University of Aarhus
Department of computer science
Åbogade 34
8000 Århus C
Denmark

Masters Thesis

Ronni Laursen
rage@daimi.au.dk
19993343

Daniel Nielsen
djn@daimi.au.dk
19992541

18th September 2005

Investigating small scale combat situations in real time strategy computer games

Supervisor: Ole Caprani
ocaprani@daimi.au.dk

Abstract

This thesis presents and analyses the problem of small scale combat (SSC) in real time strategy (RTS) computer games. An RTS game is a war simulator where several opposing factions battle in a virtual world. The problem of SSC appears when soldiers, called *units*, of opposing factions meet in this virtual world.

In commercial RTS games SSC situations are handled by applying simple rules to each unit involved such as “attack the nearest enemy unit”. The result is far from optimal as several examples will show. Therefore, we will investigate other methods for handling SSC situations.

The commercial computer game industry’s name for modules handling computer controlled characters is *Game AI*. In this thesis we investigate which methods and algorithms this concept covers. This investigation includes an overview of selected computer game genres, an introduction to what quality in computer games is, and an overview of how Game AI is handled in different genres.

We will define the concept of *consistent* characters. On this basis we will discuss how consistency influences the quality of a computer game. We will argue that if units in an RTS game are to be considered consistent they ought to behave optimally regarding the rules of the game world.

In this thesis we present a method for solving SSC based on rule-induced timestamped game trees. Due to the amount of information in the nodes we investigate a machine learning approach to derive a node rating function. We achieve a reduction of the number of nodes in the tree by applying a sequence of rules to each node, thereby reducing the fanout. We present several examples of rating functions and rule sequences.

For evaluation of methods solving the SSC problem we use a complete, mature and commercially comparable Open Source RTS game called *Wargus*. We will measure the performance of each variation of the game tree-based methods. This is done by comparing the performance of the variations with the performance of the built-in module in Wargus.

The experiments performed in this thesis show that there is room for improvement in the way SSC situations are handled in Wargus. With appropriate rule sequences and rating methods the game tree-based methods perform better than the rule-based systems currently in use.

Resumé

Dette speciale præsenterer og analyserer *small scale combat* (SSC) situationer i realtids strategispil (RTS). Et RTS spil er en krigssimulator hvor forskellige fraktioner kæmper i en virtuel verden. SSC situationer opstår når soldater, kaldet enheder, fra forskellige fraktioner kæmper i denne virtuelle verden.

I kommercielle RTS spil håndteres SSC situationer ved at anvende regler til at styre hver enhed, såsom "angrib den nærmeste fjende". Resultatet er ikke optimalt, som flere eksempler vil vise. Derfor vil vi undersøge andre metoder til at håndtere SSC situationer.

Computerspilsbranchen bruger navnet *Game AI* til at betegne moduler, som håndterer computerstyrede karakterer. I dette speciale vil vi præsentere hvad dette begreb dækker over mht. metoder og algoritmer. Denne undersøgelse inkluderer et overblik over udvalgte computerspilsgenrer, en introduktion til hvad computerspilskvalitet er, samt et overblik over hvordan Game AI bliver brugt indenfor forskellige genrer. Vi vil introducere begrebet *konsistente* karakterer og på dette grundlag vil vi diskutere, hvorledes konsistens påvirker et computerspils kvalitet. Vi vil argumentere for, at hvis RTS enheder skal betragtes som konsistente så skal de opføre sig på en optimal måde med hensyn til den virtuelle verdens regler.

I dette speciale præsenterer vi en metode til håndtering SSC situationer. Metoden baseres på et regel-induceret tidsstemplet spiltræ. Grundet mængden af information i spiltræsknuderne vil vi kigge nærmere på en maskinindlæringstilgang til at afgøre knudernes værdi. Vi reducerer antallet af børn til hver knude ved at anvende sekvenser af regler på disse. Vi præsenterer adskillige eksempler på værdifunktioner og regelsekvenser.

Til at evaluere metoder der håndterer SSC situationer, bruger vi et komplet og kommercielt sammenligneligt Open Source RTS spil, kaldet *Wargus*. Eksperimenter der viser spiltræsvarianternes ydeevne er udført ved at evaluere hver spiltræsvariation på flere konstruerede SSC situationer mod det indbyggede modul i Wargus.

