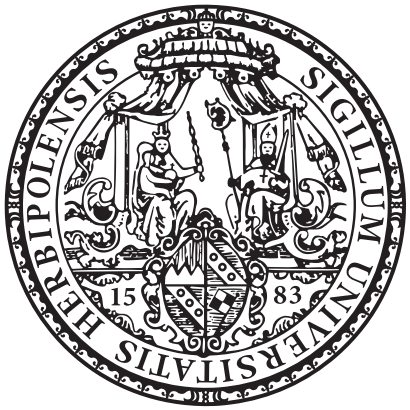


Universität Würzburg

Diplomarbeit

Konzeption und Programmierung eines Spieleagenten für StarCraft: Brood War

---

Vorgelegt von:
Jürgen Binder

Betreuer:
Prof. Dr. Frank Puppe
(Universität Würzburg)

Zweitgutachter:
Prof. Dr. Martin V. Butz
(Universität Tübingen)

9. Oktober 2012

Erklärung

Ich versichere, die vorliegende Diplomarbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Quellen angefertigt zu haben.

Würzburg, 9. Oktober 2012 _____

Jürgen Binder

Danksagungen

Mein Dank gilt allen, die dazu beigetragen haben, dass diese Diplomarbeit erfolgreich abgeschlossen werden konnte. Zuerst möchte ich mich bei meiner Familie bedanken, die mich auf jede erdenkliche Weise unterstützt und ohne die dieses Studium nicht möglich gewesen wäre. Weiterhin möchte ich mich bei allen meinen Freunden bedanken, die durch ihren moralischen Beistand mir geholfen haben, diese stressige Zeit durchzustehen.

Vielen Dank auch an Prof. Dr. Martin V. Butz, der mir dieses interessante Thema anbot und mich in schwierigen Phasen unterstützte, meinem Betreuer Stephan Ehrenfeld für wertvolle Anregungen und Fachwissen, und Prof Dr. Frank Puppe, der es durch die Annahme der Diplomarbeit möglich machte, diese zu beenden, nachdem Prof. Butz an die Uni Tübingen berufen wurde.

Zusammenfassung

Durch die Entwicklung der Programmierschnittstelle BWAPI für StarCraft: Brood War konnte die Forschungsumgebung im Bereich der künstlichen Intelligenz für Echtzeitstrategiespiele erheblich attraktiver gestaltet werden. Ziel dieser Arbeit ist es einen Bot für StarCraft zu implementieren. Dieser muss die verschiedenen Teilaufgaben wie z.B. Einheiten bauen und verwalten sowie kämpfen bewältigen können. Der gewählte Ansatz erwies sich als richtig und mit Dreadbot konnte ein ernstzunehmender Bot implementiert werden, der achtbare Resultate erzielt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Spiele sind immer ein Forschungsobjekt, das Aufsehen erregt und neugierig macht. Auch in der Künstlichen-Intelligenz-Forschung hat man sich schon lange mit Spielen befasst. Großen Erfolg konnte sie besonders bei rundenbasierten Spielen erzielen. Hier liegt der Vorteil in der meist unbeschränkten Zuglänge, die es möglich macht, Rechenzeit und -kapazität zu nutzen, um viele Optionen durchzurechnen. Durch Aufwendung von genügend Rechenleistung ist es dann möglich, menschliche Topspieler zu schlagen. Dies war auch der Grund für *Deep Blues* spektakulären Erfolg, der 1996 als erster Computer gegen den amtierenden Schachweltmeister, Garri Kasparow, gewann. Er war ein massiv paralleler Rechner, der über 100 Millionen Stellungen pro Sekunde berechnen konnte (Campbell u. a. (2002)). Ebenso wurde Dame durch das Durchrechnen aller Möglichkeiten komplett gelöst (Schaeffer u. a. (2007)). Ein Vorteil für Computer bei Schach und Dame ist auch, dass alle Züge globale Konsequenzen haben. Bei Spielen wie Go jedoch, wo viele Züge nur lokale Auswirkungen haben, konnten noch keine derartigen Erfolge erzielt werden. Diese Probleme sind für KIs wesentlich schwerer zu lösen.

Echtzeitstrategiespiele sind hingegen eine andere Herausforderung als solche klassischen Brettspiele. Hier müssen, wie der Name schon sagt, Aktionen in Echtzeit ausgeführt werden. Dadurch ist die Denkzeit extrem beschränkt. Zudem müssen sehr viele Aktionen ausgeführt werden und damit Entscheidungen getroffen werden. Viele von diesen haben auch nur lokale Konsequenzen, deren Einschätzung KI-Programmen wie bei Go zur Zeit meist noch schwer fällt. Erschwerend kommt hinzu, dass nicht perfekte Information über die Spielwelt zur Verfügung steht, sondern Teile dieser im sogenannten *Fog of War* verborgen sind. Unter diesen Unsicherheiten müssen Entscheidungen getroffen werden. Dies ist nur möglich, wenn geeignete Modelle zur Verfügung stehen. Solche müssen es auch ermöglichen, den zeitlichen und auch räumlichen Zusammenhang von Aktionen und deren Konsequenzen zu erfassen.

Buro und Furtak (2003) beschrieben diese Herausforderungen als Erste und riefen zur Forschung in diesem Bereich auf. Hauptproblem war jedoch das Fehlen einer ge-

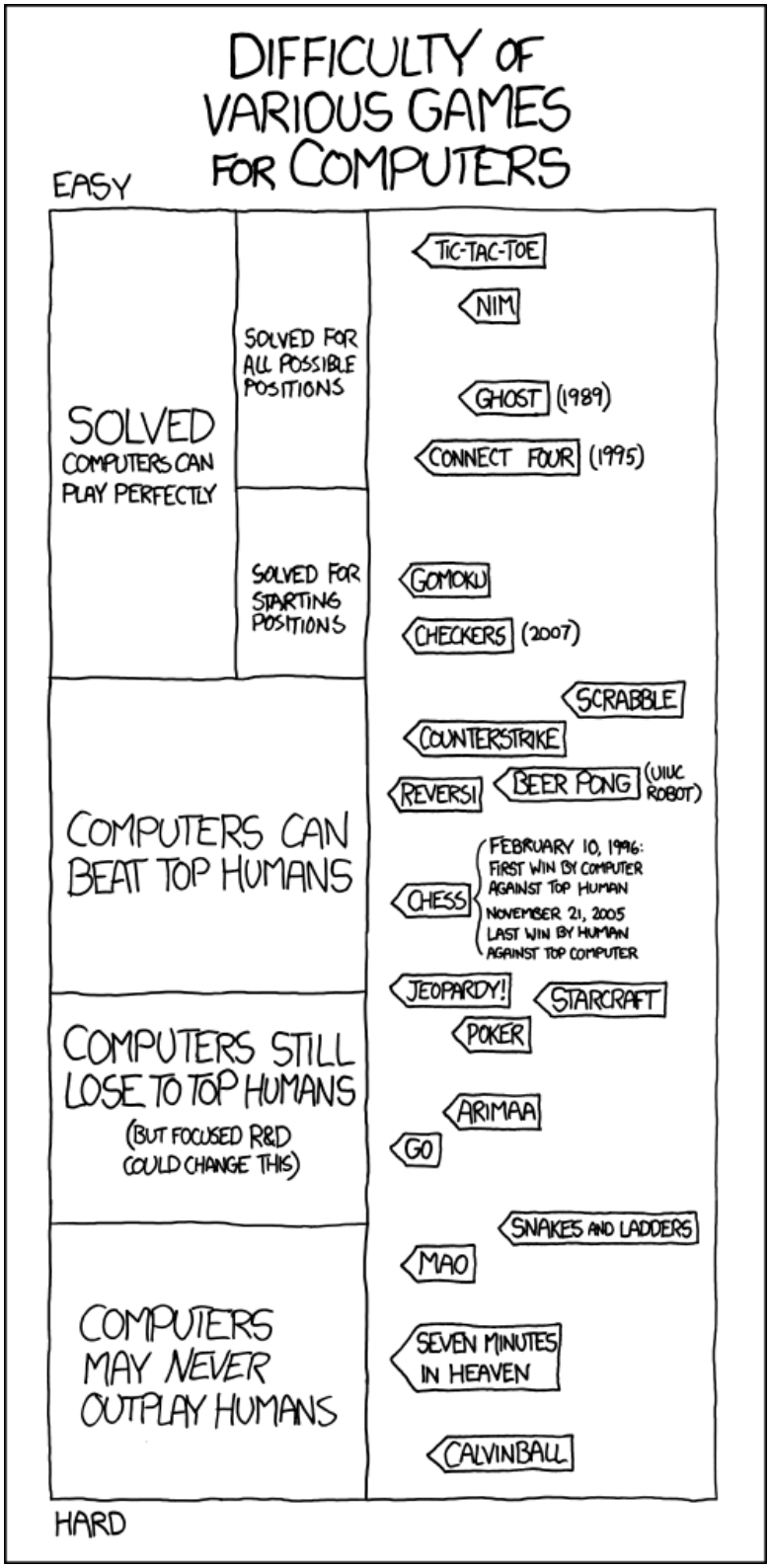


Abbildung 1.1: Humoristische und realistische Übersicht des aktuellen Forschungsstands bei Spiel-KIs durch xkcd #1002: Game AIs

eigneten Forschungsumgebung. Kommerzielle Spiele boten keinerlei Schnittstelle, um KI-Agenten zu laden. Quelloffene Spiele waren zwar vorhanden, aber konnten qualitativ nicht mit den kommerziellen Spielen mithalten. Die Situation änderte sich 2009 mit der Entwicklung der Schnittstelle BWAPI für StarCraft: Brood War durch Freiwillige. Dadurch ist es nun möglich, mit einem herausragenden Vertreter des Genres zu arbeiten und zu forschen. Dank diesem populären Spiel wird der Forschungskontext im Allgemeinen attraktiver und kann mehr Interessierte anziehen. Deswegen war und ist es möglich, Wettbewerbe zwischen KI-Agenten zu veranstalten, was zugleich die Forschung weiter vorantreibt.

Ein Ziel ist hierbei auch, regelmäßig menschliche Top-Spieler schlagen zu können. Vereinzelt gelang dies bereits in Testmatches. Im Rahmen von Schaukämpfen bei Turnieren und Konferenzen behielt jedoch der Mensch bisher stets die Oberhand.

1.2 Aufgabenstellung

In der vorliegenden Arbeit soll ein sogenannter ***Bot*** für StarCraft implementiert werden. ***Bot*** ist die Bezeichnung für ein Programm, das ein Computerspiel selbstständig spielt. Dieser Bot soll Techniken der künstlichen Intelligenz nutzen, um bestmögliche Ergebnisse zu erzielen.

Hierzu ist es nötig sich mit der Programmierschnittstelle BWAPI vertraut zu machen und eine Grundstruktur für den Bot zu entwerfen. Dann müssen die Teilaufgaben bestimmt und geeigenete Techniken zu deren Lösung entworfen werden. Eine dieser Aufgaben ist die Platzierung von Gebäuden. Hierbei muss sichergestellt werden, dass keine Einheiten eingebaut, also blockiert werden, aber auch, dass wichtige Gebäude zum Schutz hinter anderen Gebäuden gebaut werden. Weiterhin soll eine Angriffsmethodik implementiert werden. Hierzu kann ein Zustandsautomat verwendet werden. Zudem soll der Bot Methoden zum ***Scouting***, also dem Auskundschaften des Gegners und des unbekannten Terrains, besitzen und dabei strukturiert und gezielt vorgehen.

Grundsätzlich kann in jedem Teilbereich des Bots sehr viel Forschungsarbeit investiert werden, wie verschiedene Veröffentlichungen zu diesen zeigen. Somit ist das Ziel dieser grundlegenden Diplomarbeit grundsätzlich erreicht, wenn KI-basierte Strategien in den Bot eingebaut werden, die in der Evaluation den erwünschten Erfolg an den Tag legen. Zudem sollte der Bot auch erfolgreich mit dem Spiel interagieren, nicht abstürzen und gegen den eingebauten, normalen Computergegner grundsätzlich gewinnen.

1.3 Überblick über die Arbeit

Im folgenden Kapitel 2 wird das Spielprinzip von Echtzeitstrategiespielen vorgestellt sowie erläutert, warum es sich bei StarCraft um einen herausragenden Vertreter dieses Genres handelt. In Kapitel 3 wird der aktuelle Stand der Forschung anhand verwendeter KI-Methoden dargestellt sowie Bots vorgestellt, die diese implementieren und/oder bei Turnieren auffielen. In Kapitel 4 wird die Programmierschnittstelle BWAPI vorgestellt. In Kapitel 5 werden schließlich die eigene Implementierung sowie die Konzepte dahinter erläutert. Im abschließenden Kapitel 6 werden die Ergebnisse der Arbeit und das Abschneiden des Bots sowie ein Ausblick auf die Zukunft präsentiert.

Kapitel 2

Echtzeitstrategiespiele & StarCraft

In diesem Kapitel soll zuerst das Genre der Echtzeitstrategiespiele vorgestellt und diskutiert werden. Es werden das Spielprinzip und die Aufgaben des Spielers erläutert, die in allen Spielen des Genres auftreten. Weiterhin wird dargelegt, was StarCraft zu einem besonderen Vertreter dieses Genres macht und welche spezifischen Eigenheiten es bietet.

2.1 Echtzeitstrategiespiele

Echtzeitstrategiespiele stellen, gemessen an den Verkaufszahlen, heute eines der beliebtesten Spielegenres dar (ESA (2012)). Charakteristisch für diese Gattung von Spielen ist, dass alle beteiligten Spieler gleichzeitig und – wie der Name schon sagt – in Echtzeit handeln. Dementsprechend schnell müssen Entscheidungen getroffen und umgesetzt werden. Dies unterscheidet diese Sorte von Spielen von rundenbasierten Spielen, in denen jeder Spieler nacheinander seinen Zug tätig und u.U. praktisch unbegrenzte Bedenkzeit hat.

Zum Grundprinzip der Echtzeitstrategiespiele gehören nicht nur Kampfhandlungen, sondern in der Regel auch eine (vereinfachte) wirtschaftliche Komponente. Der Spieler muss Ressourcen beschaffen, die zum Bau von Einheiten und Gebäuden benötigt werden. Im Gegensatz zu Wirtschaftssimulationen wie etwa der Anno- oder Siedler-Reihe ist die Komplexität der Wirtschaftssimulation jedoch reduziert. Dafür liegt ein vergleichsweise stärkeres Gewicht auf den Kampfhandlungen.

Der Begriff des Echtzeitstrategiespiels wurde erstmal für das 1992 erschienene Dune II verwendet. Jedoch gab es zuvor schon Spiele, die wichtige Elemente von Echtzeitstrategiespielen aufwiesen. Zur schnell wachsenden Popularität des Genres trugen entscheidend die Command-&-Conquer-, WarCraft- und Age-of-Empires-Reihen und natürlich StarCraft bei.

In fast allen Fällen bieten Echtzeitstrategiespiel eine Kampagne für den Einspielermodus und einen Mehrspielermodus. Eine Kampagne besteht aus aufeinander auf-

Abbildung 2.1: Screenshot von Dune II

bauenden Missionen, die eine Geschichte erzählen. In diesen Missionen stehen nicht immer alle Einheiten zur Verfügung, es können verbesserte Einheiten vorhanden sein und es herrschen selten für den Computergegner und den Spieler die gleichen Bedingungen. Der Computergegner agiert hierbei nicht nur auf Basis einer KI, sondern auch gescriptete Ereignisse bestimmen sein Handeln.

Im Mehrspielerspielermodus hingegen herrschen für alle beteiligten Parteien die gleichen Bedingungen. Alle im Spiel verfügbaren Einheiten stehen hierbei mit den gleichen Eigenschaften zur Verfügung. In der Regel existieren verschiedene Spielmodi, häufig auch noch mit Modifikationen, die zusätzliche Komplexität bieten. Zum einen gibt es den Standardmodus, in dem es gilt, alle Gegner zu besiegen. Hierbei kann es Allianzen verschiedener Spieler geben. Zum anderen kann es Spezialmodi wie *Capture the Flag* geben, bei der man bestimmte Gebiete oder Gegenstände erobern und eine Zeit lang verteidigen muss, um das Spiel zu gewinnen. Im Mehrspielermodus kann üblicherweise sowohl gegen einen Computergegner gespielt werden, als auch gegen menschliche Gegner über lokale Netzwerke oder das Internet. Hier kann man nicht nur Einzelspiele gegen einem bekannte oder auch unbekannte Spieler spielen, sondern es ist auch möglich an weltweiten Ligensystemen teilzunehmen. Dadurch haben sich Echtzeitstrategiespiele im E-Sport als das eines der beliebtesten Genres etablieren können (Deutscher eSport-Bund (2011)).

2.2 Spielprinzip

Im Folgenden wird das Spielprinzip von Echtzeitstrategiespielen vorgestellt. Einzelne Elemente können bei verschiedenen Spielen unterschiedlich stark ausgeprägt sein, im Großen und Ganzen sind sie jedoch Grundbestandteil aller Spiele des Genres.

2.2.1 Macromanagement & Strategie

Als Macromanagement bezeichnet man das Treffen strategischer Entscheidungen. Hierzu gehören zum einen Entscheidungen über das Bauen von Einheiten: welche Einheiten gebaut werden, wann sie gebaut werden, wie viele von welchem Typ. Dieselben Entscheidungen sind auch für Upgrades nötig, falls diese in dem jeweiligen Spiel vorhanden sind. Weiterhin gehören Angriffsentscheidungen zum Macromanagement: wann angriffen werden soll, mit welchen Einheiten und was das Angriffsziel ist, z.B. ob versucht werden soll, den Gegner sofort zu besiegen oder ob vorrangig seine Wirtschaft geschwächt werden soll. Auch das Basismanagement (siehe 2.2.1.3) gehört zum Macromanagement. Die genannten Entscheidungen werden jedoch nicht isoliert voneinander getroffen. Die wichtigsten strategischen Entscheidungen sind das Abwägen zwischen Bauen, Auf- und Ausbau der eigenen Basis oder Angreifen. Diese Entscheidungen sind oft ausschlaggebend für Sieg oder Niederlage.

2.2.1.1 Ressourcenmanagement

Um Einheiten produzieren und Gebäude bauen zu können, müssen ***Rohstoffe*** gesammelt werden. Die Anzahl der Rohstoffarten unterscheidet sich von Spiel zu Spiel und dies kann für eine erhöhte Komplexität des Wirtschaftsteils sorgen. So gibt es z.B. bei den Command-&-Conquer-Spielen nur eine Art von Rohstoffen, das sogenannte Tiberium, während es in Age of Empire vier verschiedene Arten von Ressourcen gibt: Nahrung, Holz, Stein und Gold. Die Ressourcen sind üblicherweise nur begrenzt vorhanden und müssen an bestimmten Stellen auf der Karte abgebaut werden. Hierfür ist meist eine spezielle Einheit zuständig, die als *Arbeiter* bezeichnet wird. Die Sicherung der Ressourcenquellen ist somit ein entscheidender Teil des Spiels. Gibt es mehrere Rohstoffe in einem Spiel, so bestimmen auch die Rohstoffquellen, die man sich erschlossen hat, die Einheitenkomposition der Armee. Hierbei kann es ausreichen, dass einer der Rohstoffe nicht in der gewünschten Geschwindigkeit gesammelt werden kann, damit die gewünschte Zusammensetzung der Armee überdacht werden muss und man gezwungen ist, seine Einheitenkomposition und gegebenenfalls weitere strategische Entscheidungen anzupassen.

Ein weiterer limitierender Faktor für die Armee, der auch eine Art Ressource darstellt und insbesondere bei StarCraft eine wichtige Rolle spielt, kann die sogenannte ***Versorgung*** (engl. Supply) sein. Versorgung wird im Gegensatz zu gewöhnlichen Rohstoffen nicht gesammelt, sondern von bestimmten eigenen Einheiten oder Gebäuden zur Verfügung gestellt, jedoch nur solange, wie diese Einheiten leben. Kampfeinheiten verbrauchen einen Teil davon während sie am Leben sind. Die nötige Versorgung muss vor Baubeginn vorhanden sein. Versorgung wird also von eigenen Einheiten zur Verfügung gestellt und verbraucht, und auch nur solange die entsprechenden Einheiten leben. Jedoch stellen die entsprechenden Einheiten bzw. Gebäude meist nur eine geringe Menge an Versorgung für ein paar wenige verbrauchende Einheiten zur

Verfügung, so dass diese recht häufig nachgebaut werden müssen und damit ein stark limitierender Faktor für die Armeegröße sind, der gut im Auge behalten werden muss.

2.2.1.2 Tech tree & Baureihenfolge

Nicht alle Einheiten- und Gebäudetypen sind zu Beginn eines Spieles verfügbar. Die Produktion von bestimmten Einheiten oder Gebäude schalten weitere Typen frei, die dann produziert oder gebaut werden können. Dies nennt man den ***Technologiebaum*** oder auch *Tech tree*. Die Entscheidung, welche Äste des Technologie-Baums man verfolgt, also welche Einheiten man bauen will, ist ein sehr wichtiger Teil der Strategie.

Die Baureihenfolge wird, neben der längerfristigen Entscheidung der Wahl der freizuschaltenden Technologien, auch von kurzfristigen Situationen bedingt. Um auf Angriffe zu reagieren oder um günstige Zeitpunkte für Angriffe nutzen zu können, kann es notwendig sein, von der geplanten Baustrategie abzuweichen und auf Grundlage der Spielsituation eine neue zu wählen.

2.2.1.3 Basismanagement

Zum Basismanagement gehört die Platzierung von Gebäuden innerhalb einer Basis. Dazu gehört auch das Bauen von Verteidigungsanlagen. Durch geschickte Platzierung von Gebäuden unter Berücksichtigung des Geländes kann die Verteidigung einer Basis erheblich verbessert werden, wie z.B. bei der Strategie *Protoss Fast Expand Forge Walling* in StarCraft. Dabei werden die Gebäude so gebaut, dass nur ein Weg in die eigenen Basis möglich ist, wodurch es wesentlich leichter ist, diese zu verteidigen, siehe Abbildung 2.3. Weiterhin fällt unter Basismanagement die Entscheidung neue Basen zu errichten. Hierbei ist vor allem der Zeitpunkt und auch der Ort der neuen Basis entscheidend. Zum Aufbau einer neuen Basis sind Ressourcen nötig, welche dann nicht in den Bau von Kampfeinheiten investiert werden können. Daher muss eine hinreichend große Armee vorhanden sein, um sowohl die bisherige als auch die neue Basis verteidigen zu können. Alternativ kann man einen Zeitpunkt wählen, in dem der Gegner durch einen misslungen Angriff geschwächt ist oder dadurch, dass er selbst expandiert, nicht angreifen kann. Solche Gelegenheiten erlauben es dann, die neuen Basen mit einer vergleichsweise kleinen Armee zu verteidigen.

2.2.1.4 Kartensicht & Scouten

Üblicherweise ist bei aktuellen Echtzeitstrategiespielen zu Beginn eines Spieles nicht die ganze Karte sichtbar. Je nach Spieltyp können die statischen Kartendaten, d.h. die Positionen der Ressourcen, das Gelände, und die möglichen Startposition bekannt sein. Jede Einheit hat einen Sichtradius, innerhalb dem der Spieler, dem die Einheit gehört, gegnerische Einheiten und das Gelände sieht. Von erforschten, also schon

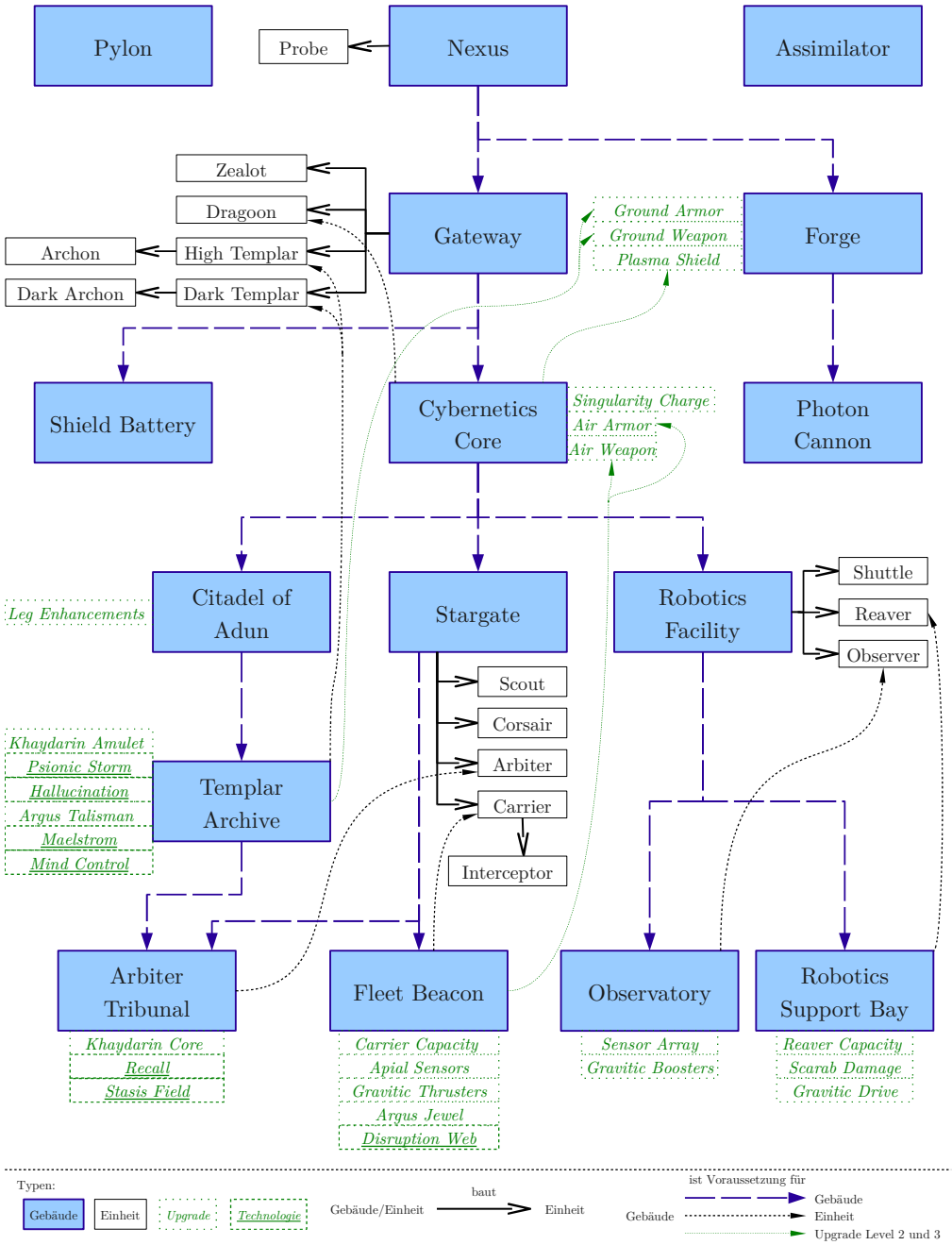


Abbildung 2.2: Tech tree der Protoss in StarCraft

Beispielsweise ermöglicht ein Gateway das Bauen der Shield Battery und des Cybernetic Core. Letzterer ermöglicht wiederum das Bauen der Citadel of Adun, Stargate und Robotics Facility.

Für das Arbiter Tribunal muss sowohl ein Stargate als auch ein Templar Archive vorhanden sein.

Pylon, Nexus und Assimilator haben keine Voraussetzungen, diese können jederzeit gebaut werden.

Abbildung 2.3: Protoss Fast Expand Forge Walling

Abbildung 2.4: Fog of War: Eine Protoss-Sonde scoutet eine terranische Basis

Sichtbereich (links): Im Sichtbereich der Sonde, sind gegnerische Einheiten sichtbar.
Fog of War (mitte): Im Fog of War sind nur Gelände, Gebäude und Ressourcen eingezeichnet, so wie sie zuletzt gesehen wurden
unerkundet (rechts): im unerkundeten Bereich ist nichts zu senden

einmal gesichteten Gebieten, bleibt das Gelände bekannt. Gegnerische Einheiten in Gebieten, die nicht mehr im Sichtradius einer eigenen Einheit liegen, verschwinden üblicherweise im sogenannten *Fog of War*.

Deshalb ist es nötig, immer wieder Einheiten loszuschicken, um Informationen über den Gegner zu gewinnen. Dies wird als *Scouten* bezeichnet. Von Interesse hierbei ist v.a. auf welche Art von Einheiten der Gegner setzt und welche Anzahl er hiervon hat, um damit auf seine Strategie zu schließen und auf diese reagieren zu können. Üblicherweise verwendet man zum Scouten billige und/oder schnelle Einheiten, deren Verlust man einkalkuliert.

2.2.2 Micromanagement & Taktik

Micromanagement bezeichnet das Steuern von einzelnen Einheiten. Je genauer man seine Einheiten steuern kann, desto zielgerichteter kann man sie einsetzen. Als Maß für das Micromanagement haben sich die Aktionen pro Minute, kurz APM, etabliert. Ein höherer Wert wird als Indiz für ein besseres Micromanagement gewertet, da man prinzipiell mehr Befehle pro Zeiteinheit absetzen und damit mehr Einheiten steuern kann. Top-Spieler erreichen bei StarCraft Werte von über 300 APM (Liquipedia (2009a)). Ein Problem beim Micromanagement kann die eingebaute Einheiten-KI sein. Einheiten greifen Gegner in ihrer Nähe von selbst an. Dies kann dazu führen, dass sie nicht an der gewünschten Stelle bleiben oder sich nicht wie gewünscht bewegen. Hierzu bedarf es dann die Kombination verschiedener Befehle, um das gewünschte Verhalten zu erzwingen.

Eng verwoben mit dieser mechanischen Fähigkeit ist die taktische Komponente es Spieles. Ohne eine Taktik, also einen Plan, wie man seine Einheiten einsetzt, ist die Fähigkeit des Micromanagements weniger effektiv. Auch kann es kontraproduktiv sein, einem kleinen Gefecht viel Aufmerksamkeit zu widmen, wenn man hierdurch die Wirtschaft vernachlässigt oder einem Überraschungsangriff an anderer Stelle zum Opfer fällt.

2.2.2.1 Kämpfen

Essentieller Teil eines jeden Echtzeitstrategiespiels sind Kämpfe. Hier kann man durch geschicktes Micromanagement und kluge Taktiken entscheidende Vorteile erzielen. Gewöhnlich kann man ein Schere-Stein-Papier-Prinzip ausnutzen, um mit einer gewissen Einheitenkombination besonders gut eine gegnerische Armee zu bekämpfen.

Es gibt eine Reihe von Standardkampftaktiken, die jeder kennen muss, auch um sie verteidigen zu können. Zu diesen gehört der sogenannte ***Rush***. Dabei versucht man möglichst schnell eine bestimmte Anzahl von Einheiten, meist eines Types, zu bauen und damit den Gegner zu überrennen. Eine Möglichkeit ist es, schnell eine große

Anzahl von billigen Einheiten, die schon zu Beginn des Spieles verfügbar sind zu bauen und den Gegner in der Anfangsphase zu überrennen. Alternativ kann man versuchen, den Technologiebaum schnell soweit auszubauen, um eine relativ starke Kampfeinheit möglichst früh im Spiel zur Verfügung zu haben und damit den Gegner zu besiegen.

Eine weitere Standardtechnik ist der ***Drop***. Hierbei werden mittels Lufttransporteinheiten Bodeneinheiten direkt in der gegnerischen Basis abgesetzt. Dadurch muss sich der gegnerische Spieler um die Verteidigung seiner Basis kümmern und gegebenenfalls zu diesem Zeitpunkt besonders wertvolle Einheiten beschützen, so dass er sich nicht um andere Aufgaben kümmern kann und idealerweise seine Einheitenproduktion gestört wird.

Außerdem gibt es noch die Taktik des ***All-ins***. Dabei wird mit allen vorhandenen Einheiten angegriffen, um das Spiel für sich zu entscheiden. Dies kann entweder geplant sein, meist sind All-ins jedoch die letzte Möglichkeit ein Spiel noch zu gewinnen, da man eine schwächere Wirtschaft als der Gegner hat und dieser dadurch auf lange Sicht eine viel stärkere Armee aufbauen wird. Für geplante All-ins ist der Umstand entscheidend, dass der Aufbau einer starken Wirtschaft selbst mit Kosten verbunden ist, die sich erst nach einer gewissen Zeit vollständig amortisieren. Hierdurch entstehen Zeitfenster, in denen der Spieler, der seine Ressourcen direkt in den Bau von Kampfeinheiten investiert, über eine stärkere Armee verfügt. Gelingt es ihm jedoch nicht, mit diesem Vorteil das Spiel zu gewinnen oder zumindest genug Schaden anzurichten, um wirtschaftlich aufholen zu können, ist er im Nachteil und wird das Spiel meist verlieren – daher die Bezeichnung All-in.

2.2.2.2 Einheitenpositionierung im Kampf

Im Kampf möchte man den Schaden, den man anrichten kann, möglichst lange möglichst hoch halten und den des Gegners möglichst schnell reduzieren. Denn verfügen beide Armeen über ähnliche Schadens- und Lebenswerte, ist dies der Schlüssel zum Erfolg. Man setzt die geschickte Positionierung von Einheiten dazu ein, dies zu erreichen. Dieses Micromanagement wird auch *Dancing* genannt.

Gewöhnlich gibt man den Schaden pro Sekunde als *DPS-Wert* (Damage per second) an. Verwendet man selbst Fernkampfeinheiten, so können diese die Technik des *Focus Fire* anwenden. Dabei visieren die eigenen Einheiten die selbe gegnerische Einheit an, um sie schnellstmöglich zu zerstören und damit den gegnerischen DPS-Wert zu senken. Hierbei muss man darauf achten, dass nicht viel mehr Geschosse auf die gegnerische Einheit abgefeuert werden als nötig sind, um diese zu zerstören (*Overkill*), da diese Geschosse nutzlos verschwendet wären. In diesem Fall sollte man seine eigenen Einheiten in Gruppen aufteilen, die sich jeweils um eine gegnerische Einheit kümmern. Zudem versucht man, stark verwundete Einheiten zurückzuziehen, um deren Verlust zu vermeiden oder zumindest hinauszuzögern. Gemeinsam kann eine Armee deutlich mehr Schaden aufnehmen, bevor es zum Verlust einer Einheit kommen muss.

Sind die Einheiten im Fernkampf in mehreren Reihen aufgestellt, versucht man möglichst viele Einheiten in die vordere Reihe zu bekommen, damit sich der Schaden auf diese verteilt. Hierbei wird ausgenutzt, dass Einheiten von sich aus meist die nächste gegnerische Einheiten angreifen, wenn sie keinen anders lautenden Befehl wie z.B. zum Focus Fire erhalten haben. Ab einer bestimmten Armeegröße ist Einheitenpositionierung jedoch viel praktikabler als Focus Fire.

Solche Stellungen gelingen am besten, wenn man die eigenen Einheiten in einem Bogen (*Konkave*) aufstellt, sodass die gegnerischen Einheiten sich bei allen in Waffenreichweite befinden. Das zwingt den Gegner in eine konvexe Position mit wenigen Fronteinheiten, was den Angriff der eigenen Einheiten auf diese konzentriert. Dies kommt Focus Fire recht nahe, ist jedoch weitaus einfacher zu erreichen und somit sehr viel praktikabler. Dies führt dazu, dass die Einheiten des Gegners schneller sterben und somit sinkt der DPS-Wert des Gegners schneller.

