eine Skriptsprache, die die Grundzüge der Programmierung mit if-else usw. beherrscht. Bei der Erstellung des Spiels wurde die KI explizit für Warcraft 3 geschrieben. Sie lässt sich zwar von außen manipulieren, aber nicht auf andere Spiele anwenden.
Ein weiteres Beispiel stellt die Fussballsimulation Fifa von Electronic Arts dar. Man steuert die Spieler einer Mannschaft gegen ein anderes Team. Das Spiel Fifa erscheint i.d.R. alle zwei Jahre neu und bekommt den Beititel des aktuellen Jahres, zum Beispiel Fifa 98, Fifa 2000 usw. Für einige der Titel des folgendes Jahrgangs wird die gleiche KI Engine verwendet. Das heißt, die Entwickler haben eine KI-Engine programmiert, die in mehreren Fifa Titeln verwendet werden kann. Diese KI ist immer noch herstellerspezifisch, da alle Spieltitel, von der gleichen Firma kommen. Bei der KI Entwicklung kann so gearbeitet werden, als sei es nur ein Spiel. Der Folgetitel des nächsten Jahres beinhaltet die gleichen Datenstrukturen und wahrt so die Kompatibilität.

2.2 Herstellerunspezifische KIs

Eine herstellerunspezifische KI sei definiert, wenn die KI-Engine von einer Firma entwickelt und verkauft wird. Andere Firmen verwenden deren KI-Engine für ihre Produkte. Diese Art der KI Entwicklung ist ungleich komplizierter, als die der herstellerspezifischen KIs. Die Entwickler der KI-Engine können die Datenstrukturen der Spiele nicht kennen, in der sie eingesetzt wird. Somit muss ein Konzept erdacht werden, um dieses Problem in den Griff zu bekommen. Zudem soll der Implementierungsaufwand für den Spielentwickler möglichst gering sein. Der Lizenznehmer soll die KI-Engine in sein Produkt schnell und unkompliziert einbinden können. Derzeit existieren sehr wenige herstellerunspezifische KIs.(siehe 2)
Die deutschsprachige Gesellschaft xaitment mbH entwickelte eine Engine für künstliche Intelligenz Namens xaitEngine. Von hier ausgehend lassen sich zu dem Hauptpaket, der xaitEngine, noch weitere Pakete hinzufügen. Ein interessanter Zusatz ist das Paket xaitThink. Es ergänzt die Charaktere einer beliebigen 3D Applikation um intelligentes Verhalten. Das bedeutet, dass sie eigenständige Bewegungen und Aktionen ausführen können. Gelenkt werden diese Aktionen über einen beigelegten grafischen Regeleditor. Die damit erstellten Regelketten sind übersichtlich und leicht zu bedienen. Der Programmierer bildet quasi Sätze, bei denen er die Verhältniswörter, wie zum Beispiel **falls** oder **istIn** etc. aus einer Liste auswählen kann. Vervollständigt wird der Satz, bzw. die Regel mit den erstellten oder vorhandenen Objekten aus der xaitEngine. Eine Regel würde dann zum Beispiel wie folgt aussehen: Falls Objekt1 enter Room1 then Shoot. Eine ähnliche Vorgehensweise verwendet der Emailclient Outlook von Microsoft.

2.3 Einordnen der gAImes-Engine

Die gAImes-Engine ist eine Middleware, die zwischen dem Spiel und den KI-Skripten steht. Sie ermöglicht dem KI-Entwickler unabhängig vom Spiel Skripte zu schreiben und auf einem Spiel auszuführen. Der Vorteil der gAImes-Engine gegenüber den grafischen Editoren kommerzieller Software ist der Einsatz von Java Quellcode. Der KI-Entwickler verfügt über den vollen Umfang der Java Sprache zur Erstellung seiner KI und ist nicht auf eine vordefinierte Skriptsprache angewiesen. Theoretisch ist auch eine Umsetzung in andere Programmiersprachen möglich, siehe dazu Kapitel 10.1.6.

3 Fragestellung

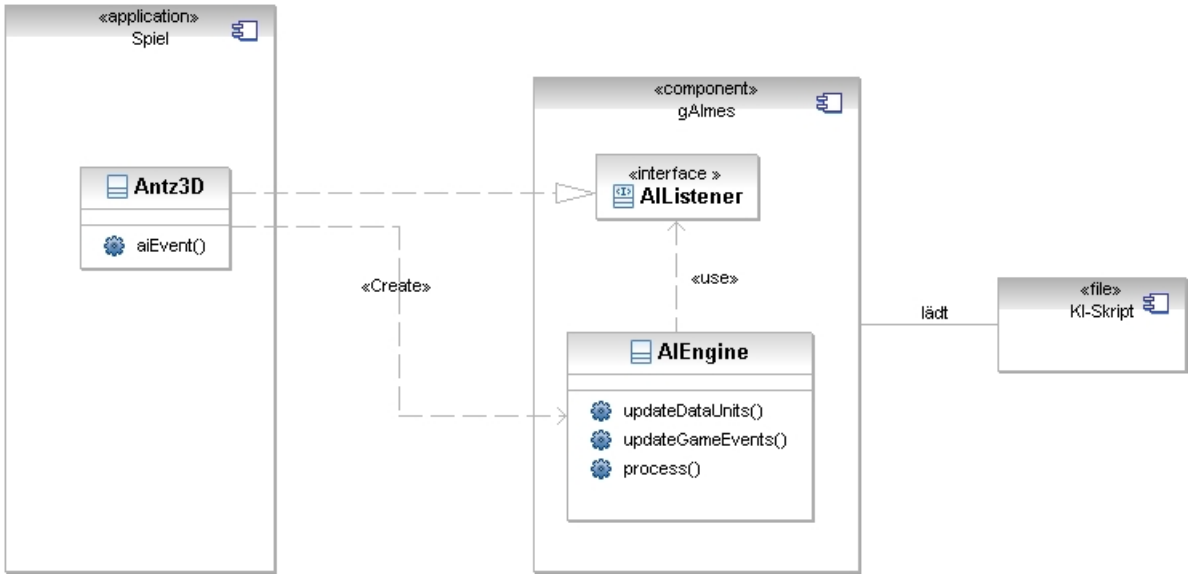


Abbildung 1: Kleiner Überblick über die Architektur

In diesem Kapitel werden die Herausforderungen dieses Projekts im Einzelnen erläutert und die zur Diskussion gestandenen Lösungsansätze beschrieben.

3.1 Ziele der gAImes-Engine

Die gAImes-Engine soll die einfache Implementierung von künstlicher Intelligenz in Computerspielen ermöglichen. Der Spielentwickler soll sich auf die Entwicklung des Spieldesigns und der Grafik konzentrieren können, ohne sich tiefgehend mit der Thematik der künstlichen Intelligenz auseinander setzen zu müssen. Dabei muss sich der Spielentwickler nur mit dem Einbinden der gAImes-Engine in sein Spiel und der Kommunikation derselben befassen. Die entsprechenden KI-Skripte für seine Spieleinheiten kann der Spielentwickler nachträglich hinzufügen.
Dabei kann man zwischen zwei grundlegenden Szenarien unterscheiden.

- Der Spielentwickler benutzt KI-Skripte, die nicht speziell für sein Spiel konzipiert wurden. Dies wäre der Fall, wenn das Spiel als einfache Visualisierung zum Test von autonom arbeitenden KI-Skripten verwendet würde. Dann müssten die KI-Skripte durch geschicktes Ausprobieren ihre virtuelle Umgebung erkennen und analysieren.

- Der Spielentwickler veröffentlicht eine Art API für seine Spieleinheiten. In diesem Fall weiß der KI-Entwickler vorher, welche Fähigkeiten seine Spielfiguren haben werden und kann das KI-Skript entsprechend definieren.

Einem KI-Entwickler soll es ermöglicht werden KI-Skripte zu schreiben, die unabhängig von einem konkreten Spieldesign sind. Hierbei implementiert er seine KI-Skripte nur unter Berücksichtigung der Schnittstelle der gAImes-Engine. Der KI-Entwickler braucht sich somit nur noch mit der Implementierung seiner Logik auseinander zu setzen. Die gAImes-Engine übernimmt für ihn die Kommunikation mit dem Spiel. Zudem soll die gAImes-Engine die modulare Entwicklung von Logik ermöglicht. Die Aufgaben der Logik können in einzelne Module aufgeteilt werden, die spezielle Aspekte, wie beispielsweise die Offensivtaktik, verarbeiten. Die gAImes-Engine soll dem KI-Entwickler hierfür ein intermodulares Kommunikationssystem bieten.

3.2 Herausforderungen

Die Herausforderung bei der Entwicklung der gAImes-Engine liegt im Umgang mit den beiden abstrakten Teile Spielentwicklung und KI-Entwicklung. Die gAImes-Engine soll für jegliche Art von 2D-Echtzeit-Strategiespiel funktionieren, (engl. 2D-Real-Time-Strategy) und im Folgenden 2D-RTS genannt. Dabei sollen KI- und Spielentwickler so wenig wie möglich eingeschränkt werden. Es soll die Entwicklung von speziellen, für ein Spiel angepassten KIs und die Entwicklung von im Kontext unabhängig handelnden KIs ermöglicht werden. Kontextunabhängig bedeutet in diesem Kontext dass eine KI ohne Kenntnisse über die Umgebung der Spiele in denen sie ausgeführt wird entwickelt wird. Hierbei lernt diese erst im fortlaufenden Spiel die Regeln ihrer Umgebung.
Um dieser Anforderung gerecht zu werden, sind viele herausfordernde Aspekte zu bewältigen. Zu diesen gehört die Entwicklung eines Protokolls zur Übermittlung der Ereignisse und Aktionen zwischen Spiel und KI-Skript. Sie muss uniform und doch erweiterbar sein. Das gilt ebenfalls für die Repräsentation der Daten, wie beispielsweise der Spielkarte, den Einheiten oder den Gedächtniseinträgen der Logik. Die Spieleinheiten sind dabei besonders wichtig, da besonders sie der Individualität des Spielentwicklers ausgesetzt sind.
Da die gAImes-Engine die Entwicklung von modularer Logik erlauben soll, benötigt sie ein Kommunikationssystem und eine Regelung zur Ausführungsreihenfolge der KI-Skripte. Modulare Logik heißt, dass für jede Einheit eine eigene KI-Skript-Datei erstellt werden kann. Zusätzlich lassen sich noch Kommandeure vergeben. So hat man eine hierarchische Befehlskette, die von oben

nach unten abgearbeitet wird.
Um eine performante Ausführung und eine wenig Komplexe Entwicklung der KI-Skripte zu gewährleisten, muss ein Konzept entwickelt werden das es dem KI-Entwickler erlaubt zustandslose KI-Skripte zu entwickeln. Zustandslos heißt, wie oben erwähnt, dass jeder Einheitentyp ein eigenes KI-Skript haben kann. Das Skript ist eine Art leere Hülle, in die eine konkrete Einheit geladen wird, wenn sie mit ihrer Ausführungszeit an der Reihe ist. Der Zustand der Spieleinheit wird dabei in der gAImes-Engine gespeichert und verwaltet. Diese Technik erlaubt der gAImes-Engine beliebige KI-Skripte für eine Einheit zu laden und diese zur Laufzeit auszuwechseln.

3.3 Schnittstelle: Kommunikation vom Spiel zu gAImes-Engine

Eines der Ziele der gAImes-Engine ist es die Spielentwicklung klar von der KI-Entwicklung zu trennen. Hierfür wird eine Schnittstelle benötigt, die alle Information, die das Spiel der gAImes-Engine mitteilen möchte, aufnimmt. Diese Informationen sind Ereignisse, welche auf dem Spielfeld geschehen sind. Es wird mitgeteilt welche der Parameter einer Einheit sich um welchen Wert geändert haben, beispielsweise die Lebenspunkte oder die Position. Zudem wird der gAImes-Engine mitgeteilt, welche neuen Kartenfelder sich in seinem Sichtfeld befinden und welche Spielobjekte oder gegnerischen Einheiten sich auf diesen befinden. Zuletzt muss die gAImes-Engine darüber informiert werden welche ihrer Einheiten zerstört wurden und welche neu gebaut wurden.

Um die Ereignisse, die das Spiel mitteilt, zu empfangen, gibt es zwei Möglichkeiten. Die erste wäre eine einzelne Übermittlung jedes auftretenden Ereignisses und dessen sofortige Einarbeitung in die Datenstrukturen der gAImes-Engine. Die zweite Möglichkeit ist eine Sammlung aller Ereignisse einer Spielrunde und der gemeinsame Versand in einem eigens entwickelten Container.

3.4 Einbinden der Spieleinheiten

Bei der Entwicklung eines 2D-Echtzeit-Strategiespiels befinden sich viele Spielfiguren auf dem virtuellen Spielfeld. Jede Spielfigur wird vom Spiel verwaltet und gezeichnet. Die Spieleinheiten benötigen eine Repräsentation, die sie definiert und identifiziert. Dabei haben alle Spielfiguren einige Eigenschaften gemeinsam, andere wiederum sind spielspezifisch. Zum Beispiel besitzen sie alle Koordinaten, die sie auf dem Spielfeld an eine Position binden. Bei der Entwicklung für ein Spiel bietet es sich von daher an eine abstrakte Klasse zu definieren von der jede Spielfigur erben muss. Die Klasse enthält die

gemeinsamen Eigenschaften, wie beispielsweise die Position. Die davon erbenden Klassen werden um die der Spieleinheiten speziellen Eigenschaften erweitert. Diese Eigenschaftenklasse kann somit die Spieleinheit definieren und identifizieren.
Soll das Spiel zum Beispiel einen Ressourcensammler haben, könnte das in etwa so aussehen:

```
class Eigenschaftenklasse
{
  Position;
  ...
}

class Ressourcensammler extends Eigenschaftenklasse
{
Ressourcenkapazität;
...
}
```

Jede der Spieleinheiten benötigt eine gewisse Art von Intelligenz. Wenn ein KI-Entwickler dieses Verhalten entwickeln möchte, so bekommt er Zugriff auf die Spieleinheit über die oben definierte Eigenschaftenklasse. Der KI-Entwickler muss den speziellen Typ der Eigenschaftenklasse zur Programmierzeit seines KI-Skriptes noch nicht kennen. Dies bedeutet jedoch, dass ein Mechanismus für den Zugriff auf diesen zur Verfügung gestellt werden muss. Zur Lösung des Problems, benutzt die gAImes-Engine die Technik Reflection, die ein Teil des grundlegendes Konzepts von Java ist.

3.5 Schnittstelle: Kommunikation von der gAImes-Engine zum Spiel

Bewegt sich eine Spieleinheit auf dem Spielfeld, dann wurde dies von einem KI-Skript berechnet und befohlen. Dazu muss der Befehl vom KI-Skript über die gAImes-Engine an das Spiel übertragen werden. Ein Befehl kann jede Art von Aktion sein, die eine Spieleinheit ausführen kann, wie zum Beispiel das Attackieren von anderen Einheiten oder Sammeln von Ressourcen.
Die Logik der Spieleinheiten, die KI-Skripte, die in der gAImes-Engine ausgeführt werden, müssen am Ende ihrer Berechnungen ihre Aktionen an das Spiel übermitteln können. Diese Aktionen könnten beispielsweise Bewegungen der Spieleinheiten oder Aktionen wie das Feuern auf eine andere Einheit sein. Alle Aktionen müssen in einer einheitlichen Struktur verpackt werden,

so dass die Schnittstelle von der gAImes-Engine zum Spiel nicht, bei neu hinzukommenden Aktionen, geändert werden muss. Befehle könnten den Regeln des Spiels entsprechend unzulässig sein und müssen daher von diesem geprüft und dem KI-Skript bestätigt werden. Dies ist nötig damit die Spieleinheit konsistente Berechnungen anstellen kann. Eine Prüfung der Befehle seitens des Spiels ist unerwünscht, da es KI-Entwicklern möglich sein soll KI-Skripte zu entwickeln, welche eigene Erfahrungen sammeln können. Diesen KI-Skripten soll es ermöglicht werden, sich in einer Spielumgebung zu bewegen, von der sie keine Kenntnis der Regeln haben(try and error).
Um die Aktionen an das Spiel abzugeben, bestehen die Möglichkeiten diese im Einzelnen oder als Packet zu übermitteln. Eine einzelne Übermittlung erhöht den Kommunikationsaufwand, vereinfacht jedoch die Datenstruktur der Nachricht und der Bestätigung. Diese gilt umgekehrt für das Senden in Paketen. Der Datenübermittlungsaufwand sinkt, jedoch steigt die Komplexität der Datenstruktur.

3.6 Ausführung

Die Logik der Spieleinheiten, die KI-Skripte, müssen in jeder Spielrunde von der gAImes-Engine ausgeführt werden, um ihre Entscheidungen treffen zu können. Hierfür müssen den Spieleinheiten die passenden KI-Skripte zugeordnet werden können. Um den Speicherverbrauch und damit auch ein Stück weit die Performanz kontrollieren zu können, müssen die KI-Skripte zustandslos implementiert werden. Methoden zur Erfassung von Informationen über ihre Umgebung und Methoden zum Merken von Entscheidungen(Gedächtnis), stellt die gAImes-Engine zur Verfügung. Hierbei müssen diese Schnittstellen so universell ausgelegt sein, dass jedes Format von gewünschtem Gedächtniseintrag gespeichert werden kann.

Bedingt dadurch dass die gAImes-Engine eine modulare Logikentwicklung erlaubt, muss auf die Ausführungsreihenfolge der KI-Skripte geachtet werden. Die KI-Skripte müssen in einer der Hierarchie entsprechenden Reihenfolge ausgeführt werden, da sonst keine Entscheidungsfindung stattfinden kann. Das heißt dass die KI-Skripte höherer Ordnung(Soldatengruppenführer) zuerst, dann der Reihe nach die jeweils niedrigeren(Fußsoldat). Es muss beispielsweise zuerst die Taktikeinheit ihre Befehle formulieren, bevor der Soldat agieren kann. Die Ausführungsreihenfolge der KI-Skripte selber Ordnung sollte variabel sein.

Um den Einheiten ihre KI-Skripte zuzuordnen und die Ausführungsreihenfolge festlegen zu können, benötigt man eine Regelung. In der gAImes-Engine

werden die KI-Skripte über spezielle Kürzel in die Klassennamen erkannt und zugeordnet.

3.7 Kommunikation der Logikkomponenten

Wie bereits an anderer Stelle erwähnt, kann eine Befehlshierarchie für die KI-Skripte erstellt werden. Damit diese sinnvoll bleibt, benötigt man eine Kommunikation untereinander. So kann das Oberkommandierende KI-Skript seine Befehle an die Offensive und Defensive delegieren. Hierfür müssen sich Logikeinheiten eindeutig adressieren lassen. In der Diskussion standen zwei Varianten. Zum einen könnten die Logikeinheiten untereinander direkte Referenzen besitzen, welche sie nutzten um Befehle oder Parameter direkt zu manipulieren. Zum Anderen könnten die Logikeinheiten über ein Protokoll mittels einer Art Briefkastensystem Befehle direkt adressiert absetzen, welche dem Empfänger entweder automatisch von der gAImes-Engine zugewiesen oder von dieser eigenständig abgefragt werden. Hierbei ist, bei der zweiten Variante, darauf zu achten, dass das Befehlsformat leicht erweiterbar ist, damit auch komplexere, zukünftige Befehle verschickt werden können. Damit die Schnittstelle von KI-Skript zu gAImes-Engine nicht zu groß wird kann in Erwägung gezogen werden diese Methode der Befehlskommunikation auch zu nutzen um Aktionen an das Spiel zu schicken. Beispielsweise durch Reservierung einer eindeutigen Befehlsadresse für das Spiel. Zudem muss ein Mechanismus bereitgestellt werden, der nicht abgeholte Befehle automatisch löscht.

3.8 Speichern von Informationen

Da die KI-Skipte zustandslos implementiert werden, bietet die gAImes-Engine KI-Entwicklern die Möglichkeit, über ein Schnittstelle, Speicherkomponenten zu nutzen in welchen sie die Erkenntnisse, die während des Spielverlaufs gesammelt wurden, speichern können. So könnten sich die KI-Skripte beispielsweise ihre Strategie und noch nicht abgearbeitete Befehle merken oder Milestones aufbauen und abarbeiten. Da es viele verschiedene Möglichkeiten gibt, Informationen abzuspeichern muss die Schnittstelle der Gedächtniskomponenten allgemeingültig angelegt sein. Hier stellt sich die Frage wie lange ein solcher Gedächtniseintrag gespeichert werden soll? Altern die Gedächtniseinträge so werden sie nach einiger Zeit automatisch gelöscht, um den Speicherverbrauch zu reduzieren. Zudem müssen Mechanismen entwickelt werden, die den schnellen und gezielten Zugriff auf die Gedächtniseinträge erlauben. So zum Beispiel ein Index, in dem alle Gedächtniseinträge aufgelistet sind, oder eine sortierte Ausgabe der Einträge mittels vorheriger Markierung selbiger

mittels Tags, wie beispielsweise MILESTONE, BEFEHL, STATUS, ALTES-ZIEL.

4 Der KI Entwickler

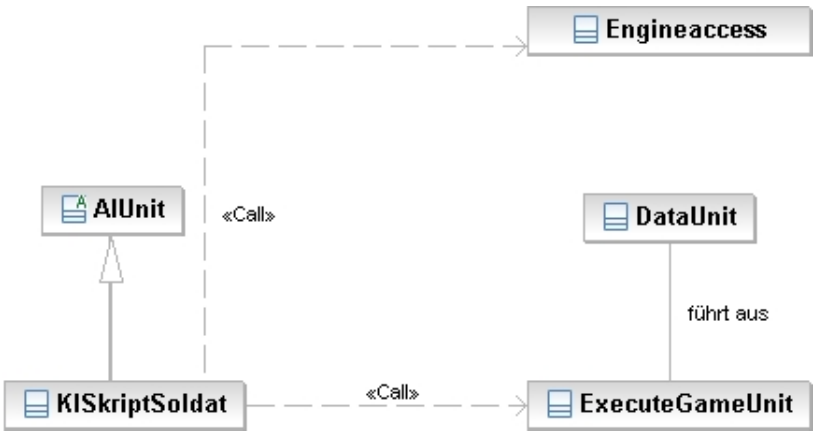


Abbildung 2: Relationen des KI-Skripts

In diesem Kapitel werden die Funktionsweisen und der Aufbau der Klassen der gAImes-Engine beschrieben, die für den Entwickler der KI-Skripte von Bedeutung sind. Zudem wird beschrieben, wie der KI Entwickler auf die Datenstrukturen der gAImes-Engine zugreifen kann und darüber mit dem Spiel kommuniziert. Im letzten Abschnitt wird ein Vorschlag beschrieben, wie die Entwicklung eines KI-Skripts aussehen könnte.

4.1 KI-Skript

```
public abstract void process(DataUnit dU, engineAccess eA);
```

Die Klasse AIUnit ist die Grundlage zur Implementierung eigener Logik, der zuvor genannten KI-Skripte. Der KI Entwickler lässt seine eigene Klasse von dieser abstrakten Klasse erben. Die Vererbung ist für die engine-interne Verarbeitung der Logikklassen nötig.
Hierbei ist die Methode ***process*** zu implementieren. Diese Methode wird zur Laufzeit ausgeführt und enthält die, vom KI Entwickler geschriebene Logik. Die Methode hat zwei Übergabeparameter. Der erste Parameter ***dataUnit*** entspricht der konkreten Instanz der Spieleinheit im Spiel (siehe Seite 20, Abschnitt 4.2). Über diesen Parameter bekommt der KI Entwickler Zugriff auf den Status der Spieleinheit. Der zweite Parameter ***engineAccess*** ist die interne Schnittstelle zur gAImes-Engine (siehe Seite 21, Abschnitt 4.4). Über diese bekommt der Entwickler Zugriff auf die Datenstrukturen der gAImes-Engine. Diese sind beispielsweise die Karte oder erhaltene Spielereignisse.

Aus Effizienzgründen wird zur Laufzeit von jeder Logikeinheit nur eine Instanz erzeugt, welche dann im Spielablauf mit der jeweilig zugewiesenen Spieleinheit geladen wird. Aus diesem Grund sollen in der Logikeinheit selbst keine Referenzen auf Datenstrukturen gehalten werden. Hierfür bietet die gAImes-Engine, über den Parameter ***engineAccess***, die Möglichkeit Informationen abzuspeichern (siehe Seite 21, Abschnitt 4.4).

4.2 Spieleinheit

Die Klasse DataUnit ist die Repräsentation der konkreten Instanz der Spieleinheit im Spiel. Diese Klasse wird jede Spielrunde vom Spiel an die gAImes-Engine übermittelt und enthält den aktuellen Status der Spieleinheit. Diese enthält zum Beispiel wie viele Lebenspunkte die Spieleinheit noch hat. Die Klasse wird auf Seite 30 im Abschnitt 5.2.2 näher beschrieben.

4.3 ExecuteGameUnit

```
public Vector<String> getClassMethods();
public Class[] getReturnClass(String methodName);
public Class[] getParameterClasses(String methodName);
public static Object invokeMethod(Object unit, String methode)

int getUnitPosX();  // für das Codebeispiel
```

Diese Klasse dient dem KI-Entwickler als Hilfsklasse, um Zugriff auf die übergebene, konkrete Einheit zu bekommen. Sie bedient sich der Java-Technology Reflection.
Über die Methode ***getClassMethods*** wird eine Liste aller Methoden zurückgeliefert, die der KI-Entwickler auf der Einheit ausführen kann. So erhält er alle Informationen über die Einheit, die der Spielentwickler seinen Einheiten für das Spiel geben will. Mit dieser Technik ist es dem Spielentwickler möglich, beliebige Funktionen zu definieren und sie dem KI-Entwickler zur Verfügung zu stellen.

Die Methode ***getReturnClass*** liefert Rückgabetyp der Methode, die man als Parameter übergibt. Als Parameter sind alle Methodennamen gültig, die von ***getClassMethods*** zurückgeliefert werden.
Analog zu ***getReturnClass*** liefert die Methode ***getParameterClasses*** eine Liste mit den Übergabeparametern zurück, die die übergebene Methode akzeptiert.

Die Ausführung erfolgt durch ***invokeMethod***, in Kombination mit den beiden genannten Methoden kann die aktuelle Position einer Einheit ermittelt werden.