De målte resultater viser at den måde Wargus håndterer SSC situationer på kan forbedres. Med egnede regelsekvenser og værdifunktioner kan en spiltræsbaseret metode yde bedre end de regelbaserede systemer der bliver brugt i RTS computerspil idag.

Preface

About the authors

The authors of this masters thesis are Ronni Laursen and Daniel Nielsen, both masters students of computer science. Ronni holds a bachelor in multimedia and computer science; while Daniel has a bachelor in mathematics and computer science.

Both authors gained an interest in computers and computer games in the late 1980's and that interest has increased ever since. The natural curiosity of what comes next in the market of computer games is a big part of being in the computer game "sphere". Both authors take great pleasure in following the media outlets dealing with computer games, to glean the latest news about new releases and technologies.

Daniel's primary interests in computer games are real time strategy games, role playing games and the ever evolving market for console games. Ronni's interests are primarily first person shooters and role playing games.

Acknowledgements

A big thank you from both authors goes to

Ole Caprani for supervising us when everybody else had left.

Thiemo Krink for countless fruitful discussions.

Brian Mayoh for several references and discussions.

Daniel also wishes to thank his family, Christine and Marie, for support and for putting up with the long hours that went into the making of this thesis.

Ronni wishes to thank his family for support and love. A special appreciation goes to all of my close friends for your support throughout the years. Love you all.

Contents

Abstract **i**

Resumé **ii**

Preface **iii**
About the authors iii
Acknowledgements iii

List of Figures **viii**

1 Introduction **1**
1.1 Motivation for improving unit behaviour 2
1.2 Goals 3
1.3 Thesis overview 3

2 Computer games **6**
2.1 Terminology 6
2.2 Computer game genres 8
2.2.1 Adventure 8
2.2.2 Board games 9
2.2.3 Strategy games 10
2.2.4 Shooters 11
2.2.5 Role playing games 12
2.2.6 Simulators 14
2.2.7 Action based games 16
2.2.8 An example of genre combination 18
2.3 Computer game quality 20
2.3.1 Immersion as a mark of quality 21
2.3.2 Consistent character behaviour 24

3 Game AI problems **28**
3.1 Information available to characters 29
3.2 Game AI problems in genres 30
3.2.1 Adventure 30
3.2.2 Board games 31
3.2.3 Strategy games 31
3.2.4 Shooters 32
3.2.5 Role playing games 33
3.2.6 Simulators 33
3.2.7 Action based games 35
3.3 Examples of applied Game AI 35
3.3.1 Black & White 36
3.3.2 No One Lives Forever 2 37
3.3.3 Half-Life 38
3.3.4 Warcraft III 39
3.4 Real time strategy (RTS) AI 40
3.4.1 RTS games in detail 40
3.4.2 Game AI problems in RTS games 44
3.4.3 Subproblems in small scale combat (SSC) 45
3.4.4 Consistent behaviour in SSC situations 46

4 The Wargus platform **48**
4.1 The Wargus game world 48
4.1.1 Scenarios 49
4.1.2 Map 49
4.1.3 Units 49
4.1.4 The built-in participant 51

5 Game trees applied to small scale combat **53**
5.1 Methods for solving SSC 53
5.1.1 Rule based methods 53
5.1.2 Evolutionary based methods 54
5.1.3 Game tree based methods 55
5.2 Game trees 55
5.2.1 Timestamped game trees 57
5.2.2 Issues 60
5.3 Representation 60
5.3.1 Threat matrix 61
5.3.2 Deriving a threat value for units 61
5.3.3 An SSC example 64
5.4 Pruning 64

5.4.1 Rules for game tree pruning . . . 67
5.4.2 Rules . . . 69
5.4.3 Rules and their sequencing . . . 73
5.5 Rating game states . . . 75
5.5.1 Handcrafted rating method . . . 76
5.5.2 Machine learning rating methods . . . 77
5.5.3 Choosing actions for units . . . 78
5.6 Measuring the performance of an SSC situation . . . 79
5.6.1 SSC situation value . . . 79
5.6.2 Experiments . . . 80