Treffen Nahkampfeinheiten auf Fernkampfeinheiten, spielt der ausgerichtete Nahkampfschaden die größte Rolle, denn die Fernkampfeinheiten können in diesem Fall ununterbrochen angreifen. Die Nahkampfeinheiten versuchen in diesem Fall einzelne Fernkampfeinheiten zu umzingeln, damit sie möglichst viel Angriffsfläche haben und maximalen Schaden ausrichten können. Im Gegensatz dazu versuchen die Fernkampfeinheiten durch kompakte Stellungen oder durch das Ausnutzen von Engstellen möglichst wenig Angriffsfläche zu bieten, damit im besten Fall nur wenige Nahkampfeinheiten überhaupt Schaden ausrichten können. (Liquipedia (2009b)).

Zudem kann der Höhenunterschied des Geländes, in dem sich die Einheiten befinden, eine Rolle spielen. Es kann für Einheiten in höherem Gelände leichter sein auf niedrigere Einheiten zu schießen und umgekehrt kann die Wahrscheinlichkeit zu treffen niedriger sein. Zum Beispiel ist in StarCraft auf Basis des Höhenunterschiedes eine unterschiedliche Wahrscheinlichkeit zu treffen festgelegt: für Einheiten, die auf höhere Einheiten zielen oder auf Einheiten, die hinter Hindernisse wie Bäumen versteckt sind, ist die Wahrscheinlichkeit zu treffen nur $\frac{136}{256} = 53{,}125\,\%$, ansonsten ist sie $\frac{255}{256} = 99{,}609375\,\%$ (Heinermann (2011)).

2.3 StarCraft: Brood War

StarCraft erschien im Jahre 1998. Im selben Jahr erschien auch Brood War, die einzige offizielle Erweiterung. Diese komplettierte die Einheiten des Spieles, indem sie jeder Rasse um eine Luft-zu-Luft-Kampfeinheit und eine neue Bodeneinheit erweiterte und so die Ausgeglichenheit der Rassen verbesserte. Durch weitere Updates perfektionierte der Hersteller Blizzard die Balance der Rassen, welche wohl der Hauptgrund für den Erfolg des Spieles wurde.

StarCraft ist ein herausragender Vertreter des Genres der Echtzeitstrategiespiel. Es ist nicht nur mit über 11 Millionen verkauften Einheiten das bisher meist verkaufte

Echtzeitstrategiespiel (Graft (2009)), sondern erfreute sich auch über 10 Jahre nach dem Erscheinen noch großer Beliebtheit. In Südkorea gibt es für StarCraft eine professionelle E-Sport-Szene mit hohen Preisgeldern, in der es selbst jetzt, zwei Jahre nach dem Erscheinen von StarCraft II, noch immer Ligen für StarCraft: Brood War gibt.

Die Story dreht sich um drei Rassen, die im 26. Jahrhundert um die Vorherrschaft im Koprulu-Sektor, einem entfernten Teil der Milchstraße, kämpfen: das sind zum einen die Terraner, menschliche Verbannte von der Erde; die Zerg, eine Rasse von insektoiden Aliens, die nach genetischer Perfektion streben, indem sie andere Rassen assimilieren; und die Protoss, eine humanoide Spezies mit fortgeschrittener Technologie und Psi-Fähigkeiten, die ihre Kultur vor den Zerg bewahren wollen.

Im weiteren Verlauf dieser Arbeit ist immer wenn von StarCraft die Rede ist, die aktuelle Version 1.16.1 inklusive der Erweiterung Brood War gemeint, da für diese auch die verwendete Programmierschnittstelle, sämtliche Bots und Turniere ausgelegt sind. Im Folgenden werden die wichtigsten Aspekte vom StarCraft: Brood War im Detail vorgestellt.

2.3.1 Balancing

Als Hauptgrund für den Erfolg von StarCraft wird im Allgemeinen die extrem gute Ausbalancierungen zwischen den drei ziemlich verschiedenen Rassen genannt. Während in anderen Spielen sich meist die Parteien sehr ähnlich sind und sich nur in relativ wenigen Punkten unterscheiden, hat Blizzard bei StarCraft das Kunststück hinbekommen, mit den Terraner, Zerg und Protoss drei Rassen zu erschaffen, die sich völlig unterschiedlich spielen, aber dennoch keine einen Vorteil hat. StarCraft gilt vielen Spielern als das best-ausbalancierte Echtzeitstrategiespiel überhaupt. Die Balance erstreckt sich jedoch nicht nur auf die drei Rassen, sondern auch auf andere Spielaspekte wie Strategie, Taktik, Macro- und Micromanagement. Es reicht nicht, sich auf einen der Aspekte zu konzentrieren, um ein guter Spieler zu werden, muss man all diese Dinge beherrschen. Die beste Strategie nützt nichts, wenn man nicht entsprechende Taktiken und Micromanagement hat, um sie umzusetzen. (mahnini (2009))

2.3.2 Rassen

In StarCraft gibt es mit den Terranern, Zerg und Protoss drei grundlegend verschiedene Rassen. Die Terraner sind menschliche Verbannte, die Zerg sind insektoide Aliens und die Protoss stellen eine hochentwickelte außerirdische Rasse mit Psi-Fähigkeiten dar.

2.3.2.1 Gebäude

Ein grundsätzlicher Unterschied zwischen den Rassen sind die Eigenschaften der Gebäude und wie und wo diese gebaut werden können:

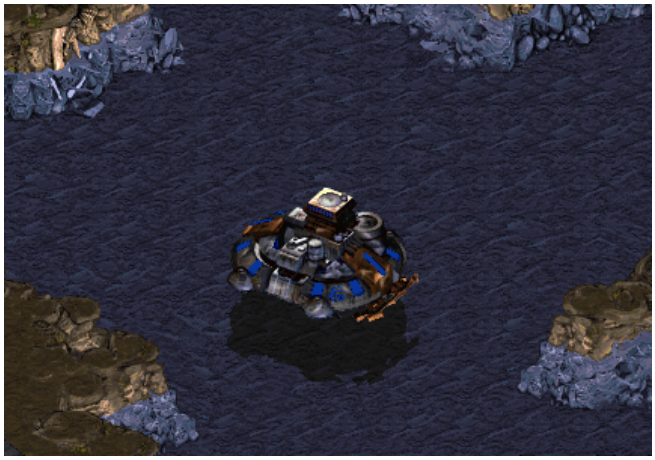
- Die ***Terraner*** können jedes Gebäude (außer Raffinerien, diese müssen auf Geysiren gebaut werden) überall in bebaubaren Gebiet platzieren. Ein Arbeiter muss während der Konstruktion am Gebäude verbleiben. Danach kann er wieder frei verwendet werden. Beschädigte Gebäude können von Arbeitern repariert werden. Stark beschädigte Gebäude beginnen zu brennen und verlieren dabei weitere Lebenspunkte bis sie zerstört sind, wenn sie nicht vorher repariert werden. Alle Produktionsgebäude, Engineering Bay und Science Facility können fliegen und an anderer Stelle wieder platziert werden. Terranische Gebäude blinken, wenn sie Einheiten produzieren oder upgraden. An bestimmte terranische Gebäude können Erweiterung angebaut werden, um ihre Fähigkeiten zu erweitern.

- Alle Gebäude der ***Zerg*** müssen auf dem sogenannten Kriecher errichtet werden. Ausgenommen hiervon sind nur neue Hauptquartiere (Hatchery) und Extraktoren, die Raffineriegebäude, welche auf Geysiren errichtet werden müssen. Fertige Gebäude führen dazu, dass sich der Kriecher weiter ausbreitet. Beim Bauvorgang morpht ein Arbeiter in das gewünschte Gebäude und ist daher danach nicht mehr anderweitig verwendbar. Verwundete Gebäude regnerieren wie alle Zergeinheiten langsam die Lebenspunkte. Zergschen Gebäuden sieht man nicht an, ob sie produzieren oder upgraden, nur Eier oder Gebäude beim Morphen zu einer höhren Stufe sieht man den Veränderungsprozess an.

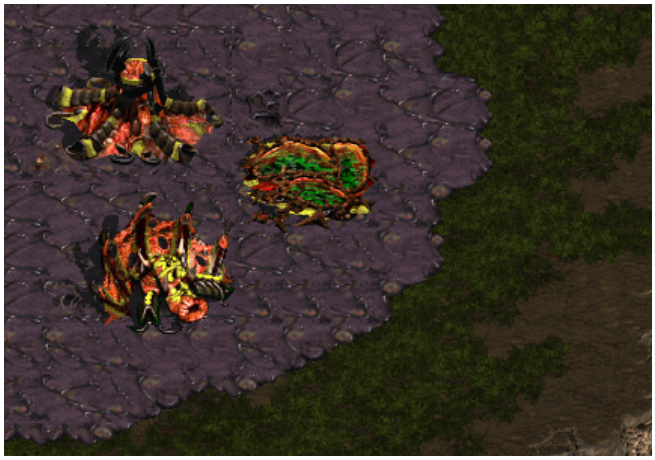
- Die Gebäude der ***Protoss*** müssen im Psi-Feld errichtet werden, dass Pylonen erzeugen. Ausgenommen davon sind nur das Hauptquartier (Nexus), die Raffinieriegebäude (Assimilatoren) und Pylonen selbst. Der Bauvorgang muss nur von einem Arbeiter initiiert werden. Sobald dieser am gewünschten Bauplatz gestartet wurde, kann der Arbeiter einer anderen Aufgabe zugewiesen werden, das Gebäude wird dann von selbst herangewarpt. Die Hälfte der Trefferpunkte der Protossgebäude sind Schilde, die sich wieder regenieren. Die andere Hälfte sind Lebenspunkte, die sich nicht regenieren oder von den Protoss repariert werden können.

2.3.2.2 Spielweise

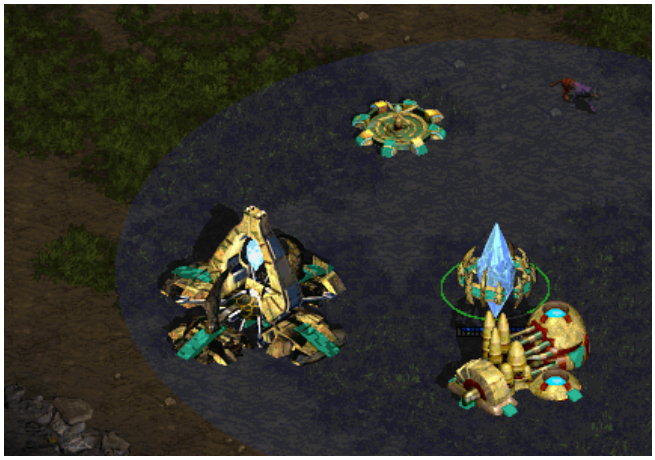
Weiterhin unterscheiden sich die Rassen in ihrer Spielweise. Dies ist besonders auf die Eigenschaften der Einheiten zurückzuführen. Verschiedene Rohstoffkosten und Beweglichkeit der Einheiten bedingen, dass die Armeegröße je nach Rasse signifikant



(a) Fliegende Kommandozentrale der Terraner



(b) Kriecher mit Zerggebäuden



(c) Protoss-Gebäude im von einem Pylonen erzeugten Psi-Feld

Abbildung 2.5: Gebäudebesonderheiten der einzelnen Rassen

unterschiedlich ausfällt, was notwendigerweise eine andere Taktik und Spielweise fordert. Im Folgenden werden die drei Rassen und ihre Spielweisen vorgestellt:

Terraner Die Terraner sind Menschen. Ihre Einheiten stellen futuristische Versionen von gepanzerten Soldaten, Panzern und Flugschiffen dar. Dementsprechend kommt man mit dieser Rasse zu Beginn meist am leichtesten zurecht. Sämtliche Einheiten und Gebäude können entweder von Arbeitern repariert oder von Medics geheilt werden. Bei den Terranern kommen dem Macromanagement und Timing große Bedeutung zu. Generell ist man auf der Suche nach Zeitfenstern, die einem der Gegner bietet, in denen man die größere Armee stellt und ihn dann attackieren kann. Terranische Einheiten werden als kosteneffektiv, aber relativ unbeweglich angesehen. Deshalb ist es wichtig, seine Einheiten zu reparieren und am Leben zu erhalten. Diese Micromanagement ist erforderlich, um die Terraner auf hohem Niveau spielen zu können. Im Vergleich zu den anderen Rassen spielt man mit einer mittleren Anzahl von Einheiten.

Zerg Die Zerg sind insektoide Aliens. Zerg gelten als die flexiblste Rasse und sind dementsprechend schwer für Anfänger zu spielen, weil man viel Erfahrung braucht, um richtig auf eine Situation zu reagieren, da kleinste Unterschiede große Auswirkungen haben können. Zergeinheiten sind relativ billig, deshalb hat man im Allgemeinen eine große Anzahl von ihnen. Sie sind zudem im Vergleich zu den anderen Rassen die schnellsten Einheiten, was für die Flexibilität wichtig ist. Zugleich haben sie im Gegensatz zu den anderen Rassen zur Einheitenproduktion nur einen einzigen Gebäudetyp und dessen zwei Ausbaustufen. Zergeinheiten und -gebäude regnerieren von selbst ihre Lebenspunkte.

Protoss Protoss sind humanoide, technologisch fortschrittliche Außerirdische, die zudem Psi-Fähigkeiten besitzen. Sie besitzen relativ teure Einheiten, deshalb befehligt man meist relativ wenige von diesen. Die Trefferpunkte der Protoss teilen sich in Lebenspunkte und Plasmaschilde auf. Die Plasmaschilde regnerieren sich selbst, die Lebenspunkte können nicht regneriert werden. Die Protoss sind in mancherlei Hinsicht die leichteste Rasse für Anfänger, da nicht so oft strategische Entscheidungen wie bei den Zerg und weniger mechanische Routine als bei den Terranern nötig ist. Hingegen sind durch die geringe Armeegröße taktische Fähigkeiten gefragt, um ein guter Protoss-Spieler zu werden. (mahnini (2009)).

2.3.3 Spielmechanik

In diesem Abschnitt werden einige Eigenschaften von StarCraft und Einheiten erläutert, um die Spielmechanik besser verstehen zu können.

Rohstoffe In StarCraft gibt es zwei Sorten von Rohstoffen: Mineralien und Vespingas. Mineralien können direkt in Mineralienfeldern von Arbeitern abgebaut werden und werden dem eigenen Konto gutgeschrieben, wenn diese zu einem Hauptgebäude gebracht worden sind. Eine solche Ladung beinhaltet hierbei 8 Mineralien. Ein Mineralienfeld beinhaltet zu Beginn eine gewisse Menge an Mineralien, in der Regel 1500. Ist es abgebaut, verschwindet es von der Karte und gibt den Platz frei. Die auf der Karte vorhandene Mineralienmenge ist also von Anfang an begrenzt. Dies spielt jedoch nur bei sehr langen Partien eine Rolle. Vespingas kommt in Geysiren vor. Um dieses abbauen zu können, muss ein Raffineriegebäude auf dem jeweiligen Geysir gebaut werden. Dann können die Arbeiter auch Gas in Kanistern mit je 8 Einheiten Gas aus dem Raffineriegebäude zum Hauptgebäude bringen, was dort dann dem eigenen Konto gutgeschrieben wird. Geysire enthalten auch nur eine bestimmte Menge an Vespingas, in der Regel 5000. Ist diese abgebaut, ist der Geysir erschöpft. Es kann trotzdem weiterhin Vespingas aus diesem gewonnen werden, dann jedoch nur in 2er-Einheiten und damit deutlich langsamer als zuvor.

Versorgung Ein weiterer limitierender Faktor für die Einheitenproduktion ist die sogenannte Versorgung. Jede Einheit verbraucht solange sie am Leben ist ein gewisses Maß an Versorgung – je größer und stärker sie ist, desto mehr. Wird die Einheit getötet, ist die Versorgung für andere Einheiten wieder frei. Neue Einheiten, die Versorgung benötigen, können nur gebaut werden, wenn genügend Versorgung für sie vorhanden ist. Neben dem Hauptgebäude, dass auch Versorgung zur Verfügung stellt, gibt es bei den Terranern und Protoss ein Gebäude und bei dem Zerg eine Einheit, die diese Aufgabe hat. Die Versorgung wird nur solange zur Verfügung gestellt, wie das Gebäude oder die Einheit am Leben ist. Werden zu viele versorgungsstellende Einheiten oder Gebäude zerstört, kann es sein, dass die zur Verfügung gestellte Versorgung unter die verbrauchte sinkt. Dadurch werden aber keine eigenen Einheiten zerstört, sondern es können nur keine neuen Einheiten mehr gebaut werden, bis wieder hinreichend Versorgung zur Verfügung steht. Pro Rasse kann ein Spieler maximal 200 Versorgung verbrauchen.

Einheitenarten In StarCraft gibt es Boden- und Lufteinheiten. Lufteinheiten können jeden Punkt auf der Karte erreichen und für sie wird auch keine Kollisionsabfrage durchgeführt. Jede Rasse hat eine Lufteinheit, die Bodeneinheiten transportieren kann. Während des Transports können die Bodeneinheiten nichts machen. Diese Einheit bzw. Fähigkeit wird erst von fortgeschrittenen Gebäuden zur Verfügung gestellt und ist noch nicht zu Beginn des Spieles verfügbar. Einige terranische Gebäude können auch abheben und fliegen.

Tarnen, Eingraben und Detektoren Weiterhin gibt es die Spezialfähigkeit der Tarnung bei Terranern und Protoss sowie das Eingraben bei den Zerg und von terranischens Spinnenminen. Getarnte und eingegrabene Einheiten sind für Gegner nicht

sichtbar und können nicht direkt angegriffen werden, Flächenschaden und verschiedene Spezialfähigkeiten können diese dennoch treffen. Um diese sehen und attackieren zu können, sind Einheiten mit Detektor-Fähigkeit nötig. Jede Rasse hat eine Lufteinheit und ein Verteidigungsgebäude mit dieser Fähigkeit. Zudem können die Terraner mit Hilfe der Scanner-Suche getarnte und eingegrabene Einheiten detektieren. Eingegrabenen Einheiten können sich weder bewegen noch angreifen, mit Ausnahme des Schleichers, der nur angreifen kann, wenn er eingegraben ist.

Schaden Es gibt Nah- und Fernkampfeinheiten. Alle Flugeinheiten, die angreifen können, sind Fernkampfeinheiten. Fernkampfeinheiten können nicht in allen Fällen sowohl Boden- als auch Lufteinheiten angreifen. Es gibt Fernkampfeinheiten, die nur eine Sorte von Einheiten angreifen kann. Normalerweise macht ein Angriff nur bei einer Zieleinheit Schaden. Aber es gibt auch Waffen, die Flächenschaden machen und damit alle Einheiten, sogar getarnte, in einem bestimmten Bereich treffen. Jede Waffe hat eine spezifische Abkühlphase, also eine gewisse Pause zwischen zwei Angriffen. Neben den konventionellen Angriffen gibt es noch Spezialfähigkeiten, die auf Gegner angewandt werden können. Diese können direkt Schaden ausrichten, oder die gegnerische Einheit auf andere Weise behindern, z.B. verlangsamen oder die Sicht nehmen. Spezialfähigkeiten – auch solche, die auf eigene Einheiten oder sich selbst angewandt werden – verbrauchen ein bestimmte Menge an Energie. Diese Energie ist unabhängig von Schilden oder Lebenspunkten, und regeneriert sich langsam mit der Zeit. Dies sorgt dafür, dass Spezialfähigkeiten mit Bedacht eingesetzt werden müssen.

2.3.4 Vermächtnis

StarCraft wird allgemein dafür gelobt, das Genre der Echtzeitstrategiespiele weiterentwickelt zu haben. Es ist das erster Spiel, dass auf drei fundamental verschiedene Rassen setzt, welche ausbalanciert sind und keiner Rasse einen Vorteil gibt (Chick (2000)). Es wurde von verschiedenen Magazinen zum Spiel des Jahres gewählt, war auf vielen Listen der besten Spiele aller Zeiten vertreten und wurde von Gamespot sogar zum besten Spiel aller Zeiten gewählt (Blizzard Entertainment (2006)).

Die U.C. Berkeley bot einen Kurs in StarCraft an (Cavalli (2009), Crecente (2009)), aus dem der Overmind-Bot hervorging, welcher die AIIDE 2010 StarCraft AI Competition gewann. Zudem nutzte die US Air Force im Aerospace Basic Course 1999-2000 StarCraft, um Krisenmanagement und Teamwork in Stresssituationen zu trainieren (AirUniversity (1999)).

Auch in der E-Sport-Szene war StarCraft bis zum Erscheinen des Nachfolgers StarCraft II eines der beliebtesten Spiele. So war StarCraft: Brood War von 2001 bis 2010 auf den World Cyber Games vertreten (World Cyber Games (2012)).

In Südkorea haben sich erfolgreich professionelle StarCraft-Ligen etabliert, die im Fernsehen übertragen werden. Dadurch haben die Spieler Berühmtheit erlangt und können durch TV- und Sponsoren-Verträge und Preisgeldern teilweise hohe Einnahmen verbuchen. Ein Spieler, Lee Yunyol, kam 2005 auf Einnahmen von ca. 200.000 $ (Rossignol (2006)). Erst eine Weile nach dem Erscheinen von StarCraft II wurden diese Ligen meist durch StarCraft-II-Ligen ersetzt, wobei es noch immer StarCraft-Brood-War-Ligen gibt.

Über das Internet kann fortwährend an Ladder-Wettbewerben teilgenommen werden. Der bekannteste ist iCCup (International Cyber Cup). Hierbei wird StarCraft mittels eines speziellen Launcher gestartet, um ein Anti-Hack-Plugins zu laden und weitere Funktion mittels dem Chaosplugin zur Verfügung zu stellen. Die Spieler erhalten Punkte basierend auf ihren Ergebnissen gegen andere Spieler und deren Stärke. Diese Statistiken werden automatisch nach jedem Spiel gespeichert und können auf der offiziellen ICCup-Webseite betrachtet werden.

Kapitel 3

Forschungskontext

Buro und Furtak (2003) riefen als Erste dazu auf, Echtzeitstrategiespiele als Objekt für die KI-Forschung zu nutzen. Einige Jahre lang konnten jedoch nur Open-Source-Spiele wie Wargus und Stratagus oder dem von Buro selbst entwickelten ORTS dafür genutzt werden. Diese hatten jedoch den Nachteil, dass sie gegenüber den kommerziellen Spielen nicht sehr ausgereift waren. Sie waren schlecht getestet und nicht gut balanciert. Die Herstellern kommerzieller Spielen stellten jedoch keine API für ihre Spiele zur Verfügung. Erst als 2009 einige Enthusiasten selbständig BWAPI, eine API für StarCraft: Brood War entwickelten, konnte mit einem sehr guten Spiel gearbeitet werden. Dies brachte die Forschung deutlich voran. Blizzard gab für die CIG 2010 dann auch seinen Segen dazu (Huang (2011)). Seither gibt es auch jährlich Turniere bei der CIG (IEEE Conference on Computational Intelligence and Games) und AIIDE (AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment). Zudem veranstaltete 2011 die Universität Bratislava zum ersten Mal ein Turnier für Studierende.

Bei der Forschung wurde sowohl auf bewährte Methoden der künstlichen Intelligenz zurückgegriffen, also auch neue entwickelt. Im ersten Abschnitt dieses Kapitels werden solche Methoden dargestellt. Danach folgt eine Vorstellung von einigen Bots und zum Abschluss folgen noch Informationen über die bisherigen Turniere.

3.1 Methoden der künstlichen Intelligenz

In diesem Abschnitt werden verschiedene Methoden der künstlichen Intelligenz vorgestellt, die bei Echtzeitstrategiespielen Anwendungen finden. Zuerst ist dies der endliche Zustandsautomat, der eine Grundlagentechnik nicht nur der künstlichen Intelligenz ist, sondern in den verschiedensten Bereich der Informatik eingesetzt wird. Dann werden Potentialfelder und deren Verwendung zur Bewegungssteuerung von Agenten präsentiert. Den Abschluss dieses Abschnittes bilden dann verschiedene andere Methoden, die speziell bei Echtzeitstrategiespielen Verwendung finden.

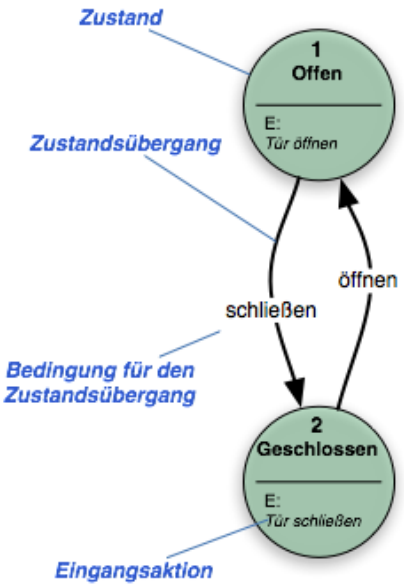


Abbildung 3.1: Zustandsautomat einer Tür

3.1.1 Endlicher Zustandsautomat

3.1.1.1 Allgemein

Ein endlicher Zustandsautomat besteht aus einer endlichen Menge von Zuständen, aus Zustandsübergangen und Aktionen. Er befindet sich immer in genau einem der Zustände. Ein Zustandsübergang besteht aus einem Ausgangszustand, einem Zielzustand und einer Übergangsbedingungen. Er wird genau dann vollzogen, wenn sich der Automat im Ausgangszustand befindet und die Eingabe die Übergangsbedingung erfüllt.

Es gibt vier verschiedene Sorten von Aktionen, die eine Ausgabe erzeugen:

Eingabeaktionen werden ausgeführt, wenn sich der Automat in einem bestimmten Zustand befindet und eine bestimmte Eingabe erhält

Eingangsaktionen werden ausgeführt, wenn der Automat in einen bestimmten Zustand wechselt

Ausgangsaktionen werden ausgeführt, wenn der Automat einen bestimmten Zustand verlässt

Übergangsaktionen werden ausgeführt, wenn ein bestimmter Zustandsübergang ausgeführt wird

Werden nur Eingangsaktionen verwendet, nennt man den Zustandsautomaten einen ***Moore-Automaten***. Die Ausgabe hängt in diesem Fall nur vom Zustand ab. Ver-

wendet ein Zustandsautomat Eingabeaktionen, so nennt man ihn einen ***Mealy-Automaten***. Die Ausgabe hängt bei diesen also von Eingabe und Zustand ab. Beide Automatentypen sind gleichwertig und können in jeweils den anderen mit gleicher Funktionalität überführt werden.

Beim Dreadbot ist der `CombatMan` als endlicher Zustandsautomat mit 6 Zuständen implementiert, siehe Abschnitt 5.6.1.

3.1.1.2 Theorie

Das mathematische Modell eines Zustandsautomaten ist ein 7-Tupel $(\Sigma, \Gamma, S, s_0, \delta, \omega, F)$ mit:

- Σ ist das Eingabealphabet
- Γ ist das Ausgabealphabet
- $S \neq \emptyset$ ist ene endliche Menge von Zuständen
- $s_0 \in S$ ist der Anfangszustand
- $\delta : S \times \Sigma \to S$ ist die Zustandsübergangsfunktion
- ω ist die Ausgabefunktion. Bei einem Moore-Automaten ist $\omega : S \to \Gamma$, bei einem Mealy-Automaten $\omega : S \times \Sigma \to \Gamma$.
- $F \subset S$ ist die Menge von Endzuständen (kann leer sein)

Ist für eine Eingabe und einen Zustand mehr als ein Zustandsübergang definiert, so handelt es sich um einen nicht-deterministischen Automaten. Dann wird willkürlich einer dieser Zustandsübergange ausgewählt. Jedoch kann jeder nicht-deterministische Automat in einen äquivalenten deterministischen Automaten mittels der Potenzmengenkonstruktion umgewandelt werden.

Einen endlicher Zustandsautomat mit nur einem Zustand nennt man kombinatorischen Zustandsautomaten. Er kann folglich nur Eingabeaktionen ausführen.

Erzeugt ein Zustandsautomat keine Ausgabe, also ist das Ausgabealphabet leer und keine Ausgabefunktion definiert, so nennt man den Automaten einen ***Akzeptor***. Dieser zeigt nur durch den Zustand an, ob er die Eingabe akzeptiert. Andernfalls nennt man den Automaten ***Transduktor***.

3.1.1.3 Vor- & Nachteile

Der Vorteil von endlichen Zustandsautomaten ist deren Übersichtlichkeit. Jeder Zustand ist klar definiert und die Übergänge sind eindeutig. Somit kann durch den aktuellen Zustand und die Eingabe der Folgezustand sicher bestimmt werden. Dadurch lassen sich endliche Zustandsautomaten leicht debuggen. Ihr Nachteil ist die daraus resultierende Starrheit. Es können eben nur vorher definierte Zustände angenommen und vorher definierte Übergänge angewendet werden. Dadurch kann nicht dynamisch auf sich neu ergebene Situationen reagiert werden. Wurde eine Situation vorher nicht bedacht und kein angemessener Zustand und/oder Übergang für sie definiert, kann unter Umständen nicht angemessen auf sie reagiert werden und der Zustandsautomat feststecken.

3.1.2 Potentialfelder

3.1.2.1 Pfadplanung

Bei Echtzeitstrategiespielen besteht eine Herausforderung darin, Einheiten durch ein sich dynamisch veränderndes Gebiet zu lotsen. Dynamisch verändernd insofern, dass sich eigene und gegnerische Einheiten bewegen und Gebäude gebaut werden. Man darf nicht die eigenen Einheiten blockieren bzw. muss um sie herum während sie andere Aufgaben ausführen, z.B. Kampfeinheiten müssen vorbei an Arbeitern. Zudem sollen, je nach Spielsituation, entweder gegnerische Einheiten gemieden werden oder man sich optimal um diese positionieren, um sie bekämpfen zu können.

Traditionell wird der A*-Algorithmus zur Planung von Pfaden verwendet. Dieser ist eine Erweiterung des Dijkstra-Algorithmuses. Er ist *vollständig*, *optimal* und *optimal effizient*. *Vollständig* bedeutet, dass falls seine Lösung existiert, diese auch gefunden wird. *Optimal* bedeutet, dass immer eine optimale Lösung, also ein kürzester Pfad, gefunden wird. *Optimal effizient* bedeutet, dass kein anderer Algorithmus die Lösung mit der selben Heuristik schneller findet. (Russell und Norvig (2004),Wikipedia (2012))

Die dynamisch ändernde Spielwelt macht es jedoch nötig, den Pfad nach jeder Änderung neuzuberechnen. Dabei kann der A*-Algorithmus Deadlocks produzieren, falls Einheiten in die selbe Richtung ausweichen und sich dann gegenseitig blockieren. Hier sind Potentialfelder geeigneter. Bei ihnen muss nicht der komplette Pfad im Voraus berechnet werden, sondern es wird jeweils nur der nächste Schritt berechnet. Dadurch werden keine Ressourcen auf die Berechnung eines Pfades verschwendet, der durch Änderungen in der Spielwelt obsolet wird. Zudem kann mit Potentialfeldern durch Änderungen der Feldstärke recht einfach komplexes Verhalten erzeugt werden, wie z.B. das Formieren der eigenen um die gegnerische Einheiten in der maximalen Waffenreichweite und das automatische Zurückziehen in der Abkühlphase der Waffe.

Die Idee hinter Potentialfeldern ist eine Ladung an einer Stellung zu platzieren, die ein Feld erzeugt, das mit dem Abstand geringer wird. Hierbei muss der Feldstärkeabfall nicht linear mit dem Abstand sein, sondern es können beliebige Funktionen dafür verwendet werden. An die Zielposition setzt man üblicherweise eine positive Ladung, die die eigenen Einheiten anzieht. In jedem Schritt wird dann in Richtung der höchsten Feldstärke gegangen. Um an Hindernissen vorbeizukommen, werden diese mit einem negativen, abstoßenden Feld belegt. Dadurch wird an den Hindernissen vorbeinavigiert und die Einheit gelangt ans Ziel. Ebenso kann man andere eigenen Einheiten mit einem abstoßenden Feld belegen, damit die Einheiten einen gewissen Abstand zueinander halten und sich nicht gegenseitig blockieren. Abbildung 3.2 zeigt das an einem Beispiel.

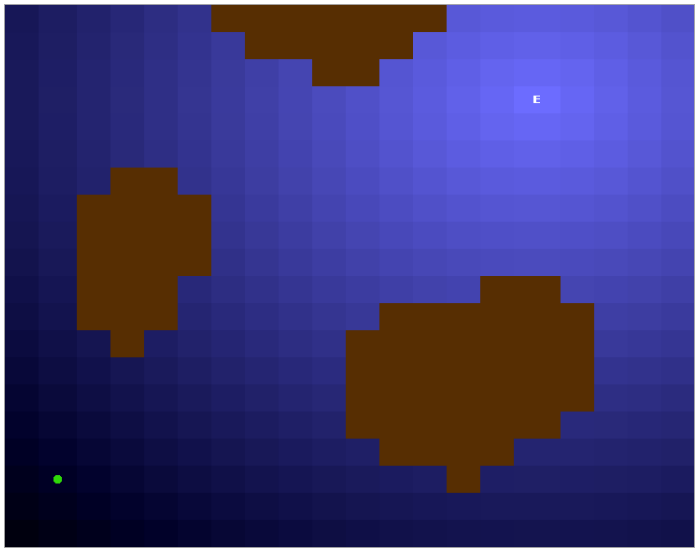
3.1.2.2 Komplexers Verhalten

Des Weiteren kann man mit Potentialfeldern auch komplexeres Verhalten erzeugen. Hat man Fernkampfeinheiten, z.B. Panzer, möchte man, dass die Einheiten nicht zu nahe an den Gegner heranfahren, sondern ihn in einem gewissen Abstand, z.B. der maximalen Reichweite der eigenen Waffen, umstellen. Dazu erzeugt man ein Feld, dass von der gegnerischen Einheit aus ansteigt, seinen Maximalwert im Abstand der eigenen Waffereichweite von der gegnerischen Einheit hat, und danach steil abfällt. Dann sammeln sich die eigenen Einheiten in diesem Maximalwert und sind in optimaler Distanz zur gegnerischen Einheit.

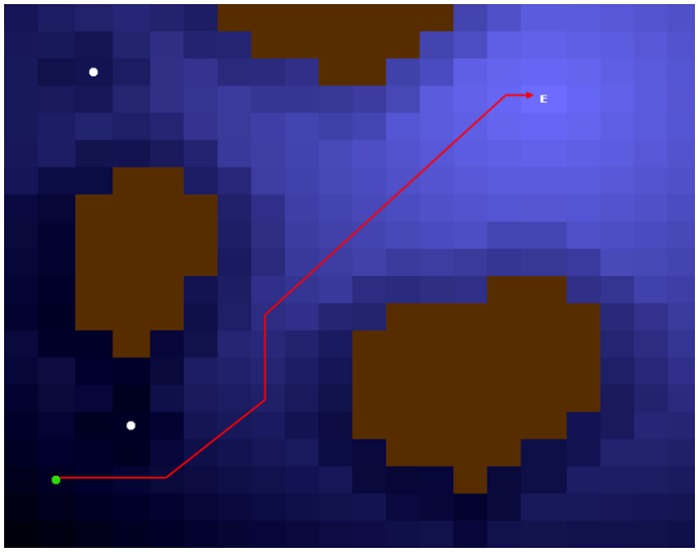
Ein weiteres Verhalten, dass sich mit Potentialfeldern erzeugen lässt, ist das automatische Zurückziehen in der Abkühlphase der Waffe und dass Hinbewegen zum Ziel, wenn die Waffe feuerbereit ist. Dazu legt man in der Abkühlphase ein abstoßendes Feld in die gegnerische Einheit und ein anziehendes, wenn die Waffe wieder feuerbereit ist, wie im Abschnitt vorher beschrieben. Damit erzeugt man professionelle Dancing-Manöver. Beim Overmind-Bot wurde dieses Verhalten mit den Mutalisken sehr gut umgesetzt, siehe Abschnitt 3.2.1.