In folgendem Codebeispiel nehmen wir an es existiert eine Methode ***getUnitPosX***, die die aktuelle X-Koordinate für den Spieler zurückliefert. Um das Beispiel übersichtlich zu halten gehen wir davon aus, wir kennen den Rückgabewert und den Namen der Methode und müssen ihn nicht mit den Methoden ***getReturnClass*** und ***getParameterClass*** ermitteln.

```
Object player_unit = curUnit;
String methode = new String ("getUnitPosX");
int xPos = ((int)ExecuteGameUnit.invokeMethod(player_unit, methode));
```

4.4 Zugriff auf die gAImes-Engine

Die Klasse engineAccess dient dem KI Entwickler als Schnittstelle zu den Datenstrukturen der gAImes-Engine. Diese Klasse ermöglicht ihm auf die von der gAImes-Engine aufbereiteten Informationen die das Spiel schickt zuzugreifen. Es gilt hierbei zwischen fünf Kategorien von Informationen zu unterscheiden, der Karte, den Spieleinheiten, den Befehlen, den Spielereignissen und den Gedächtniseinträgen.

4.4.1 Zugriff auf die Karte

```
public Map getMap();
public Map getMap(float radius);
```

Über diese Methoden erhält der KI Entwickler Zugriff auf die Karte. Hierbei können nur die Kartenelemente abgefragt werden die die gAImes-Engine vom Spiel übermittelt bekommen hat. So fehlen wenn es das Spiel unterstützt bei einem so genannten *Fog of War* die Kartenabschnitte die die Logik noch nicht erkundet hat.
Um die Komplexität der Logikeinheiten einschränken zu können, bietet die Methode ***getMap*** den Parameter ***radius*** an. Mit diesem lässt sich die Größe der zurückgegebenen Karten einschränken. Hierbei wird nur der Abschnitt der Karte zurückgegeben, der sich innerhalb des angegebenen Radius um die Spieleinheit befindet. So kann die Geschwindigkeit der Berechnungen erhöht werden, da beispielsweise eine im Aktionsradius sehr eingeschränkte Einheit nicht die ganze Karte auswerten muss. Der Aufbau der Karte wird auf Seite 44 im Abschnitt 6.2.2 näher beschrieben.

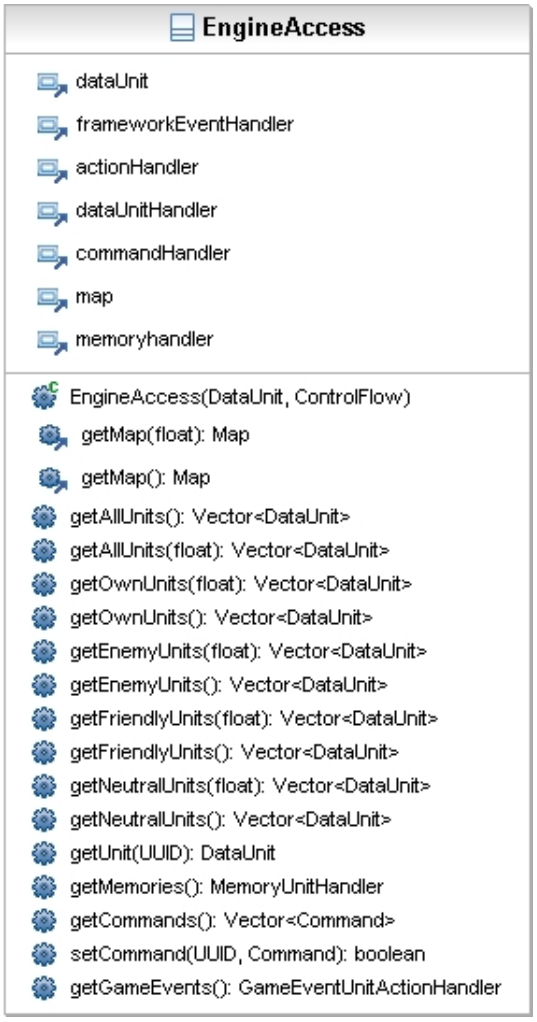


Abbildung 3: Klassendiagramm der Klasse engineAccess

4.4.2 Zugriff auf Einheiten

```
public DataUnit getUnit(UUID unitID);
public Vector<DataUnit> get...Units();
public Vector<DataUnit> get...Units(float radius);
```

Mit Hilfe dieser Methoden kann der KI Entwickler sich die anderen Spieleinheiten im Spiel ausgeben lassen. Hierbei adressiert die Methode ***getUnit(UUID unitID)*** mit Hilfe des Parameters ***unitID*** eine bestimmte Spieleinheit. Die anderen beiden Methoden gibt es für vier Kategorien. Diese sind neutrale, feindliche, freundliche oder alle Einheiten. Die ... sind hierbei nach Wunsch durch ***Neutral, Friendly, Enemy, All*** zu ersetzen. Hierbei gibt

es die Möglichkeiten sich entweder alle der gAImes-Engine bekannten Einheiten der Kategorie ausgeben zu lassen oder nur die in einem bestimmten Radius um die Spieleinheit. Die Variante mit dem Radius soll die Möglichkeit bieten die Komplexität und den Berechnungsaufwand der Logik einzuschränken. Die Verwaltung der Spieleinheiten wird auf Seite 43 im Abschnitt 6.2.1 näher beschrieben.

4.4.3 Zugriff auf das Gedächtnis

```
public Vector<MemoryUnit> getMemoryUnits();
public void setMemoryUnit(MemoryUnit memoryUnit);
public void deleteMemoryUnit(UUID memoryID);
```

Diese Methoden geben dem KI Entwickler die Möglichkeit Informationen die die Logikeinheit während des Spiels gewinnt oder Entscheidungen die sie getroffen hat, zu speichern, abzurufen oder zu löschen. Ein Gedächtniseintrag hat das Format `MemoryUnit`. Diese abstrakte Klasse verlangt lediglich eine eindeutige ID und einen Typ zugewiesen zu bekommen, damit die gAImes-Engine sie verwalten kann. Eigene Gedächtniseinträge müssen durch Erben von dieser Klasse erstellt werden. In solch einem Konstrukt könnte sich ein KI-Skript zum Beispiel seinen kompletten Laufpfad speichern und jede neue Runde ein Stück weiter ablaufen, um die Berechnung nicht jede Runde neu tätigen zu müssen. Die Verwaltung des Gedächtnisses wird im Abschnitt 6.2.5 auf Seite 46 näher beschrieben.

4.4.4 Zugriff auf die Befehle

```
public boolean setCommand(UUID receiverID, Command command);
```

Über diese Methode bietet die gAImes-Engine den KI-Skripten die Möglichkeit Befehle an das Spiel zu senden. Diese Befehle sollen dem Spiel Handlungen, die ein KI-Skipt für eine Spieleinheit bestimmt hat vorgeben. Das Spiel prüft die Handlungen, die im Befehl stehen, auf ihre Zulässigkeit nach den Spielregeln und bestätigt oder verweigert den Befehl. Somit erfährt die Logik sofort, ob sie einen zulässigen Befehl abgeschickt hat und kann entsprechend handeln. Diese Handlungsbestätigung soll es dem KI Entwickler ermöglichen, Logik zu schreiben, die die Regeln des Spiels während des Spielverlaufs erlernt. Der erste Parameter ***receiverID*** identifiziert den Empfänger. Die ID ***0*** adressiert hierbei einen Befehl an das Spiel. Der zweite Parameter ***command*** ist bei einem Befehl an das Spiel vom Typ ***CommandGameAction*** und enthält die Handlung. Diese Handlung kann zum Beispiel eine Bewegung sein.

```
Vector<String>() parameter = new Vector<String>();
parameter.add(String.valueOf(mapfield.getPosition().x));
parameter.add(String.valueOf(mapfield.getPosition().y));
parameter.add(String.valueOf(mapfield.getPosition().z));

engineAccess.setCommand(new UUID(0,0),
new CommandGameAction("move", destinationID, sourceID, parameter)));
```

Die weiteren Funktionen dieser Methode, in Bezug auf das Versenden von Befehlen an andere KI-Skripte, werden im nachfolgenden Abschnitt 4.5.1 beschrieben. Die Verwaltung und der Aufbau der Befehle werden auf Seite 46 im Abschnitt 6.2.4 näher beschrieben.

4.5 Schichtenmodell

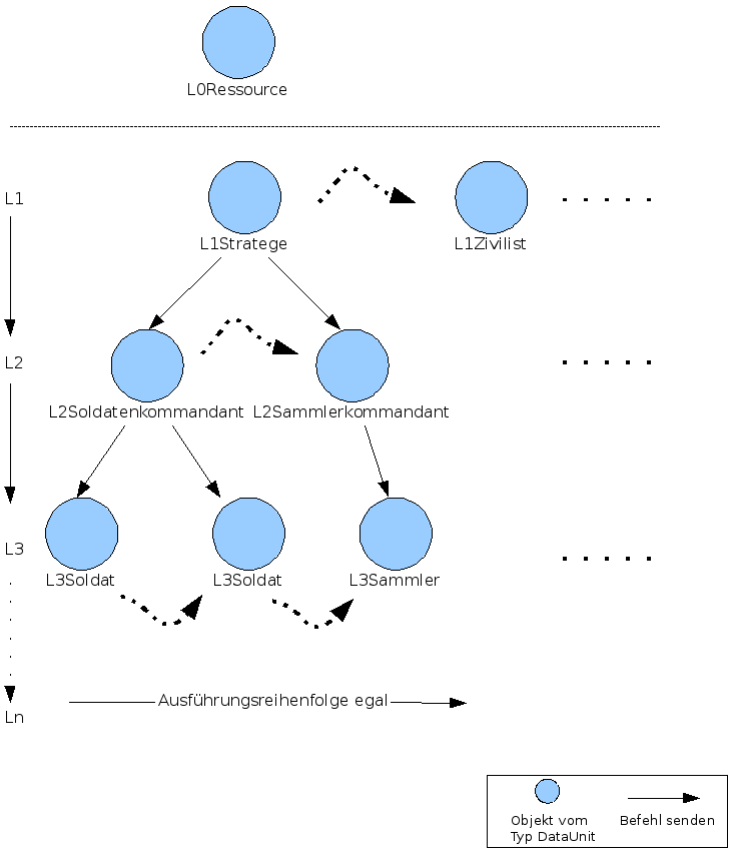


Abbildung 4: Abstrakte Darstellung des Schichtenmodells

Neben der klassischen Logikentwicklung, mit nur einer Einheit, welche alle Entscheidungen trifft, ist es dem KI Entwickler möglich, seine Logik aufzuteilen. Die Entscheidungsfindung kann hierbei auf mehrere Logikeinheiten aufgeteilt werden. Zur Verdeutlichung stelle man sich eine militärischen Befehlshierarchie vor. Die Befehle werden von einer Person an der Spitze an die darunterliegenden Personen verteilt. Es beginnt mit der Wurzel von oben und splittet sich in beliebig viele Teilbäume. Die Entscheidungen für Aktionen im und Reaktionen auf das Spielgeschehen können somit auf mehreren Schichten getroffen werden. So würden auf der obersten Schicht grundlegende strategische Entscheidungen gefällt und hier alle zur Verfügung stehenden Ressourcen den Taktikeinheiten auf der nächsten Schicht zugeteilt. Diese Ressourcen entsprechen im Spiel zu sammelnden Erzen, aber auch alle bereits gebauten Einheiten. Durch eine Zuweisung der Ressourcen auf z.B. 3 Taktikeinheiten **Angriff**, **Verteidigung** und **Ressourcensammlung**, kann die erste Schicht den Schwerpunkt des Ausbaus in der folgenden Spielrunde festlegen. Auf der zweiten Schicht wären die Taktikeinheiten angesiedelt. Diese sind spezialisiert auf Bereiche wie Angriff, Verteidigung oder Ressourcensammlung. Entsprechend der Ressourcenzuteilung(z.B. von Spieleinheiten) treffen sie taktische Entscheidungen. Der Angriff muss beispielsweise, entsprechend den Einheiten die er zugeteilt bekommen hat entscheiden, ob er angreifen will oder wartet und neue Einheiten baut. Auf der letzten Schicht würde sich die Logik der Basiseinheiten des Spiels befinden. Das sind z.B. die Soldaten oder die Ressourcensammler. Die Logik der Basiseinheiten würden autonom, entsprechend den Befehlen die sie von der Taktik erhalten, Entscheidungen treffen. So können sie ihre Wege selbst berechnen und autonom Hindernissen ausweichen, oder Gegner angreifen. Die Entscheidungen die Sie treffen, wie bspw. Bewegungen werden an das Spiel übermittelt.Dieses Schichtenmodell ist variabel angelegt. Es muss mindestens eine Schicht und eine Einheit geben, welche alle Entscheidungen trifft und Aktionen an das Spiel gibt. Es kann aber auch 4 oder mehr Schichten geben, so könnte man bspw. eine Schicht für Gruppenführer entwickeln, um Formationsverhalten aufzubauen.

4.5.1 Erweiterte Befehle

```
public Vector<Command> getCommands();
public boolean setCommand(UUID receiverID, Command command);
public Vector<Command> getCommands();
```

Damit unterschiedliche Logikeinheiten des Schichtenmodells miteinander kommunizieren können, gibt es ein internes Nachrichtensystem. Die Nachrichten werden hierbei über die Methode ***setCommand*** in der Klasse engineAccess

verschickt. Dabei adressiert der Parameter ***receiverID*** mittels der ID der Spieleinheit den Empfänger. Jede Einheit kann über die Methode ***getCommands*** die an sich gerichteten Befehle abrufen.

4.6 Entwicklungsschritte für KI-Skripte

In diesem Abschnitt wird exemplarisch gezeigt, wie die Entwicklung eines KI-Skriptes ablaufen kann. Der Vorschlag orientiert sich am Entwicklungvorgehen der Spiellogik im Abschnitt 8 und sollte als Orientierungspunkt verstanden werden. Hierbei sind Reihenfolge und Aufbau nicht bindend.

4.6.1 KI-Skripte erstellen

Als initialen Schritt zur Erstellung eines KI-Skriptes, muss eine Klasse erstellt werden, die von der abstrakten Klasse AIUnit(siehe Abschnitt 4.1) beerbt wird. Der Klassenname muss eindeutig sein, da das Spiel die Klasse mittels des Namens seinen Spieleinheiten zuweist und der Name muss die Zeichenkette AIUnit enthalten. Anhand dieser Signatur, kann gAImes alle anderen Dateien herausfiltern, die nicht als KI-Skript geladen werden sollen.

4.6.2 Datenstrukturen erweitern

Sollten die Datenstrukturen der gAImes-Engine nicht den Ansprüchen genügen, müssen diese erweitert werden. In der gAImes-Engine gibt es nur einen Befehlstyp um einer Einheit eine Aktion gegen eine andere Einheit zu befehlen. Solch ein Befehl wird mit einem CommandGameAction Objekt erzielt. Ist es gewünscht einen Befehl gegen eine Gruppe abzusetzen muss die Datenstruktur Command erweiter werden. Um dies zu tun wird eine neue Klasse erstellt, welche von Command erbt und diese um die gewünschten Parameter und Methoden erweitert. Als Erbe der Klasse Command kann die gAImes-Engine mit dem neuen Befehlstyp arbeiten. Die neue Klasse wird im Ordern ***aIDevelopment*** abgelegt und kann dort von den KI-Skripten verwendet werden.

4.6.3 Erster Schritt: Befehle abrufen

```
public Vector<Command> getCommands();
```

Als erstes werden die Befehle für die Spieleinheit abgerufen. Dies geschieht über die Klasse ***engineAccess*** und deren Methode ***getCommands***. Die Befehle werden nun einzeln ausgewertet und priorisiert. Entsprechend des

derzeit höchst priorisierten Befehls, muss über das weitere Vorgehen entschieden werden. Ein Befehl für einen Soldaten könnte zum Beispiel ein Angriff auf eine andere Spieleinheit sein.

4.6.4 Zweiter Schritt: Spielumfeld analysieren

Als zweites wird das Spielumfeld analysiert. Dies bedeutet im Falle des Angriffsbefehls wird die gAImes-Engine nach dem Standort des Ziels gefragt. Ist der Standort bekannt, werden die Daten über das Kartenfeld abgerufen und ein Pfad zum Ziel berechnet.

4.6.5 Dritter Schritt: Befehle absetzen

```
public boolean setCommand(UUID receiverID, Command command);
```

Als letzten Schritt erzeugt die Logik einen Befehl, in welchem sie ihre Handlung, beispielsweise „Schritt vorwärts“, an das Spiel übermittelt. Dieser Befehl ist ein Objekt vom Type ***CommandGameAction***. Der Befehl wird dann mittels der Methode ***setCommand*** aus der Klasse engineAccess an die Id **0**,welche das Spiel adressiert, versendet. Das Spiel prüft, die zulässigen Aktion, z.B. darf ein Feld in Schrittrichtung betreten werden und bestätigt oder verweigert den Befehl.

5 Der Spiel Entwickler

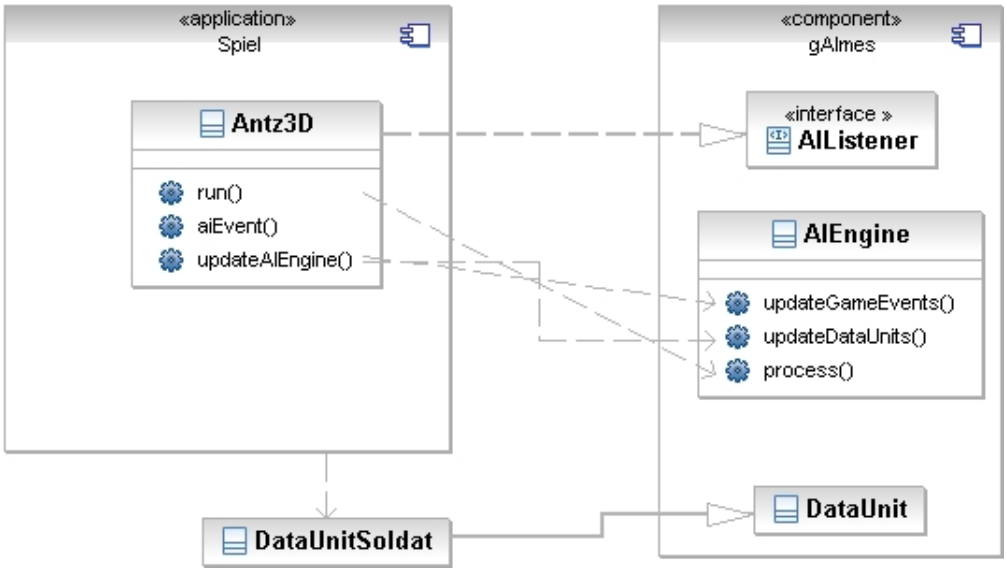


Abbildung 5: Relationen des Spiels zur gAImes-Engine

In diesem Kapitel werden die Schritte erläutert die ein Spielentwickler ausführen muss um die gAImes-Engine in sein Spiel einzubinden. Die ersten Abschnitte beschreiben wie die gAImes-Engine in das Spielprojekt eingebunden werden kann und wie die fertigen KI-Skripte in die gAImes-Engine einzufügen sind. In den darauf folgenden Abschnitten wird die Kommunikationsschnittstelle mit der gAImes-Engine und die dafür nötigen Datenstrukturen erläutert. Im letzten Abschnitt werden die initialen Anwendungsschritte aufgezeigt, die das Spiel ausführen muss um die gAImes-Engine zu konfigurieren, sowie jene Anwendungsschritte um mit ihr während des Spielablaufs zu kommunizieren.

5.1 Einbinden der gAImes-Engine

5.1.1 Einbinden der jar Datei

Um die gAImes-Engine als Spielentwickler nutzen zu können muss man das Paket **gd_gAImes.jar** einbinden.

Benutzt man eine Entwicklungsumgebung für Java, zum Beispiel Eclipse, ist das Einbinden von Paketen über einige Menüs zu erreichen. Zunächst legt man ein Projekt an, in dem das Spiel(Projekt) erstellt werden soll. Dann wählt man die Eigenschaften des Projekts aus, meist über einen Rechtsklick

auf den Projektordner zu finden. Unter der Option **Java Build Path** wählt man **Add External JARs** und sucht die **gd_gAImes.jar** aus der Verzeichnisstruktur. Damit bindet Eclipse alle entsprechenden Klassen ein.
Arbeitet man hingegen nicht mit einer Entwicklungsumgebung, muss man die ‚gd_gAImes.jar' beim Kompilierungsaufruf mit der Option **-cp** angeben. Die Option erweitert den Java Klassenpfad durch die nachfolgende ***.jar** Datei. Beispiel:

```
java -cp 'gd_gAImes.jar' Main
```

5.1.2 Erstellen des Ordners AIDevelopment

Damit die gAImes-Engine KI-Skripte laden kann, muss der Ordner AIDevelopment, als Unterordner vom Projektordner, angelegt werden. Hier sucht die gAImes-Engie nach den entsprechenden Skripten und läd diese.

5.1.3 AI Skripte in Ordner kopieren

Alle KI-Skripte, die im Spiel benutzt werden sollen, müssen in den AIDevelopment Ordner kopiert werden. Benötigt wird dabei die *.class Datei, die der Javakompiler erstellt.

5.2 Kommunikation mit der gAImes-Engine

Ein Spiel benutzt die gAImes-Engine um Algorithmen für künstliche Intelligenz und Verhalten einzubinden. In den vorangegangenen Kapiteln wurde beschrieben, wie ein Spiel seine Einheiten anpassen muss, damit die gAImes-Engine damit arbeiten kann. In diesem Kapitel wird erklärt, welche Schnittstellen das Spiel benötigt, um die Berechnungen und Informationen der KI-Skripte zu erhalten. Zudem wird erklärt mittels welcher Datenstrukturen Informationen an die gAImes-Engine übertragen werden.

5.2.1 Implementieren der AIListener Schnittstelle

```
public boolean aiEvent(GameEventUnitAction unitAction);
```

Ein Spiel arbeitet in den meisten Fällen mit einer Spielschleife. Diese Technik ist weiter oben erklärt. Aus den genannten Gründen liegt die Ausführungskontrolle beim Spiel. Zur Abarbeitung der KI-Skripte wird diese Kontrolle der gAImes-Engine übergeben. Um während dieser Phase Informationen an das Spiel weitergeben zu können wird eine Callback-Methode verwendet.

Das Spiel muss diese Callback-Methode anbieten. Hierzu implementiert es das Interface ***AIListener***, welches im Paket der gAImes-Engine mitgeliefert wird. Das Interface verlangt die Implementierung der Methode ***aiEvent***, wie oben definiert. Alle Ereignisse, die während der Berechnung einer Aktion für eine Einheit auf dem Spielfeld von der gAImes-Engine gemacht werden, erzeugen einen Aufruf dieser Methode im Spiel. Der Entwickler kann auf Grund des Übergabeparameters entscheiden, wie das Spiel darauf reagieren soll.
Zum Beispiel könnte das Spiel eine Einheit auf dem Spielfeld haben, die gerade von einer gegnerischen Einheit attackiert wird. Diese Information wird von einem KI-Skript bearbeitet, während seiner Ausführungszeit. Es entscheidet eine Gegenmaßnahme einzuleiten und schickt eine entsprechende Nachricht an das Spiel. Die Nachricht landet in der Callback-Methode ***aiEvent***. Das Spiel enthält die Nachricht, den Gegner anzugreifen. Es prüft dann zu der entsprechenden Einheit, ob der Zug legal ist, ob zum Beispiel die Einheit in Reichweite, Sichtweite usw. ist. Jetzt kann das Spiel die entsprechende Animation setzen und die Methode mit **true** beenden. Damit signalisiert es, dass der Zug legitim war. Wird die Methode mit **false** beendet, kann das KI-Skript entsprechend reagieren. Wichtig ist, dass das Spiel die Methode nicht zu lange ausführt, da diese blockend ist und die Ausführung des KI-Skripts anhält und somit das Spiel verlangsamt.

Die Klasse ***GameEventUnitAction*** enthält alle Informationen, die ein KI-Skript an das Spiel übertragen kann. Die Klasse verwaltet vier Variablen. Die ersten beiden sind vom Typ **UUID** und stellen die ***einheitenID*** der entsprechenden Einheit dar, die der Empfänger dieser Nachricht sein soll.
Der zweite Parameter ist die AbsenderID. Die Bedeutung der ID wird im direkt folgenden Kapitel erläutert.
Der dritte Parameter ist vom Typ String und enthält den Namen der Methode, die das KI-Skript verwenden möchte. Hat eine Einheit Soldat zum Beispiel die Fähigkeit **Schildblock**, dann enthält die Variable genau den Namen der Methode. Mehr zu dieser Technik wird im direkt folgenden Kapitel näher erläutert.
Der vierte Parameter ist vom Typ **Vector<float>** und kann alle Übergabeparameter für die Methode **Schildblock** aufnehmen.

5.2.2 Erstellen der Spieleinheiten

Als konkrete Spieleinheiten auf dem Spielfeld werden ***DataUnits*** bezeichnet. Für den Programmierer sind diese somit Instanzen seiner Einheitenklassen, die wiederum von der Klasse ***DataUnit*** aus der gAImes-Engine erben.

Wird im folgenden von einer DataUnit geredet, ist damit eine konkrete Einheit gemeint, wobei es egal ist um welche Art es sich dabei genau handelt.

Die Klasse ***DataUnit*** enthält vier wichtige Variablen, die für die gAImes-Engine zur Verwaltung und Steuerung notwendig sind. Dabei handelt es sich um zwei eindeutige IDs, anhand derer die Einheit identifiziert und der entsprechenden gAImes-Engine zugeordnet wird. Diese kann man als Spielerzuordnung verstehen. Eine Umgebung steht für einen Spieler. Gibt es zwei KI gesteuerte Spieler, so besitzt jeder seine eigene gAImes-Engine.
Zusätzlich gibt es eine Positionsvariable vom Typ ***Vector3f***, die von dem Java-Paket ***javax.vecmath*** zur Verfügung gestellt wird.
Und einen String, der den Namen der KI-Klasse enthält, mit der die Einheit geladen werden soll. Genauere Informationen befinden sich in Kapitel 4.6.1.