6 Extending the Stratagus engine 81
6.1 The Stratagus engine background . . . 81
6.2 The Stratagus engine . . . 82
6.2.1 Communication protocol . . . 82
6.2.2 Unit control mechanisms . . . 83
6.3 Integration with the engine . . . 84
6.3.1 Execution path . . . 84
6.3.2 The C to Java link . . . 85
6.4 Java Packages . . . 86
6.4.1 Stratagus Java package . . . 86
6.4.2 Rada Java package . . . 86

7 Results 88
7.1 Setup . . . 88
7.1.1 Experiments . . . 89
7.1.2 SSC situations . . . 90
7.2 Questions . . . 93
7.3 Presentation of results . . . 95
7.4 Presentation of movies . . . 96
7.5 Discussion . . . 98
7.5.1 A problem with the game tree method . . . 98

8 Future work 109
8.1 Engine enhancements . . . 109
8.2 Machine learning accuracy . . . 109
8.3 Improving evaluation methods . . . 110
8.4 Improving the integration with Wargus . . . 110
8.5 Game tree extensions . . . 111

9 Conclusion 113

Bibliography **115**

A A note on search for literature **124**

B Summary of diary **126**

C Design and Implementation **129**
C.1 Design 129
C.1.1 Java packages 129
C.2 Implementation 136
C.2.1 Datastructures and methods 136
C.2.2 Game tree construction 138

D Results **144**
D.1 Reading the tables 144
D.2 The tables 146

E Contents of the enclosed CD **154**

List of Figures

1.1 A sample small scale combat (SSC) situation in the game of Wargus 2

2.1 Screenshot from Monkey Island by LucasArts from 1990 9
2.2 Screenshot from Grand Master Chess v.2.5 by MediaResearch-Group from 2004 10
2.3 Screenshot from Dune II by Westwood Studios from 1992 11
2.4 Screenshot from Half-Life: Counter-Strike by Valve Software and CSteam from 1999 12
2.5 Screenshot from Tomb Raider: Angel of Darkness by Core Design from 2003 13
2.6 Screenshot from Eye of the Beholder II by Westwood Studios from 1991 14
2.7 Screenshot from Madden NFL 2004 by EA Sports from 2004 . . . 15
2.8 Screenshot from Gran Turismo II by Polyphony Digital from 2000 16
2.9 Screenshot from SimCity 4 by Maxis from 2003 17
2.10 Screenshot from Super Mario Brothers by Nintendo from 1985 . . 18
2.11 Screenshot from Street Fighter Alpha II by Capcom from 1996 . . 19
2.12 Screenshot from Warcraft III by Blizzard from 2002 19

3.1 Screenshot from Zork I: The Great Underground Empire by Infocom from 1980 31
3.2 Screenshot from Transport Tycoon Deluxe by MicroProse from 1995 34
3.3 Screenshot from Black And White by Lionhead Studios from 2001. 36
3.4 Screenshot from Warcraft III by Blizzard from 2002 depicting the cheating of the participant at the highest difficulty level. 39
3.5 Screenshot from Warcraft III by Blizzard from 2002 showing a typical game start for the Orcs. 40
3.6 Screenshot from Warcraft III by Blizzard from 2002 showing the options available at the Orc War Mill. 42

4.1 The movement possibilities for a lone melee unit 51
4.2 A scenario with several units 51

5.1 An example of a game tree in the game of *Tic Tac Toe* 56
5.2 A sample timestamped game tree 58
5.3 The situation, which figure 5.4 and figure 5.5 model 58
5.4 Two opposing units ready in the same timestamp 58
5.5 Two opposing units ready in different timestamps 59
5.6 An SSC situation in Wargus 65
5.7 The snapshot corresponding to figure 5.6 65
5.8 The threat matrix derived from the snapshot in figure 5.7 66
5.9 A sample influence map 68
5.10 A game tree and a sample rating method 76