3.1.2.3 Nachteile

Potentialfelder haben jedoch auch Nachteile: Es kann passieren, das Einheiten in einem lokalen Optimum stecken bleiben und sich nicht weiter Richtung des eigentlichen Ziels bewegen. Dies löst man am besten damit, dass man eine Art „Pheromonspur“ hinter der Einheit herzieht. Dazu werden die letzten Positionen, die die Einheit belegt hat, gespeichert und mit einer abstoßenden Ladung belegt. Dadurch wird die Einheit gezwungen, sich zu bewegen und kann dadurch aus dem lokalen Optimum herauskommen. Zudem kann man Hindernisse mit Einbuchtungen konvex ausfüllen, damit die Einheiten nicht in diesen Buchten stecken bleibt. In sehr komplexen Gelände kann es dennoch passieren, dass die Einheit stecken bleibt. Dort sind dann traditionelle Pfadsuchealgorithme die bessere Lösung.



(a) Potentialfeld mit einer Quelle (E): Je heller das Blau, desto mehr zieht es die Einheit (grüner Punkt) an.



(b) Um statische Hindernisse (braun) wie unpassierbares Gelände und dynamische Hindernisse (weiße Punkte) wie eigene Einheiten wurde ein abstoßendes Feld (dunkel) gelegt, so dass ein Pfad (rot) zwischen den Hindernissen hindurch zum Ziel gefunden wird.

Abbildung 3.2: Navigation mit einem Potentialfeld

Ein weiteres Problem ist das Finden der optimalen Feldparameter. Dies muss oftmals von Hand vorgenommen werden. Dazu kann man sich an den Einheiteneigenschaften orientieren und ihnen gewisse Prioritäten zuweisen. Eine Visualisierung des Feldes kann helfen, je mehr Quellen jedoch vorhanden sind, desto schwieriger wird es, den Einfluss der einzelnen Felder zu erkennen und anpassen zu können. Eine bessere Methode hierfür sind evolutionäre Algorithmen. Bei der Programmierung des Overmind wurde diese Methode angewendet und hat zu sehr guten Ergebnissen geführt.

Oftmals wird auch die Performance von Potentialfeldern kritisiert. Diese kann verbessert werden, indem man das statische Feld, dass sich aus dem bekannten und unveränderliche Gelände ergibt, vorher erstellt. Berechnet man zudem das Feld nur an den benötigten Orten, also an allen Positionen, die überhaupt als nächster Schritt in Frage kommen, kann die Performance nochmals gesteigert werden. Zudem kann auch zunächst über die schnell zu ermittelnde Manhattan-Distanz abgeschätzt werden, ob ein Feld für eine Einheit überhaupt berechnet werden muss. Damit kann man oftmals die relativ teure Kalkulation der euklidischen Distanz umgehen. Außerdem kann man die Berechnung auf mehrere Frames aufteilen, falls es die Spieldynamik erlaubt. Insgesamt kann man damit auf jeden Fall eine brauchbare Performance erzielen, und somit sind Potentialfelder eine praktikable Technik für Echtzeitstrategiespiele. (Hagelbäck (2009), Hagelbäck (2012))

3.1.2.4 Scheitern der Anwendung in dieser Arbeit

Potentialfelder sollten auch im Rahmen dieser Arbeit Verwendung findet. Sie sollten das Verhalten in Kampfsituationen steuern. Hierbei wurde angestrebt, dass Einheiten Platz machen, damit weitere Einheiten in Waffenreichweite kommen können und somit mehr Einheiten den Gegner attackieren können. Zudem sollten sie bei der Navigation behilflich sein. Jedoch konnte keine zufriedenstellende Implementierung erreicht werden. Das Platz machen funktionierte gar nicht, dabei blockierten sich die Einheiten gegenseitig. Zudem blieben sie bei der Navigation bei Hindernissen stecken. Das Lokal-Optima-Problem konnte hier nicht gelöst werden. Der Hauptgrund für das Scheitern war das Fehlen geeigeneter Debugging- und Tuningtools. Das Suchen der Parameter von Hand wäre sehr zeitaufwändig gewesen. Deswegen wurde auf die Potentialfelder verzichtet und sich dafür auf die anderen Teile der Arbeit konzentriert.

3.1.3 Weitere Forschungen

Im Folgenden werden noch weitere Forschungen im Bereich der Echtzeitstrategiespiele und insbesondere StarCraft vorgestellt:

Synnaeve und Bessière nutzen ein ***Bayes-Modell*** zur Erkennung der Eröffnungsstrategie sowie zur Erkennung der Baustrategie des Gegners, wobei die Parameter jeweils machinell aus Replays gelernt wurden (Synnaeve und Bessière (2011a), Syn-

naeve u. a. (2011)). Ein solches Modell wurde außerdem zur Einheitensteuerung beim Micromanagement (Synnaeve und Bessière (2011b)) und für Taktikerkennung und -findung (Synnaeve und Bessière (2012)) verwendet. Auch Hostetler u. a. (2012) verwenden ein Bayes-Modell zur Strategieerkennung und Vorhersage der Anzahl einer bestimmten Einheit des Gegners.

Ein weiteres Forschungsfeld ist die ***Optimierung der Build Order***. Kovarsky und Buro (2006) waren die Ersten, die sich mit diesem Thema befassten und die Forschung dazu angestoßen haben. Weber und Mateas verwendeten hierfür Techniken des fallbasierten Schließens (engl. case-based reasoning, CBR) (Weber und Mateas (2009a), Weber und Mateas (2009b)). Chan u.a. entwickelten einen Online-Planer, der mit menschlichen Spielern mithalten kann (Chan u. a. (2007b), Chan u. a. (2007a)).

Expertenwissen wird am besten über ***Data Mining*** gewonnen. Es stehen im Internet sehr viele Replays, Wiederholungen von Spielen, bereit, auch aus professionellen koreanischen StarCraft-Ligen, die analysiert werden können. Das Replay-Format ist bekannt und enthält u.a. alle Aktionen, die ausgeführt wurden, wodurch sich das ganze Spiel rekonstruieren lässt. Mit diesen lässt sich effektiv Data Mining betreiben, um erfolgsversprechende Strategien, BuildOrders und Taktiken zu ermitteln, wie die Arbeiten von Weber und Mateas (2009c), Weber und Ontañón (2010) und Dereszynski u. a. (2011) zeigen.

Genetische und Evolutionäre Algorithmen können auch verwendet werden, um eine Spielstrategie zu entwickeln. Louis u. a. (2004) verwenden hierbei genetische Algorithmen, in denen bekannte Fälle injiziert werden. Das hat den Vorteil, dass dadurch Expertenwissen, das diesen Fällen zugrunde liegt, mitgenutzt wird und zu besseren Ergebnissen führt. Weiterhin konnten sie zeigen, dass sich dadurch auch die Berechnungs- und Reaktionszeiten verbessern lassen (Miles und Louis (2005)). In weiteren Arbeiten wurde Co-Evolution von Spielern gegeneinander, basierend auf Bäumen von Einflusskarten, genutzt, um Strategien zu erzeugen, die robust, innovativ und erfolgreich sind (Miles und Louis (2006), Miles u. a. (2007), Miles (2007)). Lin und Ting (2011) verwendeten genetische Algorithmen, um geeignete Taktik-Formationen zu generieren, Oberberger (2010) nutzte hierfür Co-Evolution, um besser für unbekannte Situationen gerüstet zu sein.

3.2 Bots

Im Folgenden werden einige Bots vorgestellt, die auf Grund ihrer Eigenschaften oder der über sie verfügbaren Informationen sie erwähnenswert machen. Als erstes wird der Overmind vorgestellt, der die AIIDE 2010 gewann und an der Universität Berkeley in einem AI-Kurs erstellt wurde. Es folgt Nova, der als Masterarbeit der Universität Barcelona programmiert wurde. Danach werden noch verschiedene weitere Bots vorgestellt, unter anderem Skynet, der aktuell beste Bot.

Abbildung 3.3: Der Mutalisken-Schwarm des Overminds (rechts) attackiert eine gegnerische Basis

3.2.1 Overmind

Der Overmind-Bot wurde an der Universität Berkeley entwickelt. Prof. Dan Klein hielt dazu 2010 einen Kurs, der KI-Konzepte durch die Entwicklung dieses Bots vermittelte. Dort arbeiteten sowohl Doktoranden und Graduate Students als auch Undergraduate Students mit. Daraus rekrutierte sich ein Team mit 12 Mitgliedern, in dem jedes für einen spezifischen Teil des Bots zuständig war. Mit Oriol Vinyal war auch ein ehemaliger World-Cyber-Games-Teilnehmer darunter, der auch zeitweise die Nr. 1 in Spanien und Nr. 16 in Europa war. Letztendlich gelang es dem Bot, ihn zu besiegen.

Der Bot setzt nicht auf festgelegte Eröffnungsstrategien, da diese sich als nicht flexibel genug erwiesen. Stattdessen wird ein Bauplaner verwendet, der an einen Ressourcenplaner eines Betriebssystems erinnert. Verschiedene Aufgaben fordern Ressourcen wie Gas, Mineralien oder Einheiten an und bekommen diese je nach Priorität zugewiesen. Anstelle festgelegter, starrer Baureihenfolgen wurden verschiedene Kombinationen von Einheiten und Gebäuden bestimmt, und der Planer entscheidet je nach den Gegebenheiten, welche dieser Zielkombinationen angestrebt wird. Letztendlich ähnelten die Baureihenfolgen, die der Planer entwickelte, Baustrategien von StarCraft-Experten. Jedoch konnte so leichter von diesen abgewichen werden und alternative Lösungen gefunden werden, wenn es die Situation erfordert, als wenn die Baureihenfolgen starr festgelegt wären.

Die Gruppe entschied sich nur auf eine Sorte von Kampfeinheiten zu setzen, da dies die Komplexität der Kampfsteuerung erheblich reduziert. Es wurde sich für Mutalisken, eine Flugeinheit der Zerg entschieden. Diese sind vergleichsweise billig für ihre Stärke, sehr beweglich und können sowohl Boden- als auch Lufteinheiten angreifen.

Jedoch ist eine große Anzahl von ihnen schwer zu steuern für Menschen ohne dass diese eng beieinander sind, was sie sehr anfällig für Attacken mit Flächenschaden macht. Eine Computer-KI kann das jedoch besser hinbekommen. Dazu nutzte das Team Potentialfelder, vgl. 3.1.2. Hierbei haben Flugeinheiten den Vorteil, dass sie von keinerlei Gelände und Gebäuden behindert werden und sowie für sie keine Kollisionsabfrage durchgeführt wird, sie also durcheinander durch fliegen können. Die geeigenete Feldstärken für die Dancing-Manöver, wurden mittels eines evolutionären Algorithmuses in Testkämpfen ermittelt. Für die Wahl der Angriffsziele wurde mittels des DPS-Werts eine Art Erwartungswert berechnet. Wie viel Schaden eine Einheit einer anderen zufügt ist bekannt und lässt sich über BWAPI ermitteln. Aufgrund dessen lässt sich abschätzen, wie viel Schaden die eigenen Einheiten nehmen, um die gegnerischen Einheiten zu besiegen. Indem man jeder Einheit nun einen Wert basierend auf den Ressourcenkosten zuweist, konnten so lohnende und vorrangige Ziele berechnet werden.

Da die eingebaute Routenplanung der Einheiten des öfteren zu Problemen führte, z.B. dass Einheiten stecken blieben, implementierte das Team seinen eigenen Wegfindungsalgorithmus. Diesen Algorithmus kombinierten sie mit einer Gefahrenkarte, die sich aus den Positionen, Geschwindigkeiten und Waffenradien aller bekannten gegnerischen Einheiten ergaben. So konnten die Kosten eines Weges bestimmt werden und es wurden statt kurzer gefährlicher Wege lieber lange und sichere bevorzugt. Dadurch konnten die Overlords des Overminds optimal auf der Karte platziert werden, um möglichst große Bereiche der Karte sichtbar zu halten bei geringer Gefahr sie zu verlieren. Ebenso konnten mit diese Methode dynamisch diejenigen Overlords hergeholt werden, die den kürzesten sicheren Pfad haben, wenn gegnerische getarnte Einheiten detektiert werden mussten.

Mit Hilfe dieser Techniken und einem Finetuning durch Spiele gegen den ehemaligen Pro-Gamer Oriol Vinyal konnte der Bot das Fullgame-Turnier der AIIDE 2010 gewinnen. In dem Turnier, das im Double-KO-Modus in Best-of-5-Serien ausgetragen wurde, verlor der Overmind nur ein einziges Spiel – gegen Skynet. Trotz dieses Erfolges wurde der Bot nicht mehr weiterentwickelt und wurde auch bei keinen Turnieren mehr eingereicht. Dennoch belegt der Overmind bei der Botladder Platz 5 im Elo-Rating hinter den beiden Skynet- und den beiden UAlbertaBot-Versionen und knapp vor den beiden AIUR-Versionen. Nach Siegprozenten liegt er jedoch knapp hinter den AIURs. Es zeigt sich also, dass der Overmind trotz seines „Alters“ immer noch ein sehr guter Bot ist, mit dem nur drei Bots mithalten können. (Huang (2011), Klein (2010), Krastev (2012))

3.2.2 Nova

Der Bot „Nova“ wurde im Rahmen einer Masterarbeit mit dem Titel „Multi-Reactive Planning for Real-Time Strategy Games“ an der Universitat Autònoma de Barcelona

von Alberto Uriarte Pérez entwickelt. Die Arbeit wurde betreut von Santiago Ontañón Villar, der selbst Papers über StarCraft- und RTS-AIs veröffentlicht hat.

Der Bot wurde mittels eines Multi-Agenten-Systems implementiert. Dabei wurden zwei Typen von Agenten verwendet: Manager, die sich um ständige Aufgaben kümmern wie den BuildManager und von denen man nur eine Instanz benötigt, sowie normale Agenten, die einzelne Kampfeinheiten oder Kampfverbände repräsentieren. Durch die Verwendung eines Multi-Agenten-Systems können verschiedene Probleme parallel bearbeitet werden. Die Kommunikation zwischen den Agenten findet mittels einer Blackboard-Architektur statt. Dazu wurde ein Informationsmanager implementiert, der Blackboard und Working Memory Funktionalität hat. Das Blackboard speichert alle Ziele der Agenten und das Working Memory speichert alle Informationen über den Spielzustand.

Nova verwendet eine einfache Modellierung des Gegners, um Rushes zu bemerken. Zudem berechnet er eine Gefahrenkarte mit Hilfe des Schadens pro Sekunde, den die gegnerische Armee anrichten kann. Mit Hilfe derer wurde das Potentialfeld für die Einheitenbewegung erstellt. Zudem wurde eine Mechanik implementiert, die dafür sorgt, dass Kampfverbände zusammen bleiben. Dazu wird die minimale, maximale und aktuelle Ausbreitung der Einheiten berechnet. Ist die aktuelle Ausbreitung größer als der maximale, werden die Einheiten zusammengezogen, ist er kleiner als der minimale kann sich weiter Richtung Ziel des Kampfverbandes bewegt werden. Für die Auswahl des besten Ziels wurde eine Heuristik basierend auf dem Schaden pro Sekunde, der angerichtet werden kann, dem taktischen Wert der Einheit sowie deren Entfernung, verwendet. Der taktische Wert von Einheitentypen wurde vorher festgelegt. Des Weiteren wurden spezielle Micromanagement-Controller für verschiedene Einheitentypen implementiert:

Marine	Nutzung der Stim-Pack-Technologie
Marine, Firebat, Ghost	Heilung durch Medics
Wraith	Aktivieren der Tarnung
Tank	Wechseln in den Belagerungsmodus und zurück
Ghost	Tarnung und Nutzung der Lockdown-Technologie
Science Vessel	Herbeirufen als Detektor
Vulture	Ausnutzen der hohen Geschwindigkeit der Einheit als taktischen Vorteil gegenüber Nahkampfeinheiten sowie Platzierung von Spider Mines

Der BuildManager wurde als FIFO-Prioritätswarteschlange implementiert. Dabei haben Gebäude Priorität vor dem Erforschen von Uprades, welche wiederum Vorrang vor der Produktion von Kampfeinheiten haben. Die Platzierung von Gebäuden erfolgt spiralförmig, wobei keine Gebäude an die untere und rechte Seite von Produktionsgebäuden angrenzen dürfen, damit die produzierten Einheiten dort erscheinen können.

Abbildung 3.4: Visualisierung der Ausbreitungsgrenzen mit Hilfe derer Nova seine Kampfverbände zusammenhält.

Weiterhin wurden feste Vorgehensweisen implementiert, um auf *Gas Steals*[1], gegnerische Scouts, Lufteinheiten sowie getarnte Einheiten zu reagieren. Zudem werden die möglichen BaseLocations mittels der Scanner-Suche von Kommandozentralen mit ComSat-Stationen erforscht, um gegnerische Expansionen zu finden.

Zudem wurden einfache Strategien für jede Gegnerrasse implementiert. Diese wurden über endliche Zustandsautomaten verwirklicht. Gegen Terraner wird auf Tanks mit Siege Mode gesetzt. Zudem werden Vultures gebaut, wenn ein gegnerischer Marine Rush bemerkt wird. Gegen Protoss wird zuerst auf Vultures gesetzt, um gegen einen Zealot Rush gewappnet zu sein sowie um Spider Mines gegen Dragoons zu legen. Dann wird der Panzer-Anteil in der Armee schrittweise erhöht bis letztlich nur noch Tanks gebaut werden. Gegen Zerg wird auf die Standarderöffnung *1 Barracks Fast Expansion* gesetzt. Hierbei wird nach dem Bauen der ersten Kaserne (engl. Barracks) schon expandiert, außer der Zergspieler spielt einen Rush. Die Armee besteht dann aus 80 % Marines und 20 % Medics. Damit wird auch auf die Stim-Pack-Technologie gesetzt. Gegen einen Zergling-Rush werden auch Arbeiter zur Verteidigung eingesetzt.

Der Bot konnte folgende Ergebnisse erzielen: Bei der AIIDE 2011 erreichte er den 8. Platz bei 13 Teilnehmern, wobei er der beste Bot in Spielen Terraner gegen Terraner

[1]Bei einem *Gas Steal* baut man in der gegnerischen Basis ein Raffineriegebäude auf dem Geysir um ihn zu blockieren. Das wird typischerweise in der Anfangsphase des Spieles gemacht, um die Gasförderung des Gegners zu verzögern.

mit 91 % Siege war. Bei der CIG 2011 schied er in der Vorrunde mit 8 Siegen in 40 Spielen aus. Jedoch war er da mit Skynet, AIUR und UAlbertaBot in einer Gruppe. Bei der Bot-Ladder belegte er zum Zeitpunkt der Erstellung dieser Arbeit den 12. Platz von 34 Bots mit 42.85 % Siege. (Uriarte (2011))

3.2.3 Skynet

Skynet ist der aktuell beste Bot. Jedoch ist über ihn und seinen Programmierer Andrew Smith ist fast nichts bekannt. Er ist auch keinerlei Universität oder sonstigen Einreichtung zugeordnet. Der unkommentierte Quellcode von Skynet 2.0 ist jedoch unter `http://code.google.com/p/skynetbot/` verfügbar. Beim StarCraft-Turnier der AIIDE 2010 war seine Beschreibung „A Protoss bot with a few strategies under its belt“ (Weber (2010a)), auf der CIG 2011 wurde er folgendermaßen beschrieben:

> „Skynets main features include:
>
> - A fast custom terrain analyser.
> - An advanced building placer that creates tight but (mostly) non blocking bases.
> - A task based macro system that continually plans and fully understands all requirements.“ (Mahlmann und Preuss (2011a))

Bei der AIIDE 2010 schied er noch gegen den späteren Finalisten krasi0 (0-3) und den späteren Sieger Overmind (1-3) aus, war dabei jedoch der einzige Bot, der ein Spiel gegen den Overmind gewinnen konnte (Weber (2010b). Die Turniere bei der AIIDE und der CIG in den Jahren 2011 und 2012 gewann er dann allesamt. Bei der AIIDE 2011 konnte er dabei 320 von 360 Spielen (88,9 %) gewinnen (Buro (2011)). Bei der CIG 2011 überstand er die Vorrunde mit 31 Siegen in 40 Spielen (77,5 %) nur auf Platz 2 in seiner Gruppe hinter UAlbertaBot, in der Finalrunde erzielte er 26 Siege in 30 Spielen (86.7 %) und gewann damit das Turnier. Insgesamt erreichte er also 57 Siege in 70 Spielen (81,4 %) (Mahlmann und Preuss (2011b), Mahlmann und Preuss (2011a)). Bei der AIIDE 2012 gewann der 1397 von 1656 Spielen (84,4 %). Bei der CIG 2012 gewann er 78,7 % der 810 absolvierten Spielen. Zudem führen die aktuelle Skynet-Version (87,99 % Siege) vor der Skynet-Version der AIIDE 2011 (75,16 %) die Bot-Ladder an Krastev (2012).

3.2.4 AIUR

Florian Richoux von Universität Nantes hat den Bot AIUR programmiert, wobei dies als Abkürzung für „Artificial Intelligence Using Randomness“ steht. Die Hauptidee hinter diesem Bot ist es durch zufällige Entscheidungen die Vorhersagbarkeit

der Aktionen zu verhindern. Der Bot befindet sich zu Beginn zufällig in einer bestimmten „Stimmung“, die einge Entscheidungen des Macromanagements bestimmt. Zustätzlich werden noch andere Entscheidungen wie Produktionen, Angriffszeitpunkte und Verhalten in der Anfangsphase des Spieles zufällig getroffen. Die Stimmung kann in eine defensivere gewechselt werden, wenn ein Rush des Gegners bemerkt wird. Dann werden fortgeschrittene Gebäude abgebrochen und stattdessen schnellstmöglich eine Armee zur Verteidigung aufgebaut. Zudem wird gespeichert, welche Stimmungen gegen welchen Gegner wie abgeschnitten haben und nach 20 Spielen wird die Wahrscheinlichkeitsverteilung dieser geändert, um eine größere Chance zu haben, eine erfolgsversprechende Stimmung auszuwählen. AIUR gehört zu den besten Bots zur Zeit und belegte 2011 bei AIIDE den 3. und bei CIG den 5. Platz, 2012 erreichte er bei AIIDE den 2. Platz und bei CIG den 3. Platz. (Richoux (2012))

3.2.5 EISBot

Ben Weber von der University of California, Santa Cruz, der über Build-Order-Optimierung, Data Mining und den Einbau von Expertenwissen in StarCraft promoviert hat (3.1.3), hat den EISBot programmiert. Er nahm nur an der AIIDE 2011 teil und erreicht mit 218 Siegen in 360 Spielen (60,6 %) den 5. Platz von 13 Teilnehmern.

3.2.6 BroodwarBotQ

Gabriel Synnaeve (INRIA Grenoble/Collège de France), der über die Anwendung von Bayes-Modellen in StarCraft forscht (s. 3.1.3), ist der Autor von BroodwarBotQ. Seine Abschneiden ist bis auf der CIG 2011 eher mäßig. Dort erreichte er die Finalrunde, in der er aber nur ein Spiel von 30 gewinnen konnte, womit er insgesamt auf Platz 4 landete. In der Vorrunde gewann er noch 23 von 40 Spielen (57,5 %). Bei der AIIDE 2010 schied er schon recht früh aus, bei der AIIDE 2011 reichte es nur zu Platz 9 von 13 mit 32,8 % gewonnen Spielen. Bei der AIIDE 2012 gelang immerhin ein 5. Platz bei 10 Teilnehmern mit 59,1 % Siegen, während bei der CIG 2012 nur Platz 7 von 10 mit 47,9 % Siegen heraussprang.

3.2.7 BTHAI

Dr. Johan Hagelbäck (Blekinge Institute of Technology), der seine Dissertation u.a. über die Verwendung von Potentialfeldern zur Einheitennavigation geschrieben hat (siehe 3.1.2, Hagelbäck (2012)), hat den BTHAI-Bot geschrieben. Dieser ist für jede Rasse geeignet. Er basierd auf einer flexiblen und erweiterbaren Multi-Agenten-Architektur, die mittels einfacher Textdateien konfiguriert werden kann. In diesen

Konfigurationsdateien kann die BuildOrder, die gewünschten Upgrades und die Zusammensetzung der Armee mitsamt gewünschter Priorität angegeben werden. Sein Abschneiden mit allen Rassen war jedoch nicht überzeugend. Bei der AIIDE 2010 schied erschon früh im Turnierverlauf aus. Bei der CIG 2011 erreichte er 23 Siege in 40 Spielen (57,5 %)und schied damit in der Vorrunde aufgrund des schlechteren direkten Vergleichs gegen BroodwarBotQ aus. Bei der AIIDE 2011 erreichte er 115 Siege in 360 Spielen (31,9 %) und belegte damit Platz 10 von 13. Bei der CIG 2012 wurde er Letzter von 10 Teilnehmern mit 19,9 % Siegen. Auch bei der AIIDE 2012 wurde er Letzter von 10 Teilnehmern mit lediglich 8,4 % gewonnen Spielen.

3.3 Turniere

Bisher fanden folgende acht Wettbewerbe für StarCraft-Bots statt. Eine komplette Übersicht findet sich unter `http://code.google.com/p/bwapi/wiki/Competitions`.

AIIDE 2010 Bei der AIIDE 2010 wurden vier verschiedene Turniere ausgerichtet. Alle Turniere wurden im Double-KO-Format ausgetragen. Dies bedeutet, dass ein KO-Turnier gespielt wird, wobei nach der ersten Niederlage jeder Teilnehmer in die nächste Runde der Verlierrunde rutscht. Verliert er in dieser auch nochmal, ist er endgültig ausgeschieden. Der Gewinner der Loserrunde und der Siegerrunde ermitteln dann den Gesamtsieger. In jeder Runde wurde eine Best-of-Five-Serie gespielt, d.h. der Bot einer Paarung, der zuerst drei Spiele gewinnt, ist der Sieger und kommt in die nächste Runde.

Im ersten Turnier ging es um Micromanagement. Jeder Bot bekam die selbe Armee zu Spielbeginn und kämpfte gegen den anderen auf einer kleinen, leeren Karte. Es konnten also keine Einheiten gebaut werden. Es gewann FreSCBot vor Sherbrooke bei insgesamt 7 Teilnehmern. Beim zweiten Turnier wurde gegenüber dem ersten nicht in einer flachen, leeren Karte gespielt, sondern auf Karten mit Terrain, wo Geländevorteile ausgenutzt werden konnten. Wieder gewann FreSCBot vor Sherbrooke, die diesmal die einzigen beiden Teilnehmer waren. Bei Turnier 3 wurde ein vereinfachtes StarCraft-Spiel ausgetragen. Es konnten nur die Protoss gespielt werden und diese auch nicht mit allen Gebäuden, Einheiten und Upgrades. Zudem gab es keinen Fog of War. Es gewann mimicbot gegen botnik im Finale. Insgesamt nahmen 8 Bots teil. Das vierte Turnier war ein komplettes, uneingeschränktes StarCraft-Spiel. Bei 17 teilnehmenden Bots gewann der Overmind im Finale gegen krasi0.
`http://eis.ucsc.edu/StarCraftAICompetition`

CIG 2010 Bei der CIG 2010 wurde nur ein Turnier mit eingeschränkten Möglichkeiten veranstaltet. Es konnten nur Terraner gespielt werden und zudem waren die Gebäude- und Einheitentypen, die gebaut werden konnten, eingeschränkt. Informationen über

Teilnehmer oder Sieger ließen sich nicht online finden.
`http://ls11-www.cs.tu-dortmund.de/rts-competition/starcraft-cig2010`

AIIDE 2011 Am Turnier der AIIDE 2011 nahmen 13 Bots teil. Es wurde im Ligasystem ausgerichtet, wobei jeder Bot gegen jeden anderen 30 mal spielte. Es wurde auf 10 verschiedenen Karten gespielt. Erster wurde Skynet vor UAlbertaBot und AIUR.
`https://skatgame.net/mburo/sc2011/`

CIG 2011 An dem Turnier der CIG 2011 nahmen 10 Bots teil. Diese wurden auf zwei Vorrundengruppen aufgeteilt, in denen sie 10mal gegen jeden anderen, also 40 Spiele, spielten und jeweils die beiden besten Bots qualifizierten sich für die Finalrunde. Hier wurden wieder 10 Spiele gegen jeden anderen Bot gespielt. Es gewann Skynet vor UAlbertaBot, Xelnaga und BroodwarBotQ.
`http://ls11-www.cs.tu-dortmund.de/rts-competition/starcraft-cig2011`

AIIDE 2012 Das Turnier der AIIDE 2012 wurde ebenfalls im Ligasystem ausgetragen. Jeder der 10 teilnehmenden Bots spielte 184 mal gegen jeden anderen. Es wurde auf denselben 10 Karten vom Vorjahr gespielt. Wieder gewann Skynet vor AIUR und UAlbertaBot.
`http://www.starcraftaicompetition.com/`

CIG 2012 Das Turnier für die CIG 2012 wurde im Ligasystem ausgerichtet. Jeder der 10 teilnehmenden Bots spielte 15 mal auf jeder der verwendeten 6 Karten gegen jeden anderen – ingesamt also 810 Spiele. Es waren fast dieselben Teilnehmer wie bei der AIIDE 2012, lediglich IceBot nahm nur an der CIG teil, während SPAR nur an der AIIDE teilnahm. Es gewann wieder Skynet vor UAlbertaBot und AIUR.
`http://ls11-www.cs.uni-dortmund.de/rts-competition/starcraft-cig2012`

SSCAI 2011 Das Student StarCraft AI Tournament der Universität Bratislava ist nur für Studierende offen. Es nahemn 50 Bots teil, jedoch fand sich darunter kein Name, der auch bei den AIIDE- oder CIG-Turnieren vertreten war.
`http://bwapiproject.netii.net/2011/`

Bot-Ladder Eigentlich wollten die Autoren der Programmierschnittstelle BWAPI im Januar 2012 selbst ein Turnier veranstalten, doch dieses wurde mangels Teilnehmern abgesagt. Stattdessen richtete ein Freiwilliger ein sogenanntes Laddersystem ein, bei dem automatisch Bots gegeneinander antreten und mit einer Elo-Zahl bewertet werden. Dort worden schon über 80.000 Spiele durchgeführt.
`http://bots-stats.krasi0.com/`

Kapitel 4

Programmierschnittstelle

In diesem Kapitel wird die vorhandene Programmierschnittstelle vorgestellt. Diese setzt sich aus zwei Teilen zusammen: BWAPI, über die man die Informationen über das Spiel selbst erhält und Einheiten steuern kann, und BWTA, welcher Informationen über die Karte und das Gelände liefert.

4.1 BWAPI

BWAPI stellt alle Informationen über das Spiel zur Verfügung. Besonders wichtig sind hier die Informationen über die Einheiten und Einheitentypen. Aber auch über die Spieler und das Spiel selbst können zahlreiche Daten abgefragt werden. Die Programmierschnittstelle basiert auf einem eigentlich illegalen Hack von StarCraft, da ein eigenes Modul in den StarCraft-Prozess geladen wird und dieser damit verändert wird. Jedoch hat mittlerweile Blizzard die Verwendung von BWAPI für Turniere undd Forschungszwecke erlaubt (BWAPI (2012), Huang (2011)).

4.1.1 Konzeptuelles

4.1.1.1 Verfügbarkeit von Einheiteninformationen

Es gibt vier verschiedene Stufen der Verfügbarkeit von Informationen über Einheiten: vollständig, normal, teilweise und keine. Dieses Konzept muss man verstanden haben, um sinnvoll mit Einheiteninformationen, besonders über gegnerische Einheiten, umgehen zu können.

vollständig Von allen eigenen Einheiten sind sämtliche Informationen verfügbar, wenn sie am Leben sind. Falls das Flag `Complete Map Information` gesetzt ist oder das Spiel ein Replay ist, dann sind für alle Einheiten des Spieles alle Informationen verfügbar. Ansonsten fallen gegnerische und neutrale Einheiten in eine der anderen

drei Stufen. Eine Ausnahme bildet der allererste Frame. In diesem sind von allen neutralen Einheiten alle Informationen verfügbar.

normal Normale Verfügbarkeit von Informationen hat der Bot über alle sichtbaren gegnerischen und neutralen Einheiten. Das sind alle Informationen außer solche über interne Vorgänge der Einheit, wie die Anzahl von Scarabs und SpiderMines, welche Einheiten die Einheit baut, welches Upgrade oder welche Technologie erforscht und wie lange das noch dauert, welchem Carrier oder Hatchery die Einheit zugeordnet ist und ob die Einheit eine Nuklearrakete hat. Diese Funktionen werden je nach Rückgabewert immer 0, NULL oder false liefern.

teilweise Der Bot hat teilweise Verfügbarkeit von Informationen über gegnerische Einheiten, die für ihn sichtbar (`Unit::isVisible() == true`), aber nicht detektiert (`Unit::isDetected() == false`) sind. Das sind getarnte gegnerische Einheiten, die nicht vom Fog of War verdeckt sind. Eingegrabene gegnerische Einheiten sind nur sichtbar, wenn sie angreifen oder detektiert sind. Für Einheiten in dieser Stufe sind Informationen wie ID, Spieler, Typ, Position, Bewegungsrichtung und Angriffs- und Waffenzustand verfügbar.

keine Für tote Einheiten sind keine Informationen verfügbar. Falls die komplette Karteninformation nicht aktiviert wurde, sind auch Einheiten, die vom Fog of War verdeckt sind in dieser Stufe. Für neutrale Einheiten, die zu Beginn des Spieles existiert haben, sind jedoch die Informationen über den initialen Typ, initiale Position, HitPoints und Ressourcen verfügbar, da sich diese aus den statischen Kartendaten ergeben. Mittels `Game::getStaticMinerals()`, `Game::getStaticGeysers()` und `Game::getStaticNeutralUnits()` können diese Einheiten ermittelt werden.

4.1.1.2 Positionsangaben

BWAPI unterscheidet intern, wie auch StarCraft, drei verschiedene Sorten von Positionsangaben: Position, WalkTile und TilePosition.

Position Positions werden pixelgenau gemessen und sind die kleinste verfügbare Einheit von Positionsangaben. Die `Position`-Klasse implementiert diese.