Möchte der Spielentwickler eine DataUnit erstellen, muss er nur eine Instanz seiner Klasse laden. Das setzt voraus, dass wie oben erwähnt, die Einheit der Gattung Soldat ist und es existiert eine Klasse Soldat, die von ***DataUnit*** erbt. Folgend ein Codebeispiel für die Erstellung einer Einheit.

```
DataUnit soldat = new SoldatL3( String aiClassName,
                                UUID einheitenID,
                                UUID engineID,
                                Vector3f position);
```

Es ist zu empfehlen die erstellte Einheit Soldat im Spiel zu speichern. Werden Informationen über die Methode ***aiEvent*** an das Spiel übertragen, so wird zur Identifizierung der Einheit im Übergabeparameter die **einheitenID** verwendet. Man könnte zum Beispiel eine Hashmap wie folgt verwenden ***HashMap<UUID, DataUnit>***.

In obigem Abschnitt wird die Klasse Soldat für das Beispiel vorausgesetzt. Bei der Erstellung dieser Klasse ist zu beachten, dass sie von ***DataUnit*** erben muss und die Konvention für das Schichtenmodell einhält. Die Konventionen des Schichtenmodells werden im folgenden Abschnitt erklärt.
Damit ein KI-Skript mit dieser Einheit arbeiten kann, bietet die gAImes-Engine eine Technik an, die an anderer Stelle genauer erklärt wird. So lassen sich beliebige Methoden auf einer DataUnit ausführen, die der Programmierer nicht mit einem expliziten **typecast** ausführen muss. Ein **typecast** wäre nicht möglich, da die auszuführende Java Klasse im Spiel nicht bedingt verfügbar sein muss. Für das folgende Beispiel ist zu beachten, dass für jede Aktion, die eine Einheit können soll, eine Methode angeboten werden

muss. Jede Methode muss über eine Annotation **Executable** verfügen. Die Einheit Soldat soll beispielsweise folgendes können: Laufen, Schwertangriff, Schildblock. Dann muss entsprechend für diese Aktionen eine Mehtode existieren. Die Klasse Soldat könnte wie folgt aussehen.

```
class SoldatL3 extends DataUnit
{
  @Executable
  public void Laufen(flaot x)
  @Executable
  public void Schwertangriff(UUID enemyID)
  @Executable
  public void Schildblock()
}
```

Im Kapitel ***aiEvent*** 7.4 werden die Spielaktionen erwähnt, die über die Schnittstelle ***GameEventUnitAction*** kommen. Der dritte Parameter dieser Schnittstellenklasse würde analog zu den oben definierten Methoden in ***SoldatL3*** einen String mit den Methodennamen besitzen.

```
public boolean aiEvent(GameEventUnitAction unitAction)
{
    String event = unitAction.getActionEventType();
    System.out.println(event);
}
```

Die Ausgabe wäre eine der Methodennamen, aus der Klasse **SoldatL3**. Ausgabe: **Schildblock**.

Bei der Erstellung der Einheitenklasse kann der Spielentwickler den Namen der Klasse frei wählen. Er muss allerdings entsprechend der Schicht, auf der die Einheit aggieren soll, ein Kürzel anhängen. Theoretisch kann man beliebig viele Kürzel verwenden. Zur Veranschaulichung wähle ich L1, L2 und L3. In unserem aktuellen Beispiel **Antz3D** werden diese Kürzel verwendet.
L1 steht für einen Kommandeur, der die Spielleitung für einen vollständig KI gesteuerten Spieler übernimmt. Man kann das vergleichen mit einem menschlichen Spieler, der über dem Spiel sitzt und mit der Maus seinen Einheiten Befehle gibt.
L2 steht für verschiedene Zwischenkommandeur. Hier kann das Verhalten für zum Beispiel Angriff, Verteidigung, Ressourcensammeln oder Basisbau implementiert werden.

L3 ist das Verhalten der Einheiten. Hier werden zum Beispiel Bewegungsalgorithmen, wie ein **A*-Algorithmus** eingebunden. In einem Spiel ist dies mit den Einheiten auf dem Spielfeld zu vergleichen. Diese agieren stückweise autonom, wenn sie zum Beispiel auf dem Spielfeld stehen und angegriffen werden. Dann soll die Einheit mit einem Gegenangriff kontern oder weglaufen etc.
Diese Schicht ist sehr wichtig und muss für jede Einheitenklasse auf dem Spielfeld eingebunden werden, wenn sie in irgendeiner Art agieren können soll. Ein Beispiel für einen Klassennamen einer Einheit, die einen Soldaten steuern soll wäre **L3Soldat**. Die Schichten und die damit verbundenen Kürzel können beliebig vertieft werden, um komplexe Strukturen zu schaffen. Ausgeführt werden sie chronologisch von klein nach groß. Also zuerst der Kommandeur, dann der zweite Kommandeur und ann die Einheiten in unserem Beispiel.
Eine Ausnahme ist die Schicht **L0**. Diese sollte für Klassen verwendet werden, die neutral sind, oder Art nicht aktiv am Spielgeschehen teilnehmen. Dies gilt zum Beispiel für Ressourcen. Ressourcen müssen als Einheit an der gAImes-Engine registriert werden, solange sie von der KI erkannt werden sollen. So kann eine KI die Ressourcen erkennen und entsprechend handeln.

5.2.3 Erstellen des Spielfelds

Das ganze Spiel findet auf einem Spielfeld statt. In den meisten Spielen interagieren die Spieleinheiten mit dem Spielfeld. Zum Beispiel bauen sie Ressourcen ab, können nicht über oder durch Hindernisse wie Berge oder Seen laufen.
Alle initialen Informationen über das Spielfeld muss der Spielentwickler der gAImes-Engine zu Spielbeginn mitteilen. Spielfeldveränderungen während des Spielablaufs werden der gAImes-Engine zu gegebenem Zeitpunkt mitgeteilt. So zum Beispiel bei Entdeckung neuer Spielfeldgebiete **Fog of War**. Dann kann die gAImes-Engine auf Anfragen der KI-Skripte reagieren. Zum Beispiel erstellt die gAImes-Engine einen Wegegraphen für den **A*-Algorithmus** zur Berechnung der Laufpfade.

Gespeichert wird die Karte in der gAImes-Engine mittels der Klasse ***Map***, welche einen Graphen aus Kartenfeldern der Klasse ***Mapfields*** enthält. Ein Kartenfeld speichert drei Parameter, die Position des Feldes in Bezug auf das Gesamtspielfeld, ein Wert für die Bodenbeschaffenheit, zum Beispiel passierbar oder unpassierbar und ein Vector mit allen Nachbarfeldern. Nachfolgend ein Beispiel: ein Kartenfeld wird erzeugt, mit der Position (0, 0, 0) und mit dem Wert 0 für die Bodenbeschaffenheit, dass als passierbar gekennzeichnet

wird. Diese Kartenfeld hat vier Nachbarkartenfelder, die kreuzförmig angeordnet werden.

```
Mapfield node =
  new Mapfield(new Vector3f(0,0,0),
  mapLoader.getTerrain(), null);
// oben
node.setNeighbor(new Vector3f(0, 1, 0));
// unten
node.setNeighbor(new Vector3f(0, -1, 0));
// links
node.setNeighbor(new Vector3f(-1, 0, 0));
// rechts
node.setNeighbor(new Vector3f(1, 0, 0));
```

5.2.4 Erstellen der Spielereignisse

Spielereignisse sind Informationen, die während des Spiels an die gAImes-Engine übergeben werden. Sie enthalten Änderungsinformationen über die Spielumgebung. Die Änderungen sind während einer Spielrunde aufgetreten und müssen in der gAImes-Engine aktualisiert werden. Das Spiel wird auf dem Rechner rundenbasierend abgearbeitet. Doch wird dem Spieler durch die hohe Ausführungsgeschwindigkeit Echtzeit simuliert. Pro Spielrunde darf jeder Spieler für alle seine Einheiten Spielaktionen ausführen. Nach Ablauf der Runde muss das Spiel die Veränderungen an der Spielumgebung an die eingebundenen gAImes-Engine Instanzen weiterleiten, damit die KI-Skripte angemessen reagieren können.
Es existieren zwei Arten von Spielereignissen. Ereignisse welche die gAImes-Engine betreffen und solche die die Einheiten in der Spielumgebung betreffen. Eine Angriffsaktion auf eine Spieleinheit wird als ein Spielereignis an die gAImes-Engine gesendet. Eine weiteres Spielereignis ist der Tod einer Spieleinheit. Diese wird an die gAImes-Engine gesendet, die die entsprechende Spieleinheit aus der Liste aller Einheiten entfernt. Nachfolgend werden diese beiden Beispiele als Codebeispiele erzeugt.

```
// Entfernen einer Spieleinheit
  GameEvent gameEvent =
  new GameEventUnitHandle(GameEventType.kill,
   dataUnit.getOwnID());

// Ein Angriff auf eine Spieleinheit
```

```
GameEvent gameEvent =
new GameEventUnitAction (
 destination,
 source,
 "fire",
 parameter);
```

Je nach Spiel kann es von Nöten sein neue Ereignisse zu erstellen. Dies geschieht mittels Vererbung. Das neue Ereignis erbt direkt von der Klasse ***GameEvent*** oder deren Erbe ***GameEventUnitAction***. So kann zum Beispiel ein ***GameEventUnitGroupAction*** Ereignis erstellt werden als Erbe von ***GameEvent***, welches eine Gruppe von Spieleinheiten als Ziel einer Aktion hat.

5.3 Erste Anwendungsschritte vor dem Spielablauf

In diesem Abschnitt wird exemplarisch erklärt, wie die Computergegner mittels der gAImes-Engine in den Spielverlauf eingebunden werden können.

5.3.1 Instanzen der gAImes-Engine

Jeder Spieler, der von der gAImes-Engine gesteuert werden soll, benötigt eine eigene Instanz. Neutrale Einheiten, wie zum Beispiel Ambients(Lebewesen, die der optischen Aufbereitung des Spiel dienen) benötigen ebenfalls eine Instanz von der gAImes-Engine, auch wenn augenscheinlich wenig Intelligenz benötig wird.

5.3.2 Erster Schritt: Spieler registrieren

```
public void gAimes-Engine::updateGameEvents(GameEventContainer);
GameEventRegisterEngine(GameEventType, UUID);
```

Bevor der Spielentwickler die gAImes-Engine nutzen kann, muss er zunächst einige Initialisierungsschritte durchführen. Dabei ist zu beachten, dass für jeden computergesteuerten Spieler eine Instanz der gAImes-Engine benötigt wird. Dies gilt auch für neutrale Spieler.
Im Kapitel „Erstellen der Spieleinheiten“5.2.2 wurde bereits erläutert, wie man einem Spieler zugehörige Einheiten erstellt und der entsprechenden gAImes-Engine Instanz zuteilt. Dieser Schritt wird für jede *eigene* Einheit durchgeführt. Es müssen aber auch alle *fremden* Einheiten mitgeteilt werden. Wurden diese Schritte für jede gAImes-Engine durchgeführt, muss man sie

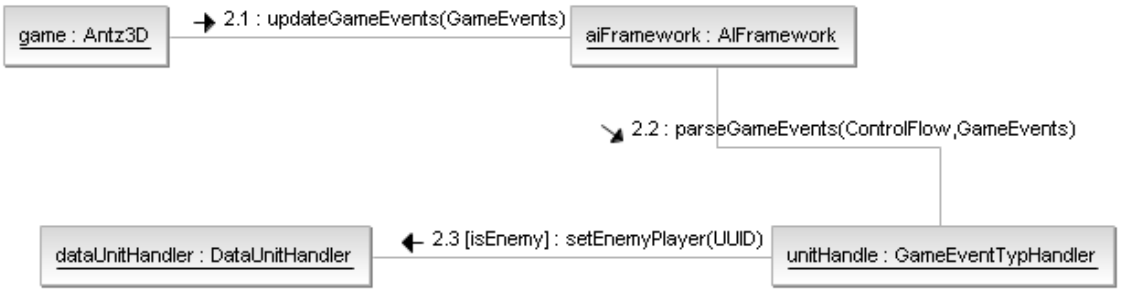


Abbildung 6: Die Spieler werden registriert

sich gegenseitig bekannt machen. Dazu ruft man die Methode ***updateGameEvents*** auf. Die Instanz der gAImes-Engine Klasse ist in diesem Fall die, an der die andere gAImes-Engine registriert werden sollen.

Der Parameter ***GameEventContainer*** kann mehrere Spielereignisse aufnehmen. In diesem Beispiel benötigen wir nur einen. Dennoch muss man einen parametrisierten Vector anlegen und diesen übergeben (***Vector<GameEvent>***). Das Spielereignis, welches wir in den GameEventContainer eintragen wollen ist vom Typ ***GameEventRegisterEngine***.
Der Parameter ***GameEventType*** sieht wie folgt aus:

```
public enum GameEventType
{
  enemy, friend, neutral,
  kill, build,
  action
}
```

Der zweite Parameter mit der **UUID** ist die ID der gAImes-Engine, die angemeldet werden soll.

Die Registrierung ist aus zwei Gründen notwendig. Durch das Registrieren aller Einheiten bei einer gAImes-Engine, kann die entsprechende KI die gegnerischen Einheiten erkennen und angemessen reagieren. Die Zuordnung der eigenen Einheiten zu einer gAImes-Engine ist für die interne Verarbeitung notwendig.

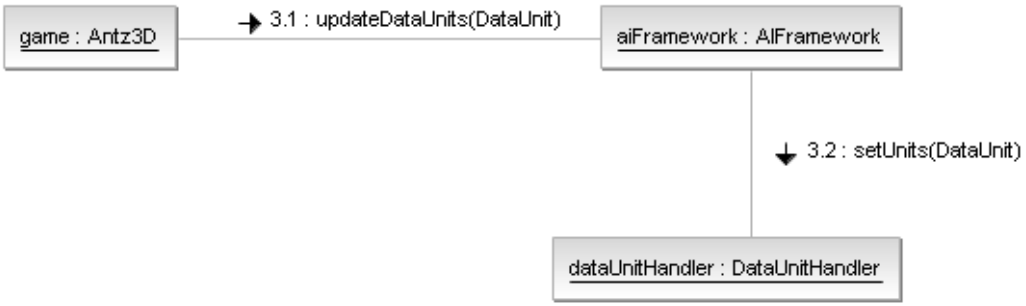


Abbildung 7: Übermittlung des Startzustands

5.3.3 Zweiter Schritt: Startzustand

Als zweiten Schritt überträgt das Spiel den Startzustand, in dem es ausgeführt werden soll. Dies beinhaltet die Übermittlung der Teile der Karte, die aktuell sichtbar sind und die Startaufstellung der Spieleinheiten im Spiel. Repräsentiert werden die Spieleinheiten als Instanzen der Klasse **DataUnit**. Ein Beispiel ist in Kapitel 5.2.2 zu finden.

5.3.4 Dritter Schritt: Spielbeginn

```
public abstract void process();
```

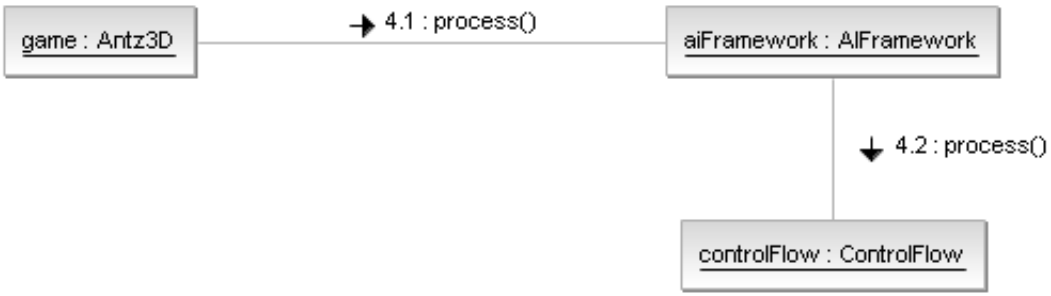


Abbildung 8: Das Spiel beginnt

Im letzten Schritt wird die Abarbeitung in der gAImes-Engine durch den Aufruf der Methode ***process*** gestartet. Diese ruft ihrerseits wiederrum den

Kontrollfluss der gAImes-Engine auf. Der Kontrollfluss aktualisiert alle rundenabhängig arbeitenden Komponenten innerhalb der gAImes-Engie, wie beispielsweise den Befehlsspeicher, welcher Befehle von Logikeinheiten(AIUnit) verwaltet. Danach startet der Kontrollfluss die Einheitenausführung(UnitExecutionControl) und somit die Logikprozesse in der gAImes-Engine. Der Ablauf ist für alle Anwender der gAImes-Engine im Detail nicht wichtig. Wissen sollte man als Anwender lediglich, dass alle Einheiten in einer Runde nacheinander ausgeführt werden.

5.4 Spielablauf

Der generelle Spielablauf eines 2D-RTS läuft in der Regel wie folgt ab. Einer Spielschleife wird solange wiederholt, bis das Spiel beendet, oder eine Pause eingelegt wird. Innerhalb der Schleife laufen alle wichtigen Ereignisse zusammen. Zunächst werden die Benutzereingaben eingelesen und verifiziert. Dann werden die Computerspieler berechnet und zu guter Letzt die Spielszene dargestellt. Exemplarisch sieht das so aus:

```
while(gameIsRunning)
{
  readPlayerInput
  calculateCPU_Players
  renderGrafic
}
```

5.4.1 Übertragung der KI-Aktionen

```
public boolean aiEvent(GameEventUnitAction unitAction);
```

Die gAImes-Engine führt in jeder Spielrunde jedes ihrer eingeladenen KI-Skripte einmal aus. Das jeweilige KI-Skript führt jeweils seine Berechnungen durch und schickt Aktionen an das Spiel. Im Spiel kommen diese Aktionen als ***GameEventUnitAction*** Objekte in der Methode ***aiEvent*** aus dem Interface ***AIListener*** an. Diese Methode liest die Aktion ein und prüft ihre Zulässigkeit in der aktuellen Spielumgebung. Ist die Aktion zulässig wird die Spielumgebung entsprechend verändert und Ereignisse erzeugt, welche alle gAImes-Engines über diese Veränderung informieren. Solch eine Aktion ist zum Beispiel eine Spieleinheitenbewegung. Es muss geprüft, werden ob das Feld auf das die Spieleinheit laufen möchte nicht von einer anderen Einheit besetzt wird oder unbetretbares Terrain, wie beispielsweise ein Berg, ist. Ist das Feld frei wird die Spieleinheit verschoben und alle gAImes-Engines

darüber informiert. Diese Information erhalten sie in diesem Fall mittels der Übertragung der DataUnit die

5.4.2 Aktualisieren der gAImes-Engine

```
public void updateDataUnits(Vector<DataUnit> dataUnits);
public void updateGameEvents(GameEventContainer gameEvents);
public void process();
```

Um die gAImes-Engine über alle Änderungen an der Spielumgebung zu informieren nutzt der Spielentwickler zwei Methoden in der Schnittstelle ***AI-Engine***.
Wenn sich aufgrund einer Aktion die Attribute einer Spieleinheit, wie beispielweise die Lebenspunkte, ändern wird dies in der entsprechenden DataUnit vermerkt. Mittels der Methode ***updateDataUnits*** werden alle Änderungen an den Spieleinheiten übermittelt indem eine Liste mit deren DataUnits übergeben wird. Mit dieser DataUnits ersetzt die gAImes-Engine diejenigen die bereits in ihr gespeichert sind.
Aktionen des KI-Skripts können zudem Ereignisse in der Spielumgebung verursachen. Ein solches Ereignis ist zum Beispiel das Schießen, welches der obigen Spieleinheit seine Lebenspunkte abgezogen hat. Im Spielverlauf kann sich ebenfalls das Spielfeld verändert, indem sich das Terrain ändert oder es sich im Spielgeschehen durch erforschen erweitert. Diese Informationen werden der gAImes-Engine mittels der Methode ***updateGameEvents*** in der Datenstruktur GameEventContainer mitgeteilt.
Die Datentypenrepräsentation von Spielereignissen und dem Spielfeld sind in verschiedenen Spielen zumeist sehr unterschiedlich. Die gAImes-Engine arbeitet jedoch auf immer gleichen Datentypen. Damit die Spielentwickler in ihren Spielen nicht mit den Datentypen der gAImes-Engine arbeiten müssen, kann der Datentyp ***GameEventContainer*** genutzt werden. Dieser kann so umgeschrieben werden das er den Datentyp des Spiels übernimmt und diesen in den Datentyp der gAImes-Engine umwandelt. Wenn der Spielentwickler sein Spielfeld beispielsweise mittels eines großen ***Arrays*** repräsentiert, könnte er den ***GameEventContainer*** erweitern, so dass dieser dieses ***Array*** in die ***Map*** Struktur der gAImes-Engine umwandelt.

6 Die gAImes-Engine

In diesem Kapitel wird die Architektur der gAImes-Engine gezeigt. Zudem werden die Klassen beschrieben die Verarbeitung und Verwaltung der Datenstrukturen übernehmen. Ebenso werden die internen Abläufe der gAImes-Engine aufgezeigt.

6.1 Die Architektur

In den kommenden drei Abschnitten wird der Aufbau der gAImes-Engine mit Hilfe von Klassendiagrammen visualisiert.

6.1.1 Kommunikation

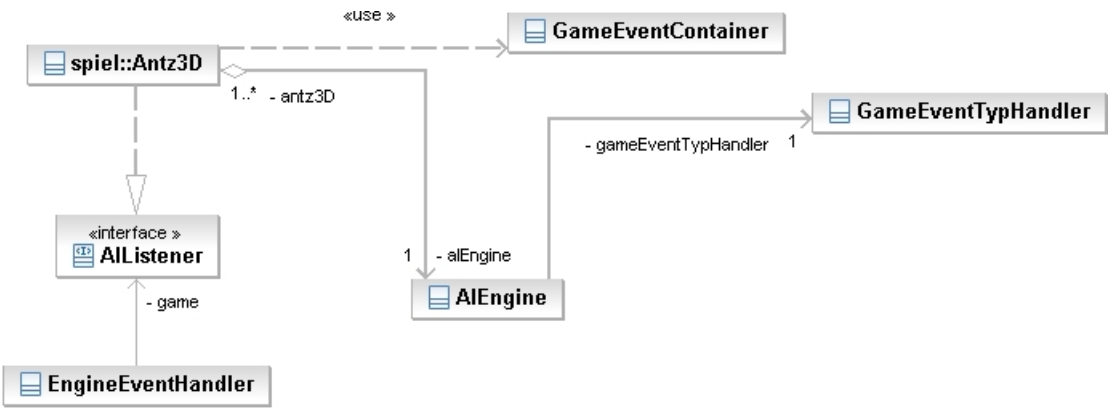


Abbildung 9: Dieses Klassendiagramm gibt eine Übersicht über die für die Kommunikation verantwortlichen Klassen an.

Die Abbildung 9 zeigt die Verbindung zwischen einem Spiel und der gAImes-Engine. Das Spiel implementiert das Interface ***AIListener***. Dieses Interface wird von der Klasse ***EngineEventHandler*** genutzt um die Spielaktionen der KI-Skripte zu übermitteln. Das Spiel erzeugt innerhalb Instanzen der ***AIEngine*** Klasse, welche die Schnittstelle zur gAImes-Engine ist. Über diese übermittelt es, neben ***DataUnits***, einen ***GameEventContainer***. Der ***GameEventContainer*** wird dann in der gAImes-Engine mittels der Klasse ***GameEventTypHandler*** abgearbeitet.

6.1.2 Verwaltung

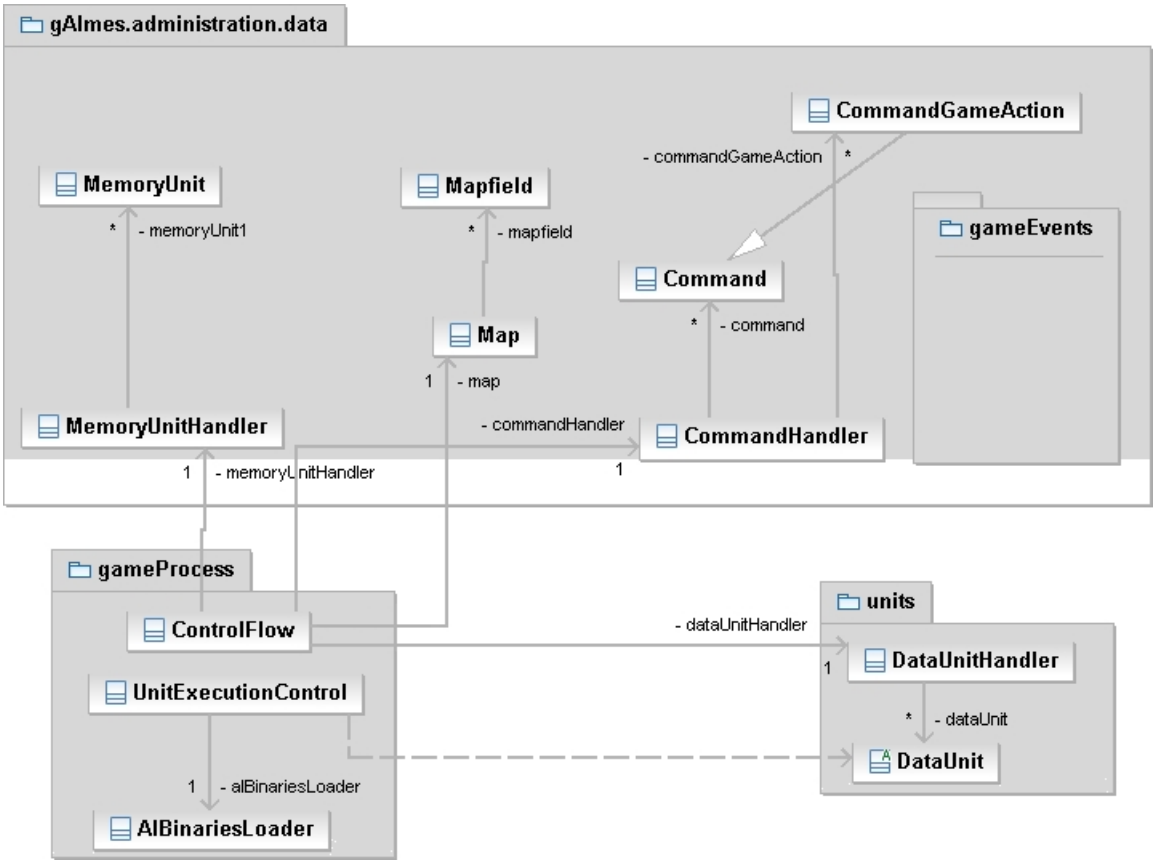


Abbildung 10: Dieses Klassendiagramm gibt eine Übersicht über die Klassen die die Verwaltung und die Verarbeitung in der gAImes-Engine übernehmen.