6.1 A callgraph depicting the overall communication between the engine and our module 84

7.1 A simple SSC situation with two opposing melee units called *Scen1vs1* 90
7.2 Two melee units opposing three melee units called *Scen3vs2* . . . 91
7.3 Two squads with four melee and three ranged units each called *Scen7vs7* 92
7.4 Two squads with two melee and four ranged units each called *archer-ambush* 92
7.5 Three melee and two ranged units surrounded by four melee units and one ranged unit called *Captured* 93
7.6 A problem with our game tree-based method 99

C.1 The UML diagram of the Stratagus package 130
C.2 The UML diagram of the Experiments component 132
C.3 The UML diagram of the rada package 134
C.4 The full recursion graph 142
C.5 The reduced recursion graph 142

Chapter 1

Introduction

"So long as there are men there will be wars"
- Albert Einstein

The commercial computer game industry has evolved into a major part of today's entertainment industry. Contributing factors to this growth are an increased interest in computer games from players of all ages and an ever increasing budget for production and marketing of computer games. This interest has grown to a point where even the enormous Hollywood movie industry has reacted by producing movies such as the Tomb Raider series, [Paramount, 2005], based on computer game worlds and the characters within.

The amount of interest in computer games has furthermore created a large consumer demand for more photo realistic graphics, more interesting game characters and a more immersed feeling when playing. This has led to a boom in the graphics branch whereas methods dealing with behaviour of the characters populating and interacting with the game world still lacks the breakthrough computer graphics have had.

Techniques for controlling computer game characters vary greatly in sophistication. All to often a static method such as scripting is chosen. Such methods can often lead to repetitive behaviour and a lack of adaptation to situations not foreseen by the designer of the scripts.

The methods which control computer game characters are called *Game AI* by the commercial game industry. A method for controlling characters should ensure a *consistent* behaviour. Consistency means that a character supposed to fill some specific part in the world has to ensure that the player believes in it. Given the concepts of the game world the player expects a certain behaviour from the game world characters. The pixels on the screen are not just RGB-values, but constitute living characters in the game world.

Real time strategy (RTS) games are war simulators where several factions battle in a virtual game world. Several characters, called *units*, are situated in this

Figure 1.1: A sample small scale combat (SSC) situation in the game of Wargus

world and belong to different factions representing the factions' soldiers. Here, consistent behaviour implies that the RTS units must convince the player that they are hardened soldiers and fight as such. When these units fight a small scale combat (SSC) situation appears and the goal in these situations is to kill the enemy units. How this goal can be reached is the main focus of this thesis.

Figure 1.1 depicts an SSC situation in the RTS game of Wargus in which units from a red faction is fighting units from a green faction in a grassy environment.

1.1 Motivation for improving unit behaviour

RTS games are among the most popular games in today's computer game market. The reasons are that RTS games include several aspects and challenges which all must be tackled by the player in real time. Combat must be conducted while planning logistic routes and constructing defensive buildings. The actual gameplay in RTS games vary from game to game but many focus on a player controlled low-level handling of units. This reduces the attention from the high-level aspects which often leads to a sub-optimal handling of these.

This thesis proposes an alternative. We will argue that RTS games are often won on high-level decisions rather than handling a handful of units better than the opponent. By giving the player a near-optimal autonomous handling of SSC situations a shift in focus from low-level handling of units to high-level strategic decision making is made. We believe that this shift in focus and thus in gameplay

is needed to ensure victory in those RTS games where both low-level and high-level aspects are included.

RTS games are also interesting from a computer science perspective since solving the problems within this domain is difficult. Methods handling the sub-problems of RTS must furthermore find solutions without violating the real time demands.

1.2 Goals

The notion of Game AI includes more than just character control mechanisms. We will introduce selected computer game genres and investigate which Game AI methods are used in these. Attempts at achieving consistent character behaviour will be presented and discussed. We will argue that if a consistent character behaviour is reached the overall quality of the computer game increases.

We will argue that an optimal handling of SSC situations is required to make the RTS units exhibit consistent behaviour thereby increasing an RTS game's quality. We propose a method for handling SSC situations in an near-optimal fashion. The success criteria of the presented method is thus to handle SSC situations as optimally as possible.