WalkTile WalkTiles sind 8x8 Pixel groß. In dieser Auflösung liegen die Daten darüber vor, wo gelaufen werden kann, also wo auf der Karte sich Bodeneinheiten bewegen können. Diese Auflösung wird ausschließlich für diese Begehbarkeitsinformationen genutzt. Für WalkTiles ist auch keine eigene Klasse vorhanden. `BWTA` be-

rechnet die UnwalkablePolygons einer Karte, hierfür wird jedoch die Klasse `Position` missbraucht. Um die eigentliche Kartenposition zu erhalten, müssen diese „Positions“ mit 8 multipliziert werden.

TilePosition TilePositions sind 32x32 Pixel groß. In dieser Auflösung liegen die Daten darüber vor, wo gebaut werden kann. Für TilePosition existiert auch eine entsprechende Klasse `TilePosition`. Ein oft verwendetes Synonym für TilePosition ist Tile.

`Position`s und `TilePosition` können jeweils dem Konstruktor der anderen Klasse übergeben werden, um sie von einer Klasse in die andere umzuwandeln. Bei der Umwandlung von Position nach TilePosition geht jedoch Genauigkeit verloren. Beide Klassen haben Methoden, um die x- und y-Koordinate als Ganzzahl auszulesen und zu setzen. Des Weiteren gibt es Methoden zum Testen, ob sie eine gültige Positionsangabe auf der Karte darstellen und ob ein Bodenweg zu einer anderen Instanz der jeweiligen Klasse existiert. `Position` hat zudem noch zwei Methoden zur Abstandsberechnung zu anderen `Position`s: eine genaue, welche `double`-Werte zurückliefert, und eine näherungsweise, welche einige CPU-Zyklen einspart und `int` zurückliefert.

4.1.2 Klassen

4.1.2.1 AIModule

Die virtuelle Klasse `AIModule` ist dafür gedacht von einem Bot implementiert zu werden. Sie hat die Callbacks mittels derer der KI bestimmte Spielereignisse mitgeteilt werden:

onStart wird zu Beginn des Spieles aufgerufen. Hier kann eine KI Initialisierungen vornehmen.

onEnd wird am Ende des Spieles aufgerufen.

onFrame wird in jedem Spielframe aufgerufen.

onUnitDiscover wird aufgerufen, wenn man Zugriff auf eine Einheit bekommt. Ist das Flag `Complete Map Information` gesetzt, wird dies zur selben Zeit wie onUnitCreate, ansonsten zur selben Zeit wie onUnitShow aufgerufen.

onUnitEvade wird aufgerufen, wenn man Zugriff auf eine Einheit verliert und ist somit das Gegenstück zu onUnitDiscover. Ist das Flag `Complete Map Information` gesetzt, wird dies zur selben Zeit wie onUnitDestroy, ansonsten zur selben Zeit wie onUnitHide aufgerufen.

onUnitShow wird aufgerufen, wenn eine Einheit sichtbar wird. Ist das Flag `Complete Map Information` nicht gesetzt, bedeutet das auch, dass man nun Zugriff auf die Einheiteninformationen bekommt.

onUnitHide wird aufgerufen, wenn eine Einheit unsichtbar wird. Ist das Flag `Complete Map Information` nicht gesetzt, verliert man damit auch Zugriff auf die Einheiteninformationen.

onUnitCreate wird aufgerufen, wenn eine Einheit erzeugt wird, für die Einheiteninformationen verfügbar sind. Das bedeutet, wenn das Flag `Complete Map Information` nicht gesetzt ist, werden für Einheiten in nicht sichtbaren Bereichen dieser Callback nicht aufgerufen. Der Callback wird auch nicht aufgerufen, wenn sich Typ einer Einheit ändern. In diesem Fall wird onUnitMorph aufgerufen.

onUnitDestroy wird aufgerufen, wenn eine Einheit zerstört oder anderweitig aus dem Spiel entfernt wird (z.B. ein komplett abgebautes Mineralienfeld), für die Einheiteninformationen verfügbar sind. Das bedeutet, wenn das Flag `Complete Map Information` nicht gesetzt ist, wird für Einheiten in nicht sichtbaren Bereichen dieser Callback nicht aufgerufen.

onUnitMorph wird aufgerufen, wenn der Typ einer Einheit wechselt, für die Einheiteninformationen verfügbar sind. Dies ist inbesondere der Fall bei Zerg-Einheiten, die aus Larven gemorpht werden. Hierbei morpht die Larve erst zu einem Ei und dann z.B. zu einer Drohne oder Hydralisken. Für beide Übergänge wird dieser Callback aufgerufen. Auch für das Bauen einer Raffinerie, Assimilators oder Extraktors wird dieser Callback aufgerufen, da der Geysir dann zu dem Gebäude morpht.

onUnitRenegade wird aufgerufen, wenn der Eigentümer einer Einheit, für die Einheiteninformationen verfügbar sind, wechselt.

onNukeDetect wird aufgerufen, wenn ein nuklearer Abschuss festgestellt wird. Falls die Zielposition sichtbar ist, wird sie dem Callback übergeben. In jedem Frame können alle bekannten Atomraketenziele über `Game::getNukeDots()` abgefragt werden.

onSendText wird aufgerufen, wenn ein Spieler Text in den Chat eingibt.

onReceiveText wird aufgerufen, wenn ein Spieler Text von einem anderen Spieler empfängt.

onPlayerLeft wird aufgerufen, wenn ein Spieler das Spiel verlässt.

4.1.2.2 Client

Neben der Möglichkeit, eine KI als `AIModule` zu implementieren, gibt es bei BWAPI noch die Alternative sie als Client zu realisieren. Dabei dient eine Implementierung dieser Klasse dazu, mit einem Clientprogramm zu kommunzieren und dessen Befehle an die StarCraft-Instanz mit BWAPI weiterzuleiten, damit sie dort ausgeführt werden. Da die `Client`-Klasse keine Callbacks wie `AIModule` besitzt, stehen für jeden Callback ein entsprechendes Ereignisse zur Verfügung. Diese sind mittles der Klasse `Event` implementiert. Über `Game::getEvents()` kann die Liste aller Events des aktuellen Frames abgefragt werden.

4.1.2.3 Game

Die Klasse `Game` repräsentiert das laufende Spiel. Über sie können u.a. alle verfügbaren Spieler, Einheiten, statische Kartendaten, wie die Ressourcenpositionen, Kartengröße und -name, sowie die vergangene Spielzeit und Frames abgefragt werden. Zudem besitzt sie Methoden, um auf die Karte oder den Bildschirm zeichnen oder Text ausgeben zu können, was für Debugging-Zwecke nützlich ist.

4.1.2.4 Player

Die Klasse `Player` repräsentiert einen Spieler. Sie speichert u.a. seinen Namen, Rasse, die Einheiten, gesammelte und ausgegebene Rohstoffe, Anzahl der gebauten und zerstörten Einheiten. Zudem ist es mittels dieser Klasse möglich, die erforschten Upgrades und deren Auswirkungen abzufragen. Dazu gehören die tatsächliche aktuelle maximale Energie, Höchstgeschwindigkeit, Sichtradius und Rüstung eines Einheitentyps sowie Waffenreichweite und -abkühlphase in Abhängigkeit von den zu dem Spielzeitpunkt erforschten Upgrades und Technologien.

4.1.2.5 Race

Mittels dieser Klasse kann für jede Rasse der `UnitType` des Arbeiters, des Hauptquartiers, der Raffinerie, der Einheit, welche andere transportieren kann, und der Einheit, welche Supply für neue Einheiten zur Verfügung stellt, abgefragt werden.

4.1.2.6 Unit

Die Unit-Klasse ist eine der wichtigsten Klassen. Mittels dieser Klasse können alle verfügbaren Informationen über eine Einheit abgerufen werden. Welche das sind, richtet sich nach der Stufe der Verfügbarkeit von Einheiteninformationen, in der sich

eine Einheit befindet, siehe 4.1.1.1. Weiterhin können eigenen Einheiten mittels dieser Klasse Befehle erteilt werden. Jede Einheit wird durch eine eindeutige Instanz dieser Klasse repräsentiert. Diese wird angelegt, sobald die Einheit erzeugt wird. Dann wird auch `AIModule::onUnitCreate()` aufgerufen. Für Einheiten, die in einem Gebäude oder einer anderen Einheit trainiert werden, ist dies der Fall sobald der Bauprozess begonnen hat und nicht erst, wenn die Einheiten fertiggestellt sind und auf der Karte erscheinen. Es wird kein Event ausgelöst, wenn die Einheit fertiggestellt ist, daher muss man wie der `WorkerManager` dies selbst überwachen, siehe 5.4.1. Die Instanzen der Einheiten werden nicht gelöscht, wenn die Einheiten zerstört werden, sondern erst am Ende des Spieles, so dass es dadurch nicht zu Nullpointer-Exceptions kommen kann.

4.1.2.7 UnitType

Diese Klasse symbolisiert die Typen, die Einheiten haben können. Sie wird sowohl für Gebäude- als auch andere Einheitentypen verwendet. Sämtliche Eigenschaften, die ein Einheitentyp mit sich bringt, können über diese Klasse abgefragt werden. Dazu gehören u.a. der Mineralien- und Gaspreis, welche Einheit diesen Typ produziert, seine Größe, Waffen und Lebenspunkte. Im Namespace `UnitTypes` befinden sich alle in StarCraft verfügbaren Einheitentypen.

4.1.2.8 UpgradeType & TechType

Die Klassen `UpgradeType` und `TechType` repräsentieren die Upgrades bzw. Technologien, die erforscht werden können. Mittels dieser Klassen können Informationen wie der Mineralien- und Gaspreis, welches Gebäude dies erforscht und welche Einheiten oder Waffen von dem Upgrade oder Technologie verbessert werden, abgefragt werden. Über den Namespaces `UpgradeTypes` und `TechTypes` kann auf alle verfügbaren Upgrades und Technologien zugegriffen werden.

4.1.2.9 Order

Mittels der `Order`-Klasse können genaue Informationen über das, was eine Einheit gerade erledigt, abgefragt werden. Über Unit::getOrder und Unit::getSecondaryOrder erhält man die entsprechenden Order-Instanzen. Eine einzige Aufgabe kann aus mehreren Orders bestehen, z.B. besteht das Sammeln von Mineralien aus MoveToMinerals, HarvestMinerals2, MiningMinerals, ReturnMinerals mit optionalem WaitForMinerals. Über den Namespace Orders sind alle möglichen Order-Instanzen verfügbar.

4.1.2.10 WeaponType

Diese Klasse implementiert die im Spiel verfügbaren Waffen. Mittels dieser können die minimale und maximale Reichweite, die möglichen Ziele (Luft, Boden, mechanische oder organische Einheiten, etc), welcher Einheitentyp die Waffe nutzt, der Schaden, Schadentypen, ExplosionType, Abkühlphase usw. abgefragt werden. Um die aktuellen Werte für Waffenreichweite und Abkühlphase unter Berücksichtigung der schon erforschten Upgrades abzufragen, muss auf `Player::weaponMaxRange()` und `Player::groundWeaponDamageCooldown()` zurückgegriffen werden.

4.1.2.11 UnitCommand

Die Klasse `UnitCommand` bietet einen alternativen Weg zu `Unit::`<*BefehlTyp*>, um Befehle an Einheiten zu geben. Mittels `Unit::issueCommand()` können Instanzen dieser Klasse an Einheiten übergeben werden. Durch diese Klasse ist es möglich Funktionen zu schreiben, die jeden Einheitenbefehl verarbeiten können.

Die Klasse `UnitCommandType` implementiert alle Befehlstypen. Zu jedem Befehlstyp existiert eine Methode `Unit::`<*BefehlTyp*> mittels der der Befehl direkt an eine Einheit gegeben werden kann. UnitCommandType-Instanzen werden beim Erstellen einer UnitCommand-Instanz genutzt. Über den Namespace `UnitCommandTypes` kann auf alle möglichen UnitCommandType-Instanzen zugegriffen werden.

4.1.2.12 Weitere Klassen

Die Klasse `Bullet` repräsentiert einzelne Geschosse, Raketen, Zauber und andere Fernwaffen. Die Klasse `BulletType` repräsentiert deren mögliche Arten und der Namespace `BulletTypes` enthält eine Auflistung dieser.

Die Klasse `ExplosionType` repräsentiert die verschiedenen Arten von Explosionen wie z.B. Nuclear_Missile, Maelstrom oder Yamato_Gun. Über den Namespace `ExplosionTypes` kann auf all diese zugegriffen werden.

Die Klasse `DamageType` spezifiziert für eine Waffe die Art des Schadens, den diese anrichtet. Dieser kann Concussive, Explosive, Ignore_Armor, Independent, Normal, None oder Unknown sein.

Die Klasse `Color` repräsentiert eine Farbe für das Zeichnen auf dem Bildschirm. Über den Namespace `Colors` kann auf benannte Farbe zugegriffen werden. StarCraft verwendet eine Palette von 256 Farben aus der auch diese Farbe gewählt werden muss. Teile dieser Palette können wechseln. Mittels des Konstruktors `Color(int red, int green, int blue)` wird eine Instanz erzeugt, die die Farbe der Palette repräsentiert, die am nächsten an dem angegebenen RGB-Wert ist.

Die Klasse `Error` kann genutzt werden, um mittels `Game::getLastError()` auf den letzten aufgetretenen Fehler zuzugreifen. Über den Namespace `Errors` kann auf alle möglichen Fehlerarten zugegriffen werden.

Die Klasse `Force` modelliert ein Team von Spielern. Über sie kann der Name des Teams und dessen Spieler abgefragt werden. In nicht-team-basierten Spieltypen sind alle Spieler einem Team zugeordnet, auch wenn sie gegeneinander kämpfen.

Die Klasse `GameType` symbolisiert alle Spieltypen wie Nahkampf (Melee) oder Duell (1on1) von StarCraft. Mittels des Namespaces `GameTypes` kann auf diese zugegriffen werden.

Die Klasse `PlayerType` repräsentiert alle möglichen Arten von Spielern. Über den Namespace `PlayerTypes` kann auf diese zugegriffen werden. Diese sind neben *Computer* und *Player* für einen menschlichen Spieler oder KI auch *Neutral*, dem z.B. alle Ressourcen und neutralen Einheiten zugeordnet sind, auch Spezialtypen für die Kampagne oder andere gescriptete Missionen. Zudem existieren die Typen *ComputerLeft* und *PlayerLeft*, denen Einheiten zugeordnet werden, wenn deren Spieler das Spiel verlassen hat.

Die Klasse `UnitSizeType` spezifiziert alle möglichen Größen von Einheiten. Diese sind Independent, Small, Medium, Large, None und Unknown und über den Namespace `UnitSizeTypes` verfügbar.

4.2 BWTA

Das BWTA-Projekt hat einen Terrain-Analyzer für Karten von StarCraft: Brood War erstellt. Dieser analysiert die statischen Kartendaten, also die Karten wie sie zu Beginn des Spieles sind ohne die Änderungen, die sich durch ein laufendes Spiel ergeben. Dadurch stehen schon zu Beginn das Spiel die Informationen über die Beschaffenheiten der Karte zur Vefügung. Dazu gehören die Positionen der Ressourcen, die möglichen Startpositionen, die Gebietsgrenzen und v.a. Engstellen zwischen diesen. Dies ist jedoch keine Bevorzugung der Bots, denn ebenso wie bei den Botturnieren sind bei den Turnieren der eSport-Ligen die Karten vorher bekannt und die menschlichen Spieler kennen diese in- und auswendig. BWTA sorgt dafür, dass diese Informationen ebenso den Bots zur Verfügung stehen BWTA (2011).

Folgende Klassen werden von BWTA zur Verfügung gestellt, um diese Informationen zu repräsentieren:

BaseLocation Diese Klasse repräsentiert eine Basisposition. Als eine Basisposition wird die nächstmögliche Bauposition von Hauptquartieren bei Rohstoffvorkommen bezeichnet. Dieser werden ebendiese Rohstoffe zugeordnet. Neben Methoden, um die

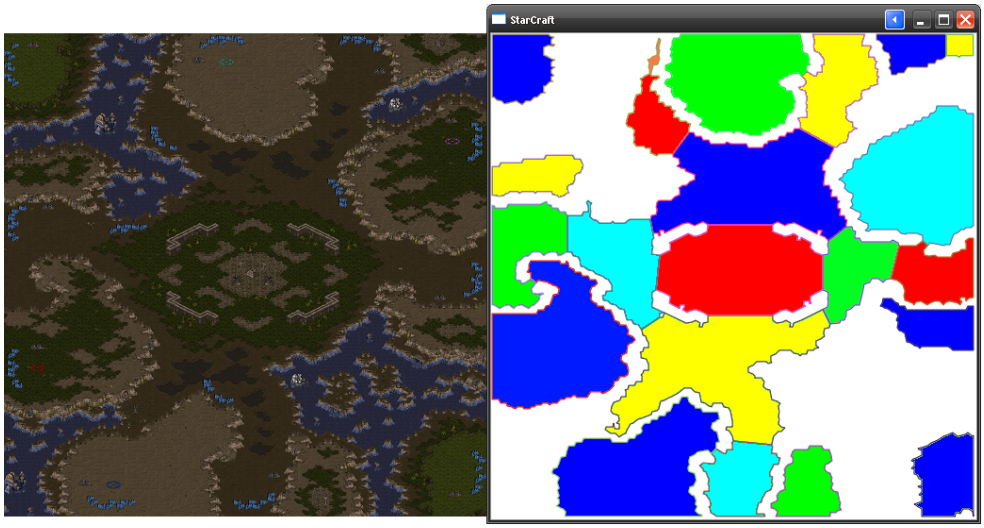


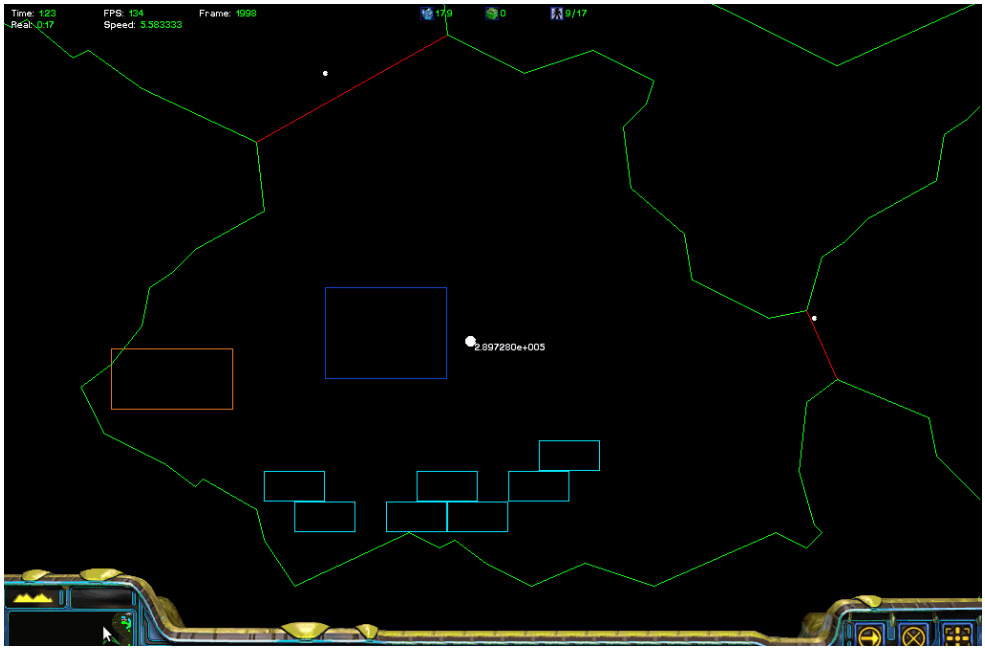
Abbildung 4.1: Repräsentation der BWTA-Analyse der Karte „Lost Temple“

Gesamtzahl der Mineralien und Gasvorkommen abzufragen, gibt es auch welche, die die Abstände zu Boden und in der Luft zu anderen Regionen berechnen. Wichtig hierbei ist zudem, ob die BaseLocation von einer anderen auf dem Bodenweg überhaupt erreicht werden kann. Hierfür gibt es die Funktion `isIsland()`. Weiterhin wird die Information gespeichert, ob diese BaseLocation auch eine Startposition sein kann. Zudem wird sie einer Region zugeordnet.

Region Eine Region ist ein zusammenhängendes Gebiet, das von anderen Regionen durch `Chokepoints`, sinngemäß Engstellen, getrennt ist. Für sie werden die in ihr liegenden `BaseLocations`, die `Chokepoints`, dessen Zentrum sowie die von dieser Region aus erreichbaren Regionen gespeichert. Die Geometrie der Region, also deren Eckpunkte, wird in derem zugehörigen Polygon gespeichert.

Polygon Die `Polygon`-Klasse ist abgeleitet von `std::vector<Position>`, also eine geordnete Liste von `Position`en. Gewöhnlich sind dies sämtliche Eckpunkte einer Region. Die Polygon-Klasse implementiert zusätzlich zu den vector-Funktionen noch Methoden, um die Fläche, den Durchmesser und den Mittelpunkt des Polygons abzufragen. Weiterhin stehen Funktionen zur Verfügung um zu testen, ob ein Punkt innerhalb des Polygons liegt und um zu einem gegebenen Punkt den nächsten Punkt auf dem Rand des Polygons zu berechnen.

Chokepoint Ein Chokepoint ist eine Engstelle, die zwei Regionen voneinander trennt. Dies ist die Strecke zwischen den beiden gemeinsamen Eckpunkten der Regionen. Eine Instanz der Klasse speichert diese beiden Regionen, die beiden Endpunkte sowie die den Mittelpunkt und die Länge der Engstelle, also der Strecke zwischen den beiden Endpunkte.



(a) nicht erkundet

(b) sichtbar

Abbildung 4.2: Die von BWTA gelieferte Informationen, visualisiert für einen Kartenausschnitt, wenn er a) noch nicht erkundet ist und b) sichtbar ist:

blau: `BaseLocation`
orange: `Geysir`
türkis: `Mineralien`
grün: `Region`
rot: `Chokepoint`

BWTA Im Namespace BWTA werden weitere Funktionen zur Verfügung gestellt. Dazu gehören die Funktionen zum Einlesen und Analysieren der Karte. Diese sind getrennt worden, um das Analysieren der Karte in einen gesonderten Thread auslagern zu können. Das erstmalige Einlesen einer Karte kann mehrere Minuten dauern. Danach werden die gewonnen Informationen im StarCraft-Ordner gespeichert, so dass danach das Einlesen der Karteninformationen in wenigen Sekunden erfolgt. Je nach Turnier werden diese eingelesenen Informationen schon zur Verfügung gestellt, so dass es nicht nötig ist, das Einlesen der Karteninformationen in einen separaten Thread auszulagern. Erst nach dem Analysieren der Karte können die anderen Funktionen dieses Namespace aufgerufen werden.

Diese liefern alle `Region`en, `Chokepoint`s, `BaseLocation`s, StartLocations, sowie die nicht-begehbaren Polygonen. Dies sind die Bereiche der Karte, die in keiner Region liegen. Zudem gibt es Funktionen, die zu einer gegebenen `Position` die nächste Region, Chokepoint, BaseLocation, unwalkable Polygon zurückliefern. Weiterhin werden Methoden zur Verfügung gestellt, die den kürzesten Pfad oder den Abstand von einer gegeben `TilePosition` zu anderen liefern, die nächste TilePosition aus einer Menge berechnen oder testen, ob zwei TilePositionen auf dem Bodenweg miteinander verbunden sind.

Kapitel 5

Implementierung

In diesem Kapitel wird die Implementierung des *Dreadbots* vorgestellt. Zuerst wird die Grundstruktur des Bots und anschließend dessen einzelne Fähigkeiten und die dafür verantwortlichen Klassen beschrieben.

5.1 Grundstruktur

5.1.1 Vorabentscheidungen

Der Bot wurde in C++ implementiert. Es wäre zwar möglich gewesen, dank verschiedener Projekte den Bot in anderen Sprachen wie Lua, Python, Java oder Haskell zu implementieren, doch diese Projekte waren und sind jeweils nicht mit der aktuellen BWAPI-Version kompatibel. Deshalb wurde der Bot in C++ geschrieben, um mit der aktuellen Version 3.7.2 arbeiten zu können.

Weiterhin gab es die Wahl zwischen der Implementierung als Modul oder Client. Die Implementierung als Modul erzeugt eine DLL, die direkt in den StarCraft-Prozess geladen wird. Bei der Implementierung als Client wird eine Verbindung zu einem speziellen BWAPI-Modul aufgenommen, das als Server dient. Diese Verbindung verursacht einen gewissen Overhead, da der gesamte Status des Spiels in jedem Frame in den gemeinsamen Speicher kopiert werden muss. Dadurch sind Client-Programme grundsätzlich langsamer. Da alle Funktionen von Dreadbot mittels der Modulimplementierung realisiert werden konnten, wurde diese gewählt, um den Geschwindigkeitsvorteil zu nutzen.

Als Rasse wurde Protoss gewählt. Das Hauptargument für diese Wahl war, dass man schon mit nur zwei Sorten von Kampfeinheiten und einer recht simplen Strategie schon als menschlicher Spieler die vorhandene StarCraft-KI leicht besiegt. Die beiden Kampfeinheitentypen sind der Beserker (Zealot) als Nahkampfeinheit und der Dragooner (Dragoon) als Fernkampfeinheit, die sowohl Boden- als auch Lufteinheiten angreifen kann. Hinzu kommen noch Beobachter (Observer), um unsichtbare gegneri-

sche Einheiten zu detektieren. Nur bei Bedarf werden Träger (Carrier) mit Interceptoren gebaut, nämlich dann, wenn der Gegner Basen hat, die nicht auf dem Landweg zu erreichen sind.

5.1.2 Hauptklasse Dreadbot

Die Klasse `Dreadbot` ist die Hauptklasse des Programmes. Sie ist von der virtuellen Klasse `BWAPI::AIModule` abgeleitet und implementiert die Callbacks `onStart()`, `onCreate()`, `onMorph()`, `onDestroy()` und `onEnd()`. Alle möglichen Callbacks, auch die, für die keine weitere Funktionalität implementiert wurde, beeinhalten Debug-Statements, die, falls das Präprozessor-Makro `PRINT_UNIT_EVENTS` beim Komplilieren definiert war, den Aufruf des Callbacks in der Logdatei dokumentieren. Dabei werden der Name des Callbacks sowie die ID, der Typ und der besitzende Spieler der Einheit, für die der Callback aufgerufen wurde, ausgegeben.

onStart() Zu Beginn des Spieles wird in der onStart()-Methode BWTA zum Einlesen und Analysieren der Karte genutzt.

onFrame() Die onFrame()-Methode ist die Hauptschleife des Programms. Sie wird in jedem logischen Frame von StarCraft von BWAPI aufgerufen. Diese ruft wieder ihrerseits die onFrame()-Methoden von `Baseman`, `CombatMan`, `CIA`, `ScoutMan` und `BuildManager` auf. Der `Baseman` (s. 5.4.2) ruft wiederum von dem `WorkerManager` jeder Basis die onFrame()-Methode auf, um die Arbeiter und Raffinerien zu verwalten (s. 5.4.1. Weiterhin überprüft er, ob expandiert werden soll oder ob zerstörte Basen aus seiner Liste entfernt werden müssen. Der für Kampfaktionen zuständige `CombatMan` (s. 5.6.1) ist als endlicher Zustandsautomat implementiert. Er bestimmt in jedem Frame seinen aktuellen Zustand und führt die zugehörigen Aktionen dann aus. Die `CIA` (s. 5.5) speichert die über den Gegner gesammelten Informationen und aktualisiert diese in jedem Frame. Der `ScoutMan` (s. 5.5) ist für das Aussenden einer Einheit zum Auskundschaften des Gegners zuständig. Er speichert in jedem Frame jede sichtbare `BaseLocation` und sendet gegebenenfalls einen neuen Scout los oder, falls schon ein Scout unterwegs ist, diesen zu seinen Zielen. Der `BuildManager` (s. 5.2.1) überprüft in jedem Frame, ob die aktuellen Gebäudebauvorhaben korrekt ausgeführt werden (s. 5.3.2) sowie fragt die `HardcodedBuildOrder` nach neuen Bauaufträgen an (s. 5.2.3).

onCreate(Unit*) Der onCreate()-Callback weist eigene neue Einheiten einen Manager gemäß ihrer Funktion zu. Arbeiter werden `Baseman::assignWorkerToBase()` übergeben, damit dieser sie einer Basis zuweist. Gebäude werden der nächsten Basis zugewiesen und `BuildManager::buildingCreated()` übergeben, damit das Gebäude der entsprechenden `PendingBuildOperation` zugewiesen wird. Alle anderen Einheiten werden dem CombatMan zugeteilt.

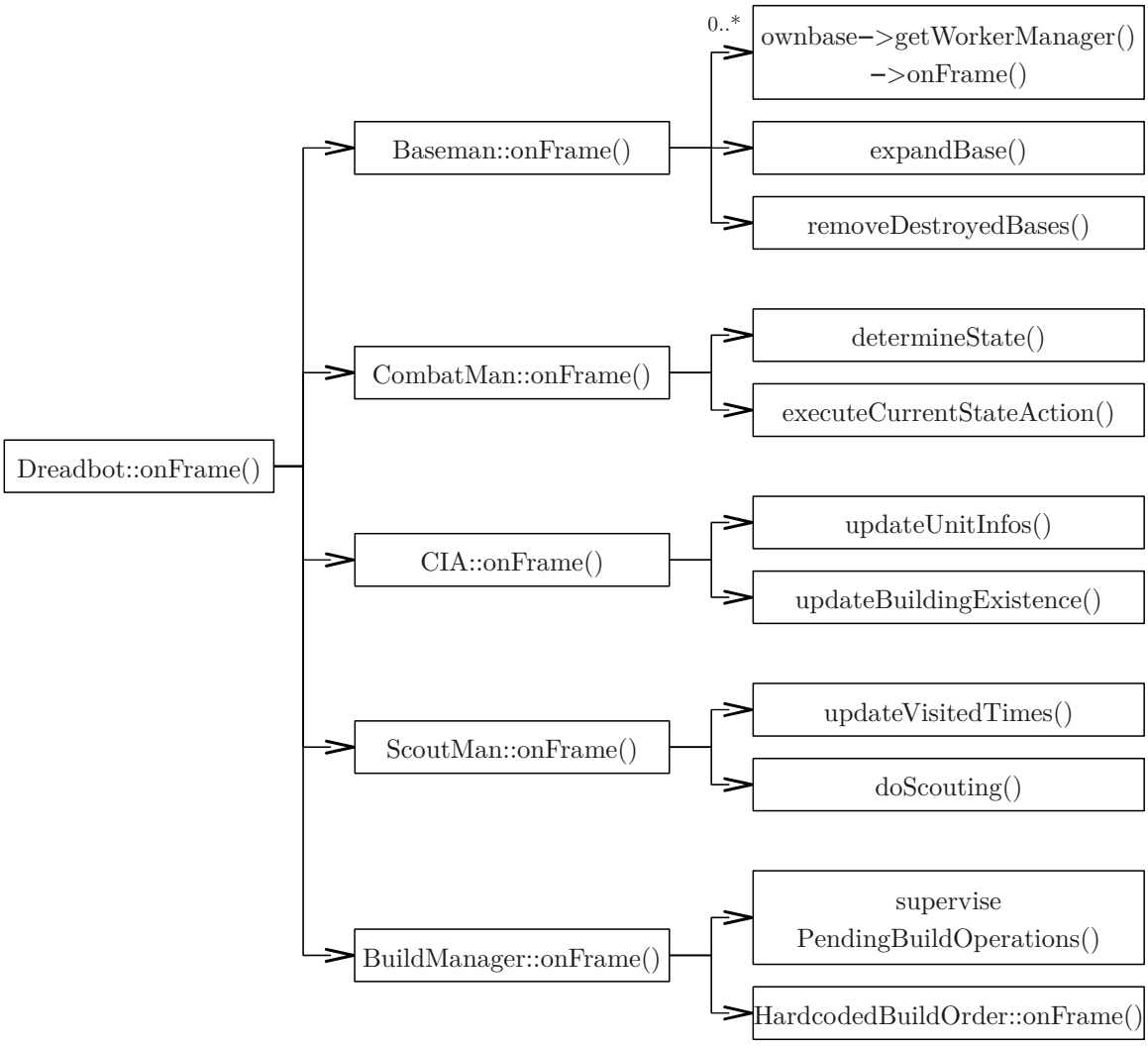


Abbildung 5.1: Ablaufdiagramm eines onFrame()-Aufrufes. Es wird in der Reihenfolge einer Tiefensuche durchlaufen.

onMorph(Unit*) Der onMorph()-Callback kümmert sich um Raffinerien. Da diese intern aus dem Geysir morphen, auf dem sie gebaut werden, wird der onCreate()- und onDestroy()-Callback für diese nicht aufgerufen. Neu gebaute Raffinerien werden wie andere Gebäude auch der nächsten Basis und dem `BuildManager` zugewiesen, aber auch dem `WorkerManager` der nächsten Basis übergeben, damit er der Raffinerie Arbeiter zuweisen kann, sobald diese fertig ist. Bei einer zerstörten Raffinerie wird diese nicht nur aus der Basis und dem WorkerManager entfernt, sondern auch `CIA` übergeben. Denn es könnte sich um eine gegnerische Raffinerie handeln, da der ursprüngliche Besitzer nicht mehr angezeigt wird, sondern nur noch der Neutral-Player als aktueller Besitzer des Geysirs. Weiterhin finden sich in diesem Callback zerg-spezifische Funktionen, damit die Grundfunktionen des Bots rassenunabhängig funktionieren.

onDestroy(Unit*) Im onDestroy()-Callback werden entsprechend die eigenen Einheiten aus den Managern entfernt. `CIA::enemyUnitDestroyed()` wird für gegnerische Einheiten aufgerufen, damit diese Information in der `UnitInfo`-Instanz der Einheit gespeichert wird. Siehe 5.5 für Details.

onEnd() Der onEnd()-Callback wird am Ende des Spieles aufgerufen. Er kann genutzt werden, um den Ausgang des Spieles in eine Logdatei zuschreiben, was für Ergebnisanalysen nützlich sein kann.

Die Klasse wurde als Singleton implementiert. Dieses Entwurfsmuster garantiert, dass nur eine Instanz einer Klasse zur Laufzeit erzeugt wird. Hierbei wurde auf die Datei `Singleton.h` von Skynet zurückgegriffen, die es ermöglich, eine Klasse sehr einfach als Singleton zu definieren. Dazu muss eine beliebige Klasse `A` von `Singleton<A>` abgeleitet werden, `Singleton<A>` muss als befreundete Klasse von `A` deklariert werden sowie der Konstruktor und Destruktor von `A` als `protected` oder `private` gekennzeichnet werden. Dann kann man mittels des Aufrufs `A::Instance()` auf die einzige Instanz von `A` zugreifen, die beim ersten Aufruf dieser Methode angelegt wird. Von BWAPI wird die Datei Dll.cpp zur Verfügung gestellt, die dafür sorgt, dass beim Kompilieren aus der `Dreadbot`-Instanz die DLL erzeugt wird, die in StarCraft geladen wird.

5.2 Bauen

Grundlegender Bestandteil eines jeden Bots ist die Fähigkeit Einheiten und Gebäude zu bauen. Während zum Erstellen von Einheiten nur ein produzierendes Gebäude[1] ausgewählt werden muss, um diese zu erzeugen, müssen bei Gebäuden nicht nur ein geeigneter Bauplatz gewählt werden, sondern auch der Bauprozess überwacht werden. Zudem soll das Baumodul auch fähig sein, das Erforschen von Upgrades und Technologien zu starten. Weiterhin müssen die Baubefehle durch eine vorgegebene Baureihenfolge gegeben werden.