Die Abbildung 10 zeigt den Teil der Architektur der gAImes-Engine der sich mit der Verwaltung der Daten und der Ausführung der KI-Skripte beschäftigt. Dieser ist in die drei **Packages *units***, ***data*** und ***gameProcess*** unterteilt. Das **Package *units*** beinhaltet die Spieleinheitenrepräsentation und die Klassen für deren Speicherung und Verarbeitung.
Das **Package *data*** beinhaltet die Datenstrukturen für die Gedächtniseinträge, die Befehle und die Kartenfelder, sowie deren Verwaltungs- und Verarbeitungsklassen. Zudem beinhaltet es das **Package *gameEvents***, welches die Spielereignisse beinhaltet und dessen Inhalt im Abschnitt refsec:gaimes-engine-verwaltung-spielereignisse auf Seite 45 näher beschrieben wird.
Das dritte **Package *gameProcess*** beinhaltet die Klasse die die Prozesse in der gAImes-Engine steuern. Diese sind die Klasse ***ControlFlow***, welche alle

Verbindungen zu den Verwaltungs- und Verarbeitungsklassen hält, sowie die Klasse ***UnitExecutionControl***, welche die KI-Skripte der Spieleinheiten ausführt und die Klasse ***AIBinariesLoader***, welche die KI-Skripte von der Festplatte lädt.

6.1.3 Logik

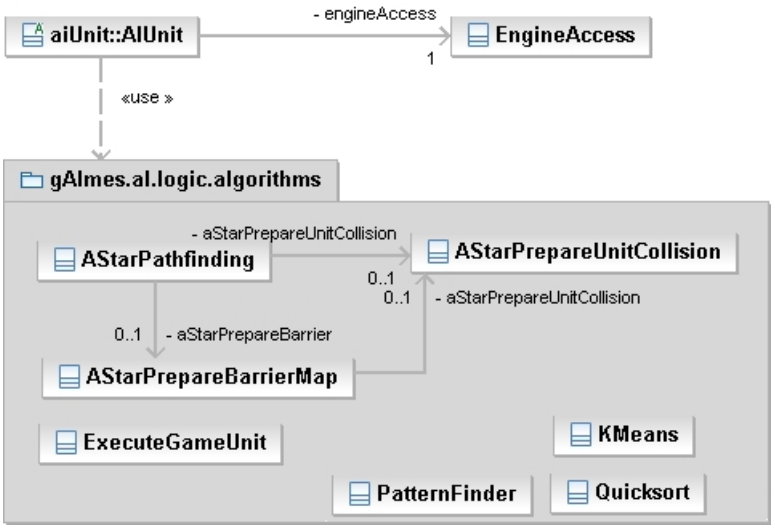


Abbildung 11: Dieses Klassendiagramm gibt eine Übersicht über die Klassen die für den KI-Entwickler wichtig sind.

Das Klassendiagramm in Abbildung reffig:gaimes-logic zeigt die Beziehungen zwischen den Klassen die für die Logikentwicklung von Bedeutung sind an. Zentraler Klasse dieser Beziehungen ist die Klasse ***AIUnit***, welche die Basisklasse aller KI-Skripte ist. Sie nutzt die Klasse ***EngineAccess*** um Zugriff auf die Daten in der gAImes-Engine zu bekommen. Um die Entwicklung der Logik in den KI-Skripten zu unterstützen gibt es das **Package *algorithms***, welches Implementationen von Algorithmen, die von allgemeinem Nutzen sind, beinhaltet.

6.2 Verwaltung der Datenstrukturen

6.2.1 Spieleinheiten

Beschrieben wird die Klasse DataUnitHandler.java
Für Informationen zu DataUnits siehe Abschnitt 4.2 auf Seite 20.

Für die Verwaltung der ***DataUnit*** Objekte, der Repräsentation der Spieleinheiten, wurde die Klasse ***DataUnitHandler*** entwickelt. Diese Klasse speichert alle ***DataUnit*** Objekte, die der gAImes-Engine vom Spiel geschickt werden. Zudem werden in dieser Klasse alle anderen am Spiel beteiligten gAImes-Engines(Mitspieler) registriert. Die ***DataUnit*** Objekte werden der

jeweiligen gAImes-Engine zugeordnet, in Listen indiziert. Dabei wird nach den drei Kategorien BEFREUNDET, VERFEINDET und NEUTRAL unterschieden. Neutrale gAImes-Engines sind solche die am Spielgeschehen nicht aktiv teilnehmen, wie beispielsweise herumlaufende Zivilisten. Die Klasse EngineAccess (siehe Abschnitt /refsec:kiEinheiten, Seite /pagerefsec:kiEinheiten) greift bei Aufruf ihrer Methoden get...Units() auf diesen Handler(Anwender) zu.

6.2.2 Karte

Beschrieben werden die Klasse Map.java und die Klasse Mapfield.java

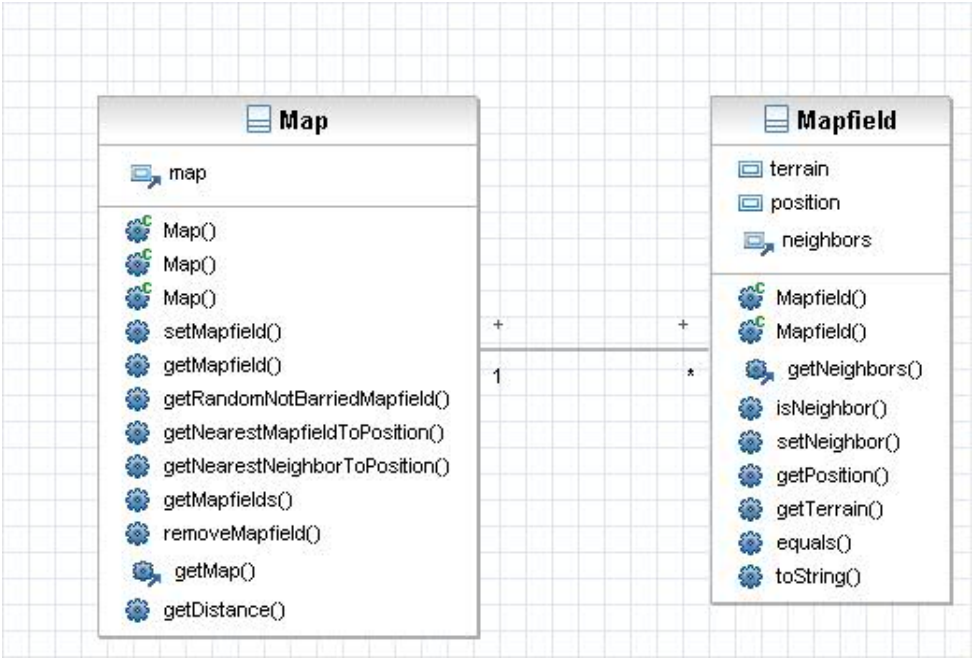


Abbildung 12: Relation zwischen Karte und den Kartenfeldern

Die Karte ist eine Datenstruktur zur Repräsentation des Spielfelds. Sie enthält das Spielfeld in Form eines Graphen. Dabei sind die einzelnen Kartenfelder die Knoten und die Nachbarschaftsbeziehungen die Kanten. Jedes Kartenfeld kennt alle seine direkten Nachbarfelder und wird, über seine Position auf dem Spiel referenziert, abgespeichert. So können Informationen sehr effizient abgerufen werden, ohne den ganzen Graphen durchsuchen zu müssen.
Die Kartenfelder haben zudem die Variable ***int terrain***, welche die Art von Beschaffenheit angeben soll aus dem es besteht. Diese Angabe kann als Gewichtung für die Wegberechnung genutzt werden. So könnte einer um so höherer Terrainwert ein schwieriger zu durchlaufendes Spielfeld sein.

6.2.3 Spielereignisse

Beschrieben werden die Klasse GameEvent.java und ihre Erben GameEventUnitAction.java, GameEventRegisterEngine.java und GameEventUnitHandle.java

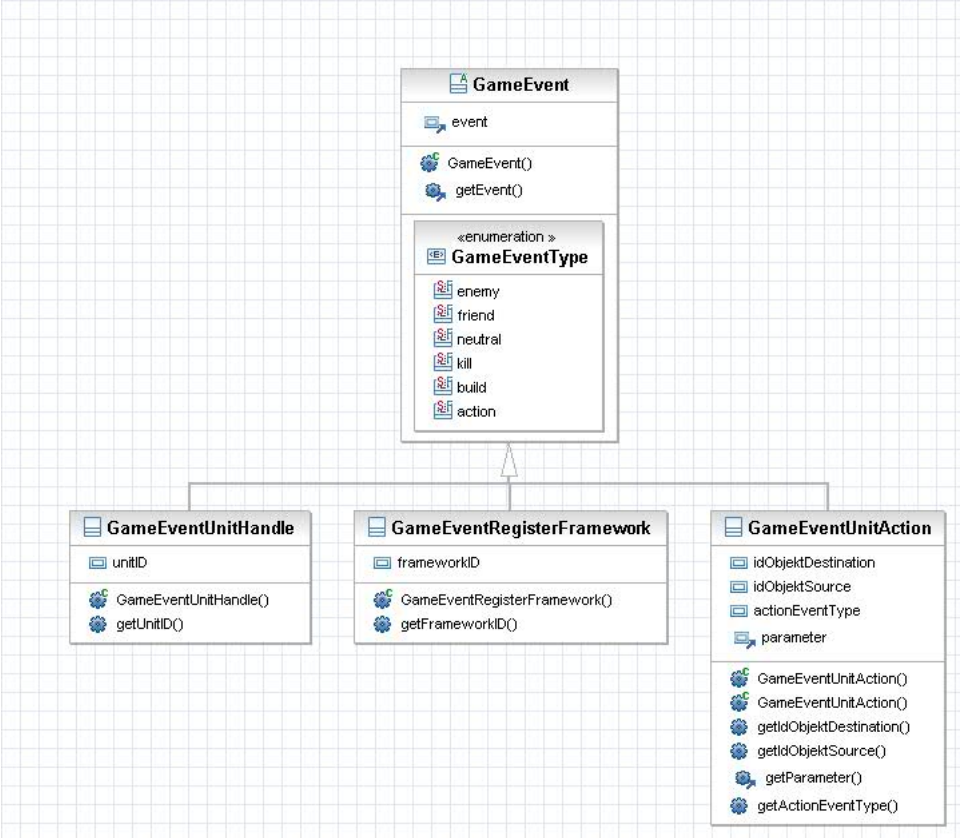


Abbildung 13: Vererbungshierarchy der Klasse GameEvent

Im Spiel gibt es festgelegte Kategorien von Ereignissen die das Spiel an die gAImes-Engine verschickt. Zum einen ein ***GameEventRegisterEngine*** Ereignis, welches einen Gegenspieler an der gAImes-Engine registriert. Zum anderen ein ***GameEventUnitHandle*** Ereignis. Mit Hilfe dieses Ereignisses teilt das Spiel der gAImes-Engine mit, ob eine Einheit auf dem Spielfeld gestorben ist oder der Bau einer eigene Einheit abgeschlossen wurde. Zum letzten werden die Aktionen die auf dem Spielfeld geschehen in Form von ***GameEventUnitAction*** Ereignissen mitgeteilt. In diesen steht welche Einheit gegen welche andere Einheit eine Aktion ausgeführt hat und welche Parameter diese Aktion hat. Mit diesem Ereignis kann beispielsweise mitgeteilt werden, ob einer der eigenen Soldaten von einem anderen Soldaten angegriffen wird oder wie viel Erz eine der Sammlereinheiten gerade abbaut.

Beschrieben wird die Klasse GameEventUnitActionHandler.java
Ein Ereignisse vom Typ ***GameEventUnitAction*** wird durch den Ereignishandler verwaltet. Diese schickt das Spiel an die gAImes-Engine um Ereignisse wie beispielsweise Einheit **A feuert auf Einheit B** mitzuteilen. Entsprechend den oben beschriebenen Attributen kann sich ein KI-Entwickler Ereignisse gefiltert nach Tag, Quelle und Ziel zurückgeben lassen. Während eines Spiels werden im Normalfall sehr viel Ereignisse an die gAImes-Engine verschickt. Aufgrund dessen werden alte Ereignisse, dass sind in diesem Fall Ereignisse aus vorherigen Runden, nach einer zuvor angegebenen Anzahl von vergangenen Runden automatisch gelöscht, um Speicher einzusparen.

6.2.4 Befehle

Beschrieben werden die Klasse Command.java und ihr Erbe CommandGameAction.java
Die KI-Skripte kommunizieren über ein Nachrichtensystem, welches Befehle aufnimmt. So besitzt jede Einheit die Möglichkeit ihre Befehle abzurufen und eigene Befehle abzusetzen. Zu diesem Zweck gibt es die Klasse ***Command***. Sie gibt als abstrakte Klasse allen Befehlen vor einen **Tag(Kennzeichnung)** zu tragen. Mittels dieses Attributs soll die Menge an Befehlen sortiert und gefiltert werden.
Ein Befehl vom Typ ***CommandGameAction*** ist eine Erweiterung eines ***Command***. Er wird gebraucht um Befehle an das Spiel senden zu können. Die gAImes-Engine verlangt dies, da Befehle an das Spiel Aktionen sind, die zumindest ein Ziel und eine Quelle haben müssen. Zudem hat die Klasse noch ein Attribut, welches die Werte der Aktion übermitteln soll. So zum Beispiel bei einem Feuerbefehl, wieviel Schaden angerichtet wurde (siehe Abschnitt 4.5.1, Seite 25).
Beschrieben wird die Klasse CommandHandler.java
Befehle werden im ***CommandHandler*** entsprechend den Spielrunden in denen sie abgegeben wurden gespeichert. Einem entsprechend zu konfigurierenden Parameter nach werden veraltete Befehle automatisch nach einer vorgegebenen Rundenanzahl gelöscht. Dies ist nötig um den Arbeitsspeicher nicht zu überfüllen. Die Befehle können entsprechend der Spielrunde in der sie ausgesprochen wurden und der Einheit an die sie gerichtet sind abgerufen werden.

6.2.5 Gedächtniseinträge

Beschrieben wird die Klasse MemoryUnit.java

Dem KI-Entwickler wird die Möglichkeit gegeben sich eigene Vorgehensweisen, Ereignisse, Aktionen etc. zu speichern. Hierfür nutzt er eine ***MemoryUnit***. Eine MemoryUnit verlangt einzig die Angabe einer Identifikationsnummer und eines Tags. Der Tag erlaubt es bestimmte MemoryUnits zu gruppieren und auszufiltern. So zum Beispiels alle MemoryUnits in denen der KI-Entwickler sich gespeichert hat welche Angriffsbefehle er an seine Soldaten geschickt hat. Mithilfe ihrer Identifikationsnummer wird eine MemoryUnit eindeutig identifiziert. Dies ermöglicht es den KI-Skripten Daten die schon berechnet wurden auszutauschen, indem sie sich die Identifikationsnummern von MemoryUnits mittels des Befehlssystems zuschicken. Hat eine Einheit die gegnerischen Soldaten bereits gruppiert mittels eines Cluster-Algorithmus, kann sie ihr Ergebnis in einer MemoryUnit abspeichern und dessen Identifikationsnummer an andere Einheiten, in einem Befehl verpackt, schicken.

Beschrieben wird die Klasse MemoryUnitHandler.java
Die gAImes-Engine hat eine große Gedächtnisverwaltungseinheit, welche alle Gedächtniseinträge speichert. Wenn ein KI-Skript seine Gedächtniseinträge abruft wird ihm eine neue Instanz der Klasse MemoryUnitHandler übergeben, welche nur noch die ihm zugehörigen Gedächtniseinträge enthält. Über Methoden lassen sich einzelne Gedächtniseinträge mittels ihrer ID oder gefiltert nach ihrem Tag zurückgeben.

6.3 Verarbeitung der Datenstrukturen

6.3.1 Spielereignisse

Beschrieben wird die Klasse GameEventTypHandler.java
Die Ereignisse die das Spiel an die gAImes-Engine schickt haben verschiedene Formate. Dabei handelt es sich um Daten von den Typen **Mapfield** und **GameEvent**. Der **GameEventTypHandler** überprüft das Format der Ereignisse und gibt sie an die entsprechenden Verwaltungseinheiten weiter. Wenn es sich um Ereignisse betreffend der Aktualisierung der Kartenfelder handelt werden diese entsprechend an die Karte weitergegeben. Sollte es sich um ein Spielereignis handeln, wird der Typ geprüft. Bei einem **GameEventRegisterEngine** wird die angegebene Engine beim **DataUnitHandler** als freundlich, feindlich oder neutral angemeldet. Bei einem GameEventUnitHandle wird die angegeben **DataUnit** aus dem **DataUnitHandler** gelöscht. Sollte es sich um eine **GameEventUnitAction** handeln, wird diese an den **GameEventUnitActionHandler**(siehe Abschnitt 6.2.3) weitergegeben.
Beschrieben wird die Klasse EngineEventHandler.java

Die Befehle der KI-Skripte, wie beispielsweise Bewegungen, müssen vor dem Versand an das Spiel in Spielereignisse umgewandelt werden. Dies übernimmt die Klasse ***EngineEventHandler***. Sie wandelt die Befehle mit Format **CommandGameAction** in das Format ***GameEventUnitAction*** um. Dies macht Sinn, da sich der Spielentwickler im Umgang mit der gAImes-Engine mit möglichst wenigen Datenstrukturen auseinander setzten soll. Er kennt in diesem Fall das Format ***GameEventUnitAction*** schon für Ereignisse an, welche an die gAImes-Engine gerichtet sind.

6.3.2 Ausführungskontrolle

Beschrieben wird die Klasse UnitExecutionControl.java
Wenn die gAImes-Engine vom Spiel das Signal bekommt ihre KI-Skripte auszuführen wird hierfür die Ausführungskontrolle tätig. Sie ist dafür verantwortlich die passenden KI-Skripte zu den gegebenen Spieleinheiten zu finden. Hat sie diese Zuordung herstellen können lädt sie die KI-Sripte mit den zugeordneten Spieleinheiten(DataUnit) und führt sie aus.

Die Zuordnung eines KI-Skripts zu einer DataUnit erfolgt mittels einer Zeichenkette die in jeder DataUnit angegeben ist. Diese Zeichenkette dient als Schlüssel der im Namen eines passenden KI-Skripts steht. Solch eine Zeichenkette könnte zum Beispiel **SoldatenCommander** sein. Die KI-Skripte sind in einer Liste gespeichert und werden der Reihe nach geprüft. Gibt es eine Übereinstimmung wird das KI-Skript mit der DataUnit geladen und gestartet. Folgend ein Pseudocodebeispiel:

```
aiScripts : AIList // LogicUnit
units     : UnitList // DataUnit

for(DataUnit curDataUnit : dataUnits) // Liste aller Einheiten
{
 for(AIUnit curAIScript : aiScripts) //Liste aller KI-Scripte
 {
   // Entsprechende KI zu Einheit gefunden
   if (curDataUnit.getAIType() == curAIScript.getAIType())
   {
     (1) curAIScript.loadUnit(curUnit); //Lädt aktuelle DataUnit in KI-Skript
     (2) curAIScript.doWork(); //Startet den Denkprozess für diese Einheit
   }
  }
}
```

Dieser Vorgang wird für jede Spielfigur in jeder Spielrunde einmal durchlaufen. Diese Vorgehensweise ist üblich in der Spielentwicklung und lehnt sich stark an die Entwicklung von Computerspielen an.

6.3.3 Kontrollfluss

Beschrieben wird die Klasse ControlFlow.java
Der Kontrollfluss ist das Zentrum der gAImes-Engine welches alle Referenzen auf die zur Spiellaufzeit relevanten Klassen behält. Er koordiniert zudem den Arbeitsablauf dieser. Zu den Referenzen die er hält gehören:

```
private UUID ownID;
private AIListener game;
private DataUnitHandler dataUnitHandler;
private UnitExecutionControl unitExecutionControl;
private CommandHandler commandHandler;
private Map map;
private GameEventUnitActionHandler actionHandler;
private EngineEventHandler engineEventHandler;
private MemoryUnitHandler memoryHandler;
```

Des weiteren behandelt der Kontrollfluss die Aktualisierung der Verwaltungsklassen während des Spielablaufs. So wird von ihm zum Beispiel die Klasse ***CommandHandler*** darüber informiert das eine neue Spielrunde begonnen hat, welche diese Information zur Speicherung der Historie von Befehlen braucht.

6.3.4 Laden der KI-Skripte

Beschrieben wird die Klasse aIBinariesLoader.java
Wenn die gAImes-Engine zu Beginn eines Spiels instanziert wird lädt sie alle ihr zur Verfügung gestellten KI-Skripte. Diese hat der Spielentwickler vor der Ausführung des Spiels in ein von der gAImes-Engine festgelegtes Verzeichnis kopiert. Die KI-Skripte liegen dort in Form von binären Javaklassen vom Typ ***.class**. Zum Einlesen dieser Dateien wird die Klasse ***aIBinariesLoader*** genutzt. Sie wird im Konstruktor der Klasse ***UnitExecutionControl*** aufgerufen. Die gefundenen Dateien werden mithilfe der Java Reflectionstechnology instanziert und in eine Liste eingelagert. Diese Liste wird von der ***UnitExecutionControl*** verwendet um das passende KI-Skripte zu einer Spieleinheit zu laden.

...

```
// Der Pfad zum Verzeichnis der KI-Skripte wird geladen.
URI uri = new URI(aIPath);
File aIUnitDIR = new File(uri);

// Listet alle Dateinamen auf die "AIUnit" im Namen
// und die Endung ".class" haben
String[] classfiles = aIUnitDIR.list(new FilenameFilter()
  {
    public boolean accept(File d, String name) {
      return name.contains("AIUnit") && name.endsWith(".class");
}});
...
// Instanzierung der KI-Skript Dateien die zuvor gefunden wurden.
for(String fileName : pureClassNames) {
  Class<?> classForFileName = Class.forName(loadClassPath + fileName);
  classInstances.add((AIUnit)classForFileName.newInstance());
}

return classInstances;
```

7 2D-RTS Spiel als Testumgebung

Um die gAImes-Engine testen und implementieren zu können, haben wir eine Testumgebung geschrieben. Sie ist aufgebaut, wie ein 2D-RTS Spiel und stellt ein Spielfeld in der Vogelperspektive dar, auf dem sich theoretisch beliebig viele kontrahierende oder alliierte Fraktionen befinden können. Das Spielfeld besteht aus einer Ebene, auf der sich die Spielfiguren bewegen können. Zusätzlich gibt es Gebirge, die als natürliche Hindernisse dienen und Ressourcenfeldern, die als Ressourcesammelpunkte von allen Fraktionen genutzt werden können. Getauft haben wir die Testumgebung **Antz3D**.

Spielfiguren werden zylidnrisch in unterschiedlichen Farben und Beschriftung dargestellt. Dabei ist der Körper des Zylinders in der Spielerfarbe zu sehen und die kreisförmige Oberseite mit einem Buchstaben versehen, der die Art der Einheit zeigt. Ein S steht für Soldier(Soldat) und ein H für Harvester(Sammler). Zusätzlich gibt es violette Kreise, die unparteiische, bewegliche Hindernisse darstellen und grüne Quadrate mit einem R für Resource(Ressource). Grüne Flächen sind begehbare Bodenareale und graue Flächen sind unpassierbare Berge. Abbildung 12 zeigt eine solche Szene. Die weißen Linien ausgehend von den Spieleinheiten, sind die jeweiligen anvisierten Ziele.

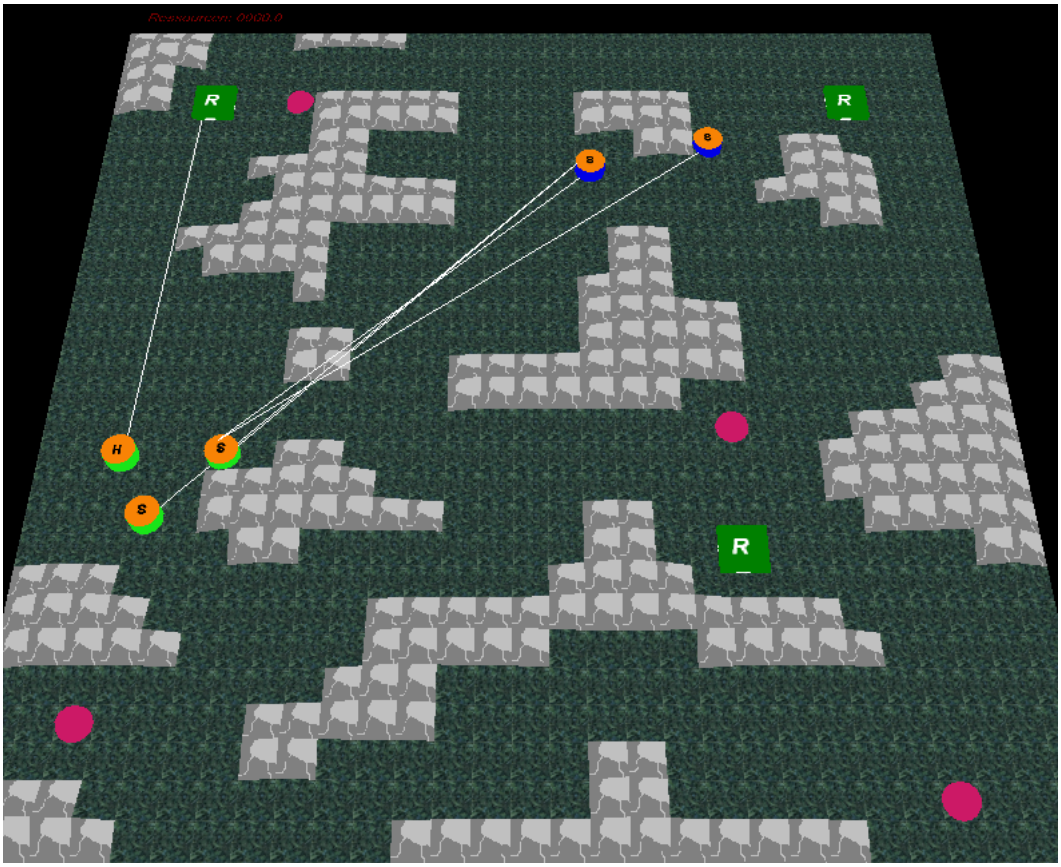


Abbildung 14: Ansicht des Spiels im Spielermodus

7.1 Technologie

Für das Spiel Antz3D setzen wir Java3D ein. Java3D bietet umfangreiche Mechanismen zur Darstellung von 3D Szenen an, dabei greift es auf die Technik von OpenGL zurück. Es existiert eine umfangreiche Dokumentation mit vielen Codebeispielen.
In aktuellen 3D-Engines wird häufig mit 3D-Objekten gearbeitet. Ein 3D-Objekt wird von der Grafikengine in einer 3D-Szene dargestellt. Diesen Vorgang nennt man eine Szene rendern. Dabei werden die Objekte unter Berücksichtigung des Blickwinkels und Lichtquellen berechnet. Aus mathematischer Sicht stellt sich die darzustellende Szene als Koordinatensystem mit drei Achsen dar. Jedes Objekt besitzt einen dreidimensionalen Vektor als Positionsangabe, ebenso wie der Blickpunkt und die Lichtquellen. Mit Hilfe von Translation, Rotation und Scherung werden die Objekte manipuliert.
Die 3D-Szene wird von einem Augpunkt, der mit einer Fernsehkamera zu

vergleichen ist, erstellt. Berechnet werden die Objekte vom weitesten zum nächsten Objekt. Dadurch können nahe Objekte dahinter liegende verdecken.