We investigate how SSC situations are handled by contemporary commercial RTS games and present a game tree-based method which allows for near-optimal solutions to SSC. An evaluation-suite is set up to test and analyse which of several game tree-based methods perform best. Ian Lane Davis, [Davis, 1999], has the following remark concerning game trees used in RTS games:

> "Optimal solutions such as game trees [...] that try to predict several moves ahead (useful for tic-tac-toe, and some other small games) are inappropriate and infeasible due to the relatively enormous branching factor", [Davis, 1999]

Although we acknowledge the problematic issues involved in using game trees in real time games we find it interesting and challenging to investigate such a method and measure its performance against methods currently in use.

1.3 Thesis overview

This thesis consists of the following chapters:

Chapter two presents computer games and selected computer game genres which make up the background this thesis is based on. Chapter two furthermore introduces and discusses computer game quality and which elements are included in a game of high quality.

Chapter three presents which methods are used to create high-quality characters in the presented computer game genres. Chapter three continues with a detailed description of RTS games. An examination of SSC is given special attention.

Chapter four presents the RTS game of Wargus in which our method for handling SSC situations is based. The presentation will describe the rules of Wargus in detail.

Chapter five presents game trees and the modifications needed to handle the real time constraints and concurrent actions of RTS games. Furthermore, the chapter will present and elaborate on the remaining issues associated with game trees.

Chapter six describes how the chosen RTS platform works and how our game tree-based method is integrated with this platform.

Chapter seven presents the collected results along with a discussion of these. The results are based on several SSC situations where the game tree-based methods are evaluated when playing against the built-in methods of the Wargus game.

Chapter eight presents and elaborates on several future work-topics.

Chapter nine concludes the thesis.

Appendix A elaborates on the background and validity of the collected material and literature.

Appendix B presents an extract from the diary held during the course of this thesis.

Appendix C presents the full design and implementation details of our game tree-based method.

Appendix D shows the result-tables collected during evaluation of our game tree method.

Enclosed CD We have recorded a number of movies illustrating several issues encountered during the making of this thesis. A description of the enclosed CD is given in appendix E on page 154.

Chapter 2

Computer games

"It's a me, Mario!"
- Mario

A computer game is a game set in a computer-controlled virtual universe that players interact with. In this chapter several computer game genres will be introduced and examples of game which are stereotypical for their genres will be presented.

We seek a foundation to answer the seemingly easy question – when is a computer game said to be of high quality? This innocent question has been the topic of research in human computer interaction and aesthetics. The answer depends very much on which definition of quality is used. Does it mean that a game is technically superior to its contemporary peers? Is it the overall impression the game has on a human player? Or is it the kind of interactivity which is available in the game?

It is obvious that ensuring a technical superior game requires insight into several branches of computer science, but what else is required to make a high quality game? We will argue that the concept of *consistent characters* is equally as important to game quality as consistent graphics.

2.1 Terminology

Before discussing the different genres in the realm of computer games, some common terminology for describing computer games is needed. Since most concepts in the area of computer games are interpreted differently, the following definitions will be adopted throughout this thesis. An complementary survey of terminology is presented in [Nareyek, 2001], where Alexander Nareyek defines modules of artificial intelligence (AI) in commercial computer games along with their area of responsibility within the game. Our list is based partly on Nareyek's definitions along with definitions from [Champandard, 2003, Chap. 1], [Rabin, 2004b], [Ra-

bin, 2004c] and our own experiences as computer game players and students of several AI related courses.

Player is the term used for the human being playing the computer game. Throughout this thesis the human player will be referred to as *he*.

Avatar is the game's representation of a player. In games, where only one player plays at a time, this often equals the lead role in the story of the game – although some games lack an avatar altogether. An example of this is strategy games where the player often assumes the role of a general over vast armies and is not explicitly represented in the game world.

Game world is the virtual universe in which the game takes place. The game world is presented to the player by means of graphics, sounds and the story of the game.

Game objects are objects which appear in a computer game. Game objects can either be *active* or *reactive*. Active means that the object can actively conduct actions upon itself or upon the game world. Reactive means that the object cannot by itself perform actions, but only react as an effect of an action.

Characters are active game objects situated in the game world. Characters do not have a direct motive, e.g. mood-setting animals in digital forests. Characters are said to be *consistent* if these follow the game's rules and theme and if these are able to portray their part in the game world convincingly. For more detail on consistency, see section 2.3.2 on page 24.