5.2.1 BuildManager

Der `BuildManager` ist zuständig für das Ausführen von Baubefehlen. Er überprüft, ob alle notwendigen Voraussetzungen für den Bau erfüllt sind, wählt eine Einheit, die das Bauvorhaben ausführen kann, aus und gibt ihr den Befehl zum Bauen. Bei Gebäuden werden zudem mittels `Baseman::getBaseForNewBuilding()` die Basis bestimmt, in der das Gebäude errichtet werden soll, ein Bauplatz gesucht und dann eine zugehörige `PendingBuildOperation`-Instanz für das Überwachen des Bauprozesses erstellt sowie Mineralien, Gas und `TilePosition`s reserviert. Die Details dazu sind im Abschnitt 5.3 beschrieben.

[1] Interceptoren werden nicht von Gebäuden, sondern von Trägern gebaut. Das macht hier aber keinen Unterschied bei der Funktionalität des Programms.

Durch den direkten Aufruf von `makeBuilding()` kann ein Gebäudebauvorhaben erzwungen werden, auch wenn nicht alle Voraussetzungen erfüllt sind, sodass dieses schnellstmöglich konstruiert wird, sobald die Voraussetzungen erfüllt sind. Dies ist wichtig, um Gebäudebauaufträgen Vorzug über Routinebaumaßnahmen wie z.B. für Arbeiter geben zu können, indem die nötigen Ressourcen reserviert werden.

In jedem Frame überprüft der `BuildManager` die korrekte Ausführung der `PendingBuildOperations` und fragt die `HardcodedBuildOrder` nach den nächsten Aufträgen.

5.2.2 BuildType

Damit der BuildManager sowohl Einheiten als auch Upgrades und Technologien handhaben kann, wurde die Klasse `BuildType` als Wrapper-Klasse für `UnitType`, `UpgradeType` und `TechType` geschaffen. Diese enthält nicht nur Methoden, die den Mineralien- und den Gaspreis, die Voraussetzungen an Einheiten, Technologie und Supply liefern, sondern auch die `make(Unit* maker)`-Methode, welche den entsprechenden Typ produziert. Dort wird nicht nur das unterschiedliche Vorgehen für `UnitType`, `UpgradeType` und `TechType` berücksichtigt, sondern auch berücksichtigt, dass es bei `UnitTypes` verschiedene Weisen gibt, diesen zu erstellen. Es muss zum einen unterschieden werden zwischen Einheiten, die in Gebäuden ausgebildet werden, und Einheiten, die aus ihrem Macher gemorpht werden. Zum anderen baut diese Funktion auch Gebäudeerweiterungen der Terraner und morpht Zerggebäude zu höheren Stufen wie z.B. Hatchery zu Lair und Spore zu Greater Spore. Diese Gebäudefunktionalitäten wurden hier implementiert, da im Gegensatz zu einem Neubau eines Gebäudes kein Bauplatz gesucht werden muss und der Bauprozess nicht überwacht werden muss, sondern analog zum Ausbilden von Einheiten abläuft.

5.2.3 BuildOrder

5.2.3.1 Bauziele

Grundlage des Bauplans ist die Entscheidung, nur auf *Zealots* und *Dragoons* als ***Kampfeinheiten*** zu setzen. Zealots sind starke Nahkampfeinheiten, Dragoons sind Fernkampfeinheiten, die sowohl Boden- als auch Lufteinheiten bekämpfen könnten. Mit dieser einfachen Kombination bekommt man schon zu Beginn ganz gute Erfolge hin. Auf Carrier als Lufteinheit wird nur zurückgegriffen, wenn der Gegner Basen hat, die von den eigenen Basen aus nur mit Lufteinheiten erreichbar sind. Auf Spellcaster wie High Templar und Dark Archons wird verzichtet, da für diese ein weit komplexeres Angriffsverhalten implementiert werden müsste, was den Rahmen dieser grundlegenden Arbeit sprengen würde.

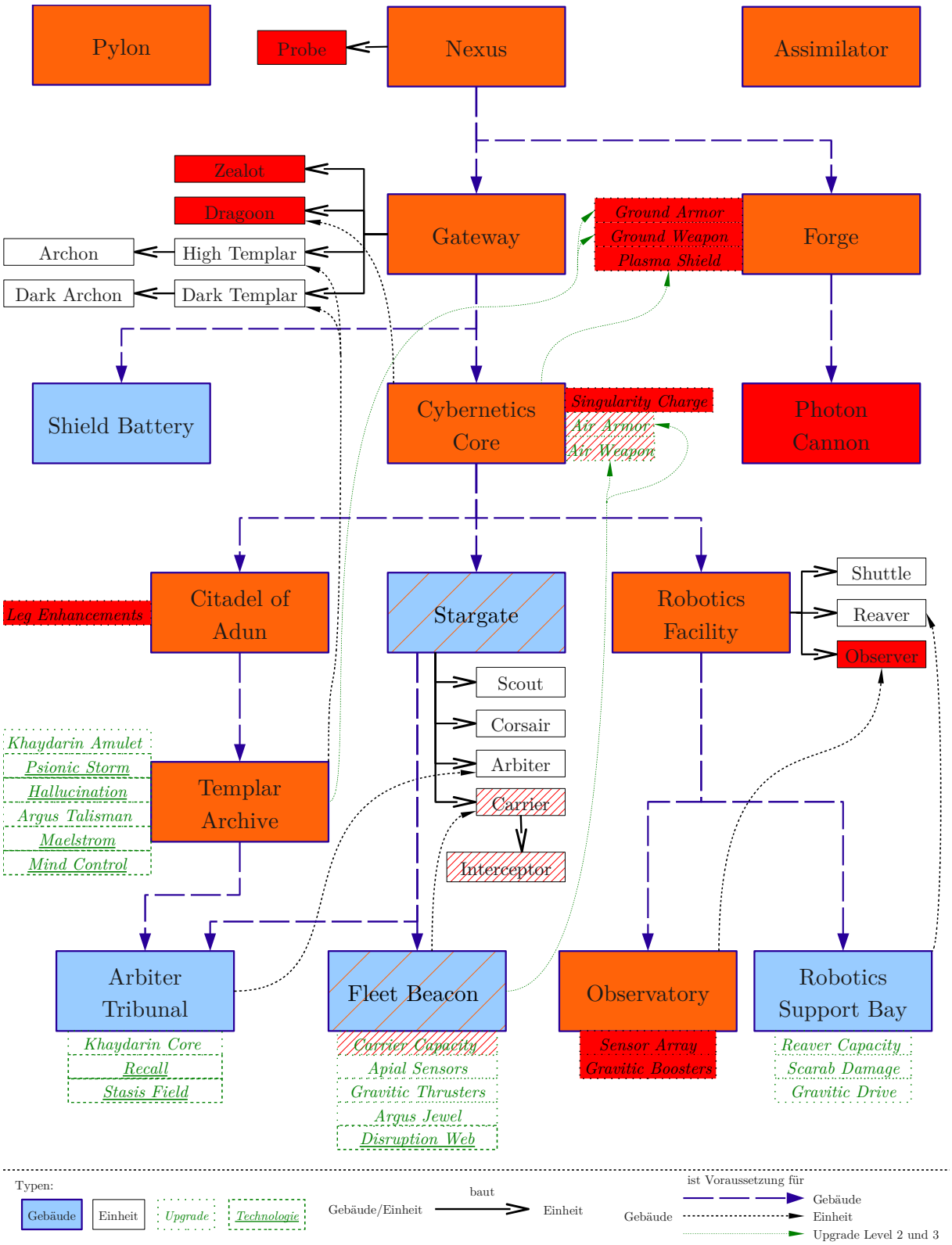


Abbildung 5.2: Tech tree der Protoss mit hervorgehobenen Bauzielen (rot) und den dafür benötigten Gebäuden (orange); schraffiert, falls es sich um nur bei Bedarf gebaute Lufteinheiten handelt

Weiterhin werden *Observer* als ***mobile Detektoren*** benötigt, so dass man gegnerische getarnte oder vergrabene Einheiten bekämpfen können. Ansonsten wäre man diesen wehrlos ausgeliefert und eine solche Einheit könnte die ganze Armee zerstören. Da der Observer eine Flugeinheit ist, kann er auch dazu verwendet werden, alle Bereiche der Karte zu scouten und kann somit auch gegnerische Basen entdecken, die man nicht auf dem Landweg erreichen kann.

Zudem muss entschieden werden, welche ***Upgrades*** erforscht werden. Da Upgrades eine sehr kosteneffektive Möglichkeit sind, die Fähigkeiten aller betroffenen Einheiten zu erhöhen, wurde entscheiden, alle Upgrades, die auf die eigenen Einheiten einen Effekt haben, zu erforschen. Dies sind *Ground Armor* und *Plasma Shield*, welche die Panzerung aller Bodeneinheiten und die Plasmaschilde aller Einheiten verstärken. Diese beiden Upgrades erhöhen auch die Stärke der Panzerung und der Plasmaschilde von Gebäuden. Zudem wird mit dem Upgrade *Ground Weapons* die Stärke der Waffen von Zealot und Dragoon erhöht. Außerdem werden die einheitenspezifischen Updates *Leg Enhancements*, welche die Geschwindigkeit der Zealots erhöht, und *Singularity Charge*, welche die Waffenreiche der Dragoons erhöht, erforscht. Für die Observer werden auch noch zwei Upgrades erforscht: *Sensor Array*, welches deren Sichtreichtweite vergrößert, und *Gravitic Boosters*, welches deren Geschwindigkeit erhöht.

Außerdem wurde die Entscheidung getroffen, in der Anfangsphase des Spieles *Photon Cannons* zu bauen. Diese sind zu Beginn des Spieles eine effektive und kostengünstige Verteidigung, welche zusammen mit den ersten gebauten Einheiten effektiv gegen Rushes verteidigt, was Erfahrungswerte gezeigt haben.

Aus der Wahl der Einheiten und Upgrades ergeben sich die Gebäude, die errichtet werden müssen:

- *Gateway* zum Bauen von Zealots und Dragoons
- *Forge* zum Erforschen der Upgrades Ground Weapons, Ground Armor und Plasma Shield (jeweils Level 1-3) sowie zum Freischalten von Photon Cannon
- *Cybernetic Core* zum Freischalten des Dragoons und Erforschen der Singularity Charge sowie zum Freischalten des Levels 2 und 3 des Upgrades Plasma Shields
- *Citadel of Adun* zur Erforschen des Upgrades Leg Enhancements
- *Templar Archive* zum Freischalten von Level 2 und 3 der Upgrades Ground Weapons und Ground Armors
- *Robotics Facility* zum Bauen von Observern
- *Observatory* zum Freischalten von Observer sowie zum Erforschen der Upgrades Sensor Array und Gravitic Boosters

Dazu kommen die Grundgebäude Nexus, Pylon und Assimilator, welche auf jeden Fall vorhanden sein müssen. Nexus ist das Hauptgebäude, welches Sonden produziert und bei dem Rohstoffe abgegeben werden. Pylonen erzeugen das Psi-Feld, in dem die anderen Gebäude errichtet werden müssen. Assimilatoren sind die Raffinerien der Protoss, welche Vespingas aus Geysiren gewinnen.

Falls der Gegner Basen hat, die nur mit Flugeinheiten von den eigenen Basen aus erreichbar sind, werden *Carrier* als Luftkampfeinheiten gebaut. Hierzu benötigt man dann ein *Stargate*, das die Einheiten baut, und einen *Fleet Beacon*, der die Carrier freischaltet. Durch den Fleet Beacon sind dann auch die Upgrades *Air Armor* und *Air Weapons* freischaltet, welche dann erforscht werden. Zudem wird *Carrier Capacity* erforscht, so dass jeder Carrier 8 statt 4 Interceptoren haben kann. Für jeden Carrier wird die maximale Anzahl an Interceptoren gebaut.

5.2.3.2 Baureihenfolge

Bei der Baureihenfolge wird zwischen verschiedenen Arten von Baumaßnahmen unterschieden. Zum einen gibt es die Routinebaumaßnahmen, wie z.B. neue Arbeiter bauen, Pylonen bauen, falls Supplyknappheit droht oder Kampfeinheit bauen. Zum anderen gibt es die Baumaßnahmen, um den Tech tree aufzusteigen, d.h. um fortgeschrittene Einheiten oder Upgrades freizuschalten. Diese werden ausgeführt, wenn gewisse Voraussetzungen erfülllt. Die Baumaßnahmen sind unterschiedlich gewichtet, es gibt Routinebaumaßnahmen, die wichtiger sind als den Tech tree aufzusteigen, wie z.B. Arbeiter nachzubauen, und welche die niedrigere Priorität haben, wie z.B. neue Kampfeinheiten bauen.

Spielbeginn Am Anfang des Spieles kommt es sehr darauf an, die ersten Rohstoffe gut zu nutzen, eine gute Grundlage für die spätere Strategie zu schaffen und vor allem keine Zeit zu verlieren, sonst ist man Rushes mehr oder weniger schutzlos ausgeliefert. Deshalb gibt es eigens auf dem Spielbeginn abgestimmte Bauregeln, damit da alles reibungslos funktioniert. Es sind immer ein Nexus und vier Arbeiter gegeben. Der Nexus stellt 9 Supply zur Verfügung, wovon 4 schon für die vier gegebenen Sonden verbraucht sind. Zunächst werden Sonden gebaut und sobald 100 Mineralien auf dem Konto sind, wird der erste Pylon gebaut. Dies fällt erfahrungsgemäß meist genau mit dem Zeitpunkt zusammen, wo die 9. Sonde gebaut wird, so dass da keine Supplyknappheit entsteht. Wenn dieser fertig gestellt ist, wird der erste Gateway gebaut. Sobald dessen Bauprozess gestartet ist, wird eine Schmiede gebaut. Ist dieser fertig gestellt, können Photonenkanonen gebaut werden. Um gegen Rushs gewappnet zu sein, werden zwei von ihnen errichtet. Der Cybernetic Core und der Assimilator in der Startbasis werden gebaut, sobald 12 Kampfeinheiten fertig gestellt oder 40 Supply verbraucht sind. Damit sind die Sonderregeln für den Spielbeginn abgearbeitet.

Aufbau des Tech trees Ist der Cybernetic Core fertig gestellt, werden die Upgrades Singularity Charge sowie Ground Armor und Ground Weapon erforscht. Zudem wird eine Citadel of Adun gebaut. Ist diese fertig, können die Leg Enhancements erforscht sowie ein Templar Archive gebaut werden, welches es ermöglicht, Level 2 und 3 der Upgrades Ground Weapons und Ground Armor zu erforschen.

Wenn der Cybernetic Core fertig gestellt ist, kann man zudem in einem anderen Ast des Technologiebaumes weitergehen. Es wird eine Robotics Facility und ein Observatory gebaut. Dies ist nötig, um Observer produzieren zu können und die Upgrades Sensor Array und Gravitic Boosters zu erforschen, welche die Sichtreichtweite und die Geschwindigkeit der Observer erhöhen. Wenn das Templar Archive fertig gestellt ist, aber die Robotics Facility noch nicht in Bau ist, wird deren Bau mittels der Methode `BuildManager::makeBuilding()` erzwungen. Sobald dann auch die Robotics Facility fertig ist, erzwingt man ebenso den Bau des Observatory. Dies ist nötig, da nur angegriffen wird, wenn mindestens ein Observer zur Verfügung steht, vergleiche die Angriffsbedingungen in Abschnitt 5.6.2. Andernfalls würde man immer mehr Kampfeinheiten produzieren, aber in der eigenen Basis verharren, da eben die Angriffsregel nicht erfüllt wird.

Falls festgestellt wurde, dass der Gegner eine Basis hat, die nur mit Lufteinheiten von der eigenen Basis aus erreichbar ist, werden zwei Stargates sowie ein Fleet Beacon gebaut, um Carrier bauen zu können. Zudem werden die Upgrades Air Armor sowie Carrier Capacity erforscht, damit die Carrier 8 statt nur 4 Interceptoren haben können. Für jeden Carrier wird die maximale Anzahl an Interceptoren produziert.

Routinebaumaßnahmen Die wichtigste Routinebaumaßnahmen ist das Bauen von Pylonen, um genügend Supply für andere Einheiten zur Verfügung zu stellen. Ein neuer Pylon wird gebaut, wenn folgende vier Bedingungen erfüllt sind:

1. Es steht gibt schon mindestens einen eigenen Pylonen (ansonsten ist man noch in der Frühphase des Spiels, siehe entsprechenden Abschnitt),

2. das nicht verbrauchtes Supply ist kleiner oder gleich 4 mal die Anzahl der Gateways,

3. das Supplylimit ist noch nicht ausgeschlpft,

4. und es sind weniger Pylonen im Bau als fertige Gateways vorhanden.

Regel 4 dient der Begrenzung der im Bau befindlichen Pylonen, sonst würde aufgrund von Regel 2 solange der Bau neuer Pylonen gestartet, bis genügend fertig sind, damit Regel 2 nicht mehr erfüllt ist.

Dies wird zu Beginn jedes Frames geprüft. Danach kommen die Baumaßnahmen zum Aufbau des Tech tree, siehe vorheriger Abschnitt, und danach werden folgende Routinebaumaßnahmen ausgeführt:

Es wird für jede eigene Basis überprüft, ob sie ein Hauptgebäude, Arbeiter, Pylonen oder einen Assimilator benötigt und falls dies der Fall ist, werden diese gebaut. Hierbei werden auf die `needsCenter()`-, `needsWorker()`-, `needsPylon()`- und `needsRefinery()`-Methoden von `OwnBase` zurückgegriffen. Falls die Basis jedoch angegriffen wird, wird dieser Check übersprungen, da nicht Ressourcen in eine Basis gesteckt werden sollen, die womöglich zerstört wird. Zudem wird überprüft, ob jede Basis eine Photon Cannon hat. Dies wird auch ausgeführt, wenn die Basis angeriffen wird, da Photon Cannons der Verteidigung dienen und helfen können, die Basis zu halten.

Als nächstes wird getestet, ob man drei Observer hat, wenn nicht, werden diese in Auftrag gegeben. Diese werden den Kampfeinheiten gegenüber bevorzugt, da sie relativ leicht zu zerstören sind und daher schnell nachgebaut werden müssen. Dadurch, dass versucht wird, drei Observer zu haben, ist erfahrungsgemäß mindestens einer verfügbar, sodass gegnerische getarnte Einheiten detektiert werden können.

Anschließend wird überprüft, ob man einen neuen Gateway bauen soll. Dies ist der Fall, wenn alle Gateways ausgelastet sind, also Einheiten ausbilden, und noch genügend Mineralien übrig sind, um diesen zu bauen. Dann wird angenommen, dass hinreichend schnell Ressourcen abgebaut werden, um noch einen Gateway mehr auslasten zu können. Drei Einschränkungen gibt es hierbei: es muss schon mindestens einen Gateway geben, sonst befindet man sich noch am Spielbeginn, wo gesonderte Bauregeln gelten, siehe entsprechenden Abschnitt. Weiterhin soll maximal ein Gateway gerade im Bau sein, damit auch die Heuristik für die Auslastung der Gateways nach jedem Gateway neu getestet wird und nicht fälscherlicherweise zu viele neue Gateways auf einmal gebaut werden. Zudem kann es in der Anfangsphase des Spieles vorkommen, wenn man noch wenige Gateways hat, dass diese sehr schnell ausgelastet sind und man dadurch unnötig viele Gateways baut. Es hat sich als sinnvoll erwiesen, die Anzahl der Gateways auf maximal sechs zu beschränken, wenn man nur ein oder zwei eigene Basen hat.

Zuletzt werden die Kampfeinheiten produziert. Werden Luftkampfeinheiten benötigt, weil der Gegner eine Basis hat, die nur auf dem Luftweg zu erreichen ist, produziert man zuerst Carrier und Interceptoren, falls das möglich ist. Bei den Bodenkampfeinheiten wird versucht ein vorher definiertes Verhältnis von Dragoons und Zealots zu erreichen. Steht jedoch kein Gas zur Verfügung steht oder sind noch nicht die notwendigen Gebäude (Assimilator, Kybernetikkern) zum Bau von Dragoons gebaut, werden nur Zealots gebaut und das Zielverhältnis ignoriert.

5.3 Gebäude bauen

Das Bauen es eines Gebäudes ist, wie oben bereits angedeutet, komplizierter als das Trainieren oder Morphen einer Einheit. Hierbei muss nicht nur ein Bauplatz für das Gebäude gewählt werden, sondern der gesamte Bauprozess überwacht werden.

5.3.1 Wahl der Bauposition

Für die Bauplatzwahl ist die Klasse `FengShuiMan` zuständig. Sie ist als Singleton implementiert. Der Bauplatz eines Gebäudes wird über dessen linken, oberen `TilePosition` beschrieben. Die weiteren benötigten `TilePositions` ergeben sich aus dem `UnitType` des Gebäudes.

Die Bauplatzsuche geht grundsätzlich in „rechteckigen Spiralen“ ausgehend von einer Startposition los. Mit „rechteckiger Spirale“ ist hierbei gemeint, dass in jeder Iteration alle Punkte auf dem Rand des Rechtecks mit der linken oberen Ecke $startposition - (radius, radius)$ und der rechten unteren Ecke $startposition + (radius, radius)$ getestet werden. Es werden maximal 15 Werte für $radius$ durchprobiert, außer zu Beginn, da wird $radius$ von 0 bis 25 iteriert. Wird dabei kein gültiger Bauplatz für das Gebäude gefunden, wird die Suche abgebrochen und `TilePositions::None` zurückgegeben. Da dieser Vorgang sehr lange dauern kann, aber der Bot das Spiel nicht lahm legen soll, wird die Bauplatzsuche zeitlich beschränkt. 38ms nach Framebeginn wird die Suche abgebrochen, wenn bis dahin kein Bauplatz gefunden wurde. Dies lässt noch knapp über 3ms für andere Operationen übrig, denn StarCraft läuft mit 24 Frames pro Sekunde, was eine Framedauer von $\frac{1000\,\text{ms}}{24} = 41{,}\overline{6}\,\text{ms}$ ergibt.

5.3.1.1 Wahl der Startposition

Die Startposition wird in Abhängigkeit vom Gebäudetyp gewählt. Grundidee hierbei ist es, die Gateways an den Ausgängen zu platzieren und dahinter die Photonenkanonen, damit diese von den Gateways geschützt werden und die Ausgänge abdecken. Die anderen Gebäude, außer Pylonen, sollen zwischen den Kanonen und dem Hauptquartier platziert werden, um geschützt durch die Verteidigungsanlagen an den Ausgängen und außer Reichweite von Einheiten mit Fernwaffen zu sein. Pylonen werden abwechselnd in beiden Bereichen platziert, damit alle Gebäude und möglichst große Bereiche mit Psi versorgt werden. Wenn zwischenzeitlich ein Gebäude ohne Psi ist, wird versucht, den nächsten Pylon in dessen Nähe zu positionieren, damit dieses Gebäude wieder funktionstüchtig ist. Die Startposition für allgemeine Gebäude errechnet sich als Mittelwert der Mittelpunkte der Ausgänge und der Position des Hauptquartiers, außer es handelt sich um eine Basis mit großer Fläche (siehe 5.4.3), dann wird die Position des Hauptquartiers genommen. Für die Startposition der Gateways wird abwechselnd ein Ausgang gewählt, der nicht in eine Sackgasse führt. Von dessen Mittelpunkt aus wird senkrecht in das Basisinnere gegangen: 200px, falls der Ausgang schmaler als 150px ist, sonst 100px. Bei diesem Punkt wird dann die Suche nach einem Bauplatz für Gateways begonnen. Dieser Punkt wird auch als die *Bauposition des Ausgangs* bezeichnet. Bei der Startposition für Photonenkanonen wird ganz analog vorgegangen, nur die Abstände sind größer, da die Photonenkanonen hinter den Gateways platziert werden sollen. Es wird 250px nach innen gegangen, falls der Ausgang weniger als 150px breit ist, ansonsten 150px.

Abbildung 5.3: Die Startpositionen dieser Basis:
grün: Gateways
pink: Photonenkanonen
rot: allgemeine Gebäude

5.3.1.2 Zulässigkeit des Bauplatzes

Für die Überprüfung, ob eine Position als Bauplatz für ein Gebäude geeignet ist, ist die Funktion `FengShuiMan::testTilePosition(TilePosition tileposition, UnitType type, OwnBase* ownbase, int radius, bool saveLastPositionAndRadius)` zuständig. Diese prüft zuerst, ob die `TilePosition` überhaupt in der verlangten Basis liegt. Weiterhin darf keine der benötigten Tiles schon für einen anderen Bauprozess reserviert sein. Mittels der von BWAPI zur Verfügung gestellten Funktion `Game::canBuildHere(Unit*, TilePosition, UnitType, bool)` wird getestet, ob dort grundsätzlich gebaut werden kann. Dazu gehört, dass die Tiles bebaubar sind, dass keine Gebäude dort schon konstruiert sind oder sich Einheiten dort befinden. Zudem überprüft diese Funktion die rassenabhängigen Bedingungen: für die Protoss, ob Psi vorhanden ist, für die Zerg, dass dort Kriecher ist und für Terraner und Protoss, dass sich dort kein Kriecher befindet. Danach müssen noch zwei weitere Bedingungen erfüllt werden, bevor diese Position als Bauplatz für das Gebäude ausgewählt wird: Sie darf sich in keinem sogenannten illegalen Bereich befinden und es muss sichergestellt werden, dass keine Einheiten eingebaut werden, also die Ausgänge der Basis erreichbar bleiben.

Abbildung 5.4: Die roten Polygone sind die illegalen Bereiche dieser Basis.

5.3.1.3 Illegaler Bereich

Ein Gebäude darf nicht in einem sogenannten *illegalen Bereich* errichtet werden. Von diesen gibt es zwei Arten: zum einen das Gebiet zwischen Ressourcen und Hauptquartier. Dort soll nicht gebaut werden, da dies die Arbeiter beim Ressourcenabbau behindern und so die Abbaugeschwindigkeit verringern würde. Zu dessen Berechnung werden die Ecken des Hauptquartiers und aller Ressourcen zu einem `Polygon` hinzugefügt und dessen konvexe Hülle mittels des Graham-Scan-Algorithmuses berechnet. Die anderen illegalen Bereiche befinden sich um die Basisausgänge herum, damit diese nicht zugebaut werden und keine Einheiten in der Basis eingesperrt werden. Dort wird jeweils die Verschiebung der Bauposition des Ausgangs zum Ausgangsmittelpunkt genommen. Dieser Vektor wird auf die beiden Endpunkte des Ausgangs jeweils addiert und subtrahiert und die so erhaltenen vier Eckpunkte bestimmen das Rechteck des nichtzubebauenden Bereichs um diesen Ausgang herum. Keine Ecke des geplanten Gebäudes darf sich in einem dieser illegalen Bereiche befinden. Zudem wird getestet, ob sich der Mittelpunkt des Gebäudes in einem der illegalen Bereiche befindet. Andernfalls könnte es bei einem großen Gebäude und einem entsprechend gelegenen illegalen Bereich vorkommen, dass, obwohl sich das Gebäude zu einem Teil im illegalen Bereich befindet, alle Eckpunkte des Gebäudes außerhalb des Bereiches liegen.

Abbildung 5.5: Der Bereich in der oberen Hälfte des Bildes, in dem keine Zahlenwerte stehen, ist durch Gebäude von den Ausgängen der Basis abgetrennt.

5.3.1.4 Erreichbarkeit der Ausgänge

Zum Schluss wird noch ein aufwendiger Test durchgeführt, um sicherzustellen, dass kein Bereich der Basis von einem Ausgang abgeschnitten wird. Hierbei ist es nötig, dass dies für alle Ausgänge getestet wird, da sonst die Einheiten dennoch in einem Bereich der Karte feststecken könnten, wenn die Ausgänge, die erreicht werden können, eben nicht zu allen möglichen Bereichen der Karte führen können. Dieser Test wird mit Hilfe eines Einflussfeldes durchgeführt. In diesem Feld bedeuten positive Werte, dass die Quelle erreicht werden kann, negative Werte stehen für Hindernisse auf dem Weg zur Quelle und Tiles mit dem Wert 0 können die Quelle nicht erreichen.

Hier sind die Hindernisse die Tiles, die durch existierende oder geplante Gebäude, Mineralien und Geysire belegt sind, sowie die Tiles, die das Gebäude, für das ein Bauplatz gesucht wird, belegen würde. Weiterhin gehören dazu die Tiles, die aufgrund der Gegebenheiten der Karte immer als blockiert gefunden werden würden. Diese werden beim Anlegen der Basis berechnet, in dem selbige Methode genutzt wird, jedoch nur mit Geysiren und Mineralien als Hindernisse. Durch die Belegung mit einem negativen Wert sind diese Tiles als Hindernis gekennzeichnet. Alle anderen Tiles werden mit 0 initialisiert. Dann wird der Mittelpunkt des Ausgangs als Quelle des Potentialfeldes mit dem Wert 1 hinzugefügt. Diese Tile wird auch einer Queue hinzugefügt, die alle Tiles enthält, deren Wert sich geändert hat. Jeweils das erste Element der Queue wird herausgenommen, und dann der Wert deren Nachbartiles aktualisiert. Es werden hierfür die links und rechts sowie oben und unten, aber nicht die diagonal benachbarten Tiles, die sich in der Basis befinden, genommen. Diagonal benachbarte Tiles werden ausgenommen, da sich das Feld sonst zwischen Gebäuden, die diagonal Ecke an Ecke grenzen, ausbreiten könnte, wo jedoch keine Einheit durchkommen kann, siehe Abbildung 5.6. Diese Nachbartiles werden auf 99 % des Werts der Tile gesetzt, außer sie haben schon einen Wert, der größer als dieser neue Wert ist, oder einen negativen Wert. Alle Tiles, deren Wert geändert wurde, werden der

Queue hinzugefügt. Dies wird solange ausgeführt, bis die Queue leer ist. Befindet sich nun innerhalb der Basis eine Tile, die den Wert 0 hat, so ist diese von diesem Ausgang nicht erreichbar und der Bauplatz wird verworfen.

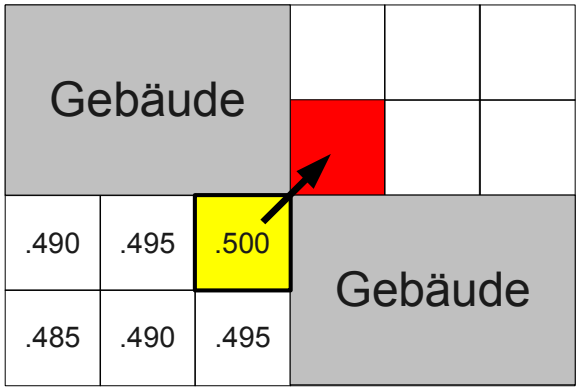


Abbildung 5.6: Würde sich das Potentialfeld auch diagonal ausbreiten, könnte es passieren, dass es sich zwischen zwei Gebäuden, die sich nur an den Ecken berühren, ausbreitet, obwohl dort keine Einheiten durchkommen können.

Für jede Tile, die benachbart ist und der eventuell ein neuer Wert zugewiesen werden soll, muss getestet werden, ob sie in der Basis ist. Dieser Test, ob die Tile im Polygon der Basis ist, ist nicht trivial und, da er sehr oft durchgeführt werden muss, in der Summe dann auch sehr zeitintensiv. Daher wurde nicht auf die Methode `Polygon::isInside()` zurückgegriffen, sondern eine effizientere Variante implementiert. Hierbei wird der Rand des Basispolygons berechnet und im Potentialfeld gekennzeichnet. Die Eckpunkte des Polygons liegen in Pixelauflösung vor, während das Potentialfeld auf Tiles operiert. Deswegen werden die Kanten des Polygons mittels des Bresenham-Algorithmus berechnet. Der Bresenham-Algorithmus ist ein Linienrasterisierungsalgorithmus. Ein solcher wird normalerweise verwendet, um eine Line, deren Eckpunkte aus einem kontinuerlichen Wertebereich stammen, auf einen diskreten Wertebereich abzubilden und alle Punkte der Linie zu berechnen. Dieses Verfahren gehört zu den grundlegenden Aufgaben der Computergrafiken. Es eignet sich hier auch, um aus einem feineren auf einen gröberen Wertebereich abzubilden. Die so berechneten Kantentiles werden mit -1 in das Potentialfeld eingetragen. Das obige Verfahren erkennt diese dann als Hindernis und weist diesen keinen Wert zu. Die berechneten Randitles weisen auch keine Lücke auf, sodass das Potentialfeld nicht fälscherlicherweise über den eigentlichen Rand hinaus berechnet wird. Für große Basen (vgl. 5.4.3) wird nicht das ganze Polygon genommen, sondern ein kleineres Basispolygon berechnet. Die Eckpunkte dieses Polygons sind die Ecken des Regionpolygons, die weniger als die 1500 Pixel entfernt sind, die als Grenze der Zugehörigkeit definiert wurde. Durch diese Methode konnte ein signifikanter Geschwindigkeitszuwachs erzielt werden.

Ein Spezialfall ist noch bei produzierenden Gebäuden zu beachten. Da die neu produzierten Einheiten bevorzugt an der Gebäudeunterseite erscheinen, muss darauf geach-

tet werden, dass zwischen dieser und dem Basisrand eine Tilereihe frei ist. Andernfalls kann es vorkommen, dass die Einheiten in dem geringen Platz zwischen Rand und Gebäude herauskommen, aber dort gefangen sind, weil an anderen Stellen der Abstand zwischen Rand und Gebäude geringer ist, was jedoch durch die Abbildung auf Tiles nicht festgestellt werden kann. Auf Abbildung 5.7 ist diese Situation zu sehen.

Abbildung 5.7: Die Zealots fanden Platz zwischen dem Gateway und dem Rand der Basis (grün), jedoch am linken unteren Eck des Gateways ist der Abstand zu gering zum Rand, sodass die Einheiten dort eingebaut sind.

Wurde letztendlich ein Bauplatz für das Gebäude gefunden, so werden diese Tiles reserviert, damit sie nicht von anderen Gebäuden belegt werden, bevor das Gebäude gebaut wird. Die Reservierung wird aufgehoben, wenn dort wirklich ein Gebäude errichtet wird, da sie dann nicht mehr nötig ist, oder wenn das Bauvorhaben abgebrochen wird.

5.3.2 PendingBuildOperation

Die Klasse `PendingBuildOperation` modelliert den Bauprozess eines Gebäudes vom Zeitpunkt der Entscheidung ein Gebäude zu bauen bis zu dessen erfolgreichen Abschluss. Die wichtigste Aufgabe dieser Klasse ist es, diesen Bauprozess zu überwachen und eventuell auftretende Probleme zu korrigieren. Hierzu wird die Methode `checkExecution()` aller existierenden `PendingBuildOperation`-Instanzen in jedem Frame vom `BuildManager` aufgerufen. Der Rückgabewert dieser Methode ist ein `boolean`, welcher anzeigt, ob der Bauprozess korrekt abläuft (`true`) oder ob der Bauprozess abgebrochen werden muss (`false`), weil ein schwerwiegender Fehler auftrat.