7.2 Architektur

Das Visualisierung Antz3D besteht hauptsächlich aus folgenden vier Klassen.

- Antz3D
- Initialzustand
- EnvironmentalCreation
- Antz3DGraphics

Die Hauptklasse ist ***Antz3D***. Sie bindet alle notwendigen Interfaces und Klassen ein, um ein Fenster in Windows zu zeigen. Anschließend wird ***EnvironmentalCreation*** geladen, um abhängig von den Konstanten ***Initialzustand*** eine Spielumgebung zu erzeugen. Zuletzt wird die 3D Visualisierung von der Klasse ***Antz3DGraphics*** angezeigt.

7.2.1 Antz3D

Antz3D ist die Hauptklasse der Visualisierung. Sie erbt von der Klasse ***Frame***, um alle notwendigen Eigenschaften zu erhalten, die man zur Fensterdarstellung in Windows benötigt. Zusätzlich implementiert sie das Interface ***Runnable***. Das ist notwendig, wenn man ein Thread erzeugen möchte. Diesen Thread verwendet Antz3D für die Spielschleife, die ununterbrochen ablaufen soll. Eine einfache Endlosschleife in der Klasse ***Antz3D*** würde nicht ausreichen, da sie sonst nicht mehr auf Maus-/ oder Tastatureingaben reagieren könnte. Die Spielschleife würde die gesamte Rechenzeit verbrauchen. Das Interface ***AIListener*** aus der gAImes-Engine wird hier ebenfalls implementiert. Den Zweck dieses Interfaces wird in Kapitel 5.2.1 erläutert.

7.2.2 EnvironmentalCreation

Die Klasse ***EnvironmentalCreation*** dient der Erstellung der Spielumgebung. Dazu gehört das erzeugen des Spielfeldes und der Spieleinheiten. Abhängig von den Parametern der Klasse ***Initialzustand*** aus dem nächsten Kapitel werden unterschiedliche Spielfelder erzeugt und die Anzahl, Art und Spielerzugehörigkeit der Spieleinheiten verändert. Die Startposition aller Spieleinheiten kann nach Bedarf vollständig zufällig auf dem Spielfeld erzeugt werden.

7.2.3 Initialzustand

Die Klasse ***Initialzustand*** enthält verschiedene Konstanten. Diese lassen sich schnell verändern und somit das Spiel gezielt und einfach modifizieren, ähnlich einer Konfigurationsdatei.

7.2.4 Antz3DGraphics

Die Klasse Antz3DGraphics beinhaltet alle Funktionen zur Darstellung der Spielszene mit Java3D. Dazu gehört das erzeugte Spielfeld sowie alle Spielfiguren.

Java 3D ist eine API, die auf der Hierarchie von Java Klassen aufbaut und dient als Schnittstelle zur dreidimensionalen Darstellung von Grafik in einem Rendersystem. Dem Programmierer werden stark abstrahierte Konstrukte an die Hand gegeben, um die Szenen zu erstellen oder zu manipulieren.
Java 3D arbeitet in einem eigenständigen Universum, in dem Objekte dreidimensional beliebig positioniert werden können. Die Details der Darstellung werden dabei verborgen, so dass der Programmierer sich nur um die Erstellung der Szene kümmern muss. Im Hintergrund arbeiten Optimierungsalgorithmen, die einen schnellen und flüssigen Ablauf gewährleisten.
Antz3DGraphics bindet dieses Rendersystem in das von ***Antz3D*** erzeuge Fenster ein.
Java3D arbeitet mit einer Baumstruktur. Jedes 3D-Objekt wird dabei an einen Pfad, ausgehend von der Wurzel, angehängt. Möchte man eine Szene mit zwei Würfeln erzeugen, so hängt man an die Wurzel eben diese zwei Objekte an. Jedem Objekt können weitere Objekte angefügt werden, die Eigenschaften, wie Material, Textur oder Transparenz enthalten.
Um ein Objekt in der Szene Bewegen zu können, bietet Java3D Verhaltensklassen an. Diese werden **Behavior** benannt. Diese Klassen werden in Antz3D genutzt, um die Spieleinheiten zu bewegen. Die Behaviorklassen werden nur dann von Java3D ausgeführt, wenn sie einen Auslöser besitzen, der gefeuert wurde. Diese Auslöser werden Trigger genannt. Für unsere Anwendung verwenden wir einen Trigger, der auf eine Veränderung des Objekts reagiert. Möchten wir die Position einer Spielfigur verändern, speichern wir die neue Position in dem entsprechenden Objekt. Dadurch wird der Trigger von Java3D ausgelöst und die Grafik verändert neu gezeichnet.

7.2.5 Initalisierung

Für Antz3D erstellen wir zunächst zwei Spieler, die gegeneinander Antreten sollen. Wir benötigen dafür zwei Instanzen der gAImes-Engine. Engine eins

wird einen Ressourcensammler und einen Soldaten bekommen, Engine zwei zunächst nur eine neutrale Einheit. Die Erstellung der Einheiten geschieht in der Klasse ***EnvironemntalCreation***. Die Anpassung des Szenarios geschieht über einen Parameter, der der Klasse ***Initialzustand*** übergeben wird. ***EnvironemtalCreation*** kann auf Grund des Parameters eine entsprechende Erstellungsmethode ausführen. Der folgende Code listet eine der Erstellungfunktionen aus dieser Klasse auf.

```
1   private HashMap<UUID, DataUnit> unitList;
2   private HashMap<UUID, AIEngine> interfaces;
3   private GameEventContainer gameEventContainer;

4   AIEngine ai1 = new AIEngine(this,
      new UUID(new Random().nextLong(),
      new Random().nextLong()));
5   Vector<Vector<DataUnit>> initUnits =
      getStartFriendlyUnitsAI3(ai1.getOwnEngineID());

6   AIEngine ai2 = new AIEngine(this,
      new UUID(new Random().nextLong(),
      new Random().nextLong()));
7   Vector<DataUnit> neutralUnits =
      getStartNeutralUnits(ai2.getOwnEngineID());

8   interfaces.put(ai1.getOwnEngineID(), ai1);
9   interfaces.put(ai2.getOwnEngineID(), ai2);

10  Vector<UUID> enginesToSend = new Vector<UUID>();
11  enginesToSend.add(ai2.getOwnEngineID());
12  registerEngines(GameEventType.neutral,
     sn1.getOwnEngineID(), enginesToSend);
13  enginesToSend = new Vector<UUID>();
14  enginesToSend.add(ai1.getOwnEngineID());
15  registerEngines(GameEventType.neutral,
              ai2.getOwnEngineID(), enginesToSend);

16  registerDataUnitsOnEngines(initUnits.get(1));
17  registerDataUnitsOnEngines(neutralUnits);
18  ai1.updateDataUnits(initUnits.get(0));
```

```
19  for(DataUnit unit: initUnits.get(1)) {
20 unitList.put(unit.getOwnID(), unit);
    }
21 for(DataUnit unit: neutralUnits) {
22     unitList.put(unit.getOwnID(), unit);
    }
```

In Zeile 1 und 2 wird jeweils eine HashMap für die Einheiten, die auf dem Spielfeld erscheinen sollen und für jede gAImes-Engine Instanz angelegt.

Zeile 3 wird die Ereignisse enthalten, welche zur Registrierung gesendet werden müssen.

In Zeile 4 wird eine Instanz der gAImes-Engine für Spieler eins angelegt. Der erste Parameter ist die Klasse Antz3D selbst. Diese beinhaltet das Spielfenster vom Typ Frame. Aus der Java Dokumentation kann man entnehmen, dass diese auch als Observer zum Laden von Bildern dienen kann. Dies wird benötigt, um im späteren Verlauf Texturen zu laden. Man könnte jedoch auch jede beliebige Observerklasse erstellen und übergeben. Der zweite Parameter erstellt eine eindeutige ***ID*** vom Typ ***UUID*** zur Identifikation der gAImes-Engine. Die nächste Zeile 5 lädt einen Vector von DataUnits, also den Spieleinheiten für Spieler eins. Dieser ist wiederum ein Vector, um beliebig viele Spieler laden zu können. Die Erstellung der Einheiten kommt der Übersicht halber aus einer Hilfsfunktion und erstellt lediglich die zwei oben erwähnten Einheiten für Spieler eins. Das Erstellen von Einheiten wird in Abschnitt 8 gezeigt.

Zeile 6 und 7 wiederholen das Erstellen von Spieleinheiten für Spieler 2.

In Zeile 8 und 9 werden die beiden gAImes-Engines in einen Vector gespeichert.

Zeile 10 und 11 speichern die ***UUID*** der gAImes-Engine, an der die andere Engine registriert werden soll. Dies wird in Zeile 13 und 14 für die Engine zwei wiederholt.

Zeile 12 führt die Registrierungsmethode aus. In Zeile 15 geschieht das Gleiche für Engine 2.

In Zeile 16 und 17 werden nun alle Einheiten, die das Spiel enthalten soll für beide gAImes-Engines registriert. Jetzt kennt jeder Spieler alle seine eigenen

Einheiten.

Zeile 18 erstellt noch eine Level 0 Einheit, also einen Kommandeur für Spieler eins. Die Zeilen 19-22 speichern alle erstellten Einheiten für das Spiel Antz3D ab, so dass bei einem Event über die Callback, die entsprechende Einheit gefunden werden kann. Dieser Schritt ist zur Verwaltung im Spiel und nicht nötig für die gAImes-Engine, wird jedoch empfohlen.

Jetzt sind alle Initialisierungschritte für zwei KI-Skript gesteuerte Spieler abgeschlossen.

7.3 Implementierung der Spieleinheiten

Um eine Spielfigur in Antz3D zu erzeugen, sollte der Programmierer zwei Dinge beachten. Jede Einheit soll bestimmte Eigenschaften, Fähigkeiten, individuelles Aussehn besitzen und für gAImes zugänglich sein.
Wie bereits in Kapitel 5.2.2 erläutert, muss jede Klasse, die eine Spieleinheit repräsentieren soll von **DataUnit** erben. Dies ist ausreichend für gAImes.
Für die Grafikdarstellung in Java3D und die Verwaltung der Einheiten in Antz3D benutzen wir vier weitere Klassen. Nachfolgend ein Beispiel für die Sammlereinheit. Die Vererbungshierarchie ist chronologisch von oben absteigend. Der Vollständigkeit halber ist die Klasse ***DataUnit*** mit aufgelistet, da von ihr geerbt wird.

- DataUnitL3Harvester
- DataUnitGraphicHarvesterSphere
- DataUnitBasics
- DataUnitGraphic
- DataUnit

7.3.1 DataUnitGraphic

DataUnitGraphic ist eine abstrakte Klasse, die von ***DataUnit*** erbt. Sie enthält grundlegende Grafikinformationen, die für jede Einheit gleich sind.

```
abstract public class DataUnitGraphic extends DataUnit
{
  protected Color3f playerColor;
  protected Vector3f destination = new Vector3f(0.0f,0.0f,0.0f);
```

```
  protected boolean showDest;

  public DataUnitGraphic(UUID ownID,
                         UUID engineID, Vector3f position)
  {
    super(ownID, engineID, position);
  }

  abstract public void createUnit(BranchGroup objRoot, Antz3D game);
  abstract public void triggerUnitBehavior();
  abstract public void showDest(boolean show);
  abstract public void setDestination(Vector3f dest);
}
```

7.3.2 DataUnitBasics

DataUnitBasic erbt von ***DataUnitGraphic*** und enthält grundlegende Informationen zu den Spieleinheiten. Hier sind zum Beispiel Trefferpunkte, Angriffskraft usw. gespeichert. Folgend nun ein Codeauszug aus der Klasse ***DataUnitBasics***.

```
public class DataUnitBasics extends DataUnitGraphic
{
  protected float healthPoints;
  protected float unitRadius;

  ...

  public DataUnitBasics(UUID ownID, UUID engineID,
    Vector3f position, float healthPoints, float unitRadius)
    {
    super(ownID, engineID, position);
    this.healthPoints = healthPoints;
    this.unitRadius = unitRadius;
    }
}

...

@Override
public void createUnit(BranchGroup objRoot, Antz3D game) {
```

```
}
```

7.3.3 DataUnitGraphicHarvesterSphere

```
public void createUnit(BranchGroup objRoot, Antz3D game);
private Node createBody(Antz3D game);
```

DataUnitGraphicHarvesterSphere ist eine Klasse für eine konkrete Spielfigurenklasse. Sie erbt von ***DataUnitBasics*** und überschreibt die oben gezeigte Methode ***createUnit***. In der Klasse wird das Aussehen der Spielfigur RessourcenSammler erstellt und ihr Verhalten geregelt.
Das Aussehen der Spielfigur wird in ***createUnit*** erstellt.

```
createUnit(BranchGroup objRoot, Antz3D game)
{
1 Transform3D T3D = new Transform3D();
2 T3D.set(new Vector3f(0.0f, 0.0f, 0.0f));
3 TransformGroup objTG = new TransformGroup(T3D);
4 TransformGroup objTGCaps = new TransformGroup();
5 objTGCaps.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
6 objRoot.addChild(objTGCaps);
7 objTGCaps.addChild(objTG);

8 objTG.addChild(createBody(game)); // Harvester erstellen

    ...
```

Der Übergabeparameter objRoot, ist der erwähnte Baum, an den alle Objekte angehängt werden. In Zeile 1-4 wird eine ***TransformationsGruppe*** erstellt, das ist die übliche Struktur, mit der Java3D arbeitet. Hier wird die Position des Objekts in dem virtuellen Universum festgelegt und als Ast definiert. Alle weiteren Objekte werden dort angehängt, wie in Zeile 5-7 , bis sie schließlich am Gesamtbaum hängen.
Das Verhalten dieser Spielfigur wird ebenfalls hier definiert. Dazu wird eine Verhaltensklasse zugewiesen, die jede Spieleinheitenklasse besitzt. So agieren alle gleichartigen Einheiten ähnlich, könnten aber individuell gesteuert werden.

```
  ...
  unitBehavior = new BehaviorUnitHarvesterMobile(objTGCaps, this);
  unitBehavior.setSchedulingBounds(new BoundingSphere());
  objRoot.addChild(unitBehavior);
}
```

In obigem Codeausschnitt in Zeile 8 wird die Methode ***createBody*** aufgerufen, diese erstellt die eigentliche Figur. Nachfolgend der Erstellungscode.

```
private Node createBody(Antz3D game)
{
// Erstelle Texture mit Hilfe des TextureLoaders
1 Texture texture = new TextureLoader(
    Initialzustand.getTexturepathHarvester(), game).getTexture();

// Transparenz erstellen
2 TransparencyAttributes ta = new TransparencyAttributes();
3 ta.setTransparencyMode(TransparencyAttributes.BLENDED);
4 ta.setTransparency (0.5f);

// Objektfarbe erstellen
5 ColoringAttributes ca = new ColoringAttributes();
6 ca.setColor(playerColor);
7 ca.setCapability(ColoringAttributes.ALLOW_COLOR_WRITE);

// Erstelle Appearance und weise ihr die Textur zu
8 Appearance appearance = new Appearance();
9 appearance.setTransparencyAttributes(ta);
10 appearance.setTexture(texture);
11 appearance.setColoringAttributes(ca);
12 appearance.setCapability(Appearance.ALLOW_COLORING_ATTRIBUTES_WRITE);
// Bastel eine Kugel und überziehe sie mit des Textur-Apearance
13 return new Sphere(0.5f, Sphere.GENERATE_TEXTURE_COORDS |
                        Sphere.GENERATE_NORMALS, 80, appearance);
}
```

In Zeile 1 wird aus einer Hilfsklasse der Texturpfad ermittelt. Texturen sind 2D Bilder, wie zum Beispiel JPEGs oder BMPs, die über eine 3D Objekt gelegt werden können, um ihm ein realistischeres Aussehen zu verpassen. In unserem Fall von **Antz3D** dienen die Texturen zur Unterscheidung der Einheiten, wie Soldat oder Sammler, da wir nur einfache, geometrische Formen, wie Kugel oder Würfel verwenden.
In Zeile 2-4 werden die Werte für die Transparenz der Textur gesetzt, damit die Spielerfarbe darunter zu erkennen ist. Besagte Spielerfarbe wird in Zeile 5-7 zugewiesen. Die Variable ***playerColor*** ist eine proteced Variable aus der beerbten Klasse ***DataUnitGraphic***.
In Zeile 8-12 werden die vorher erstellten Attribute-Klassen einer übergeordneten **Appearance-Class** zugewiesen, wie der Name bereits sagt, sammeln

sich hier alle Daten für das Erscheinungsbild des Objekts.
Letztendlich wird in Zeile 13 eine Kugel erstellt und mit ihrem neuen Erscheinungsbild versehen.

7.3.4 DataUnitL3Harvester

Die Klasse ***DataUnitL3Harvester*** erbt von ***DataUnitGraphicHarvesterSphere***. Hierbei handelt es sich lediglich um eine Namenskonvention, wie in Kapitel 5.2.2 erläutert.

7.4 Implementierung der Ereignisschnittstelle

Die Kommunikation in der heutigen Software funktioniert häufig ereignisgesteuert. Das grundlegende Prinzip von Ursache und Wirkung wird dabei verfolgt. Hierzu gibt es immer Sender und Empfänger. Dabei ist es theoretisch egal, wie viele Sender und Empfänger zum Einsatz kommen.
Die ereignisgesteuerte Kommunikation wird von Microsoft Windows und Java für die Steuerung des GUI und der damit verbundenen Benutzereingaben verwendet. Dabei horcht eine so genannte Listener Methode auf das Eintreffen bestimmter Ereignisse, zum Beispiel Mausereignisse (Mouse Events). Ein Event könnte ein Linksklick sein. Dieser wird vom System an eine Listenerschnittstelle gesendet. Hier können sich theoretisch alle Mouse Listener anmelden und mithören. Ist die Taste in einem Fenster betätigt worden, bekommt der Mouse Click Event als Empfänger, das Fenster, in dem geklickt wurde. Jetzt kann der Programmierer dieser Fensterapplikation entsprechend auf den Klick reagieren.
Nach dem gleichen Prinzip arbeitet Antz3D mit der gAImes-Engine zusammen. In Antz3D muss eine Listener Methode implementiert werden, die auf die Ereignisse der gAImes-Engine reagieren kann. Die entsprechende Methode wird durch die Implementierung des Interfaces ***AIListener*** erzwungen. Der Spiel Programmierer muss die Methode

```
public boolean aiEvent(GameEventUnitAction unitAction)
```

implementieren. Diese ist der Listener, der auf die gAImes-Ereignisse hören kann und ist wie folgt definiert.

```
public class GameEventUnitAction extends GameEvent
{
  private UUID idObjektDestination;
  private UUID idObjektSource;
  private String actionEventType;
  private Vector<String> parameter;
  ...
}
```

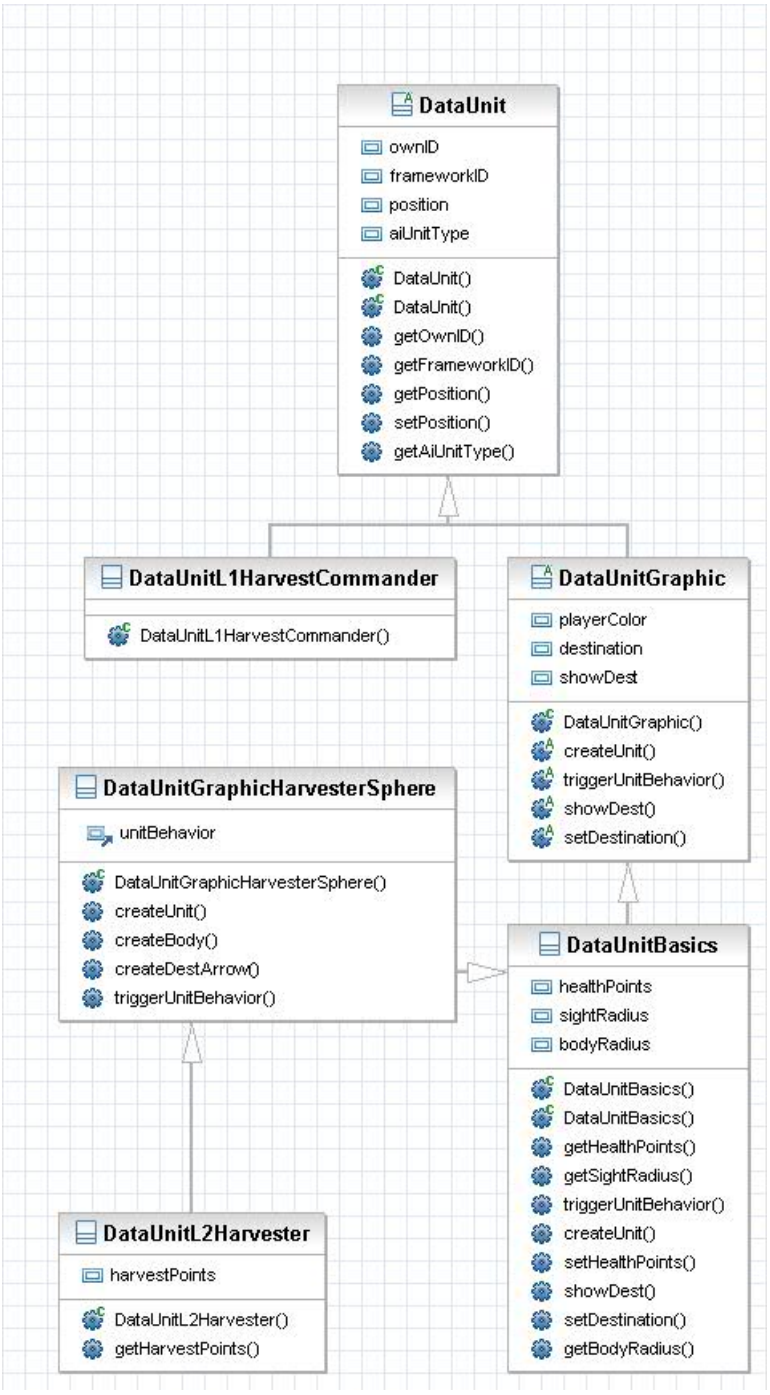


Abbildung 15: Vererbungshierarchie der Klasse der Harvester Spieleinheit

Die Variablen idObjektDestination und idObjektSource sind die ***UUIDs*** von Sender und Empfänger der Nachricht. ActionEventType vom Typ String

ist die eigentliche Nachricht selbst. Der Datentyp **String** ist notwendig, da die Ereignisse vom Spielentwickler definiert werden. Dies geschieht implizit über die Erstellung der Spieleinheiten. Aus allen Methoden, die mit der Annotation ***Executeable*** gekennzeichnet sind, macht die gAImes-Engine ein übertragbares Event. So können die KI-Skripte mit Methoden arbeiten, die sie nicht zum Erstellungszeitpunkt kennen müssen.

7.4.1 Was sendet die gAImes-Engine

Da die gAImes-Engine für intelligentes Verhalten der Spieleinheiten im Spiel zuständig ist, müssen die Befehle für die Spielfiguren dem Spiel mitgeteilt werden. Diese Informationen werden über das aiEvent Interface gesendet. Für jede Spielfigur, für die auf dem Spielfeld Entscheidungen getroffen werden, generiert die gAImes-Engine eine Nachricht, die an das aiEvent Listener Interface gesendet wird. Über den Parameter ***GameEventUnitAction*** werden mitgesendet: Absender, Empfänger, aufgetretenes Event(Befehl) und eventuelle Parameter.

7.4.2 Wie reagiert man auf die Ereignisse

Die aiEvent Schnittstelle wird von der gAImes-Engine als Callback-Methode ausgeführt. Sie sollte somit nicht Teil der Spielschleife sein, sondern wird je nach Bedarf ausgeführt. Um angemessen auf die eintreffenden Ereignisse reagieren zu können, ist es zu empfehlen, einige Konventionen einzuhalten. Der wichtigste Punkt, ist die Speicherung der Spieleinheiten. Sinnvoll ist eine Aufbewahrung, wie in Kapitel 5.2.2 erwähnt, denn über den Parameter ***GameEventUnitAction*** werden die UUIDs der Einheiten zur Identifikation gesendet.
Zu Beginn der Listener Methode sollte man die referenzierte Einheit heraussuchen. Angenommen die Einheitendaten liegen als DataUnit-Objekte in einer HashMap Namens unitList mit dem Schlüssel UUID vor, so könnte der Code wie folgt aussehen.

```
public boolean aiEvent(GameEventUnitAction unitAction)
{
  DataUnit destDataUnit =
    this.unitList.get(unitAction.getIdObjektDestination());
  DataUnit srcDataUnit =
    this.unitList.get(unitAction.getIdObjektSource());
  ...
```

Nachdem die beteiligten Einheiten vorliegen, kann nur das erzeugte **Event** bearbeitet werden. Es empfiehlt sich eine Verzweigung für alle auftretenden Ereignisse zu erstellen. Die einfachste Art ist eine **if...else** Anweisung zu erstellen. Für wenige Events ist dies eine gute und schnelle Lösung.

```
...
if( unitAction.getActionEventType().compareTo("move") == 0 )
{
  Vector<String> vs = unitAction.getParameter();
  ...
  return true;
}
else
 if ( unitAction.getActionEventType().compareTo("attack") == 0 )
...
```

Bei vielen Events wird das Konstrukt allerdings sehr unübersichtlich, vor allem, wenn viel Quellcode in den einzelnen **if...else** Blöcken steht.

Eine übersichtliche Lösung für das Problem ist, jedes Ereignis in einer eigenen Klasse zu bearbeiten. Allerdings bedarf dies erheblich mehr Schreibaufwand. Das Prinzip ist nicht neu und wird sowohl von Java, als auch von Microsoft, seit der MFC Klassensammlung verwendet.
Die eingetretenen Events werden ebenfalls, analog zu obigem Codebeispiel sortiert. Trifft ein Event zu, wird eine neue Klasse instanziert und mit den entsprechenden Parametern ausgeführt. Angenommen wir haben eine Klasse **Move**, dann könnte der Code so aussehen.

```
...
  if( unitAction.getActionEventType().compareTo("move") == 0 )
  {
    Move move = new Move(destDataUnit, srcDataUnit, unitAction);
    return move.do();
  }
  else
   if ( unitAction.getActionEventType().compareTo("attack") == 0 )
  ...
```

Die vollständige Verarbeitung ist in die Klasse ***Move*** ausgelagert worden. So bleibt die Methode ***aiEvent*** übersichtlich. Diese Technik haben wir in Antz3D auf Grund der geringen Quantität der auftretenden Events nicht verwendet.

7.4.3 Verarbeiten der Events

Angenommen es tritt ein **move** Event ein, wie im direkt vorangegangenem Kapitlel beschrieben, soll die Spielfigur entsprechend reagieren. Dazu müssen eventuell Werte, wie Lebenspunkte, Position, oder Munition verändert werden. Hier ein vollständiges Beispiel aus Antz3D. Es trifft ein move Event ein und wird für die entsprechenden Einheit verarbeitet.

```
if( unitAction.getActionEventType().compareTo("move") == 0 )
{
  1 Vector<String> vs = unitAction.getParameter();
  2 Float f1 = new Float(vs.elementAt(0));
  3 Float f2 = new Float(vs.elementAt(1));
  4 Float f3 = new Float(vs.elementAt(2));
  5 Vector3f desiredPosition = new Vector3f(f1,f2,f3);

  6 for(DataUnit dataUnit: this.unitList.values())
    {
  7   if(dataUnit.getPosition().equals(desiredPosition))
      {
  8     System.out.println("Game: Event Move Denyed -
                Position Blocked By Unit " + dataUnit);
  9     return false;
      }
    }
  10 srcDataUnit.setPosition(desiredPosition);

     return true;
}
```

In Zeile 1-5 werden die Parameter für den **move** Befehl extrahiert. Der **move** Befehl besteht aus drei Koordinaten x,y,z, zu denen sich die Spieleinheit bewegen soll. Zur Verarbeitung werden diese drei Koordinaten wieder in einen Vector3f gespeichert.
Zeile 6 ist eine Schleife, die durch die komplette HashMap mit DataUnits, also den Spieleinheiten, läuft und in Zeile 7 prüft, ob die gewünschte Zielposition in der Variable ***desiredPosition*** bereits durch eine andere Einheit belegt ist. Ist dies der Fall, kann die Suche mit Zeile 9 beendet werden. Ist die gewünschte Position noch frei, wird in Zeile 10 die neue Position zugewiesen. Dabei verändern wir die Positon der Spielfigur in dem Datenbestand von Antz3D. Am Ende der Runde werden diese Daten der gAImes-Engine mitgeteilt.

8 Implementierte KI-Skripte

In diesem Kapitel soll anhand der Implementierung von KI-Skripten gezeigt werden, wie KI-Skripte die Funktionen der gAImes-Engine nutzen können, um mit einem Spiel zu interagieren. Hierbei wird die Komplexität der KI-Skripte stufenweise höher. Sie verwenden in jeder Stufe mehr und mehr der von der gAImes-Engine zur Verfügung gestellte Konstrukte und Algorithmen. So wird erst die Implementierung eines Zivilisten beschrieben, welcher wahrlos auf dem Spielfeld umherläuft. Daraufhin wird die Implementation eines Ressourcensammlers beschrieben, welcher sich gezielt auf dem Spielfeld bewegt und mit Ressourcenobjekten interagiert. Zuletzt werden Implementationen von Soldaten beschrieben. Diese gruppieren sich strategisch, schießen aufeinander und reagieren auf Schussereignisse.

8.1 Zivilist

Beschrieben wird die Klasse AIUnitL1Zivilist

Mit der Implementierung eines Zivilisten wird gezeigt, wie ein KI-Skript im Spiel eine Spieleinheit bewegen kann. Das verwendete KI-Skript enthält nur eine grundlegende Implementierung. Das Spielfeld wird bezüglich den Bewegungsmöglichkeiten einer Spieleinheit untersucht. Dabei wird nur das Terrain untersucht, andere Spieleinheiten sind für den Zivillisten nicht von Bedeutung. Er soll lediglich ein bewegliches Hinderniss darstellen und läuft auf zufälligem Pfade umher und behindern so eventuell andere Spieleinheiten. Damit die zufällige Wegfindung des Zivillisten nicht auf zwei Felder, hin und her, beschränkt bleibt, sondern sich Pfade quer über das Spielfeld ergeben, wird das zufällig nächst gewählte Feld mit einer Wahrscheinlichkeit versehen. Es ist wahrscheinlicher, dass ein Feld gewählt wird, das sich bereits in Laufrichtung befindet.

Um in einer Spielrunde wissen zu können, aus welcher Richtung der Zivillist in der vorherigen Runde kam, speichert er diese Information im Gedächtnis mit Hilfe der gAImes-Engine.

```
MemoryUnitHandler memoryHandler = this.engineAccess.getMemories();
Vector<MemoryUnit> memories = memoryHandler.getMemoryUnitsByTag("lastMove");
MemoryUnit memory = memories.firstElement();
MemoryUnitCommand prevDirectionMemory = (MemoryUnitCommand)memory;
Vector3f prevPosition = prevDirectionMemory.getGoal();
```

Daraufhin filtert er die Spielkarte (siehe Abschnitt 9.1, Seite 88) nach freien Spielfeldern in seiner direkten Umgebung.

```
// Ausfiltern aller unpassierbaren Spielfelder aus der Karte.
Map barrierPreparedMap = AStarPrepareBarrierMap.prepareMap(
                    engineAccess.getMap());

// Aussortieren der eigenen Einheit aus der Liste aller Einheiten.
Vector<DataUnit> preparedUnitList = engineAccess.getAllUnits();
DataUnit tmpOwnUnit = null;
for(DataUnit tmpUnit : preparedUnitList){
 if(tmpUnit.getOwnID().equals(ownUnit.getOwnID())){
   tmpOwnUnit = tmpUnit;
   break;
 }
}
preparedUnitList.remove(tmpOwnUnit);

// Ausfiltern aller Spielfelder auf denen Spieleinheiten stehen.
Map unitAndBarrierPreparedMap = AStarPrepareUnitCollision.
                    prepareMap(barrierPreparedMap, preparedUnitList);
// Ausgabe aller Spielfelder die um die Spieleinheit betreten werden können.
Vector<Vector3f> neighborMapfieldPositions = unitAndBarrierPreparedMap.
                    getMapfield(ownPosition).getNeighbors();
```

Die gefundenen Spielfelder werden so gewichtet, dass Spielfelder die sich in bereits eingeschlagener Laufrichtung befinden stärker gewichtet werden als die Spielfelder die sich hinter der Spieleinheit befinden. So wird gewährleistet, dass sich die Verrechnung mit einer Zufallszahl bei weniger gewichteten Spielfelder schwächer auswirkt. Das Spielfeld dessen berechneter Faktor am höchsten ausgefallen ist, wird das Ziel der nächsten Bewegung sein und an das Spiel gesandt.

```
Vector<String> parameter = new Vector<String>();
parameter.add(String.valueOf(goalPosition.x));
parameter.add(String.valueOf(goalPosition.y));
parameter.add(String.valueOf(goalPosition.z));
engineAccess.setCommand(new UUID(0,0), new CommandGameAction("move",
                    ownUnit.getOwnID(), ownUnit.getOwnID(), parameter));
```

8.2 Ressourcensammler

Ein Ressourcensammler in RTS-Spielen ist allgemein für die Infrastruktur zuständig. Darunter fällt hauptsächlich die Beschaffung von Ressourcen und das errichten einer Basis. Die Ressourcen gelten allgemein als Zahlungsmittel und sind je nach Spielumsetzung konkret als Gold, Holz, Erz, usw. betitelt. In der, in dieser Arbeit beschriebenen Visualisierung Antz3D, können Ressourcensammler zwei Befehle **move** und **harvest** verarbeiten, die sie von ihrem Kommandant bekommen. Der Befehl **move** gibt eine Position an, zu der sich der Ressourcensammler bewegen soll. Der Befehl **harvest** identifiziert mit Hilfe einer eindeutigen Nummer eine Ressource auf dem Spielfeld. Ein Sammler kann mit der Nummer die Position und Kapazität der Ressource ermitteln.
Der Ressourcensammler-Kommandant analysiert das Spielfeld und kann alle Ressourcenfelder finden. Diese bewertet er an Hand ihrer Kapazität und der Entfernung zu den Ressourcensammlern auf dem Spielfeld. Anschließend delegiert er entsprechende Befehle an die Einheiten auf dem Spielfeld.

8.2.1 Kommandant der Ressourcensammler

Beschrieben wird die Klasse AIUnitHarvestCommander

Ein Ressourcensammler-Kommandant arbeitet in jeder Spielrunde zwei Phasen ab.
In der ersten Phase erstellt der Kommandant eine Liste von allen eigenen Spieleinheiten und filtert diese mit Hilfe der Klasse **PatternFinder** (siehe Abschnitt 9.2, Seite 92 nach Spieleinheiten, die den **String Harvester** im Namen tragen.

```
// Suche alle eigenen Harvester
  for(DataUnit ownArmyDataUnit : this.engineAccess.getOwnUnits())
  {
    if(PatternFinder.foundMatch("Harvester",
      ownArmyDataUnit.getClass().getName()))
      {
       harvester.add(ownArmyDataUnit);
      }
  }
```

Wurden alle eigenen zur Verfügung stehenden Ressourcensammler gefunden, werden nach dem gleichen Schema alle Ressourcenfelder auf dem Spielfeld

ermittelt.

In der zweiten Phase prüft der Kommandant, ob er Gedächtniseinträge in der vorherigen Runde erstellt hat. Gedächtniseinträge können z.B. die Position enthalten, an die der Kommandant einen Ressourcensammler geschickt hat. Ist es notwendig, werden diese Einträge bearbeitet.

```
Vector<MemoryUnit> memories = this.engineAccess.getMemoryUnits();

// In dem Vector notProcessedHarvesterUnits werden
// zu Anfang alle Ressourcensammler gespeichert.
// Diese werden dann während der Verarbeitung wieder einzeln
// heraus genommen wenn sie schon befehligt wurden. So soll
// ausgeschlossen werden dass ein Harvester zweimal in einer Runde
// befehligt wird.

Vector<DataUnit> notProcessedHarvesterUnits = new Vector<DataUnit>();
notProcessedHarvesterUnits.addAll(harvester);


if(!notProcessedHarvesterUnits.isEmpty() && !memories.isEmpty()){

// Es wird eine Liste der Ressourcensammler zurückgegeben,
// welche schon befehligt wurden, und somit gelöscht werden können.

   notProcessedHarvesterUnits.removeAll(
      processMemories(memories, notProcessedHarvesterUnits,
      ressources));
}
```

In der Methode ***processMemories*** wird geprüft, ob die Ressourcensammler ihre alten Befehle schon ausgeführt haben. Hier ist konkret benannt, ob sie bei einem **Move-Befehl** das Ziel schon erreicht haben oder bei einem **Harvest-Befehl** die Ressource schon eingesammelt haben. Sollten die Ressourcensammler die Befehle noch nicht ausgeführt haben, werden sie aus der Liste der noch zu befehligenden Einheiten ***notProcessedHarvesterUnits*** entfernt.

```
for(MemoryUnit memory : memories) {
 if(memory.getMemoryType() == "move") {
```

```
  MemoryUnitCommand moveMemory = (MemoryUnitCommand)memory;
   DataUnit harvesterUnit = engineAccess.getUnit(moveMemory.getDataUnitID());
   if(harvesterUnit.getPosition().equals(moveMemory.getGoal())) {
     deleteMemoryUnits.add(moveMemory);
     processedUnits.add(harvesterUnit);
   }
  }
}
```

Die Ressourcensammler die noch keine Befehle haben, werden in der Methode ***processOwnLogic*** abgearbeitet. Hier wird geprüft, wie viele Ressourcen sich auf dem Spielfeld befinden. Ist es nur eine, werden alle Ressourcensammler zu dieser geschickt. Sind es mehrere, wird berechnet, welcher Ressourcensammler der jeweiligen Ressource am nächsten stehen. Diesem wird daraufhin befohlen die Ressource einzusammeln.

```
   createHarvestCommand(harvesterUnit, nearestRessource);
```

Die Methode ***createHarvestCommand*** erzeugt einen Harvest-Befehl für den jeweiligen Ressourcensammler und speichert einen entsprechenden Vermerk im Gedächtnis des Kommandanten.

```
private void createHarvestCommand(DataUnit harvester,
   DataUnit resource)
{
  CommandActionOnUnit harvestCommand =
    new CommandActionOnUnit("harvest", resource.getOwnID());
  this.engineAccess.setCommand(harvester.getOwnID(), harvestCommand);
  this.engineAccess.setMemoryUnit(new MemoryUnitCommand("harvest",
    harvester.getOwnID(), null, ressource.getOwnID()));
}
```

Wurden alle Ressourcensammler abgearbeitet, enden die Arbeit des Kommandanten.

8.2.2 Ressourcensammler

Beschrieben wird die Klasse AIUnitHarvester

Jeder Ressourcensammler wird pro Spielrunde einmal ausgeführt.
Zuerst prüft er, ob neue Befehle vom Kommandanten empfangen wurden. Neue Befehle werden älteren vorgezogen, da sich die Spielsituation jede Runde ändern kann und die Einheit entsprechend neu reagieren können soll. Ein

alter Befehl würde die Einheit eventuell an eine Position führen, die bereits keine Ressourcen mehr enthält.

```
Vector<Command> commands = this.engineAccess.getCommands();
Vector<MemoryUnit> memories = this.engineAccess.getMemoryUnits();

if(!commands.isEmpty()) {
  processCommands(dataUnit, commands);
}else if(!memories.isEmpty()) {
  processMemories(dataUnit, memories);
}
```

In der Methode ***processCommands(dataUnit, commands)*** werden die neuen Befehle abgearbeitet. Es gibt nur zwei Typen von Befehlen, **Harvest-** und **Move-Befehle**. Im Falle eines **Harvest-Befehls** wird geprüft, ob sich der Harvester neben der Ressource befindet. Wenn es so ist, schickt er mittels der Methode ***sendHarvestCommands*** die Aktion **Ressource einsammeln** an das Spiel. Wenn es nicht so ist, bewegt er sich zum Ziel. In beiden Fällen merkt er sich, was er diese Runde beabsichtigte.

```
if(actualCommand.getCommandType().compareTo("harvest")==0)
{
  System.out.println("Harvester: Command Harvest");
  CommandActionOnUnit commandHarvest =
    (CommandActionOnUnit)actualCommand;
  DataUnit ressource =
    this.engineAccess.getUnit(commandHarvest.getGoalID());
  Vector3f ressourcePosition = ressource.getPosition();

// Wenn der Harvester auf der Ressource steht
   if(dataUnit.getPosition().equals(ressourcePosition))
   {
     // Sammle solange es das Spiel zulässt.
     this.sendHarvestCommands(dataUnit,
         ressource, ownHarvestPoints);
     }else{
     // Gehe zur Ressource.
     this.sendMoveCommands(dataUnit, ressourcePosition);
   }
   // Speichere dein Vorhaben.
   this.engineAccess.setMemoryUnit (
```

```
      new MemoryUnitCommand("harvest",
         dataUnit.getOwnID(), ressourcePosition,
         ressource.getOwnID()));
}
```

Die Methode ***sendHarvestCommands*** schickt solange **Ressource einsammeln** an das Spiel, bis dieses mit einem ***false*** die Aktion beendet.

```
do
{
  parameter = new Vector<String>();
  parameter.add(String.valueOf(harvestPoints));
} while(this.engineAccess.setCommand(
new UUID(0,0),
createCommand("harvest", ownUnit.getOwnID(),
ressource.getOwnID(),parameter)));
```

Die Methode ***sendMoveCommands*** schickt solange die Aktion **Laufe zu Position** an das Spiel, bis dieses mit einem ***false*** die Aktion beendet. Den Weg berechnet sie mittels des A-Stern Algorithmus.

```
Vector<Mapfield> moveRoute =
 AStarPrepareUnitCollision.findPathInBarrierMap(
 map, harvesterPosition, ressourcePosition, allGameUnits);

if(moveRoute != null)
{
  Vector<String> parameter;
  for(Mapfield mapfield : moveRoute)
  {
    parameter = new Vector<String>();
    parameter.add(String.valueOf(mapfield.getPosition().x));
    parameter.add(String.valueOf(mapfield.getPosition().y));
    parameter.add(String.valueOf(mapfield.getPosition().z));
    if(!this.engineAccess.setCommand(new UUID(0,0),
        createCommand("move", dataUnit.getOwnID(),
        dataUnit.getOwnID(), parameter)))
      break;
  }
}
```

8.3 Soldaten

Mit der folgenden Implementierung des Soldaten erweitert sich der Umfang der Funktionalitäten, bereits verwendet für Zivillist und Ressourcensammler, um die Reaktion auf Spielereignisse. Der Soldat kann feindliche Spieler attackieren und vernichten. Ebenso, wie die Angabe einer Ressource beim Ressourcensammler, bekommt der Soldat Befehle für Angriffsziele von seinem Kommandanten. Für **Antz3D** gibt es zwei Implementierungen für Soldaten und zwei für Soldaten-Kommandanten. Für den Soldat existiert der **einfache Soldat** und der **Soldat mit Gruppenzugehörigkeit**. Für die Kommandanten gibt es die Strategien **Direktangriff** und **eigene Soldaten gruppieren**.

Ein **einfacher Soldat** reagiert nur auf Befehle seines Kommandanten und kann nicht eigenständig handeln, sobald einer seiner Kameraden oder er angegriffen wird. Er erhält Befehle vom Typ ***attack***, die ihn anweisen sich zu einer gegnerischen Einheit zu bewegen und auf diese zu schießen, sobald sie in seinem Aktionsradius ist. Es wird immer nur eine gegnerische Einheit als Ziel ausgewählt.

Der Soldaten-Kommandant mit der **Strategie Direktangriff** schickt jedem der ihm unterstellten Soldaten jede Runde einen Angriffsbefehl. Ein Angriffsbefehl enthält die ID der anzugreifenden Spieleinheit. Der Soldat bewegt sich solange auf diese Einheit zu, bis sie im Angriffsradius ist, dann attackiert er. Der Soldaten-Kommandant wählt immer den dem eigenen Soldaten räumlich nächst gelegenen gegnerischen Soldaten aus.

Ein **Soldat mit Gruppenzugehörigkeit** ist Teil einer Gruppe von Soldaten. Dieser sammelt sich mit seinen Gruppenmitgliedern an einem Punkt in der geografischen Mitte der Gruppe. Er bewegt sich bei einem Angriff immer räumlich nahe zu seinen Gruppenmitgliedern auf das Ziel zu. Hierbei richtet er sich nach den Werten zweier Konstanten. Die Erste legt fest wie weit er sich maximal von seinen Gruppenmitgliedern entfernen darf. Die Zweite gibt an zu wie vielen seiner Gruppenmitglieder er diese Mindestdistanz halten soll. So könnte diese beiden Konstanten dem Soldaten zum Beispiel vorgeben dass er sich maximal 3.0 Distanzeinheiten von den zwei ihm am nächsten gelegenen Gruppenmitgliedern entfernen darf. Das Angriffsziel eines **Soldaten mit Gruppenzugehörigkeit** ist eine gegnerische Gruppe, kein einzelner Soldat. Wurde die gegnerische Gruppe erreicht, suchen sich die Soldaten selbstständig ein geeignetes Angriffsziel in der gegnerischen Gruppe. Diesbezüglich prüft jeder Soldat ob ein eigener Soldat bereits auf einen Gegner

schießt. Sollte dies der Fall sein und der Gegner befindet sich ebenfalls im eigenen Aktionsradius, schießt er ebenfalls auf dessen Ziel. Ein Soldat reagiert somit selbstständig auf Ereignisse in seiner Umgebung ohne auf einen expliziten Befehl seines Kommandanten warten zu müssen. Die einzigen Informationen die er benötigt sind eine Liste seiner Gruppenmitglieder und eine Liste der Mitglieder der gegnerischen Gruppe.

Der Soldaten-Kommandant mit der **Strategie eigene Soldaten gruppieren** erstellt Gruppen der Gegner und der eigenen Soldaten. Zuerst werden die gegnerischen Soldaten zu Gruppen zusammengefasst. Anschließend wird die Anzahl an Gruppen der eigenen Soldaten an die gegnerischen Gruppenstärken angepasst. Die gegnerischen Gruppen werden mittels des **k-means++** Algorithmus erstellt. Hierbei wird die Anzahl an gegnerischen Gruppen die gebildet werden soll solange angepasst, bis die durchschnittliche Distanz zwischen den Gruppenmitgliedern eine zuvor festgelegte Konstante unterschreitet. Die Mitglieder einer Gruppe gegnerischer Soldaten dürfen zum Beispiel räumlich nur vier Distanzeinheiten voneinander entfernt sein um als Gruppe erkannt zu werden.
Die eigenen Soldaten werden daraufhin ebenfalls mittels des **k-means++** Algorithmus gruppiert. Hierbei wird so gruppiert dass die Summe der möglichen Siege aus Gruppenkämpfen zwischen den einzelnen Gegnergruppen und den aufgebauten eignen Soldatengruppen über einem zuvor in einer Konstanten festgelegten Prozentsatz liegt. So soll die Gruppierung der eigenen Soldaten dann aufhören, wenn zum Beispiel 75% der Gruppenkämpfe gewonnen werden könnten. Ob ein Zweikampf gewonnen werden kann oder nicht wird Aufgrund der Differenz der Gruppenstärken entschieden. Die Möglichkeit festlegen zu können wie viele der Zweikämpfe mindestens zu gewinnen sind ermöglicht es eine Art „Guerilla“ Strategie zu verfolgen. So kann auf dem Schlachtfeld auch eine schwächere Gruppe an eigenen Soldaten eine überlegene Gegnergruppe angreifen um diese somit an sich zu binden. Dies ermöglicht es den anderen Gruppen an eigenen Soldaten ihre Zweikämpfe in Ruhe zu gewinnen. Sollte die Überlegenheit der gegnerischen Soldatengruppe zu hoch werden würde der Algorithmus einen neuen Gegner aussuchen, wodurch die eigene Soldatengruppe aus dem Zweikampf entfliehen würde.

8.3.1 Ein einfacher Soldat

Beschrieben wird die Klasse AIUnitSoldierSimple
In der ersten Phase wird überprüft ob neue Befehle angekommen sind. Wenn dies der Fall, ist werden diese auf ihren Typ hin geprüft und entsprechend verarbeitet.

Wenn es sich um einen ***attack*** Befehl handelt wird geprüft, ob sich die gegnerische Einheit im Aktionsradius befindet. Entsprechend wird angegriffen oder auf die Einheit zugelaufen.

```
if(inAttackRadius(dataUnit, enemy, (Float)ExecuteGameUnit.invokeMethod(
        dataUnit, "getSightRadius")))
{
  this.sendFireCommand(dataUnit, enemy);
}else{
  this.sendMoveCommand(dataUnit, enemy);
}
```

Handelt es sich um einen ***move*** Befehl, läuft die Einheit das angegebene Ziel zu.

Wurden keine neuen Befehle geschickt, werden die Gedächtniseinträge überprüft nach den Zielen der letzten Runde. Wurde ein Eintrag gefunden, wird geprüft, ob es sich um einen ***attack*** Befehl handelt und das Ziel noch existiert, woraufhin dieses angegriffen wird. Wenn es sich um einen ***move*** Befehl handelt, bewegt sich der Soldat zu der angegebene Position.