Ein Bauprozess startet mit einen Gebäudetyp, einer Tileposition als Bauplatz und einer Basis, in der das Gebäude errichtet werden soll. Diese Informationen werden dem

Konstruktor übergegeben, wenn eine `PendingBuildOperation`-Instanz erstellt wird. Als erstes muss ein Arbeiter der Basis angefordert werden, der dieses Gebäude errichtet. Kann diese Basis keinen Arbeiter bereitstellen, werden alle anderen Basen angefragt, solange bis ein Arbeiter abgestellt werden kann oder alle Basen durchprobiert wurden. Konnte kein Arbeiter gefunden werden, wird der Bauprozess abgebrochen.

Ist dem Bauprozess ein Arbeiter zugeteilt, wird anschließend geprüft, was er tut. Ist er auf dem Weg ein Gebäude zu konstruieren, konstruiert es oder weicht aus, um zum Bauplatz zu kommen, ist alles in Ordnung und die Überprüfung beendet. Andernfalls wird zuerst überprüft, ob alle Tiles, auf denen das Gebäude errichtet werden soll, sichtbar sind, denn nur dann kann ein Baubefehl erfolgreich gegeben werden. Ist dies nicht der Fall, wird der Arbeiter zu dem Bauplatz geschickt, bis er komplett sichtbar ist und der Baubefehl gegeben werden kann. Ansonsten erhält der Arbeiter den Befehl, das gewünschte Gebäude an der entsprechenden TilePosition zu bauen.

Ein weiterer Grund, warum nicht gebaut werden kann, ist, dass der Bauplatz unbrauchbar geworden ist. Dies ist zum einen der Fall, wenn man ein Protoss-Gebäude bauen will und der Bauplatz nicht mehr im Psi-Feld liegt. Zum anderen kann das Problem mit dem Kriecher auftreten: wenn man ein Zerggebäude errichten will und mittlerweile kein Kriecher mehr vorhanden ist, oder umgekehrt, wenn man ein Nichtzerggebäude bauen will, aber sich der Kriecher auf den gewünschten Bauplatz ausgebreitet hat. Dann wird eine neue Position für das Gebäude gesucht. gesucht. Zwar werden diese Bedingungen auch während der Bauplatzsuche abgefangen, aber durch das zwischenzeitliche Zerstören oder Errichten von Gebäuden können sich diese Bedingungen am Bauplatz ändern.

Weiterhin kann es noch sein, dass sich Einheiten auf dem Bauplatz befinden. Steht bereits ein Gebäude dort, muss ein neuer Bauplatz gesucht werden. Dies wird auch getan, wenn sich gegnerische Bodeneinheiten auf diesem befinden. Stören eigene Bodeneinheiten den Start der Konstruktion des Gebäudes, werden diese weggeschickt.

Wurde der Bauprozess erfolgreich an der gewünschten Stelle gestartet, so wird für das Gebäude `Dreadbot::onUnitCreate(Unit*)` oder `Dreadbot::onUnitMorph(Unit*)` aufgerufen. Diese Funktionen weisen das Gebäude der zugehörigen `PendingBuildOperation` zu.

Dann verkürzt sich unter Umständen die Überprüfungsprozedur. Ist es nämlich ein Zerg- oder Protoss-Gebäude, muss nichts weiter überprüft werden, da der Bauprozess dann von alleine seinen Gang geht. Bei terranischen Gebäuden wird weiterhin überprüft, ob ein Arbeiter das Gebäude konstruiert. Ist dies nicht der Fall, wird ein Arbeiter dazu angefordert.

Ist der Instanz ein Gebäude zugewiesen worden, dies aber zerstört worden, wird der Bauprozess nicht neu gestartet, sondern die `PendingBuildOperation`-Instanz zerstört. Es ist in diesem Fall angedacht, dass ein BuildOrderPlaner entscheidet, ob

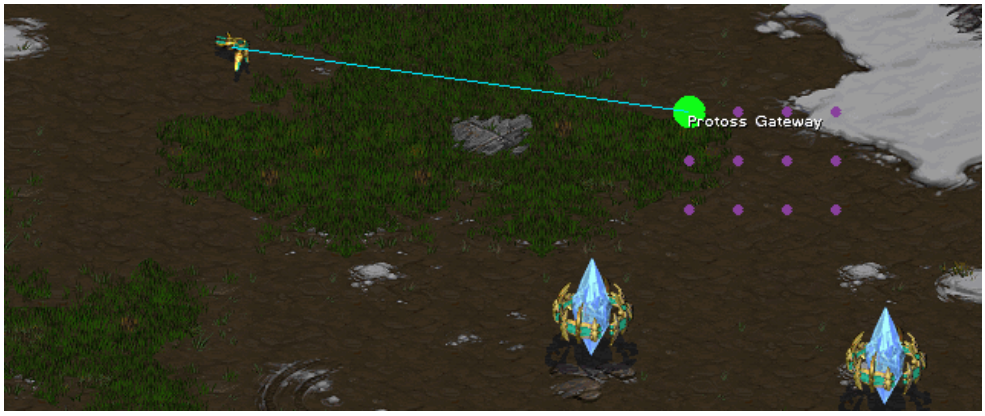


Abbildung 5.8: Visualisierung einer `PendingBuildOperation`-Instanz
Der Arbeiter ist auf dem Weg zur grün markierten Position, um einen Gateway zu bauen. Die lila markierten Tiles sind für diesen reserviert.

dieses Gebäude neu gebaut werden soll oder ob es mittlerweile andere Prioritäten gibt.

Ist dieses Gebäude dann fertig gestellt, so wird die Instanz vom `BuildManager` vernichtet, da sie nicht mehr benötigt wird.

5.4 Basismanagement

5.4.1 OwnBase, Woman & Platoon

Eine Basis definiert sich über ihre `BaseLocation`. Für jede eigene Basis wird eine Instanz der Klasse `OwnBase` erstellt. Sie verwaltet die Gebäude und `PendingBuildOperations` der Basis. Jeder Basis ist auch als WorkerManager eine Instanz der Klasse `Woman` zugewiesen. Diese verwaltet alle Arbeiter und Raffinerien der Basis. Ziel ist es, dass für jedes Mineralienfeld zwei und für jede Raffinerie drei Arbeiter abgestellt sind. Zudem wählt `Woman` die Arbeiter aus, die für Bauprozesse oder zum Scouten abgestellt werden können. Hierbei werden zuerst Arbeiter, die Mineralien sammeln ausgewählt, danach Raffineriearbeiter. Da es keinen Callback gibt, der mitteilt, dass eine Einheit oder Gebäude fertiggestellt ist, werden Arbeiter und Raffinerien schon bei ihrem Entstehen dem `WorkerManager` der Basis übergeben. Dieser verwaltet die unvollständigen Arbeiter und Raffinerien gesondert und überprüft in jedem Frame, ob sie fertig gestellt sind. Wenn dies der Fall ist, werden den Arbeitern eine Aufgabe bzw. den Raffinerien Arbeiter zugewiesen. Zudem wird in jedem Frame überprüft, ob arbeitslose Arbeiter vorhanden sind und diesen Aufgaben zugewiesen, oder sie dem `Baseman` zur Verfügung gestellt, um sie anderen Basen zuzuweisen. Jeder Basis wird zudem eine `Platoon`-Instanz zugeordnet, in dem die Kampfeinheiten organisiert sind, die diese Basis bewachen.

Abbildung 5.9: Der Bereich in der rechten Hälfte des Bildes innerhalb des grünen Randes, in dem keine Zahlenwerte stehen, ist aufgrund der Kartengegebenheiten abgetrennt. Dieser Bereich wird vorab berechnet und bei der Gebäudeplatzierung als belegt markiert.

Beim Erstellen der `OwnBase`-Instanz einer Basis werden zudem die illegalen Bereiche dieser Basis berechnet (siehe Abschnitt 5.3.1.3). Dadurch müssen diese nur einmal berechnet werden und können bei jeder Gebäudeplatzierung von `FengShuiMan` einfach abgefragt werden. Weiterhin werden die Tiles berechnet, welche schon aufgrund der Gegebenheiten der Karte wie Mineralien, Geysire und dem Rand der Basis immer als solche berechnet werden, die den Ausgang nicht erreichen können. Diese werden vorab ermittelt und gespeichert, so dass sie bei jeder Gebäudeplatzierung als besetzt markiert werden. Andernfalls würden diese bei jeder Bauposition der Basis als Tiles, die den Ausgang nicht erreichen können, gefunden werden und damit würde jede Position der Basis als nicht geeignet verworfen werden, siehe hierzu Abbildung 5.9.

5.4.2 Baseman

Der `Baseman` ist zuständig für die Verwaltung der eigenen Basen. Er wird auch angefragt, wenn ein neues Gebäude gebaut wird. Er entscheidet, in welcher Basis dies errichtet werden soll, damit die Gebäude möglichst gleichmäßig auf die Basen verteilt werden. Weiterhin teilt er neue oder unbenötigte Arbeiter Basen zu, die diese benötigen. Er kümmert sich auch um das Entfernen von zerstörten Basen, dabei werden die der Basis zugeteilten Arbeiter und Kampfeinheiten auf andere Basen verteilt und `PendingBuildOperations` abgebrochen. Eine Basis gilt als zerstört, wenn sie keine existierenden Gebäude mehr hat.

Weiterhin kümmert der `Baseman` sich um das Expandieren. Grundvoraussetzung hierfür ist, dass mindestens 12 Kampfeinheiten, also Dragoons und Zealots vorhanden sind, da man sonst nicht mehrere Basen verteidigen kann. Zum einen wird eine neue Basis eröffnet, wenn es an Ressourcen mangelt, d.h. wenn nur noch 5 oder weniger

Mineralienfelder abgebaut werden oder keine Raffinierie mit mehr als 250 Gas und zudem kein ungenutzter Geysir vorhanden ist. Ebenso wird expandiert, wenn man mindestens doppelt so viele Gateways wie Basen hat. Man benötigt dann mehr Ressourcen, um die Gateways besser auslasten zu können, weswegen expandiert wird. Als Ort für die neue Basis wird diejenige ungenutzte `BaseLocation` gewählt, die am nächsten zu einer eigenen Basen liegt. Falls entdeckt wird, dass sich dort doch schon eine gegnerische Basis befindet, wird die Expansion abgebrochen und eine andere gewählt. Der neuen Basis wird gleich ein Arbeiter zugewiesen, um einen Nexus und einen Pylon zu bauen, so dass dort weitere Gebäude errichtet werden können.

5.4.3 Basis mit großflächiger Region

Viele Karten enthalten, zumeist in der Mitte, eine sehr große Region, die meist auch mehrere `BaseLocation`s hat. Errichtet man dort eine Basis und würde wie sonst die ganze Region als der Basis zugehörig definieren, würde man einen sehr großen Bereich der Karte als die eigene Basis ansehen, obwohl sich die eigene Basis nur auf einem Bruchteil der Fläche befindet. Deshalb wurden an verschiedenen Stellen Fallunterscheidungen implementiert, um diese Fälle geeigenet zu behandeln.

Hierbei ist zu bedenken, dass auf den verwendeten Turnierkarten frühestens die zweite Expansion, also die dritte eigene Basis, in einer solchen Region gebaut wird. Diese Basen dienen dann hauptsächlich dazu, die Rohstoffversorgung sicherzustellen. Dort werden dann gemäß der Verteilung der Gebäude auf die verschiedenen Basen eigentlich nur Pylonen und Gateways gebaut. Die Technologiegebäude befinden sich primär in der Anfangsbasis und nur vereinzelt in der ersten Expansion. Dadurch kann man gewisse Regeln der Gebäudeplatzierung hier lockern, solange sichergestellt ist, dass die Arbeiter nicht behindert werden.

Als Grenze der Regionengröße haben sich 1.500.000 Pixel als zweckmäßig erwiesen. Für andere Abschätzungen hat sich zudem als hilfreich erwiesen, einen Abstand von 1500 Pixel als Grenze der Basiszugehörigkeit einer Position zu definieren.

Wenn berechnet wird, ob ein Punkt sich innerhalb einer Basis befindet, wird bei Basen, die sich in einer Region mit großer Fläche befinden, neben der Zugehörigkeit zu der Region zusätzlich getestet, ob der Punkt höchstens einen Abstand von 1500 Pixeln von der BaseLocation hat. Nur dann wird der Punkt als in der Basis angesehen, andernfalls als außerhalb. Dies spielt auch eine Rolle bei der Berechnung der Ausgänge. Würde das für die ganze Region mit großer Fläche getestet werden, würde das viel zu lange dauern. Deswegen wird nicht das ganze Regionspolygon genommen, sondern nur das Polygon aus den Eckpunkten, die weniger als 1500 Pixel von der BaseLocation entfernt sind. Als „Ausgang“, der erreicht werden muss, wird hierbei der Mittelpunkt der Strecke genommen, die dieses Teilpolygon vom Restpolygon trennt. Die initiale Bauposition bei der Bauplatzsuche für Basen mit großer Region ist immer die TilePosition der BaseLocation selbst und wird nicht nach Gebäudetyp

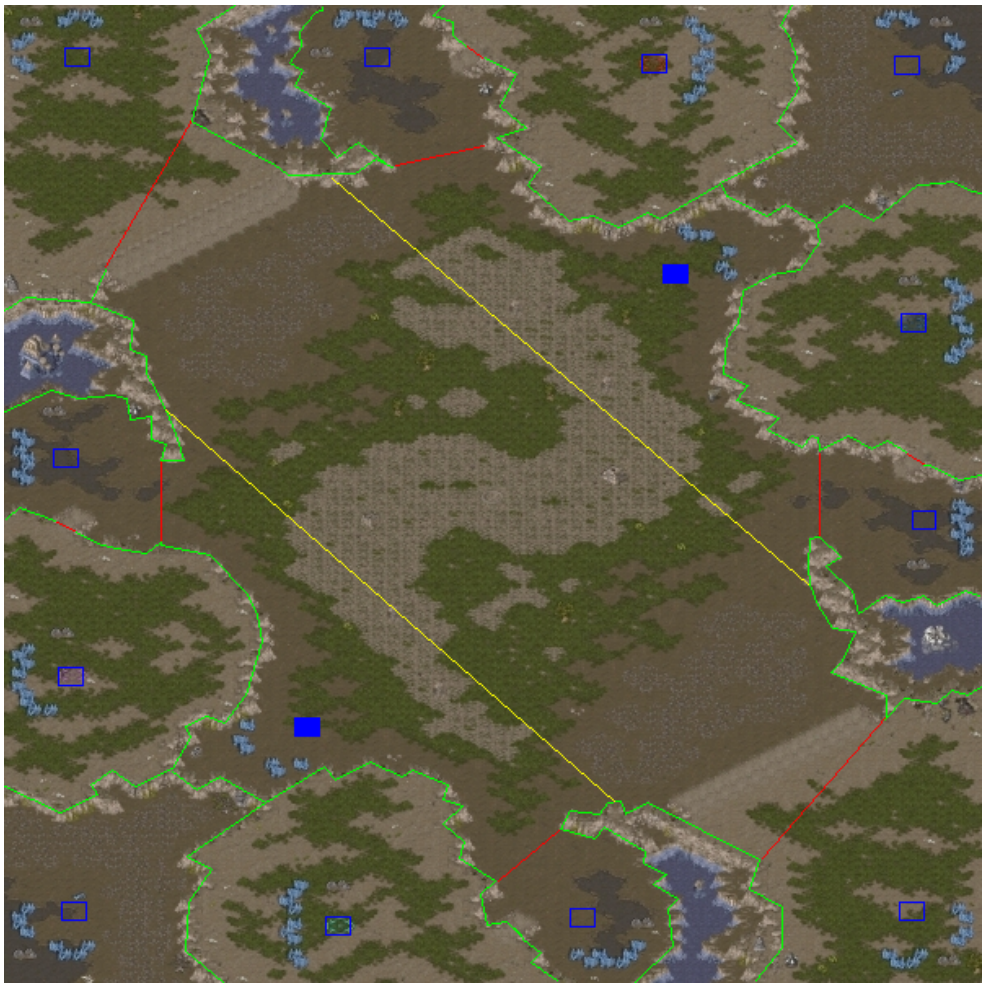


Abbildung 5.10: Karte „Python“ mit eingezeichneten Regionen.

Blaue Rechtecke sind die `BaseLocations`, grün die `Regionengrenzen` und rot die `Chokepoints` zwischen diesen.

Beginnend von links oben haben die Regionen im Uhrzeiger die Größe 783.008, 364.897, 884.672, 582.784, 877.152, 330.087, 803.200, 302.784, 829.376, 530.656, 868.000, 319.936. Die große mittlere Region mit der Schlange hat die Größe 6.520.136.

Die beiden gefüllten blauen Rechtecke stellen BaseLocations dar, die sich in dieser Region mit großer Fläche befinden. Durch die gelben Linien werden kleinere Bereiche abgetrennt, die die zur Basis gehörende Fläche bestimmen. Deren Polygon besteht nur aus Eckpunkten, die weniger als 1500 Pixel von der BaseLocation entfernt sind. Die dadurch entstanden Bereiche haben die Größen 1.233.288 (oben) und 1.341.248 (unten).

unterschieden. Ebenso ist der Sammelpunkt der Kampfeinheiten beim Bewachen der Basis nicht wie sonst der Mittelpunkt der Region, da sich dieser in diesem Fall weit entfernt von den eigenen Gebäuden befindet. Stattdessen wird der Punkt auf der Verbindungslinie zwischen diesem Mittelpunkt und der BaseLocation gewählt, der sich 200 Pixel von der BaseLocation entfernt befindet. Eine Rolle spielt die Regiongröße zudem bei der Berechnung des nächsten Sammelpunktes der Kampfeinheiten bevor die gegnerische Basis angegriffen wird. Nähere Information hierzu sind in Abschnitt 5.6.1 zu finden. Beim Scouten ist das auch relevant. Während bei normalen Basen die Rohstoffe und alle Eckpunkte des Regionspolygons Scout angesteuert werden, wird bei großen Regionen hierauf verzichtet.

5.5 Scouten

Für das Scouten sind vier Klassen zuständig: der `ScoutMan` kümmert sich um die Steuerung der scoutenden Einheit, die `Central_Intelligence_Agency` (Synonym `CIA`) verwaltet und analysiert die gefundenen Informationen. Dazu nutzt sie die Klassen `UnitInfo`, welche für jede gesichtete gegnerische Einheit deren Zustand speichert, und `EnemyBase`, welche gegnerische Basen, also deren Position und vor allem die zugehörigen Einheiten, verwaltet.

Der `ScoutMan` schickt einen Scout[2] los, wenn 9 und wenn 25 Supply erstmals verbraucht sind, und danach immer 2500 Frames nach dem Ende des vorigen Scoutvorgangs. Beim allerersten Scouten werden gezielt die möglichen StartLocations der Karte angesteuert, ansonsten werden alle BaseLocations durchgegangen. Dazu wird für jede BaseLocation der Frame gespeichert, in dem sie zuletzt sichtbar war. Dann wird immer die BaseLocation als Ziel ausgewählt, die am längsten nicht sichtbar war. Ist der Scout in der Zielbasis angelangt, steuert er dort alle Eckpunkte des Basispolygons und alle Ressourcen an, bis diese mal sichtbar waren, um so möglichst die ganze Basis zu sichten. Hat er diese alle gesehen oder wird er angegriffen, wird eine neue BaseLocation als Ziel gewählt. Das Scouten wird nur beendet, wenn der Scout zerstört wird oder Feinde bekämpft werden.

Für das Spielende wurde noch eine besondere Scouting-Funktion eingebaut. Hier kann es vorkommen, dass man als überlegener Bot die gegnerische Armee vernichtet hat und alle einem bekannten gegnerischen Gebäude zerstört hat, aber noch nicht gewonnen hat, weil man noch nicht alle gegnerischen Gebäude gefunden hat. Wenn man diese nicht findet, wird das Spiel nach einer gewissen Zeit, die vom Wettbewerb abhängt, nur als Unentschieden gewertet und man verschenkt somit den Sieg. Daher ist es wichtig, sicherzustellen, dass alle Gebäude gefunden werden. Zuerst wird wie gewöhnlich ein Scout losgeschickt, der alle BaseLocations absucht. Findet der nichts, werden in jedem Frame alle sichtbaren Tiles ausgelesen und gespeichert, und der

[2]Es gibt auch eine Protoss-Einheit mit dem Namen Scout, aber hier ist mit Scout immer die scoutende Einheit gemeint, welche eine Sonde oder ein Observer ist.

Scout gezielt zu Tiles geschickt, die seitdem noch nicht sichtbar waren. Zusätzlich werden alle 1000 Frames alle eigenen Einheiten außer den Arbeitern zufällig zu irgendwelchen bis dato nicht sichtbaren Tiles geschickt, um mit mehr Einheiten die Karte zu durchkämmen und möglichst schnell einen großen Bereich der Karte abzudecken. Die Funktion hierfür ist `CombatMan::helpSearchingEnemyBases()`. Obwohl dies eine Funktion zum Scouten ist, ist sie beim `CombatMan` angesiedelt, da dieser für die Steuerung der Kampfeinheiten zuständig ist. Durch die Benennung ist auch klargestellt, dass dies nur eine Hilfsfunktion ist, die vom `ScoutMan` aufgerufen werden soll.

Für jede gesichtete gegnerische Einheit wird eine `UnitInfo`-Instanz angelegt. Diese speichert alle relevanten Informationen, wie Typ, Position, ob die Einheit existiert, zu welcher Basis sie gehört und wann diese Informationen zuletzt aktualisiert wurden. Diese Informationen werden in jedem Frame geupdatet, in dem die Einheit sichtbar ist. Eine Einheit wird als zerstört markiert, wenn ein entsprechernder `onUnitDestroy()`-Aufruf für diese Einheit erfolgt. Gebäude werden zudem als zerstört markiert, wenn deren Position sichtbar ist, aber das Gebäude nicht.

Jede gegnerische Einheit wird nach Möglichkeit einer gegnerischen Basis zugewiesen. Dazu wird die nächste BaseLocation berechnet. Ist die Einheit ein Gebäude und ist der BaseLocation noch keine EnemyBase zugeordnet, wird für die BaseLocation eine `EnemyBase`-Instanz erzeugt. Existiert für die BaseLocation eine EnemyBase so wird die Einheit dieser zugeordnet. Ansonsten wird sie keiner EnemyBase zugeordnet.

Die `CIA` verwaltet die Informationen über gegnerischen Bases und Einheiten und sorgt für deren Updates. Zudem hat sie die für die Zustandsübergänge des CombatMans sehr wichtige Methode `areThereEnemyUnitsWeMustAttack()`. Diese entscheidet, falls gegnerische Einheiten sichtbar sind, ob diese bekämpft werden sollen oder ob es nur vom Scout erspähte Gegner sind. Ohne diese Unterscheidung würde die eigene Armee immer in die gegnerische Basis stürmen, wenn der Scout diese auskundschaftet.

5.6 Kämpfen

5.6.1 CombatMan

Der `CombatMan` ist zuständig für die Kampfhandlungen des Bots. Er ist als endlicher Zustandsautomat mit den Zuständen **Guard Base**, **Defend Base**, **Search Enemy Base**, **Rally before Attack**, **Attack Enemy** und **Retreat** aufgebaut. In jedem Frame wird der aktuelle Zustand anhand der Regeln, die in Abbildung 5.11 visualisiert sind, neu bestimmt. Danach werden die Aktionen für diesen Zustand ausgeführt.

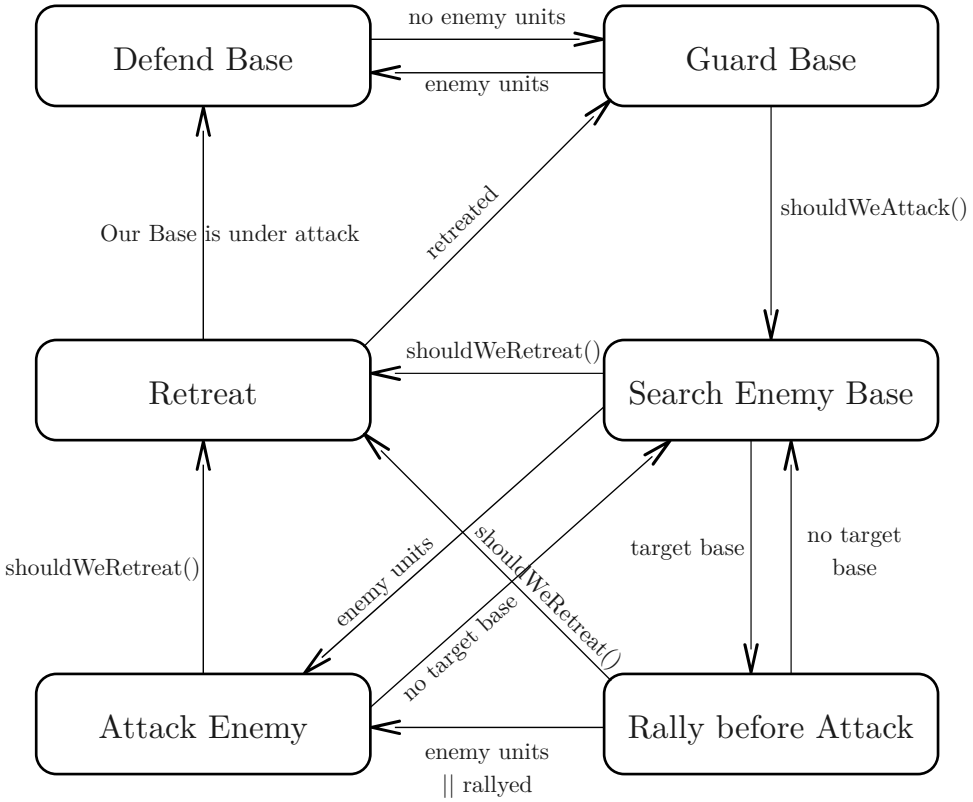


Abbildung 5.11: Die Zustände von `CombatMan` und deren Übergänge

Im Zustand **Guard Base** bewachen die Kampfeinheiten die eigenen Basen. Dazu wird jede Einheit, die sich nicht innerhalb ihrer Basis befindet oder die in illegalen Bereichen steht, an den Guard Point der Basis geschickt.

Im Modus **Defend Base** wird die eigene Basis verteidigt. Dazu werden alle gegnerischen Einheiten angegriffen. Hierfür wird der eingebaute AttackMove verwendet. Das heißt, allen Einheiten wird eine Position auf der Karten zugewiesen, die angegriffen werden soll. Die Einheiten bewegen sich dann auf diese Position zu und greifen alle gegnerischen Einheiten an, die sie sehen. Für Einheiten, die Lufteinheiten angreifen können, ist dies die Position einer gegnerischen Flugeinheit, ansonsten ist das die Position einer beliebigen gegnerischen Bodeneinheit. Die Methode `determineTargetUnits(set<Unit*> enemyUnits)` bestimmt diese Zieleinheiten. Sie wählt jeweils die erstbeste gegnerische Einheit aus der Menge aller gegnerischen Einheiten aus. Bei Flugeinheiten wird noch darauf geachtet, dass die Einheit nicht außerhalb der Waffenreichweite in vom Boden aus unerreichbaren Gebiet ist. Ansonsten kann es vorkommen, dass die Armee eine Einheit sieht und als Ziel auswählt, aber diese nicht erreichen kann und so komplett blockiert ist.

Im Zustand **Search Enemy Base** befindet man sich, wenn man den Gegner angreifen will, aber keine Zielobjekte bekannt sind. Dazu wird die Methode `CIA ::chooseTargetEnemyBase()` aufgerufen. Falls der CIA eine existierende gegnerische Basis

bekannt ist, wird diejenige Basis ausgewählt, die vermeintlich die schwächste Armee hat. Ist keine existierende gegnerische Basis bekannt, wird ein Scout losgeschickt, um nach gegnerischen Einheiten zu suchen.

Im Zustand **Rally before Attack** werden alle Kampfeinheiten gesammelt bevor angegriffen wird, damit möglichst viele gleichzeitig beim Gegner ankommen, sodass die Armee eine größere Schlagkraft entfalten kann. Dazu wird schrittweise ein Sammelpunkt berechnet und sobald sich 80 % der Einheiten bei diesem befinden oder als Fallback 1000 Frames vergangen sind, wird der nächste berechnet. Zur Bestimmung des ersten Sammelpunktes wird die eigene Basis bestimmt, die als nächstes zur gegnerischen Zielbasis liegt. Bei der Berechnung des nächsten Sammelpunktes wird auch wieder zwischen Regionen mit großer und normaler Fläche unterschieden. Liegt der bisherige Sammelpunkt in einer Region mit großer Fläche, wird zuerst deren Chokepoint mit geringstem Abstand zur gegnerischen Basis berechnet. Dieser wird dann um 500px Richtung Regionsmittelpunkt verschoben, was den nächsten Sammelpunkt ergibt. Große Regionen befinden sich oftmals im Mittelpunkt der Karte und an diese grenzen relativ kleine Regionen mit kleiner Basis. Würde man sich direkt im Chokepoint sammeln, würde die erste Einheit, die am Sammelpunkt angelangt ist, sofort die Basis sehen. Dadurch würden die Einheiten in den Angriffsmodus übergehen, obwohl sie sich noch nicht gesammelt haben. Befindet sich der bisherige Sammelpunkt in einer normal großen Region, ist der Mittelpunkt der Nachbarregionen, der am nächsten zur Zielbasis liegt, der neue Sammelpunkt.

Im Zustand **Attack Enemy** werden Angriffskampfhandlungen mit dem Gegner durchgeführt. Sind gegnerische Einheiten sichtbar, werden diese angegriffen. Wie auch im Zustand *Defend Base* wird dazu der eingebaute AttackMove verwendet. Die beiden Zieleinheiten werden ebenfalls über die Methode `determineTargetUnits(set<Unit*> enemyUnits)` bestimmt. Diese liefert, sofern vorhanden, eine gegnerische Flugeinheit, die von den Dragoons angegriffen werden kann, sowie eine Bodeneinheit, die alle angreifen können. Sind keine gegnerischen Einheiten in diesem Zustand sichtbar, werden die eigenen Kampfeinheiten zur gegnerischen Zielbasis geschickt.

Im Zustand **Retreat** werden alle Einheiten in die Basen zurückgeschickt, denen sie zugeordnet sind. Sind 75 % der Einheiten wieder dort angelangt, gilt der Rückzug als abgeschlossen.

5.6.2 Angriff auf die gegnerische Basis

Für die Entscheidung, ob angegriffen wird, ist die Funktion `shouldWeAttack()` zuständig. Grundlegende Voraussetzungen für einen Angriff sind, dass man mindestens vier Kampfeinheiten hat, und, falls schon mal ein Rückzug stattfand, die eigene Armee größer ist als sie zum Zeitpunkt des letzten Rückzugs war. Die allgemeine Angriffsregel lautet, dass angegriffen wird, wenn man mindestens einen Observer und zusammen mehr als 20 Dragoons und Zealots hat, sowie die Armeestärke um 25 % und um

20 Supply größer ist als die gegnerische Armeesträrke beim letzten Rückzug war. Immer angegriffen wird, wenn mehr als 180 Supply verbraucht sind. Dann hat man dann eine so große Armee, und kann auch nicht mehr viele Einheiten bauen, dass es keinen Sinn mehr hat, abzuwarten. Für die Anfangsphase eines Spieles gibt es noch zwei weitere Angriffsregeln. Die Anfangsphase wird hier als die ersten 10.000 Frames des Spiels definiert. Hier wird angegriffen, wenn zum einen die Kosten der eigenen Kampfeinheiten um 20 % über denen der gegnerischen Kampfeinheiten liegen. Zum anderen wird ein *„earlyGameAttackValue“* berechnet, der sich aus der Anzahl der Basen, der Technologieeinheiten, der Arbeiter und der Verteidigungsgebäude von einem selbst und dem Gegner berechnet. Übersteigt dieser einen gewissen Wert, wird angegriffen. Nur ein einziges Mal wird aufgrund einer dieser beiden Anfangsphase-Regeln angegriffen. Danach werden nur noch die normalen Regeln berücksichtigt. Das hat sich als zweckmäßig erwiesen, da die Regeln sonst viel zu oft einen Angriff auslösen würden. Das liegt daran, dass die Informationen über den Gegner zu selten aktualisiert werden und somit nur das Anwachsen der eigenen Armee beobachtet wird, aber keine Informationen über das Anwachsen der gegnerischen Armee vorliegen.

Wird der Entschluss zum Angriff gefasst, wird zuerst in den Zustand **Search Enemy Base** gewechselt. Ist eine Zielbasis bestimmt, wird die Armee mittels des Zustand **Rally before Attack** gesammelt. Sobald in einem der beiden Zustände gegnerische Einheiten gesichtet werden, wird in den **Attack Enemy**-Zustand gewechselt, um sie zu bekämpfen. Der Sammelzustand wird auch verlassen, wenn ein ungültiger Sammelpunkt berechnet wird oder ein neuer Sammelpunkt weiter von der Zielbasis entfernt ist als der bisherige Sammelpunkt. Es ist nötig, diese Fälle abzufangen, da ansonsten der Bot hin und wieder das Sammeln nie beendete und die Einheiten fortlaufend von einem Punkt zum nächsten schickte, ohne den Gegner anzugreifen.

Die Funktion `shouldWeRetreat()` entscheidet über den Rückzug, wenn man sich in einem der Angriffszustände **Search Enemy Base**, **Rally before Attack** oder **Attack Enemy** befindet. Diese gibt den Befehl zum Rückzug, wenn die eigene Armee weniger als 30 Supply und um 5 weniger Supply als die gegnerische Armee hat, oder wenn die Stärke der eigenen Armee weniger als 75 % der Stärke der gegnerischen Armee ist. Erstere Regel sorgt dafür, dass sich kleine Armeen (30 Supply entsprechen zusammen 15 Dragoons und Zealots) relativ schnell zurückziehen, während zweitere Regel dafür sorgt, dass größere Armeen möglichst länger kämpfen und so mehr Schaden beim Gegner anrichten. Bei Schlachten von größeren Armeen sind nämlich nicht alle Einheiten direkt an Kampfhandlungen beteiligt und ein Teil der Stärke der gegnerischen Armee kann von Verteidigungsgebäuden kommen, die zwar sichtbar sind, aber die eigene Armee außerhalb deren Reichweite ist. Deshalb macht es da auch Sinn, mit einer auf den ersten Blick unterlegenen Armee weiter zu kämpfen, weil die Stärke in direkten Kampfhandlungen ungefähr gleich ist und man so den Gegner schwächt.

Die Armeestärke ist die Summe des verbrauchten Supplys aller Einheiten. Für die Berechnung der Stärke der gegnerischen Armee werden auch noch Verteidigungs-

gebäude berücksichtigt. Hier zählt eine Photonenkanone 3,5, eine Tiefenkolonie 5,5 und ein Bunker 12 Supply.

5.7 Hilfsklassen

Es wurden verschiedene Hilfsklassen implementiert, um die Programmierung des Dreadbots zu erleichtern.

Mit Hilfe der Klasse `Singleton` lässt sich auf einfache Weise eine Klasse als Singleton definieren. Diese ist vom `Skynet`-Quellcode entnommen.