8.3.2 Kommandant der Soldaten: Strategie Direktangriff

Beschrieben wird die Klasse AIUnitSoldierCommander1on1

In dieser Implementierung sucht der Kommandant zu jeder seiner Einheiten die gegnerische Einheit die ihr am nächsten ist. Hierfür bestimmt er die Entfernung zwischen den Einheiten per Luftlinie .

```
// Speichert die aktuell niedrigste Distanz zu einem Gegner.
   double primaryGoalDistance = Double.MAX_VALUE;
// Aktuell nächstgelegenster Gegner.
   DataUnit primaryGoal = null;

for(DataUnit ownSoldier : ownSoldiers)
{
   for(DataUnit enemyUnit : allEnemyUnits)
   {
//   Berechnung der Luftlinie
     tmpGoalPosition = new Vector3f(enemyUnit.getPosition());
     tmpGoalPosition.sub(ownSoldier.getPosition());
     float tmpDistance = tmpGoalPosition.length();
```

```
      if(tmpDistance < primaryGoalDistance){
        primaryGoal = enemyUnit;
        primaryGoalDistance = tmpDistance;
      }
    }
// Senden einer Schussaktion an das Spiel
    createAttackCommand(ownSoldier, primaryGoal);
}
```

8.3.3 Ein Soldat mit Gruppenzugehörigkeit

Beschrieben wird die Klasse AIUnitSoldierGroupmover

Die Konstante ***groupmembersDistanceTo*** gibt die Distanz vor die sich ein Soldat maximal von seinen Gruppenmitgliedern entfernen darf. Wenn er diese Distanz überschreitet bewegt er sich automatisch in Richtung des nächstgelegenen Gruppenmitglieds. Die Konstante ***groupmembersBeNearTo*** legt fest zu wie vielen Gruppenmitgliedern der Soldat den Maximalabstand halten soll.

```
int groupmembersBeNearTo = 2;
float groupmembersDistanceTo = 3.5f;
```

Zu Anfang seiner Ausführung prüft der Soldat ob er neue Befehle von seinem Kommandanten erhalten hat. Sollte dies der Fall sein überschreibt er seine alten Befehle im Gedächtnis mit den neuen. Als Befehle bekommt er von seinem Kommandanten eine Liste seiner neuen Gruppenmitglieder und eine Liste der Mitglieder der gegnerischen Gruppe übersandt.
Daraufhin prüft er zuerst ob er sich in der Nähe seiner Gruppenmitglieder befindet. Hierfür wird die Liste seiner Gruppenmitglieder nach deren Distanz zum Soldaten aufsteigend sortiert. Die Sortierung erfolgt mit dem **Quicksort** Algorithmus. Nun wird entsprechend der Konstante ***groupmembersBeNearTo*** überprüft ob sich der Soldat nahe genug an seinen Gruppenmitgliedern befindet. Wenn sich der Soldat nicht im von der Konstanten ***groupmembersDistanceTo*** vorgegebenen Radius zu seinem Gruppenmitglied befindet, bewegt er sich in der Methode ***moveToGroupmember*** mittels des **A-Stern** Wegfindungsalgorithmus auf diesen zu.

```
// Sortierung der Gruppenmitglieder mit dem Quicksort
// Algorithmus
    Vector<DataUnit> sortedOwnGroupmember =
```

```
   GameUnitSorter.sortGameUnitsByDistance(owngroup,
                                          ownUnit.getPosition());

// Abbruchbedingung: Alle Einheiten geprüft oder über der
// Konstante die die Mindestanzahl an nahen Einheiten angibt.
   for(int i=0; i < sortedOwnGroupmember.size() &&
       i < groupmembersBeNearTo; i++)
   {
//   Prüfen ob die Distanz zwischen der eigenen Einheit und dem
//   Gruppenmitglied kleiner ist als die Konstante.
     if( !inRadius(ownDataUnit, sortedOwnGroupmember.get(i),
                   groupmembersDistanceTo))
     {
       moveToGroupmember(sortedOwnGroupmember.get(i));
     }
   }
```

Wenn der Soldat die Kriterien der beiden oben genannte Konstante erfüllt und nahe dem Gegner ist, sucht er sich einen gegnerischen Soldaten aus auf den er schießt. Hierfür prüft er erst ob eines seiner Gruppenmitglieder auf einen gegnerischen Soldaten schießt. Dadurch dass der Soldat selbst auch auf diesen Gegner feuert, kann dieser schneller ausgeschaltet werden. Um herauszufinden ob eines seiner Gruppenmitglieder feuert, wird der ***GameEventUnitActionHandler*** genutzt. Dieser verwaltet alle Ereignisse die auf dem Spielfeld aufgetreten sind. Für jedes Gruppenmitglied wird geprüft ob dieses in der letzten Spielrunde überhaupt Ereignisse ausgelöst hat. Sollte dies der Fall sein wird geprüft ob sich ein „fire“ Ereignis darunter befunden hat. Sollte dies ebenfalls zutreffen wird die **ID** des Ziels abgefragt und geprüft ob sich dieses Ziel ebenfalls im Aktionsradius unseres Soldaten befindet. Wenn der gegnerische Soldat angegriffen werden kann, wird die ***DataUnit*** des gegnerischen Soldaten zurückgegeben. Mit dieser ***DataUnit*** kann dann die Aktion „Schießen“ an das Spiel geschickt werden.

```
GameEventUnitActionHandler eventHandler =
engineAccess.getGameEvents();

for(DataUnit groupmember : sortedOwnGroupmember){

  GameEventUnitActionHandler sourceEventHandler =
  eventHandler.filteredBySource(groupmember.getOwnID());
```

```
  GameEventUnitActionHandler fireEventHandler =
  sourceEventHandler.filteredByType("fire");

  Vector<GameEventUnitAction> fireEvents =
  fireEventHandler.getGameEventsCurrent();

  if(!fireevents.isEmpty())
  {

    UUID destUnitID = fireevents.firstElement().getIdObjektDestination();

    DataUnit destEnemyUnit = engineAccess.getUnit(destUnitID);

    if(destEnemyUnit != null)
    {

      if(inAttackRadius(destEnemyUnit, ownUnit,
        (Float)ExecuteGameUnit.invokeMethod(ownDataUnit,
                                            "getSightRadius")))
      {
        return destEnemyUnit;
      }
    }
  }
}
```

8.3.4 Kommandant der Soldaten: Strategie eigene Soldaten gruppieren

Beschrieben wird die Klasse AIUnitSoldierKommanderGroupOwnSoldier

Als Abbruchkriterium für die Gruppierung der Soldaten werden die folgenden beiden Konstanten verwendet.

```
float maxAvgGroupmemberDist = 3.5f;
float minWins = 0.7f;
```

Die Konstante ***maxAvgGroupmemberDist*** ist das Abbruchkriterium für die Gruppierung der gegnerischen Soldaten. Sie legt fest wie weit die Gruppenmitglieder einer gegnerischen Gruppe von einander entfernt seinen dürfen. Wird dieser Wert überschritten, so wird die Anzahl der Gruppen die der **k-means++** Algorithmus berechnen soll erhöht.
Die Konstante ***minWins*** gibt an wie viel Prozent der Zweikämpfe zwischen den gegnerischen Gruppen und den eigenen Gruppen mindestens gewonnen werden müssen. Sie dient als Abbruchkriterium für die Gruppierung der eigenen Soldaten.

```
//  Startheuristik: Wurzel der Anzahl an gegnerischen Soldaten.
    int enemyClusterCount = (int)Math.round(Math.sqrt(
                                enemySoldiers.size()));
```

Diese Heuristik gibt dem **k-means++** Algorithmus einen initialen Wert an Gruppen die er erstellen soll vor. Hierfür wird die Wurzel der Anzahl an gegnerischen Soldatengruppen gewählt.

In dieser folgenden Schleife werden die gegnerischen Soldaten gruppiert. Die Schleife bricht erst ab wenn die durchschnittliche Distanz zwischen den Gruppenmitgliedern kleiner ist als die zuvor festgelegte Konstante oder jeder Soldat eine Gruppe für sich ist.

```
do{
  maxAvgDist = 0;
  enemyGroups = KMeans.groupUnitsByKMeansPlusPlus(enemySoldiers,
                  enemyClusterCount);

  for(GameUnitGroup group : enemyGroups){
    if(group.getAvgDist() > maxAvgDist){
      maxAvgDist = group.getAvgDist();
```

```
    }
  }
  enemyClusterCount++;

}while(  maxAvgDist >= maxAvgGroupmemberDist &&
  enemyClusterCount <= enemySoldiers.size());
```

Der **k-means++** Algorithmus wird in der ersten Runde mit der Anzahl an gegnerischen Gruppen gestartet. Daraufhin wird für jede der Gruppen die erstellt wurden eine gegnerische Gruppe ausgewählt. Dies geschieht in der nachfolgend beschriebenen Methode ***getBestEnemyGroup***. Die eigenen Gruppen werden zuvor noch ihrer Gruppenstärke entsprechend aufsteigend sortiert, sodass die schwächsten Gruppen ihre Ziele zuerst festlegen können. Dies ist in sofern sinnvoll da gegnerische Gruppen die schon anvisiert wurden in der Methode ***getBestEnemyGroup*** als unattraktivere Ziele gewichtet werden. Sollte die Stärke der jeweilig ausgewählten gegnerischen Gruppe geringer sein als die der eigenen Soldatengruppe, wird davon ausgegangen das ein Zweikampf der beiden Gruppen von der eigenen Gruppe gewonnen wird. Wenn der Prozentsatz an gewonnen Zweikämpfen größer ist als der Mindestwert in der Konstanten ***minWins***, wird aus der Schleife ausgebrochen. Die anderen beiden Fälle in denen aus der Schleife gesprungen werden kann sind die Extremfälle wenn es nur noch eine große Gruppe oder viele aus nur noch einem Soldaten bestehende Gruppen gibt.

```
// Initiale Gruppenanzahl ist die Anzahl an gegnerischen Gruppen.
   int ownClusterCount = enemyGroups.size();
   float wins;
   do{
     wins = 0;
//   In dieser HashMap wird festgehalten welche Gegnergruppe als
//   Angriffziel aufgewählt wurde.
     attackCommands = new HashMap<GameUnitGroup, GameUnitGroup>();

     ownGroups = KMeans.groupUnitsByKMeansPlusPlus(ownSoldiers,
                  ownClusterCount);

//   Alle gegnerischen Gruppen werden initial als nicht anvisiert
//   gekennzeichnet.
     for(GameUnitGroup tmpEnemyGroup : enemyGroups){
       tmpEnemyGroup.setTarget(false);
     }
```

```
//   Die erstellten Gruppen werden so sortiert das die schwächsten
//   Gruppen zuerst ihre Ziele aussuchen können.
     ownGroups = GameUnitSorter.sortGameUnitGroupsByStrength(ownGroups);

     for(GameUnitGroup group : ownGroups){

     GameUnitGroup bestEnemyGroup =
       getBestEnemyGroup(group, enemyGroups);

//     Die gegnerische Gruppe wird als schon anvisiert gekennzeichnet.
       bestEnemyGroup.setTarget(true);

       attackCommands.put(group, bestEnemyGroup);
       if(group.getGroupStrength() >= bestEnemyGroup.getGroupStrength())
         wins++;
     }

     float winChance = wins/ownGroups.size();

     if(winChance < minWins){
//     Wenn Gewinnchance zu klein erhöhe Gruppenmitgliederanzahl.
       ownClusterCount--;
     }else{
//     Wenn die Gewinnchance groß genug ist bricht die Gruppierung ab.
       break;
     }

     if(ownClusterCount <= 0){
//     Wenn nur noch eine große Gruppe besteht.
       break;
     }else if(ownClusterCount >= ownSoldiers.size()){
//     Wenn Gruppenmitgliederanzahl bei 1.
       break;
     }
   }while(true);
```

Nachfolgend wird der Inhalt der Methode ***getBestEnemyGroup*** geschrieben. In dieser Methode wird für eine eigene Soldatengruppe eine geeignete gegnerische Soldatengruppe ausgewählt. Dies geschieht auf Basis der Distanz zu der jeweiligen Gegnergruppe und deren Gruppenstärke im Vergleich zur

eigenen Gruppenstärke. Im Ablauf wird zuerst die Gruppenstärke jeder gegnerischen Gruppe mittels der eigenen Gruppenstärke normiert. Die Funktion lautet: (Gegnerische Gruppenstärke - Eigene Gruppenstärke) / Eigene Gruppenstärke. Die hieraus folgenden Werte sind positiv wenn die Gruppenstärke der Gegnergruppe höher ist als die eigene und negativ wenn die eigene Gruppenstärke höher ist. Daraufhin werden die Distanzen zwischen den Gegnergruppen und der eigenen Gruppe normiert. Hierfür wird zuvor die größte Distanz einer gegnerischen Gruppe zur eigenen Gruppe ermittelt. Die Funktion zur Normierung lautet: Direkte Distanz / Maximale Distanz. Im letzten Schritt werden die beiden genormten Werte summiert und eine zusätzliches Gewicht addiert, wenn die derzeit betrachtete Gegnergruppe bereits von einer anderen eigenen Gruppe als Ziel ausgewählt wurde. Letztendlich ausgewählt wird die gegnerische Gruppe deren Summe aus normierter Gruppenstärke, normierter Distanz und Zusatzgewichtung am geringsten ist.

```
// Diese Datenstruktur speichert die Zuordnung der normierten
// Gruppenstärken zu den Gruppen.
   HashMap<GameUnitGroup, Float> normStrengths =
   new HashMap<GameUnitGroup, Float>();

// Berechung der Unterschiede zwischen der eigenen Gruppenstärke
// und der der einzelnen Gegnergruppen.
   for(GameUnitGroup tmpEnemyGroup : enemyGroups){

//   Normiert mit der eigenen Gruppenstärke.
     float normGroupStrength = (tmpEnemyGroup.getGroupStrength() -
       ownGroup.getGroupStrength()) / ownGroup.getGroupStrength();
     normStrengths.put(tmpEnemyGroup, normGroupStrength);
   }

// Aussuchen der maximalen Distanz zwischen den einzelnen
// Gegnergruppen und der eigenen Gruppe.
   float maxDistance = 0f;

   for(GameUnitGroup tmpEnemyGroup : enemyGroups){
//   Berechnen des einzelnen Abstands zwischen den Gruppenzentren.
     Vector3f tmpGoalPosition =
     new Vector3f(tmpEnemyGroup.getGroupCenter());
     tmpGoalPosition.sub(ownGroup.getGroupCenter());
     float tmpDistance = tmpGoalPosition.length();
```

```
      if(tmpDistance > maxDistance)
        maxDistance = tmpDistance;
    }

    float bestEnemyGroupNormSum = Float.MAX_VALUE;

    for(GameUnitGroup tmpEnemyGroup : enemyGroups){
      Vector3f tmpGoalPosition =
      new Vector3f(tmpEnemyGroup.getGroupCenter());
      tmpGoalPosition.sub(ownGroup.getGroupCenter());
      float tmpDistance = tmpGoalPosition.length();

//    Normiert mittels der Maximaldistanz.
      float normDistance = tmpDistance / maxDistance;

//    Addition der normierten Gruppenstärke und der
//    normierten Distanz.
      float normSum = normStrengths.get(tmpEnemyGroup)
                    + normDistance;

//    Addition eines Zusatzgewichts wenn gegnerische
//    Gruppe bereits Ziel einer anderen eigenen Gruppe ist.
      if(tmpEnemyGroup.isTarget())
        normSum += isTargetWeight;

//    Auswahl der kleinsten Summe.
      if(bestEnemyGroupNormSum >= normSum){
        bestEnemyGroup = tmpEnemyGroup;
        bestEnemyGroupNormSum = normSum;
      }
    }
```

8.3.5 Ein Beispiel

Die folgenden beiden Abbildungen 16 und 17 zeigen ein Spiel zweier KIs. Die Einheiten in rot sind **einfache Soldaten**. Sie werden von einem Kommandanten mit der **Strategie Direktangriff** befehligt. Die Einheiten in blau sind **Soldaten mit Gruppenzugehörigkeit** und werden von einem Kommandanten mit der **Strategie eigene Soldaten gruppieren** befehligt. Die farbigen Kreise um die Einheiten zeigen die Gruppierung mit dem **k-means++** Algorithmus an. Die Zahlen neben den Einheiten bezeichnen

ihre **Health Points**.

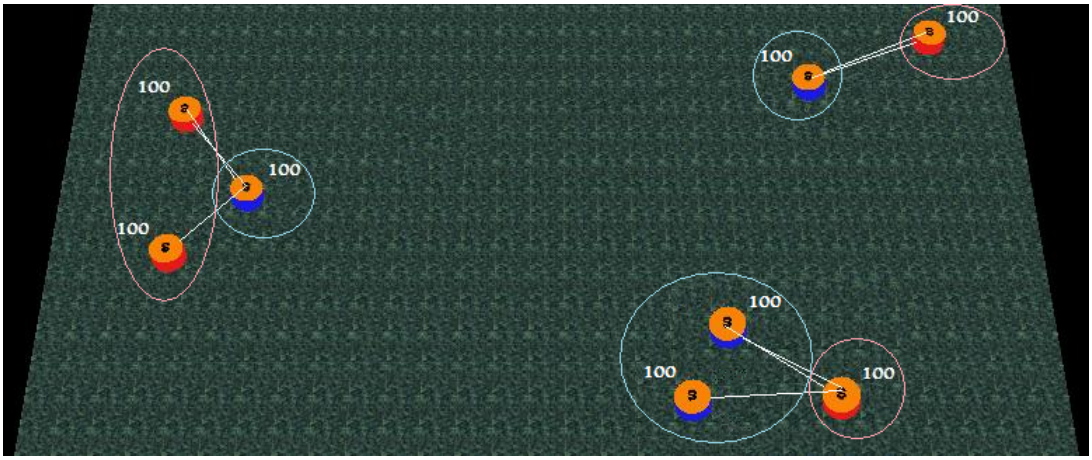


Abbildung 16: Aufstellung zu Beginn des Spiels

Auf dem ersten Bild ist zu sehen dass der Kommandant mit der **Strategie eigene Soldaten gruppieren** die gegnerischen Soldaten in drei Gruppen aufgeteilt hat. Darauf hin hat er seine eigenen Soldaten ebenfalls in drei Gruppen aufgeteilt. Der Soldat links ist trotz seiner Unterlegenheit gegenüber den beiden gegnerischen Soldaten dazu befehligt worden diese zu bekämpfen. Dies resultiert aus der geringen räumlichen Nähe in der er sich zu ihnen befindet.

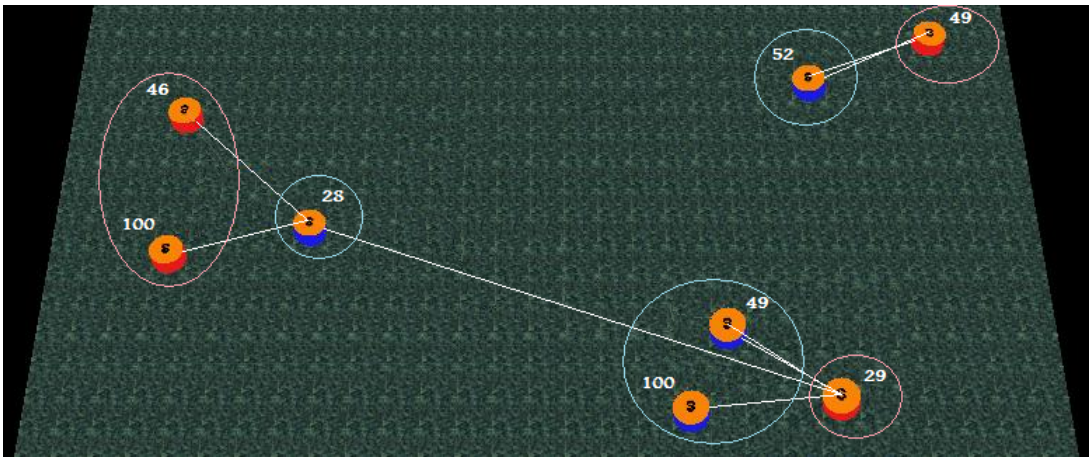


Abbildung 17: Veränderung nach einigen Spielrunden

Erst nach einigen Schusswechseln ist die Differenz seiner Gruppenstärke und der der gegnerischen Gruppe so hoch, dass er zu dem schwächeren Ziel unten

rechts befehligt wird. Während dieser Zeit, und der Zeit die der angeschlagene Soldat braucht um zu ihnen zu stoßen, können die beiden Soldaten unten rechts ihren anzahlmäßigen Vorteil zur Vernichtung des gegnerischen Soldaten nutzen.

9 Ausbau der gAImes-Engine

Wir bieten zum Umfang der gAImes-Engine noch einige Klassen an, die der KI Entwickler nahezu immer gebrauchen wird und somit nicht immer neu programmieren muss.

9.1 Wegfindung

In diesem Abschnitt wird die Funktionsweise der drei Klassen ***AStarPathfinding***, ***AStarPathfindingBarrierMap*** und ***AStarPathfindingUnitCollision*** beschieben. Diese Klassen werden in den KI-Skripten für das Antz3D Spiel verwendet um Pfade durch das Spielfeld zu berechnen. Das Spielfeld ist in der gAImes-Engine als Graph abgespeichert. Jedes Spielfeld hat mindestens die Attribute: Position, Beschaffenheit und Nachbarspielfelder. Das Attribut Beschaffenheit gibt an um welche Art von Untergrund es sich handelt, so zum Beispiel ob er begehbar ist. Im Attribut Nachbarspielfelder werden die Verknüpfungen zu den direkten Nachbarfeldern gespeichert.

9.1.1 A*-Algorithmus

Beschrieben wird die Klasse AStarPathfinding

```
public static Vector<Mapfield> findPath(Map map, Vector3f start,
Vector3f goal);
```

Mit dem A*-Algorithmus kann ein Pfad zwischen zwei Punkten auf dem Spielfeld berechnet werden. Auf Implementierungsebene berechnet der A* den kürzesten Pfad zwischen zwei Knoten im Graphen der das Spielfeld darstellt. Dieser Algorithmus ist ein informierter Suchalgorithmus, das heißt er verwendet eine Schätzfunktion (Heuristik) um zielgerichtet zu suchen. Dieser Fakt verringert die Laufzeit.

- h(x) := Geschätze Kosten vom Knoten x bis zum Zielknoten
- g(x) := Bisherige Kosten vom Startknoten bis zum Knoten x
- f(x) = g(x) + h(x)

Der A*-Algorithmus verwendet zur weiteren Pfadermittlung immer den Knoten zuerst, der am vielversprechendsten ist, dass ist der Knoten, dessen f(x)

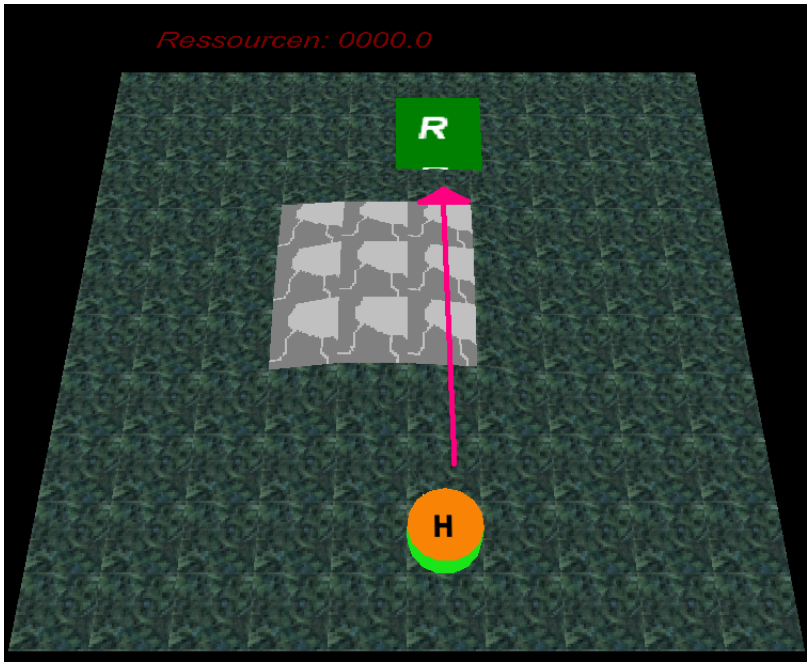


Abbildung 18: Pfadfindung ohne Berücksichtigung des Terrains

am geringsten ist. Eine Heuristik h(x) ist hierbei eine Funktion, die die Kosten zum Zielknoten nie überschätzen darf, da sonst ein Abbruch des Algorithmuses nicht gewährleistet werden kann. Eine gute Heuristik wäre beispielsweise die Berechnung der Luftlinie zum Zielknoten, da die tatsächliche Strecke nie kürzer sein kann als die direkte Verbindung.

9.1.2 Barrieren auf dem Spielfeld

Beschrieben wird die Klasse AStarPathfindingBarrierMap

```
public static Map prepareMap(Map map);
```

Auf dem Spielfeld gibt es Quadranten die starre Hindernisse darstellen sollen, z.B. Berge. Diese Hindernisse sollen von den Spieleinheiten nicht überschritten werden können. Zu diesem Zweck wird die Klasse AStarPathfindingBarrierMap genutzt. Sie filtert die Spielfelder die nicht begangen werden können aus dem Graphen. Dies tut sie, indem Sie die Spielfelder einzeln darauf prüft, ob ihr Attribut Beschaffenheit angibt, dass das Feld ein Hindernis darstellt. Das betroffene Spielfeld wird aus der Nachbarschaftsliste der Nachbarspielfelder entnommen und ist somit nicht mehr von diesen aus erreichbar. Danach wird das Spielfeld selbst aus dem Graphen entnommen.
Das so gefilterte Spielfeld kann an den A* Algorithmus übergeben werden.

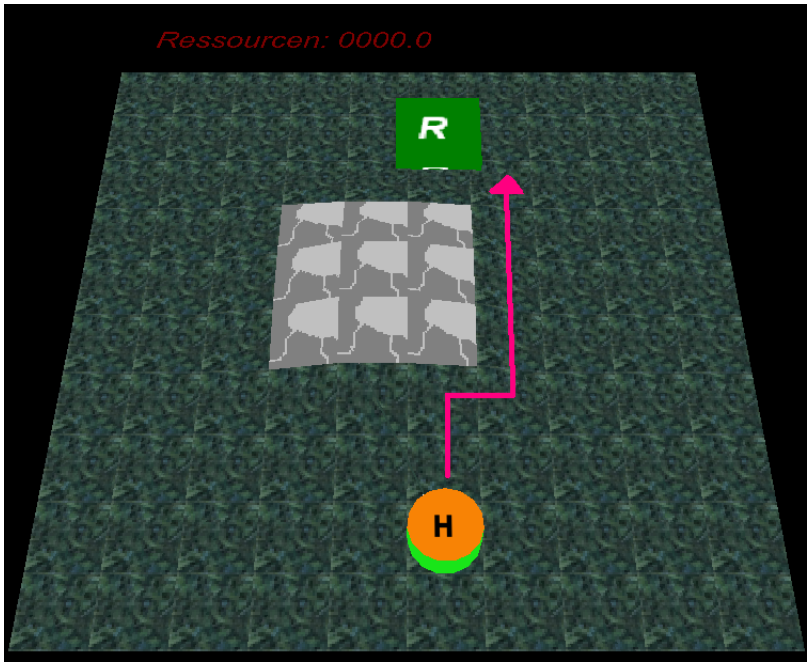


Abbildung 19: Pfadfindung mit Berücksichtigung des Terrains

9.1.3 Kollisionen mit anderen Spieleinheiten

Beschrieben wird die Klasse AStarPathfindingUnitCollision

```
public static Map prepareMap(Map map, Vector<DataUnit> gameUnits);
```

Auf dem Spielfeld befinden sich zumeist mehrere Spieleinheiten. Die Pfade der Spieleinheiten werden sich in vielen Fällen kreuzen, dies führt zu Kollisionen. Durch Kollisionen erhöht sich die Ankunftszeit der Spieleinheiten. Zudem kann die Wegfindung verbessert werden, indem von Einheiten versperrte Engpässe erkannt werden und frühzeitig eine anderer Pfad gesucht werden kann. Die Klasse ***AStarPathfindingUnitCollision*** verhindert dies, indem sie die Spielfelder auf denen mobile Einheiten stehen und die somit unpassierbar sind, aus dem Graphen herausfiltert. Dies geschieht auf die gleiche Weise wie in der zuvor beschriebenen AStarPathfindingBarrierMap Klasse. Die Liste der Spieleinheiten deren Position herausgefiltert werden sollen, wird als Parameter ***Vector<DataUnit> gameUnits*** übergeben.
Das so gefilterte Spielfeld kann an den **A* Algorithmus** übergeben werden.

Die Spieleinheit Zivilist (siehe Abschnitt 8.1, Seite 68) wurde zum Zweck eines beweglichen Hindernisses erstellt und kann somit zum Test dieser Klasse eingesetzt werden.

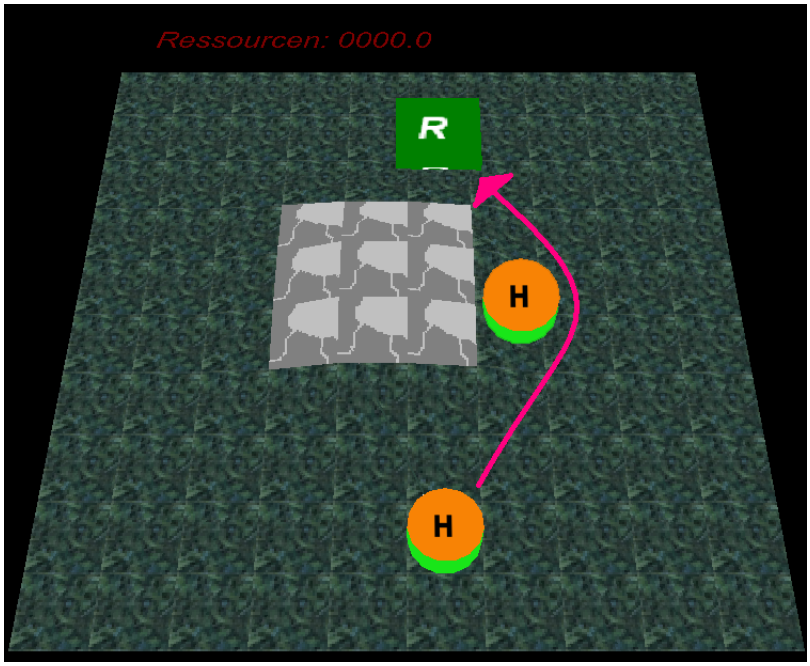


Abbildung 20: Pfadfindung mit Berücksichtigung des Terrains und Einheiten

9.1.4 Kopplung der Wegfindungsklassen

Die Klassen ***AStarPathfindingBarrierMap*** und ***AStarPathfindingUnitCollision*** bieten spezielle Methoden an die es erlauben die jeweils anderen Filterklassen mit in ihre Berechnungen zu integrieren.

```
public static Vector<Mapfield> findPath(Map map, Vector3f start,
Vector3f goal);
```

Die Klasse ***AStarPathfindingBarrierMap*** bietet die Methode ***findPath***, welche die Karte nach starren Hindernissen filtert und daraufhin auf dieser Karte den **A* Algorithmus** ausführt.

```
public static Vector<Mapfield> findPath(Map map, Vector3f start,
Vector3f goal, Vector<DataUnit> dataUnits);
public static Vector<Mapfield> findPathInBarrierMap(Map map,
Vector3f start, Vector3f goal,
Vector<DataUnit> dataUnits);
```

Die Klasse ***AStarPathfindingUnitCollision*** bietet die Methode ***findPath***, welche die Karte nach den übergebenen Einheiten filtert und daraufhin auf dieser Karte den **A* Algorithmus** ausführt. Zudem bietet sie die Methode ***findPathInBarrierMap*** welche die Karte zuerst mithilfe der Klasse ***AStarPathfindingBarrierMap*** nach starren Hindernissen filtern, dann nach den übergebenen Einheiten und zuletzt auf dieser Karte den **A* Algorithmus** ausführt.

9.2 Finden von Mustern in Zeichenketten

Beschrieben wird die Klasse PatternFinder

```
public static boolean foundMatch( String pattern, CharSequence sequence );
public static Vector<MatchResult> findMatches( String pattern,
CharSequence sequence );
```

Die Klasse ***PatternFinder*** sucht bestimmte vorgegebene Muster in Strings. Hierfür verwendet sie die Java Methodensammlung für **Reguläre Ausdrücke**. Der Methode ***foundMatch*** übergibt man im erstem Parameter das Suchmuster ***String pattern***, welches im zweiten Parameter ***CharSequence sequence*** gesucht wird. Die Methode ***findMatches*** gibt alle gefundenen Ergebnisse in einem Vektor aufgelistet zurück.
Die Klasse soll dem KI-Entwickler helfen, Stichworte in Methodennamen oder Klassennamen zu finden. So wird sie beispielsweise in der Ausführungskontrolle (siehe Abschnitt 6.3.2, Seite 48) verwendet, um das in der ***DataUnit*** angegebene KI-Skript in der Liste aller geladenen KI-Skripte zu finden.

9.3 Zugriff auf DataUnits

Beschrieben wird die Klasse ExecuteGameUnit
Die Klasse ExecuteGameUnit wird vom KI-Entwickler verwendet. Mit ihrer Hilfe lassen sich Methoden auf den Spieleinheiten, den DataUnits ausführen. Die DataUnits werden von der gAImes-Engine gespeichert und lassen den KI-Programmierer nicht erkennen, von welchem obersten Klassentyp sie sind. Eine Anforderung des gAImes-Projekts ist es, dass Spiel- und KI-Entwicklung getrennt voneinander passieren können. In diesem Fall hat der KI-Programmierer die Klassentypen der Spieleinheiten nicht zur Verfügung. Dann kann er nur den Aufruf über die ExecuteGameUnit vornehmen.

Ein Anwendungsbeispiel findet man in Kapitel 4.3.

9.4 Der Sortieralgorithmus Quicksort

Beschrieben wird die Klasse Quicksort
Der Sortieralgorithmus Quicksort kann vom KI-Entwickler eingesetzt werden, um Einheiten oder Einheitengruppen nach bestimmten Attributen zu sortieren. So kann er zum Beispiel eingesetzt werden um eine Menge von Spieleinheiten aufgrund ihrer Distanz zu einem bestimmten Ziel aufsteigend zu sortieren. Die nächstgelegene Spieleinheit könnte dann zum Angriff auf das Ziel geschickt werden.

Der Algorithmus läuft folgendermaßen ab:

Der Algorithmus wählt ein so genanntes Pivotelement aus der zu sortierenden Liste aus. Daraufhin wird die Liste in zwei Teillisten, linke und rechte aufgeteilt. Alle Elemente die kleiner als das Pivotelement sind kommen in die linke Teilliste und alle die größer als das Pivotelement sind in die rechte Teilliste. Die Elemente, die dem Pivotelement gleich sind, kommen in eine beliebige der beiden Teilliste. Als Ergebnis dieses ersten Schritts sind alle Elemente der linken Teilliste kleiner oder gleich den Elementen der rechten Teilliste. Im nun folgenden Schritt müssen noch die beiden Teillisten in sich sortiert werden. Hierfür rufen wir den Quicksort-Algorithmus rekursiv mit je einer dieser Teilliste auf. Abgebrochen wird die Rekursion wenn die Teillisten die Länge eins oder null erreicht haben.

Die Wahl des Pivotelements bestimmt die Geschwindigkeit des Algorithmus. Es sollte so gewählt werden das die Teillisten möglichst gleich lang sind und der Algorithmus somit möglichst effizient arbeiten kann.

9.5 Gruppieren mit k-means und k-means++

Beschrieben wird die Klasse k-means
k-means kann eingesetzt werden um Einheiten auf dem Spielfeld zu gruppieren. So können nahe beisammen stehende gegnerische Einheiten zu einer Gruppe zusammengefasst und ihre Kampfstärke ermittelt werden. Dieser Wert kann mit der eigenen Kampfstärke verglichen werden. Der Algorithmus kann für die KI-Entwicklung, mittels **gAImes-Engine**, eingesetzt werden, da jede Spieleinheit ein Attribut ***Position*** als Vektor hat.
Das folgende Bild zeigt ein Beispiel aus der Spielgrafik von Antz3D. Der k-means Algorithmus wird für den Spieler der Soldaten ausgeführt. Es werden die Soldaten(mit S markiert) und die Sammler(mit H markiert) in Gruppen zusammengefasst.

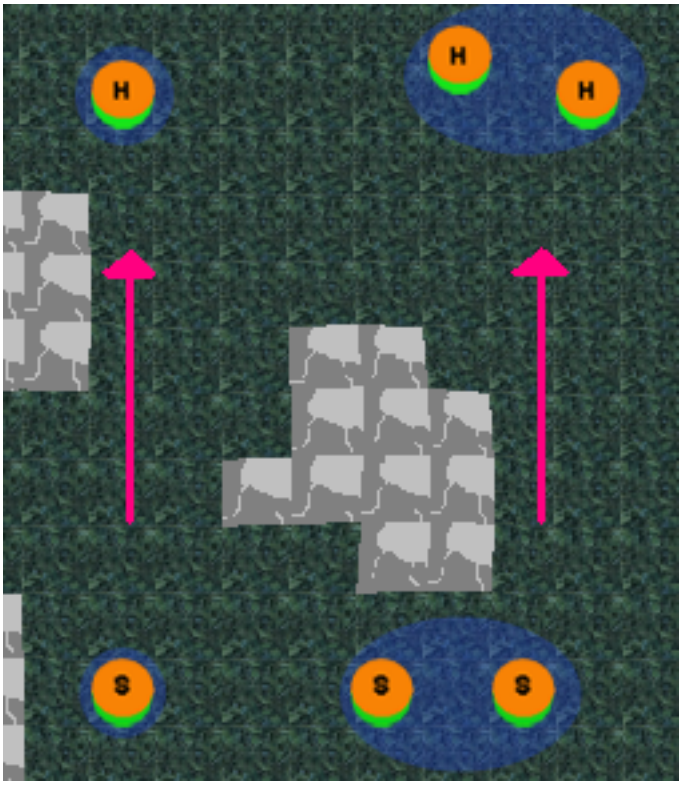


Abbildung 21: K-Means Gruppierung

Der k-means Algorithmus erstellt eine Anzahl von k Clustern aus Vektoren in einem Raum. k wird vor dem Start festgelegt. Die Funktion zur Bestimmung der Zentren der Cluster muss vorher bekannt sein.

Der Algorithmus läuft folgendermaßen ab:

1. Initialisierung: (zufällige) Auswahl von k Clusterzentren
2. Zuordnung: Jedes Objekt wird dem ihm am nächsten liegenden Clusterzentrum zugeordnet
3. Neuberechnung: Es werden für jeden Cluster die Clusterzentren neu berechnet
4. Wiederholung: Falls Abbruchkriterium erfüllt Abbruch, ansonsten weiter mit Schritt 2

Abbruchkriterien können beispielsweise eine festgelegte Anzahl von Durchläufen, eine unveränderte Clusterzuordnung oder gleichbleibende Clusterzentren sein.

Je nachdem wo die initialen Clusterzentren liegen, können sich verschiedene Cluster bilden. Dieses Problem versucht k-means++ abzuschwächen. k-means++ gibt eine Funktion zur Auswahl der initialen Clusterzentren vor.

Der Algorithmus läuft folgendermaßen ab:

X := Die Menger aller zu gruppierenden Vectoren.

1. Wähle ein initiales Zentrum c_1 zufällig aus der Menge X aus
2. Wähle das nächste Zentrum c_i, wähle $c_i = x' \in X$ mit der Wahrscheinlichkeit $\frac{D(x')^2}{\sum_{x \in X} D(x)^2}$
3. Wiederhole Schritt 2 bis alle k Centren ausgewählt wurden.
4. Weiter mit den Schritten 2-4 des standard K-Means Algorithmus

9.6 Eine Gruppe von Spieleinheiten

Beschrieben wird die Klasse GameUnitGroup

Eine GameUnitGroup ist eine Datenstruktur die genutzt werden kann um einen Gruppe von DataUnits zu erfassen. Diese Datenstruktur wird in unserer Implementation des k-means++ Algorithmus verwendet um Cluster zu erfassen. Sie wird zudem in den KI-Skripten der Kommandanten der Soldaten genutzt, um in Befehlen einem Soldaten mitzuteilen, welche seine Gruppenmitglieder sind oder welche Gruppe von Gegnern er angreifen soll.
Eine ***GameUnitGroup*** beinhaltet einen **Vector** von ***DataUnits***, mit allen Gruppenmitglieder. Zudem berechnet diese Datenstruktur das räumliche Gruppenzentrum, die durchschnittliche Distanz zwischen den einzelnen Gruppenmitgliedern, sowie die Kampfstärke der Gruppe. Das Gruppenzentrum wird für die Objektzuordung des k-means++ Algorithmus benötigt, zudem kann dieses als Ziel für den Pfad zu einer Gruppe genommen werden oder als Distanzabschätzung zwischen einer eigenen und einer feindlichen Gruppe von Soldaten. Die Durchschnittsdistanz der Gruppenmitglieder dient dem Kommandanten der Soldaten als Indikator für eine Neuberechnung der Gruppenzugehörigkeit seiner Soldaten. Die Kampfstärke einer Gruppe gibt Aufschluss darüber, welcher anderen Gruppe sie ebenbürtig ist.

9.7 Spieleinheitensortierer

Beschrieben wird die Klasse GameUnitSorter

Diese Klasse bietet Methoden an, um Spieleinheiten und Spieleinheitengruppen zu sortieren. Diese Methoden werden vom KI-Entwickler oft benötigt und sind daher in diese Klasse ausgelagert. Zu diesen Methoden gehört eine Methode, die aus einer Liste aus Spieleinheitengruppen diejenige Gruppe auswählt, die einer bestimmten Koordinate am nächsten ist. Zudem gehört zu ihnen eine Methode, die eine Liste von Spieleinheiten entsprechend ihrer Distanz zu einer bestimmten Koordinate sortiert. Zuletzt gehört zu ihnen eine Methode, die eine Liste von Spieleinheitengruppen ihrer Angriffsstärke entsprechend sortiert. Alle diese Methoden bereiten die ihnen übergebenen Daten auf und verwenden zur letztendlichen Sortierung den Quicksort Algorithmus.

10 Zusammenfassung

Nach Abschluss der Arbeit kann zusammenfassend gesagt werden, dass die gAImes-Engine alle zu Anfang an sie gestellten Ansprüche erfüllt.
Wie im Kapitel 7 ersichtlich wird kann die gAImes-Engine die Kommunikation zwischen Spiel und KI-Skripten abwickeln. Hierfür bietet sie dem Spiel, wie auch den KI-Skripten, ausreichend mächtige Schnittstellen an(siehe Abschnitt 5.3). Die Schnittstelle zur gAImes-Engine bietet eine Methode zur Übermittlung von Spielereignissen und Spielfeldveränderungen an. Diese werden mittels einer Container-Klasse übermittelt. Dieser Container erlaubt es ebenfalls die Schnittstelle einfach zu erweitern indem man diesen ausbaut(siehe Abschnitt 5.2. Zudem bietet sie eine Schnittstelle für die Übermittlung neuer Spieleinheiten und Änderungen an bekannten Spieleinheiten. Die Spieleinheiten werden durch Klassen repräsentiert die Eigenschaften und Fähigkeiten der Spieleinheiten beinhalten. Diese vermitteln den KI-Skripten ihre Aktionsmöglichkeiten.
Die Ausführung der KI-Skripte in der gAImes-Engine wird mittels Erbung der hierfür nötigen Schnittstellen von einer vorgegeben Klasse names AIUnit ermöglicht. Die Ausführungreihenfolge der Spieleinheiten wird über eine Namenskonvention ihrer implementierten Klassen geregelt. Die Zuordnung von Spieleinheit zu deren KI-Skript erfolgt mittels einer Variablen in der Spieleinheitenklasse, welcher den Namen des KI-Skriptes identifiziert.
Die Schnittstelle von der gAImes-Engine zum Spiel hin wird mittels der Übertragung von Spielaktionen realisiert. Die Spieleinheiten schicken ihre Aktionen als Ereignisse an das Spiel, welches diese in einer direkte Reaktion bestätigt oder verweigert. Die Semantik und die Syntax der möglichen Aktionen wird vom Spiel- und vom KI-Entwickler festgelegt. Je nachdem ob KI-Skript oder Spiel zuerst existiert muss sich der jeweile Part an das Protokoll des anderen anpassen. Hierbei ist die Datenklasse die eine Spielaktion repräsentiert so gewählt, dass sie einfach von beiden Parts erweitert werden kann.
Die Kommunikation zwischen KI-Skripten in der selben gAImes-Engine Instanz findet über Befehle statt. Die erteilten Befehle werden von der gAImes-Engine verwaltet und auf Anfrage des Empfängers diesem zugeschickt. Hierbei sind Sender und Empfänger mittels eindeutiger Identifikationsnummern adressierbar. Das Datenformat diese Befehle ist mittels Vererbung leicht erweiterbar.
Die gAImes-Engine bietet den KI-Skripten eine uniforme Schnittstelle an um im Spielverlauf gewonnene Erkenntnisse oder Berechnungsergebnisse speichern zu können. Diese Informationseinheiten werden von der gAImes-Engine

verwaltet und können von den KI-Skripten im Spielverlauf jederzeit wieder abgefragt werden.

Im Kapitel **2D-RTS Spiel als Testumgebung**(siehe Kapitel 7) wird anhand einer eigens dafür entwickelten Spielumgebung gezeigt, dass die gAImes-Engine die an sie gestellten Herausforderungen(siehe Abschnitt 3.2) erfüllen kann. Für diese Spielumgebung sind Spieleinheiten und KI-Skripte entwickelt worden die die von der gAImes-Engine zur Verfügung gestellten Mechanismen ausnutzen. Das beste Beispiel hierfür ist die Implementierung des **Soldaten mit Gruppenzugehörigkeit**(siehe Abschnitt 8.3.3). Dieser höhrt auf die Befehlen seines Kommandanten, reagiert auf Ereignisse im Spielgeschehen und speichert sich seine Ziele.

10.1 Vorschläge zur Weiterentwicklung

10.1.1 Evaluierungsumgebung für KI-Skripte

Die Algorithmen die ein KI-Skript nutzt, um Entscheidungen im Spielverlauf zu fällen, sind immer in irgendeiner Form parametrisiert. Diese Parameter stehen entweder im KI-Skript oder in einer separaten Konfigurationsdatei. Um diese Parameter, zur Erhöhung der Gewinnchancen, feiner zu justieren, müssen diese vor jedem Spielbeginn von Hand eingestellt werden. Dies kann sehr zeitaufwändig werden. Zur Erfüllung dieser Tätigkeit könnte eine automatisierte Evaluierungsumgebung entwickelt werden. Diese würde alle Parameter der KI-Skripte in einer globalen Konfigurationsdatei selbstständig einstellen. Sie könnte hierfür ein zuvor ausgewähltes heuristisches Optimierungsverfahren verwenden. Nach Abschluss eines Spiels würde sie dann vom Spiel einen Report über den Spielverlauf und den Ausgang bekommen, mittels dessen sie über die weitere Justage entscheidet.

10.1.2 KI-Skripte als Thread starten

Führt die gAImes-Engine ein KI-Skript aus, gibt sie die Ausführungskontrolle ab. Das KI-Skript kann von der gAImes-Engine nicht unterbrochen werden. Diese Situation kann den Ablauf des Programms stark beeinflussen. Nebenwirkungen könnten unter Anderem sein, dass das Programm sehr langsam abläuft, da das KI-Skript zu viel Rechenzeit beansprucht, oder gar nicht mehr ausgeführt wird, sollte das KI-Skript in eine Endlosschleife geraten. Um diesen Situationen vorzubeugen, kann jedes KI-Skript als Thread von der gAImes-Engine ausgeführt werden. Sollte ein KI-Skript zu viel Zeit beanspruchen, oder gar nicht mehr reagieren, kann die gAImes-Engine das Skript

beenden. Durch zusätzliche Sicherheitsmechanissmen könnte die gAImes-Engine erkennen, ob die Endlosschleife im KI-Skript auf Grund einer Fehleinschätzung einer Spielsituation geschehen ist. Der Fehler im KI-Skript könnte ein wiederholtes Senden einer Nachricht an das Spiel sein, welches einen ungültigen Zug darstellt. Das Spiel würde die Aktion verweigern, aber das KI-Skript diese immer wieder erneut senden Die Konsequenz ist eine Endlosschleife. Zusätzlich entsteht durch den Thread die Option, den KI-Skripten generell nur eine gewisse Zeit pro Runde zu geben, in der sie arbeiten können.

10.1.3 Kommunikation zwischen gAImes-Engines

In der derzeitigen Implementierung der gAImes-Engine ist es nicht möglich Nachrichten zwischen den einzelnen Instanzen von gAImes-Engines in einem Spiel auszutauschen. Kommunikation ist nur zwischen den KI-Skripten innerhalb einer gAImes-Engine möglich. Da für jeden Spieler in einem Spiel eine Instanz der gAImes-Engine erzeugt wird und es in einem Spiel auch allierte Spieler gibt, wäre es wünschenswert, dass sich KI-Skripte verschiedener Spieler austauschen können, so zum Beispiel um ihr taktisches Vorgehen abzustimmen. Zu diesem Zweck könnte die gAImes-Engine mit ein paar kleineren Anpassungen so umgeschrieben werden das alle gAImes-Engines in einer übergeordneten gAImes-Engine sitzen, die für die Komunikation zuständig ist.

10.1.4 Aufzeichnen des Spielverlaufs

Im Verlauf eines Spiels können, bei entsprechend vielen Spieleinheiten auf dem Spielfeld, viele Aktionen gleichzeitig ablaufen. Für den KI-Entwickler, der implementierte KI-Skripte evaluieren möchte, ist die Informationsflut sehr groß. So wäre es von Vorteil, wenn der Spielverlauf aufgezeichnet werden könnte. Spielsituationen könnten genauer analysiert, Fehler gefunden und mögliche Verbesserungen besser erkannt werden. Diese Funktionalität könnte erreicht werden, indem die Ausgangsschnittstelle der gAImes-Engine so erweitert würde, dass sie alle Aktionen der KI-Skripte aufzeichnet. Diese Aufzeichnungen würden für eine spätere Betrachtung des Spielverlauf von einem separatem Simulator an das Spiel übertragen werden.

10.1.5 KI-Skripte-Datenbank

Zusätzlich zur gAImes-Engine könnte es eine Datenbank für KI-Skripte geben. Kategorisiert nach unterschiedlichen Einsatzgebieten und Schwerpunkten, wie z.B. Wegfindung oder Gruppenbildung, könnten KI-Skripte gespeichert sein. Als Anwender der gAImes-Engine hat man eine große Auswahl

und kann das eigene Produkt schnell mit effizienter künstlicher Intelligenz austatten.

10.1.6 Die gAImes-Engine für andere Programmiersprachen

Die Konzepte der gAImes-Engine sind nicht an die Programmiersprache Java gebunden. Um die vorhandene Java Implementierung der gAImes-Engine an ein vorhandenes C++ Spiel anzubinden, ist eine Kommunikationsmodul nötig. Die notwendigen Daten würden dann über eine unabhängige Schnittstelle, z.B. das TCP/IP Protokoll, zwischen den beiden Parts versand. Diese Vorgehensweise ist sowohl auf Windows, Linux und Mac Systemen möglich und durchaus üblich. Dazu müsste entsprechend für das Betriebssystem eine kleine Applikation geschrieben werden, die für die Kommunikation zuständig ist. Über diese können das Spiel und die gAImes-Engine miteinander kommunizieren. Dieses Verfahren wird zum Beispiel von MySQL verwendet. Ein MySQL Datenbankserver ist systemunabhängig. Er wird für das jeweilige Betriebsystem kompiliert ausgeliefert. Der Datenaustausch mit einem Webserver auf Windows erfolgt über einen vordefinierten Port.

Literatur

[BS04] David M. Bourg, Glenn Seemann, AI for Game Developers, O'Reilly, 2004.

[C03] Alex J. Champandard, AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors, New Riders Publishing, 2003.

[H62] C.A.R. Hoare, Quicksort. Computer Journal, Vol. 5, 1, 10-15, 1962

[J3D00] Sun Microsystems Java 3D Engineering Team, Java 3D Tutorials, 2000, http://java.sun.com/developer/onlineTraining/java3d/index.html

[K08] Heiko Klinge, Warum Künstliche Intelligenz (KI) in Spielen stagniert, Vol. 2, 2008, GameStar.

[M02] Robert C. Martin, „Agile Software Development, Principles, Patterns, and Practices", 2002, Prentice Hall.

[RN04] Stuart Russell, Peter Norvig, Künstliche Intelligenz: Ein moderner Ansatz, 2004, Prentice Hall Series in Artificial Intelligence.

[U08] Christian Ullenboom, Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6, 2008, Galileo Press.

[VA07] Sergei Vassilvitskii, David Arthur, k-means++: The Advantages of Careful Seeding, 2007, Symposium on Discrete Algorithms (SODA).

[X09] http://www.xaitment.com/.