In der Klasse `Dreadbots_Little_Helper` sind Funktionen für häufig benötigte Operationen implementiert. Die Funktion `filterUnits()` liefert aus einer gegebenen Menge alle Einheiten mit bestimmten Eigenschaften zurück. Die spezifizierbaren Eigenschaften sind der `UnitType`, die Region und ob die Einheit fertiggestellt ist. Für verschiedene Kombinationen dieser Eigenschaften existieren eigene überlagerte Funktionen, um diese einfacher aufrufen zu können. Es stehen weiterhin Funktion zur Verfügung, die zurückliefern, ob eine `UnitType` ein Verteidigungsgebäude oder ein Technologiegebäude ist, da diese Information über BWAPI selbst nicht auslesbar ist. Die Berechnung der konvexen Hülle eines Polygons für die illegalen Bereiche bei der Gebäudeplatzierung wird auch über diese Klasse durchgeführt, die dann die einzelnen Funktionen der Klasse `GrahamScan` in der richtigen Reihenfolge aufruft. Des Weiteren gibt es Funktionen, um die vier Ecken einer Einheit in Pixel- oder in TilePosition-Auflösung zu bestimmen. Zudem gibt es Funktionen, um aus einer Menge von Einheiten oder Positionen die nächste zu einer Position oder anderen Einheit zu bestimmen. Zusätzlich gibt es noch Funktionen, um die nächstgelegende `BaseLocation` zu einer Position zu bestimmen. Hierbei können auch `BaseLocations` angegebenen werden, die ignoriert werden sollen und ob diese auf dem Bodenweg erreichbar sein müssen. Eine Funktion zur Berechnung des Rechtecks, welches ein Polygon einschließt ist ebenso implementiert wie eine Funktion zur Verschiebung eines Punktes in eine Richtung um eine bestimmte Strecke.

Die Klasse `Every_breath_you_take` enthält Debugging-Funktionen. Zum einen legt sie eine Logdatei an und hat Methoden, um in diese zu schreiben sowie diese zu flushen. Im Normalfall nutzt man die `print()`-Methode dieser Klasse, um in die Logdatei zu schreiben. Hierbei wird jeder Nachricht, die man schreibt, die Frame-Nummer vorangestellt, um diese zeitlich zuordnen zu können. Weiterhin sind in dieser Klasse verschiedene `toString()`-Methoden definiert, um die elementaren Datentypen sowie verschiedene Datentypen von BWAPI einfach in einen String zur Ausgabe in die Logdatei umwandeln zu können. Zudem sind diverse Zeichenfunktionen implementiert, die Debugging-Informationen in StarCraft auf den Bildschirm darstellen. Außerdem finden sich dort noch Funktionen zur Zeitmessung, um die Ausführungszeit in größtmöglicher Genauigkeit messen zu können. Es existieren zwei `typedefs`, `De-`

`bug` und `Dbg`, für diese Klasse, um mittels kürzerer Statements auf diese zugreifen zu können.

Die Datei `helpers.h` dient dazu, die anderen Hilfsklassen, die aus der Standardbibliothek verwendeten Datenstrukturen wie z.B. Vector, Map und Deque sowie verschiedene Kürzel, die via `typedef` oder `#define` erzeugt wurden, komfortabel in andere Dateien einbinden zu können. Jede Klasse bindet diese Datei ein.

Kapitel 6

Fazit & Ausblick

Zum Abschluss soll noch das Ergebnis der Arbeit präsentiert werden sowie ein Ausblick auf zukünftige Forschungsperspektiven gegeben werden.

6.1 Ergebnis der Arbeit

Zuerst wurde in Kapitel 2 das Genre der Echtzeitstrategiespiele vorgestellt und dargelegt, was StarCraft zu einem besonderer Vertreter dessen macht. Danach wurde in Kapitel 3 ein Überblick über den aktuellen Forschungsstand anhand ausgewählter Konzepte, Bots und Turniere gegeben. Anschließend wurde die verwendeten Programmierschnittstellen BWAPI und BWTA in Kapitel 4 beschrieben.

In Kapitel 5 wurde das Hauptziel der Arbeit, die Implementierung eines Bots für StarCraft unter Benutzung der Programmierschnittstelle BWAPI zu erstellen, vorgestellt. Hierbei wurden verschiedene KI-Techniken wie Einflussfelder (5.3.1.4) und Zustandsautomaten (5.6.1) eingesetzt.

Die gewählten Bauziele und die Baureihenfolge erwies sich als praktikabel, um eine schlagkräftige Armee aufzustellen (5.2.3). Die Techniken, die zur Wahl des Bauplatzes eines Gebäudes eingesetzt werden, konnten sich in der Praxis sehr gut bewähren. Die illegalen Bereiche, die zwischen den Rohstoffen und den Hauptgebäuden sowie an den Ausgängen der Basen definiert wurden, stellten zuverlässig sicher, dass die Arbeiter beim Sammeln der Rohstoffe nicht behindert wurden sowie dass die Ausgänge nicht zugebaut wurden (5.3.1.3). Der Einsatz eines Einflussfeldes erwies sich als sehr erfolgreich bei der Verhinderung des Einbauens von Einheiten und ist zudem hinreichend schnell (5.3.1.4). Das Ausführen von Bauvorhaben wurde zuverlässig sichergestellt (5.3.2). Es wurde dafür gesorgt, dass jede Basis genügend Arbeiter zu Verfügung hat und diese ihren Aufgaben nachgehen (5.4.1). Die Fallunterscheidung für Basen mit großer Fläche erwies sich als sehr zweckmäßig und konnte einige Probleme beheben (5.4.3). Das Scouten liefert schon sehr viele Information über den Gegner, diese müssten noch besser genutzt werden. Es ist zudem sichergestellt geworden, dass alle gegnerischen Gebäude immer gefunden werden (5.5). Die Verwendung eines Zustands-

automaten für die Angriffsphasen ermöglicht ein grundlegendes Kampfverhalten mit Verteidigung der Basis, Sammeln vor dem Angriff, dem eigentlichen Angriff sowie Rückzug bei übermächtigem Gegner (5.6.1).

6.2 Evaluation des Dreadbots

Unter Verwendung dieser Techniken gewinnt Dreadbot gegen die eingebaute KI 95 % der Spiele und erzielt auch achtbare Resultate gegen andere Bots. Hin und wieder ist es sogar möglich, Spitzenbots wie Skynet oder AIUR zu schlagen. Zu einer genaueren Evaluation der Leistungsfähigkeit des Bots war geplant, im Januar 2012 an einem Turnier der BWAPI-Autoren teilzunehmen. Dies fand jedoch mangels Teilnehmern nicht statt. Dafür wurde die Botladder eingerichtet, bei der die Januar-Version des Dreadbot über 6800 Spiele absolvierte und 26,95 % Siege erreichte. Die Endfassung wurde mehrfach eingereicht, doch jeweils nach wenigen Spielen deaktiviert, weil sie laut dem Administrator auf dessen System in eine Endlosschleife läuft. Dieser Fehler konnte lokal jedoch nicht reproduziert werden und trotz intensiver Kommunikationsbemühungen nicht behoben werden. Dadurch hat die Endversion nur 200 Spiele auf der Ladder absolviert, wovon sie 30 % gewann.

Die Ladderversion war auch weit nicht so ausgereift wie es die Endversion ist. Es passiert ihr immer mal wieder, dass Einheiten eingebaut werden, da dort noch nicht das Einflussfeld zur Verhinderung dessen eingebaut war (s. 5.3.1.4). Zudem kam es zu Deadlocks, wo der Bot trotz hinreichender Ressourcen nichts mehr baute und seine Arbeit einstellte. Es wurde noch auf Dark Templar und immer auf Carrier gesetzt. Auch der Angriff war noch mangelhaft. Es gab nur eine Angriffsbedingung, dass sich die eigenen Einheiten wieder zurückziehen war noch gar nicht eingebaut. Zudem versuchten Bodeneinheiten Lufteinheiten anzugreifen, was die ganze Armee blockieren konnte. Scouting war zum damaligen Zeitpunkt noch nicht vorhanden, erst wenn der Gegner angegriffen wurde, wurde die gegnerische Basis gesucht. Dies jedoch auch nicht mit einem Arbeit oder Observer, sondern die ganze Armee suchte die Basis. Mit diesen Unzulänglichkeiten konnte natürlich keine so guten Ergebnisse wie mit der Endversion erzielt werden.

Es wurden lokale Testreihen der Ladder- sowie der Endversion von Dreadbot gegen ausgewählte Bot sowie die eingebaute StarCraft-KI durchgeführt. Dabei wurde auf jeder der drei verwendeten Karten (sie Anhang B), die auch bei der Ladder im Einsatz sind, zehnmal gegen jeden Gegner gespielt. Die gegnerischen Bots waren die Teilnehmer des Turnier bei der AIIDE 2011, die sich einfach per Modul laden ließen. Bots, die als Client implementiert sind, wurden aufgrund der Komplexität derer Einrichtung weggelassen. Zudem funktionierte der Bot SPAR lokal nicht. Als Resultat der Spiele gegen diese Bots zeigte sich die Endversion gegenüber der Ladderversion teilweise deutlich verbessert. Gegen schwächere Bots konnte die Anzahl der Siege deutlich gesteigert werden. Gegen die Top-3-Bots des Turniers, Skynet, UAlbertaBot und AIUR,

jedoch zeigten die seitdem eingebauten Verbesserungen wenig Wirkung. Hier konnte nur minimal bessere Ergebnisse erzielt werden. Diese Bots spielten aber auch bei dem Turnier schon in einer anderen Liga und erreichten alle mehr als 70 % Siege, während die anderen Testgegner jeweils nicht mal die Hälfte ihrer Spiele beim Turnier gewannen. Die Ladderversion kam auf 32,0 % und die Endversion auf 48,3 % Siege gegen die Turnierbots. Zudem wurde auch gegen die eingebaute StarCraft-KI drei Testreihen durchgeführt, eine für jede Rasse. Dabei gewann die Ladderversion 66,7 % und die Endversion 95,0 % ihrer Spiele. Insgesamt konnten somit die Laddversion 40,7 % und die Endversion 60 % der Testspiele gewinnen.

Un die Resultate aus den Testspielen besser einordnen zu können, wurde ein Ranking für ein „hypothetisches Turnier“ aufgestellt. Dabei wurde für die Spiele der anderen Bots untereinander die Ergebnisse der AIIDE 2011 dieser Paarungen genommen. Dabei landet der Dreadbot mit 48,3 % gewonnen Spielen auf Platz 5 von 10. Er liegt damit knapp hinter Nova (52,2 %) und noch deutlich vor den von Doktoranden erstellten BroodwarBotQ (37,2 %, Platz 7) und BTHAI (34,4 %, Platz 8). Die genauen Ergebnis finden sich im Anhang C.2. Besonders positiv ist noch hervorzuheben, dass die Endversion keinerlei Abstürze produzierte, während diese bei der Ladderversion regelmäßig auftraten.

6.3 Verbesserungsvorschläge

Da Dreadbot nur eine grundlegende Arbeit war, auf deren Basis weitere Arbeiten vorgenommen werden sollen, gibt es natürlich vielfältige Verbesserungsmöglichkeiten. Bei der Gebäudeplatzierung kann noch für eine kompaktere Basis gesorgt werden. Zudem kann man berücksichtigen, dass Basen mit nur einem Ausgang nicht direkt die Bodenverteidigung brauchen, sondern diese in der Basis davor platziert werden können, um direkt die erste Expansion mitabzusichern. Weiterhin können die Anzahl der Arbeiter besser beobachtet werden und überzählige Arbeiter getötet werden, wenn sie zum Ende des Spieles hin Versorgung blockieren, die für Kampfeinheiten benötigt würde. Die Baureihenfolge und Angriffsbedingungen sollten auch nicht so starr festgelegt sein wie bisher, sondern hierfür sollte ein dynamischer Planer entwickelt werden, der die übergeordnete Strategie auswählt und daraus die Baureihenfolge und Angriffsbedingungen ableitet. Dazu sollten bekannten Strategien eingebaut werden sowie neue entwickelt werden.

Größtes Potential hat der Dreadbot noch beim Micromanagement, hier ist durch wenig Arbeit eine erhebliche Verbesserung leicht erreichbar. Es empfiehlt sich, die Potentialfelder zu implementieren sowie geeigenete Feldstärken über evolutionäre Algorithmen zu entwickeln. Dann sollten hier auch taktische Stellungen unter Berücksichtigung des Geländes und der Geländehöhen eingebaut werden, um das Maximum aus den Kampfeinheiten herauszuholen. Evolutionäre Algorithmen können nicht nur bei den Potentialfeldern eingesetzt werden, sondern in den verschiedensten Bereichen des Bots

zur Optimierung verwendet werden. Weiterhin sollten die im Internet zahlreich vorhanden Replays, besonders der koreanischen Profiligen, mittels Data Mining genutzt werden, um Expertenwissen in den Bot einfließen zu lassen. Neuerdings war es bei den Turnieren in 2012 möglich, Informationen über die Gegner zwischen den Spielen zu speichern, daraus Erkenntnisse zu gewinnen und diese im weiteren Turnierverlauf zu nutzen. Von dieser Möglichkeit muss Dreadbot auch Gebrauch machen, um auf die vorderen Plätze bei Turnieren vorstoßen zu können.

Ein großes Ziel der Forschung auf dem Gebiet der Echtzeitstrategiespiele ist natürlich das Besiegen menschlicher Profispieler. Hierzu müssen natürlich alle bekannten Techniken kombiniert werden. Erfolgversprechend könnte sein, auf Strategien und Taktiken zu setzen, die in der Theorie sehr gut sind, sich bisher aber praktisch für einen Menschen nicht umsetzen ließen, jedoch für einen Bot realisierbar sind. Man könnte sich auch konkret darauf verlegen, den Menschen an zu vielen Fronten zu beschäftigen, die er nicht mehr meistern kann, für eine KI jedoch zu beherrschen sind. Angesichts der Tatsache, dass Topspieler über 300 APM erreichen können, ist auch dieser Ansatz sehr schwer. Es steht also noch einiges an Arbeit an, bis dieses Ziel erreicht. Doch durch die von BWAPI geschaffene sehr attraktive Forschungsumgebung StarCraft: Brood War ist das Interesse an diesem Gebiet sicherlich weiterhin hoch und wir können uns noch auf spektakuläre Ergebnisse freuen.

Anhang A

Klassendiagramme

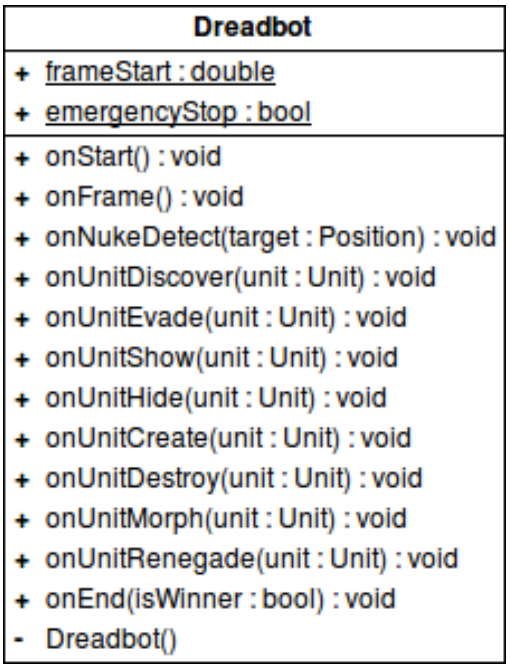


Abbildung A.1: Klassendiagramm `Dreadbot`

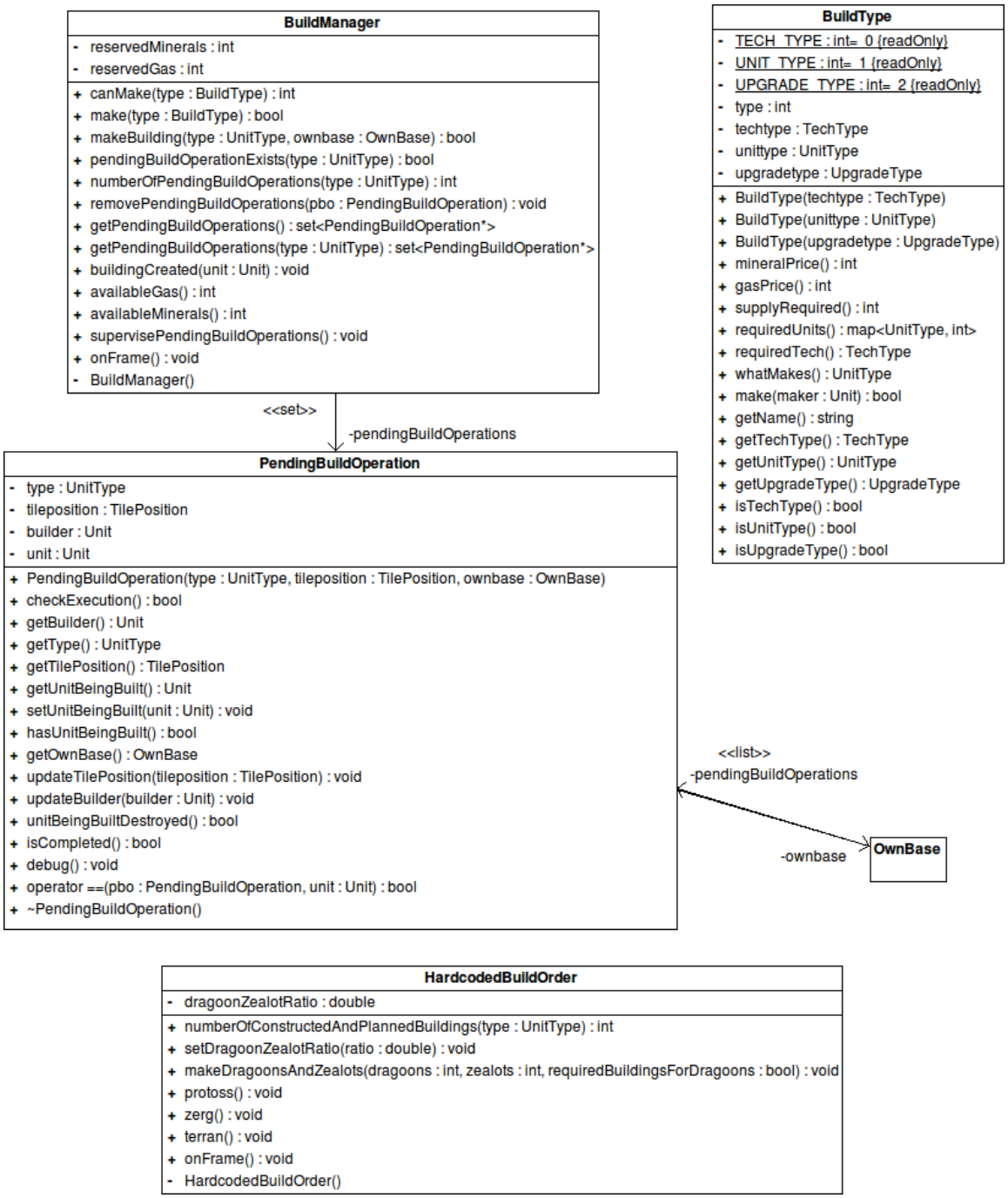


Abbildung A.2: Klassendiagramm der für das Bauen von Einheiten und Gebäuden zuständigen Klassen `BuildManager`, `BuildType`, `PendingBuildOperation` und `HardcodedBuildOrder`.
Die Klasse `OwnBase` ist in Abbildung A.4 ausführlich dargestellt.

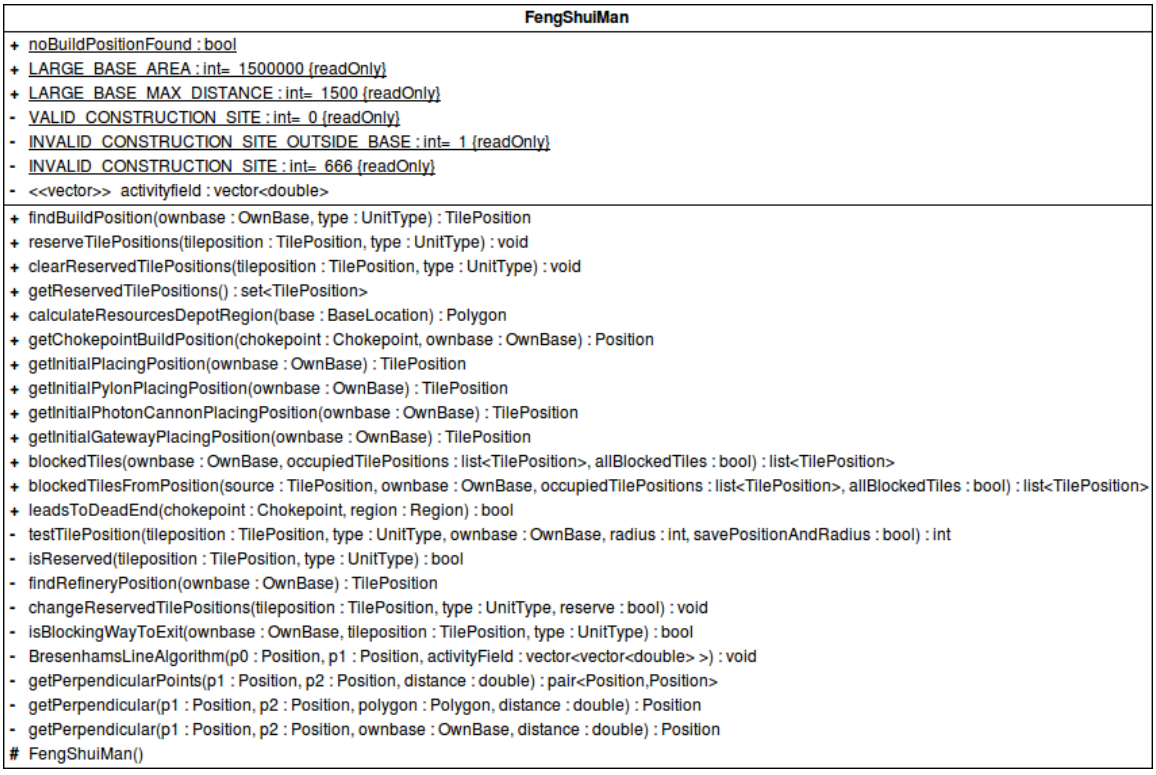


Abbildung A.3: Klassendiagramm `FengShuiMan`

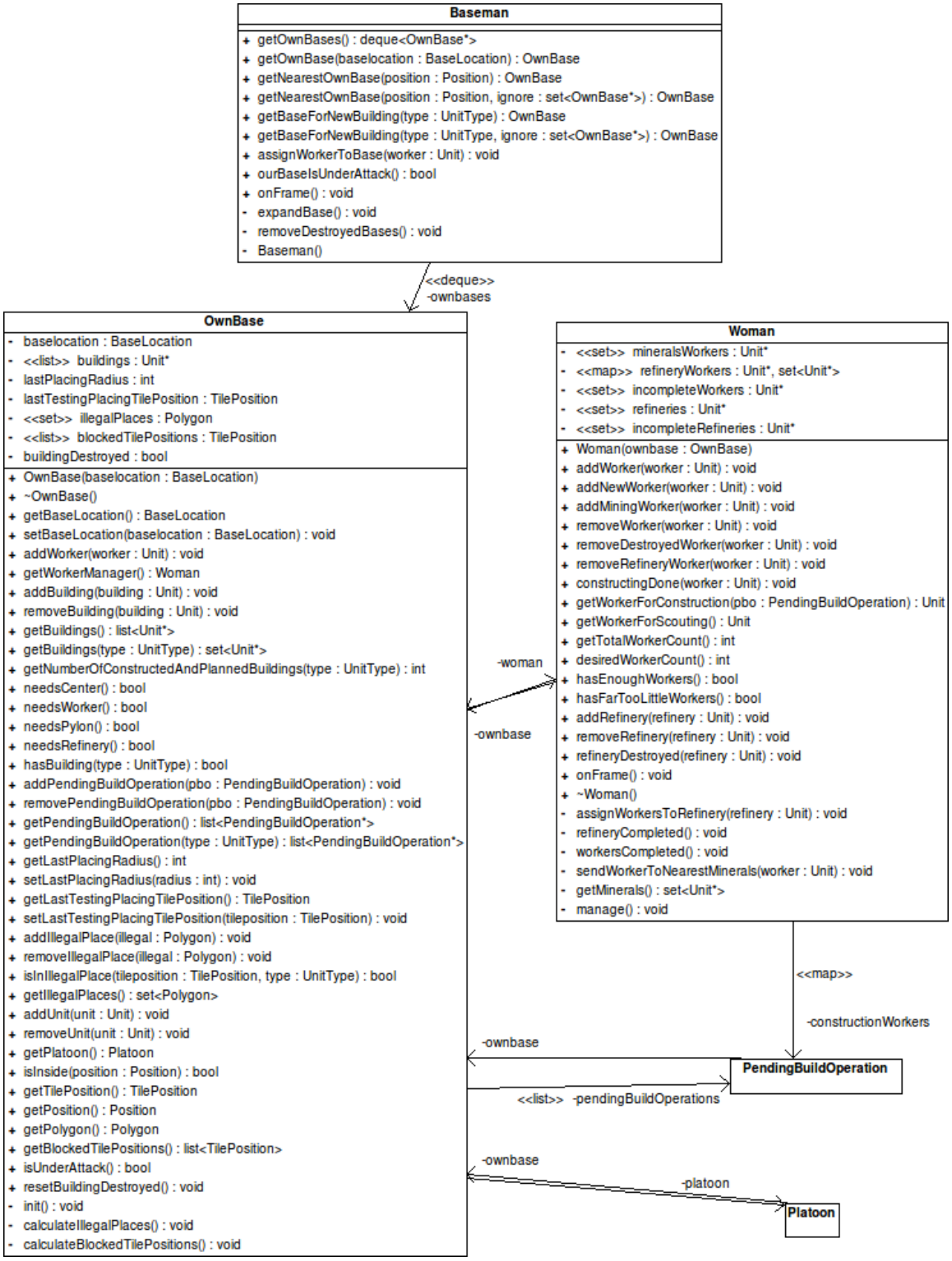


Abbildung A.4: Klassendiagramm der für die Basisverwaltung zuständigen Klassen `Baseman`, `OwnBase` und `Woman`.
Die Klasse `PendingBuildOperation` ist in Abbildung A.2 und die Klasse `Platoon` ist in Abbildung A.6 ausführlich dargestellt.

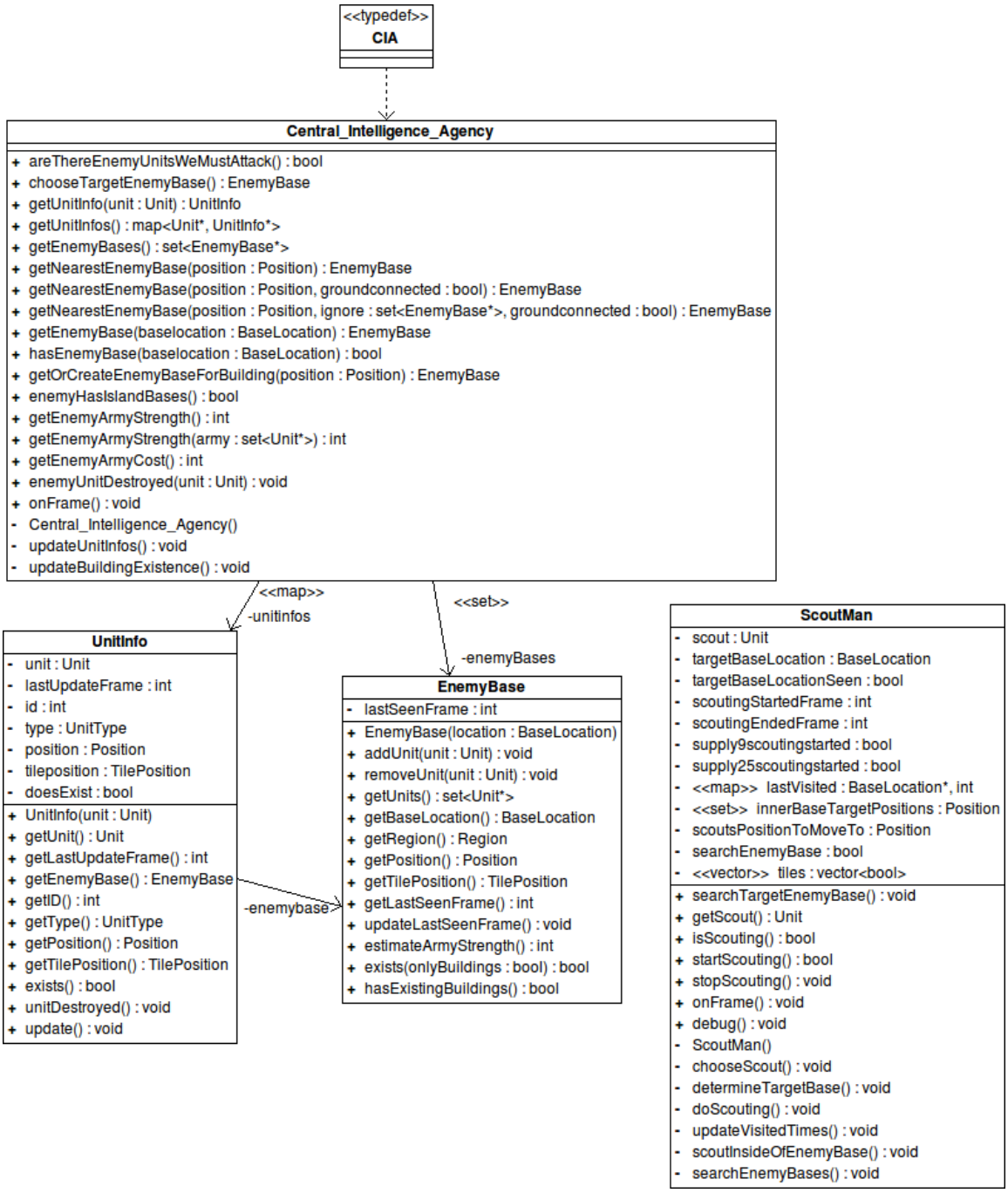


Abbildung A.5: Klassendiagramm der fürs Scouten zuständigen Klassen `Central_Intelligence_Agency`, `UnitInfo`, `EnemyBase` und `ScoutMan`

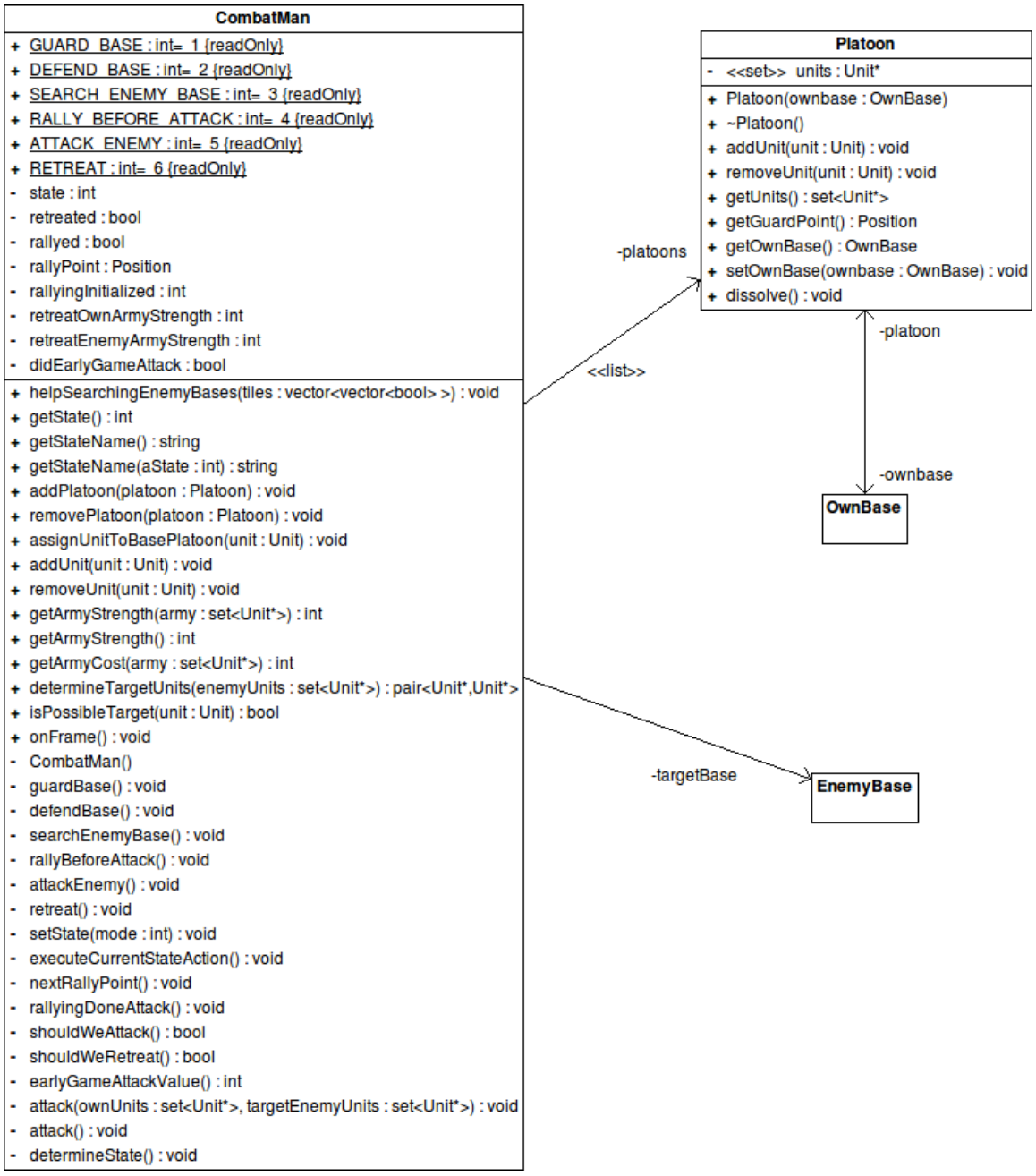


Abbildung A.6: Klassendiagramm der für Kampfhandlungen zuständigen Klassen `CombatMan` und `Platoon`.
Die Klasse `OwnBase` ist in Abbildung A.4 und die Klasse `EnemyBase` in Abbildung A.5 erläutert.

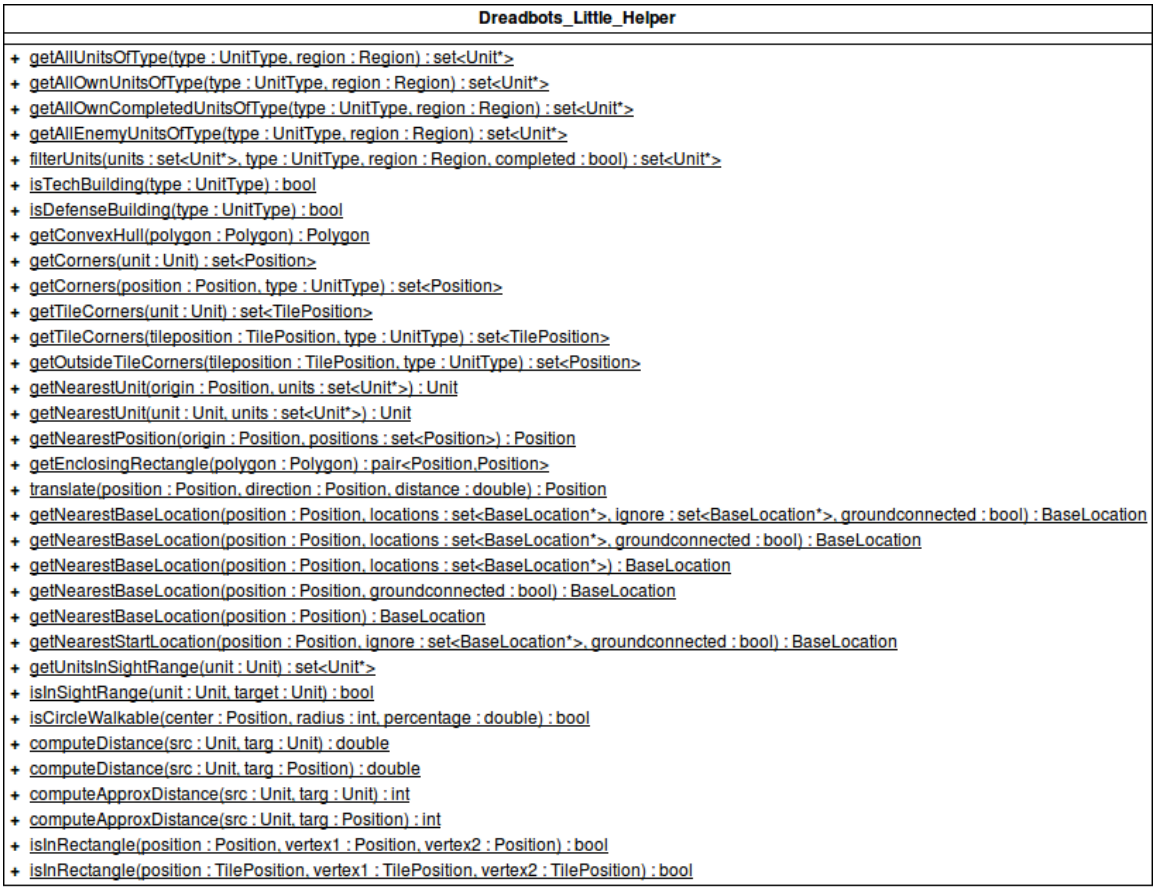


Abbildung A.7: Klassendiagramm der Hilfsklassen `Dreadbots_Little_Helper`, `GrahamScan` und `Singleton`

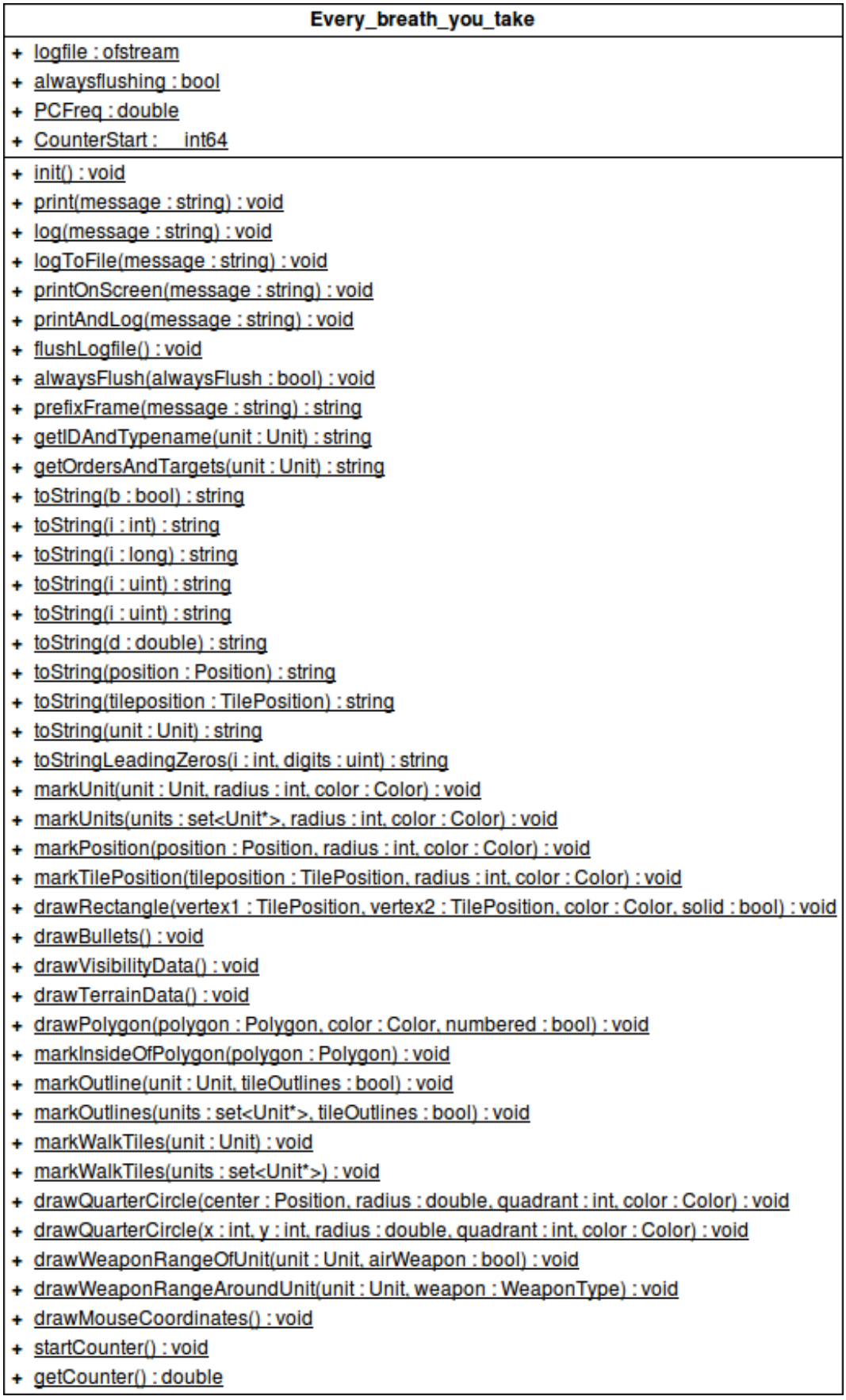


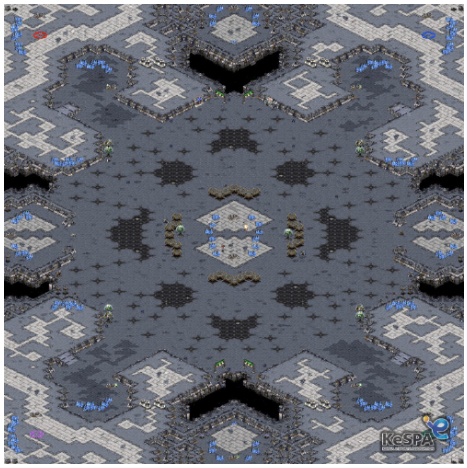
Abbildung A.8: Klassendiagramm `Every_Breath_You_Take`

Anhang B

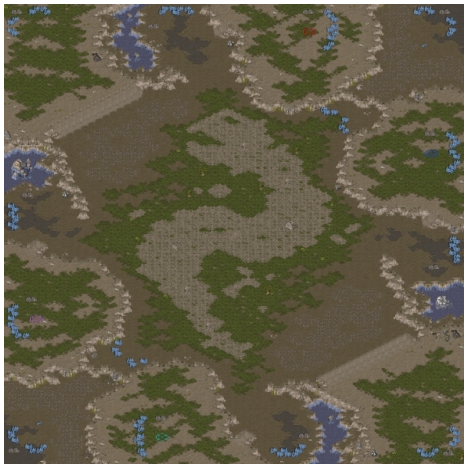
Karten

Die in der Bot-Ladder und bei den lokalen Tests verwendeten Karten:

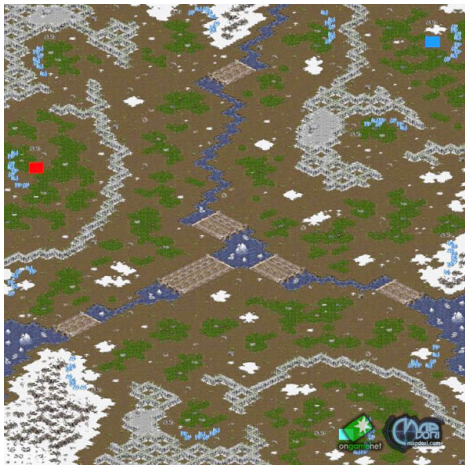
Name:	Andromeda
Größe:	128x128
Max. Spieler:	4
Startpositionen:	2, 4, 8, 10
Grafiksatz:	Weltraum-Plattform
Beschreibung:	Karte mit Fokus auf Makromanagement mit einer natürlichen Mineralienexpansion in der Hauptbasis sowie einer leicht zu verteidigenden Gas-Expansion.



Name:	Python
Größe:	128x128
Max. Spieler:	4
Startpositionen:	1, 2, 7, 8
Grafiksatz:	Dschungelwelt
Beschreibung:	Standardkarte mit leicht zu verteidigenden Expansionen. Enthält auch zwei Inselexpansionen.



Name:	Tau Cross
Größe:	128x128
Max. Spieler:	3
Startpositionen:	2, 5, 10
Grafiksatz:	Eis
Beschreibung:	Drei-Spieler-Karte mit einem schmalen Eingang, der die Hauptbasis beschützt. Es gibt keine Rampen auf dieser Karte.



Anhang C

Abschneiden von Dreadbot

C.1 Testspiele

			Andromeda					Python					Tau Cross					
Gegner	R	DB	S	N	U	C	G	S	N	U	C	G	S	N	U	C	G	Siege
StarCraft-KI	P	L	4	4	-	2	-	8	-	-	2	-	7	3	-	-	-	63,3 %
		E	10	0	-	-	-	10	0	-	-	-	10	0	-	-	-	100,0 %
StarCraft-KI	T	L	7	3	-	-	-	7	2	-	1	-	6	2	-	2	-	66,7 %
		E	10	0	-	-	-	10	0	-	-	-	9	1	-	-	-	96,7 %
StarCraft-KI	Z	L	8	2	-	-	-	9	0	-	1	-	4	5	-	1	-	70,0 %
		E	9	1	-	-	-	9	0	1	-	-	8	2	-	-	-	88,3 %
AIUR	P	L	0	9	1	-	-	0	10	-	-	-	1	7	-	1	1	8,3 %
		E	1	9	-	-	-	1	9	-	-	-	0	10	-	-	-	6,7 %
bigbrother	Z	L	1	2	-	-	7	1	0	-	-	9	6	0	-	-	4	93,3 %
		E	7	1	-	-	2	7	0	-	-	3	7	0	-	-	3	96,7 %
BroodwarBotQ	P	L	2	4	-	4	-	1	1	-	8	-	4	5	1	-	-	25,0 %
		E	5	5	-	-	-	8	2	-	-	-	4	5	1	-	-	58,3 %
BTHAI	Z	L	5	4	1	-	-	1	3	-	5	1	1	5	-	2	2	35,0 %
		E	9	0	-	-	1	9	1	-	-	-	8	2	-	-	-	90,0 %
Cromulent	T	L	8	2	-	-	-	2	7	-	1	-	4	3	-	3	-	46,7 %
		E	6	4	-	-	-	5	5	-	-	-	8	2	-	-	-	63,3 %
Nova	T	L	3	6	-	1	-	1	2	-	7	-	6	2	-	2	-	33,3 %
		E	5	5	-	-	-	4	6	-	-	-	5	5	-	-	-	46,7 %
Quorum	T	L	0	2	-	2	6	1	1	-	8	-	4	2	2	1	1	43,3 %
		E	7	2	-	-	1	5	5	-	-	-	5	5	-	-	-	60,0 %
Skynet	P	L	0	10	-	-	-	0	8	-	2	-	0	8	-	1	1	3,3 %
		E	1	9	-	-	-	1	9	-	-	-	1	9	-	-	-	10,0 %
UAlbertaBot	P	L	0	10	-	-	-	0	9	-	1	-	0	9	-	1	-	0,0 %
		E	1	9	-	-	-	0	10	-	-	-	0	10	-	-	-	3,3 %
Gesamt		L	38	58	2	9	13	31	43	-	36	10	43	51	3	14	9	40,7 %
		E	71	45	-	-	4	69	47	1	-	3	65	51	1	-	3	60,0 %

Legende R: Rasse, P: Protoss, T: Terran, Z: Zerg,
DB: Dreadbot-Version, L: Ladderversion, E: Endversion,
S: Sieg, N: Niederlage, U: Unentschieden, C: Crash von Dreadbot, G: Crash des Gegners

C.2 Hypothetisches Turnier

		Sk	UA	Ai	No	Dr	Cr	Br	BT	bi	Qu	Siege	%	'11%
1.	Skynet	–	26	11	29	27	30	28	30	30	27	238	88,1 %	88,9 %
2.	UAlbertaBot	4	–	29	17	29	26	29	30	30	30	224	83,0 %	79,4 %
3.	AIUR	19	1	–	30	28	28	18	22	29	29	204	75,6 %	70,3 %
4.	Nova	1	13	0	–	16	28	18	21	14	30	141	52,2 %	47,5 %
5.	Dreadbot	3	1	2	14	–	19	17,5	27	29	18	130,5	48,3 %	–
6.	Cromulent	0	4	2	2	11	–	19	10	30	29	107	39,6 %	30,0 %
7.	BroodwarBotQ	2	1	12	12	12,5	11	–	14	14	22	100,5	37,2 %	32,8 %
8.	BTHAI	0	0	8	9	3	20	16	–	11	26	93	34,4 %	31,9 %
9.	bigbrother	0	0	1	16	1	0	16	19	–	16	69	25,6 %	27,8 %
10.	Quorum	3	0	1	0	12	1	8	4	14	–	43	15,9 %	9,4 %

Legende '11%: Prozentsatz der gewonnen Spiele beim Turnier der AIIDE 2011

Anhang D

Inhalt der CD

Ordner	Inhalt
`\Ausarbeitung`	diese Arbeit als PDF
`\Ausarbeitung\Quellcode`	der Quellcode des Ausarbeitung-PDFs
`\Ausarbeitung\Quellen`	alle in der Ausarbeitung zitierten Quellen, sofern sie digital verfügbar sind
`\Dreadbot`	der in dieser Arbeit implementierte Bot
`\Dreadbot\Quellcode`	der Quellcode des Bots
`\Dreadbot\Dokumentation`	die Dokumentation des Quellcodes
`\Dreadbot\Ladderversion`	der unkommentierte Quellcode der Laddversion und die zugehörige DLL
`\logs`	Logfiles und Replays der Testmatches
`\Resolution Expander`	der Resolution Expander
`\sonstiges`	enthält die Downloads und `bwapi.ini`

Anhang E

Installationsanleitung

E.1 Standardinstallation

1. Visual Studio installieren

   Sprachtools → Visual C++ ist ausreichend, Rest kann abgewählt werden

2. StarCraft: Brood War installieren

   Das gewählte Installationsvereichnis, normalerweise `C:\Program Files\StarCraft`, wird im nachfolgenden mit `$StarCraft` bezeichnet

3. StarCraft auf Version 1.16.1 updaten

   `http://us.battle.net/support/en/article/starcraft-patch-information`
   oder als Direktdownload
   `http://ftp.blizzard.com/pub/broodwar/patches/PC/BW-1161.exe`

4. ChaosLauncher installieren

   a) Download: `http://winner.cspsx.de/Starcraft/`

   b) Entpacken in ein beliebiges Verzeichnis, ich empfehle `$StarCraft\ChaosLauncher`

   c) Empfehlenswert ist es eine Verknüpfung mit ChaosLauncher.exe auf dem Desktop oder in der Taskleiste zu erstellen, da der Chaoslauncher StarCraft mit dem Bot startet und daher dementsprechend häufig benötigt wird

5. BWAPI installieren

   a) Download: `http://code.google.com/p/bwapi/downloads/list`

   b) In einen beliebigen Ordner entpacken. Dieser wird nachfolgend mit `$BWAPI` bezeichnet.

c) Den Inhalt von `$BWAPI\Chaoslauncher` in das Chaoslauncher-Verzeichnis verschieben, normalerweise `$StarCraft\Chaoslauncher`

d) Den Inhalt von `$BWAPI\StarCraft` nach `$StarCraft` verschieben

e) Den Inhalt von `$BWAPI\WINDOWS` nach `C:\Windows` verschieben

6. Projekt einrichten und Bot kompilieren

a) `$BWAPI\ExampleProjects.sln` mit Visual Studio öffnen

b) Aus der Projektmappe kann AIModuleLoader, ExampleProjects, ExampleAIClient und ExampleTournamentModule gelöscht werden

c) Die schon in ExampleAIModule vorhandenen Dateien löschen

d) Projekt ExampleAIModule in Dreadbot umbennen (Wichtig, da das den Namen der generierten DLL bestimmt)

e) Sämtliche Dateien aus dem Ordner `\Dreadbot\Quellcode` auf der DVD dem Projekt Dreadbot hinzufügen (Drag&Drop des Dreadbot-Ordners auf Dreadbot im Projektmappen-Explorer)

f) Unter Dreadbot → Eigenschaften → Konfigurationseigenschaften → Allgemein das Ausgabeverzeichnis auf `$StarCraft\bwapi-data\AI` setzen

g) In `Every_breath_you_take.cpp` in der `init()`-Methode den Pfad der Log-Dateien anpassen, ansonsten werden diese einfach im StarCraft-Verzeichnis `$StarCraft` angelegt

h) Konfiguration von Debug auf Release setzen (Symbolleiste, Feld links vom Win32-Feld)

i) Kompilieren: F7 oder Menü Erstellen → Projektmappe erstellen

7. BWAPI und StarCraft einrichten

a) `\sonstiges\bwapi.ini` von der DVD nach `$StarCraft\bwapi-data` kopieren, vorhandene Datei ersetzen

b) StarCraft normal starten und im SinglePlayer einen Charakter anlegen (einfach irgendeinen beliebigen Namen eingeben). Das ist wichtig, sonst stürzt StarCraft mit dem Bot immer ab!

8. StarCraft mit Bot starten

a) ChaosLauncher starten

b) BWAPI Injector (1.16.1) RELEASE aktivieren

c) auf Start klicken

d) Es wird automatisch ein Spiel gestartet. Jedoch steht am Anfang alles und StarCraft reagiert auch nicht, da die Karte analysiert wird vom BWTA (Brood War Terrain Analyzer) und das eine ganze Weile dauert. Wenn dies fertig ist, fängt der Bot zu spielen an.

E.2 Verwendung von Resolution Expander

1. `\Resolution Expander` von der DVD an die gewünschte Stelle kopieren
2. `BWAPI.dll` von `StarCraft\Chaoslauncher` in den Resolution-Expander-Ordner ***kopieren***
3. In der ResSettings.ini kann die gewünschte Auflösung eingestellt werden. Nur ganz spezielle Auflösung funktionieren, z.B. 1024x768 und 1600x1200.
4. `InsectLoader.exe` ausführen
5. Hinweis: wenn man zum Debuggen Linien von Broodwar zeichnen lässt, kann es hier zu Darstellungsfehlern kommen.

E.3 Hinweis zum Projektnamen

Wird das Projekt Dreadbot umbenannt, erhält auch die generierte DLL den neuen Namen. Dieser muss dann auch in der bwapi.ini geändert werden: `ai = bwapi-data\AI\Dreadbot.dll` zu `ai = bwapi-data\AI\<neuer Projektname>.dll` Der Name der Projektmappe ist egal.

Literaturverzeichnis

[AirUniversity 1999] : *Air University Catalog Academic Year 1999–2000*. Vorlesungsverzeichnis. 1999. – http://www.au.af.mil/au/cf/au_catalog_1999_2000/catalog.pdf. – Zugriffsdatum: 13.07.2012, 03:31

[Liquipedia 2009a] Wiki, Liquipedia S. (Hrsg.): *Actions per Minute*. 2009. – http://wiki.teamliquid.net/starcraft/Actions_per_Minute. – Zugriffsdatum: 08.10.2012, 20:04

[BWTA 2011] : *bwta – A terrain analyzer for BWAPI*. 2011. – http://code.google.com/p/bwta/. – Zugriffsdatum: 08.18.2012. 16:16

[BWAPI 2012] : *bwapi – An API for interacting with Starcraft: Broodwar (1.16.1)*. 2012. – http://code.google.com/p/bwapi/. – Zugriffsdatum: 08.18.2012. 16:14

[ESA 2012] : *Sales & Genre Data*. 2012. – http://www.theesa.com/facts/salesandgenre.asp. – Zugriffsdatum: 08.10.2012, 22:11

[Blizzard Entertainment 2006] Blizzard Entertainment: *Blizzard Entertainment – Awards*. 2006. – http://web.archive.org/web/20060814090829/http://www.blizzard.com/inblizz/awards.shtml

[Buro 2011] Buro, Michael: *2011 AIIDE StarCraft Competition - Cumulative Results*. Tabelle. 2011. – https://docs.google.com/spreadsheet/ccc?key=0An3xUNEy2rixdHUza1BzZE50cGdUdHEwbGlhSU5rckE#gid=0. – Zugriffsdatum: 06.08.2012, 18:48

[Buro und Furtak 2003] Buro, Michael ; Furtak, Timothy: RTS Games as Test–Bed for Real–Time AI Research. In: *Workshop on Game AI, JCIS*, https://www.skatgame.net/mburo/ps/ORTS-JCIS03.pdf. – Zugriffsdatum: 29.09.2012, 01:36, 2003, S. 481–484

[Campbell u. a. 2002] Campbell, Murray ; Jr., A. Joseph H. ; Hsu, Feng hsiung: Deep Blue. In: *ARTIFICIAL INTELLIGENCE* 134 (2002), S. 57–83

[Cavalli 2009] Cavalli, Earnest ; Wired.com (Hrsg.): *U.C. Berkeley Now Offers StarCraft Class*. 2009. – http://www.wired.com/gamelife/2009/01/uc-berkeley-int/. – Zugriffsdatum: 01.10.2012, 21:18

[Chan u. a. 2007a] Chan, Hei ; Fern, Alan ; Ray, Soumya ; Wilson, Nick ; Ventura, Chris: *Extending online planning for resource production in real-time*

strategy games with search. 2007. – http://engr.case.edu/ray_soumya/papers/meals-workshop.icaps07.pdf. – Zugriffsdatum: 16.08.2012, 3:29

[Chan u. a. 2007b] Chan, Hei ; Fern, Alan ; Ray, Soumya ; Wilson, Nick ; Ventura, Chris: Online Planning for Resource Production in Real-Time Strategy Games. In: Boddy, Mark S. (Hrsg.) ; Fox, Maria (Hrsg.) ; Thiébaux, Sylvie (Hrsg.): *ICAPS*, AAAI, 2007, S. 65–72. – http://www.aaai.org/Papers/ICAPS/2007/ICAPS07-009.pdf. – Zugriffsdatum: 16.08.2012, 03:31. – ISBN 978-1-57735-344-7

[Chick 2000] Chick, Tom ; IGN Online (Hrsg.): *Starcraft – IGN*. 2000. – http://pc.ign.com/articles/152/152159p1.html. – Zugriffsdatum: 01.10.2012, 16:30

[Crecente 2009] Crecente, Brian ; kotaku.com (Hrsg.): *Competitive StarCraft Gets UC Berkeley Class*. 2009. – http://kotaku.com/5141355/competitive-starcraft-gets-uc-berkeley-class. – Zugriffsdatum: 01.10.2012, 21:27

[Dereszynski u. a. 2011] Dereszynski, Ethan ; Hostetler, Jesse ; Fern, Alan ; Dietterich, Tom ; Hoang, Thao-Trang ; Udarbe, Mark: Learning Probabilistic Behavior Models in Real-Time Strategy Games. In: *Proceedings of AIIDE*, https://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/viewFile/4073/4405. – Zugriffsdatum: 16.08.2012, 03:00, 2011, S. 20–25

[Deutscher eSport-Bund 2011] Deutscher eSport-Bund: *Wissen*. 2011. – http://www.e-sb.de/C1350/wissen.htm. – Zugriffsdatum: 08.10.2012, 22:02

[Graft 2009] Graft, Kris ; Edge Online (Hrsg.): *Blizzard Confirms One "Frontline Release" for '09*. 2009. – http://www.edge-online.com/news/blizzard-confirms-one-frontline-release-09. – Zugriffsdatum: 01.10.2012, 16:11

[Hagelbäck 2009] Hagelbäck, Johan ; AiGameDev.com (Hrsg.): *Using Potential Fields in a Real-time Strategy Game Scenario (Tutorial)*. Website. 2009. – http://aigamedev.com/open/tutorial/potential-fields/. – Zugriffsdatum: 07.08.2012, 02:23

[Hagelbäck 2012] Hagelbäck, Johan: *Multi-Agent Potential Field Based Architectures for Real-Time Strategy Game Bots*, Blekinge Institute of Technology, Dissertation, 2012. – http://www.bth.se/tek/jhg.nsf/bilagor/JHG_1_PhD_Thesis_pdf/$file/JHG_1.PhD.Thesis.pdf. – Zugriffsdatum: 07.08.2012, 4:55

[Heinermann 2011] Heinermann, Adam: *StarcraftGuide – Advanced Information Guide for Starcraft Broodwar*. Wiki. 2011. – http://code.google.com/p/bwapi/wiki/StarcraftGuide. – Zugriffsdatum: 03.10.2012, 06:57

[Hostetler u. a. 2012] Hostetler, Jesse ; Dereszynski, Ethan ; Dietterich, Tom ; Fern, Alan: *Inferring Strategies from Limited Reconnaissance in Real-time*

Strategy Games. 2012. – http://web.engr.oregonstate.edu/~afern/papers/uai12-hostetler.pdf. – Conference on Uncertainty in Artificial Intelligence

[Huang 2011] Huang, Haomiao ; Ars Technica (Hrsg.): *Skynet meets the Swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition*. 2011. – http://arstechnica.com/gaming/2011/01/skynet-meets-the-swarm. – Zugriffsdatum: 04.08.2012, 21:34

[Klein 2010] Klein, Dan: *The Berkeley Overmind Project*. Website. 2010. – http://overmind.cs.berkeley.edu/. – Zugriffsdatum: 04.08.2012, 23:04

[Kovarsky und Buro 2006] Kovarsky, Alex ; Buro, Michael: A First Look at Build-Order Optimization in Real-Time Strategy Games. In: *Proceedings of the GameOn Conference*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.5338&rep=rep1&type=pdf. – Zugriffsdatum: 16.08.2012, 3:36, 2006, S. 18–22

[Krastev 2012] Krastev, Krasimir: *Starcraft Broodwar ladder for BWAPI bots*. Website. 2012. – http://bots-stats.krasi0.com/. – Zugriffsdatum: 06.08.2012, 03:25

[Lin und Ting 2011] Lin, Chih-Sheng ; Ting, Chuan-Kang: Emergent Tactical Formation Using Genetic Algorithm in Real-Time Strategy Games. In: *Technologies and Applications of Artificial Intelligence (TAAI), 2011 International Conference on*, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6120766&abstractAccess=no&userType=inst. – Zugriffsdatum: 18.08.2012, 04:44, nov. 2011, S. 325–330

[Liquipedia 2009b] Liquipedia: *Unit Dancing — Liquipedia StarCraft Wiki*. 2009. – http://wiki.teamliquid.net/starcraft/index.php?title=Unit_Dancing&stableid=13966. – Zugriffsdatum: 24.08.2012, 03:39

[Louis u. a. 2004] Louis, Sushil J. ; Miles, Chris ; Cole, Nicholas ; McDonnell, John: Playing to Train: Case-Injected Genetic Algorithms for Strategic Computer Gaming. In: *Proceedings of the first Workshop on Military and Security Applications of Evolutionary Computation, Seattle, WA*, http://www.cse.unr.edu/~miles/papers/msaec2004.pdf, 2004, S. 6–12

[Mahlmann und Preuss 2011a] Mahlmann, Tobias ; Preuss, Mike: *CIG 2011 StarCraft competition: final round*. Präsentation. 2011. – http://ls11-www.informatik.uni-dortmund.de/_media/rts-competition/final-round.pdf. – Zugriffsdatum: 06.08.2012, 18:18

[Mahlmann und Preuss 2011b] Mahlmann, Tobias ; Preuss, Mike: *CIG 2011 StarCraft competition: qualifying round*. Präsentation. 2011. – http://ls11-www.informatik.uni-dortmund.de/_media/rts-competition/first-round.pdf. – Zugriffsdatum: 06.08.2012, 18:18

[mahnini 2009] MAHNINI ; TEAMLIQUID.NET (Hrsg.): *Zero Experience SC Beginners' Reference*. Blog. 2009. – http://www.teamliquid.net/blogs/viewblog.php?topic_id=94773. – Zugriffsdatum: 01.10.2012, 23:57

[Miles und Louis 2005] MILES, Chris ; LOUIS, Sushil J.: Case-Injection Improves Response Time for a Real-Time Strategy Game. In: *CIG*, IEEE, 2005. – http://www.cse.unr.edu/~miles/papers/cig2005.pdf

[Miles und Louis 2006] MILES, Chris ; LOUIS, Sushil J.: Towards the Co-Evolution of Influence Map Tree Based Strategy Game Players. In: LOUIS, Sushil J. (Hrsg.) ; KENDALL, Graham (Hrsg.): *CIG*, IEEE, 2006, S. 75–82. – http://www.cse.unr.edu/~miles/papers/cig2006.pdf

[Miles u. a. 2007] MILES, Chris ; QUIROZ, Juan C. ; LEIGH, Ryan E. ; LOUIS, Sushil J.: Co-Evolving Influence Map Tree Based Strategy Game Players. In: *CIG*, IEEE, 2007, S. 88–95. – http://www.cse.unr.edu/~miles/papers/cig2007.pdf. – ISBN 1-4244-0709-5

[Miles 2007] MILES, Christopher E.: *Co-evolving real-time strategy game players*. Reno, NV, USA, University of Nevada, Reno, Dissertation, 2007. – http://www.cse.unr.edu/~miles/papers/dissertation.pdf. – AAI3274501

[Oberberger 2010] OBERBERGER, Michael: *Evolving Potential Fields to Direct Tactics in Real Time Strategy Games*, University of Nevada, Reno, Master Thesis, 2010. – http://www.cse.unr.edu/~moberberger/oberberger_thesis_pfga_2010.pdf

[Richoux 2012] RICHOUX, Florian: *aiurproject – Artificial Intelligence Using Randomness*. 2012. – http://code.google.com/p/aiurproject/. – Zugriffsdatum: 30.09.2012, 02:44

[Rossignol 2006] ROSSIGNOL, Jim: *Korea — ¿ jim rossignol*. Website. 2006. – http://rossignol.cream.org/284/korea/. – Zugriffsdatum: 01.10.2012, 22:00

[Russell und Norvig 2004] RUSSELL, Stuart J. ; NORVIG, Peter: *Künstliche Intelligenz - ein moderner Ansatz*. 2. Auflage. Pearson Studium, 2004

[Schaeffer u. a. 2007] SCHAEFFER, Jonathan ; BURCH, Neil ; BJORNSSON, Yngvi ; KISHIMOTO, Akihiro ; MULLER, Martin ; LAKE, Robert ; LU, Paul ; SUTPHEN, Steve: Checkers Is Solved. In: *Science* (2007), July, S. 1144079+. – http://dx.doi.org/10.1126/science.1144079

[Synnaeve und Bessière 2011a] SYNNAEVE, Gabriel ; BESSIÈRE, Pierre: A bayesian model for opening prediction in rts games with application to starcraft. In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on* IEEE (Veranst.), http://emotion.inrialpes.fr/people/synnaeve/index_files/OpeningPrediction.pdf. – Zugriffsdatum: 16.08.2012, 03:43, 2011, S. 281–288

[Synnaeve und Bessière 2011b] Synnaeve, Gabriel ; Bessière, Pierre: A Bayesian model for RTS units control applied to StarCraft. In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on* IEEE (Veranst.), `http://hal.inria.fr/docs/00/60/72/81/PDF/BayesianUnit.pdf`. – Zugriffsdatum: 16.08.2012, 04:57, 2011, S. 190–196

[Synnaeve und Bessière 2012] Synnaeve, Gabriel ; Bessière, Pierre: Special Tactics: a Bayesian Approach to Tactical Decision-making. (2012). – `http://emotion.inrialpes.fr/people/synnaeve/index_files/SpecialTactics.pdf`. – Zugriffsdatum: 16.08.2012, 04:58

[Synnaeve u. a. 2011] Synnaeve, Gabriel ; Bessière, Pierre u. a.: A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. In: *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/viewPDFInterstitial/4062/4416`. – Zugriffsdatum: 16.08.2012, 05:01, 2011

[Uriarte 2011] Uriarte, Alberto: *Multi-Reactive Planning for Real-Time Strategy Games*, Universitat Autònoma de Barcelona, Master Thesis, 2011. – `http://nova.wolfwork.com/files/Multi-Reactive_Planning_for_Real-Time_Strategy_Games.pdf`. – Zugriffsdatum: 19.08.2012, 21:39

[Weber 2010a] Weber, Ben: *AIIDE 2010 StarCraft Participants*. 2010. – `http://eis.ucsc.edu/StarCraftParticipants`. – Zugriffsdatum: 06.08.2012, 18:13

[Weber 2010b] Weber, Ben: *AIIDE 2010 Tournament 4 Results*. Website. 2010. – `http://eis.ucsc.edu/Tournament4Results`. – Zugriffsdatum: 06.08.2012, 18:42

[Weber und Mateas 2009a] Weber, Ben G. ; Mateas, Michael: Case-based reasoning for build order in real-time strategy games. In: *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, AAAI Press* (2009), S. 106–111. – `http://eis.ucsc.edu/sites/default/files/bweber_aiide_09.pdf`. – Zugriffsdatum: 15.08.2012, 20:42

[Weber und Mateas 2009b] Weber, Ben G. ; Mateas, Michael: Conceptual Neighborhoods for Retrieval in Case-Based Reasoning. In: *Case-Based Reasoning Research and Development* (2009), S. 343–357. – `http://eis.ucsc.edu/sites/default/files/iccbr_2009.pdf`. – Zugriffsdatum: 15.08.2012, 20:41

[Weber und Mateas 2009c] Weber, Ben G. ; Mateas, Michael: A Data Mining Approach to Strategy Prediction. In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* IEEE (Veranst.), `http://eis.ucsc.edu/sites/default/files/cig_2009.pdf`. – Zugriffsdatum: 16.08.2012, 03:13, 2009, S. 140–147

[Weber und Ontañón 2010] Weber, Ben G. ; Ontañón, Santiago: Using Automated Replay Annotation for Case-Based Planning in Games. In: *Proceedings of*

the ICCBR Workshop on Computer Games (2010). – http://games.soe.ucsc.edu/sites/default/files/iccbr-cg.pdf. – Zugriffsdatum: 16.08.2012, 3:07

[Wikipedia 2012] Wikipedia: *A*-Algorithmus — Wikipedia, Die freie Enzyklopädie*. 2012. – http://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=106170508. – Zugriffsdatum: 06.08.2012, 23:08

[World Cyber Games 2012] World Cyber Games: *StarCraft: Brood War @ World Cyber Games*. Website. 2012. – http://www.wcg.com/renew/history/games/games_official.asp?selgame=2. – Zugriffsdatum: 01.10.2012, 21:54

Abbildungsverzeichnis