Non-Player Characters are those characters who participate in the game's story. A non-player character (NPC) can be seen and understood as subsidiary character in a movie.

Participants are active game objects which assume the role of a player. Unlike both its character-counterparts, described above, a participant is not necessarily situated in the game world.

Support routine is a software module which in some way assists the player in playing the game at hand. Examples include methods for controlling many characters simultaneously, [Blizzard, 1995], implementing path-finding algorithms, [Wikipedia, 2005a], or methods choosing near optimal weapons for specific combat situations, [Champandard, 2003, Chap. 23].

Game AI is the term used to describe the commercial game industry's methods for controlling active game objects and for support routines. Game AI includes methods which intelligently control a camera's position and orientation, [Carlisle, 2004], create interesting auto-generated worlds, [Adams and Mendler, 2002] and as used in the computer game Diablo, [Blizzard, 2000], or methods which help the player cleverly control many characters simultaneously such as pathfinding, [Patel, 2004], simple behaviours in Warcraft II, [Blizzard, 1995] and create opposing participants and characters for the player to face in the game, [Champandard, 2003].

2.2 Computer game genres

This section describes some of the more persistent and popular computer game genres and presents a game stereotypical of each genre discussed. The list is by no means exhaustive, but is meant to give the reader an overview of today's popular computer game genres. The list is based on our experiences as avid computer game players, the accumulated knowledge obtained by following many computer game media web-sites such as [IGN, 2005] and [GameSpot, 2005] and of course from literature about computer games. For complementary reviews of computer game genres the reader is referred to [Wolf, 2002] and [Wikipedia, 2005b].

A game is said to belong to a certain genre if the game has the traits which the genre represents. A game can possess traits from several genres and thereby belong to many genres. The traits are not carved in stone and the line between several of the presented genres is blurry.

2.2.1 Adventure

The adventure game genre concerns exploring a game world and conversing with the non-player characters (NPCs) which inhabit the world. Conversational details and the portraying of a detailed story are the fundamental elements in this genre. Often used styles for the story are humour, horror and love.

The lead role in the story is controlled by the player. Following the story of the game, the player is subjected to solving various puzzles – ranging from getting into a guarded mansion to obtaining a specific item from an NPC. Generally a non-violent approach is needed to solve the puzzle at hand. The stories are typically very rigid and progress is only made if certain key-puzzles are solved. Other actions have little or no effect on the main story but may reveal side-stories.

In figure 2.1 on the next page a screenshot from Monkey Island, [LucasArts, 1990], is shown. The avatar is the sailor standing up. There are three NPCs, all

Figure 2.1: Screenshot from Monkey Island by LucasArts from 1990

pirates and sitting down. In the bottom half of the screen the interface for controlling the avatar is located – a list of actions and an iconic representation of items currently in the avatar's possession. It is the responsibility of the player to discover the story of the game by means of his avatar.

Examples of popular adventure games are the Leisure Suit Larry series, [Sierra, 1987], and the Maniac Mansion series [LucasArts, 1987].

2.2.2 Board games

The classical board game genre involves two players playing against each other. Taking turns, the players move one piece at a time and each game usually has three possible outcomes – either player can win or the game can end in a draw. Computer board games are based on real games, such as Chess and made digital due to their popularity. Board games are often enhanced to enable players to play against each other via the Internet.

Figure 2.2 on the following page shows a screenshot from Grand Master Chess, [Media Research Group, 2004]. This is a digital version of Chess where the notable difference to ordinary Chess is the possibility of playing against a participant. The screenshot shows the Chess board in the middle and the game interface to the left and right hand of the screen.

The most popular board games are often implemented on community web sites where people compete. Examples of these are [TV2, 2005] and [Yahoo, 2005]. Games such as Chess, Go, Checkers and Backgammon are examples of popular board games made digital.

Figure 2.2: Screenshot from Grand Master Chess v.2.5 by MediaResearchGroup from 2004

2.2.3 Strategy games

Strategy games are war simulators where the player is controlling armies to compete for various resources and face opponents. These games often focus on planning, resource management and combat in order to achieve victory. These games can be divided into turn based strategy games and real time strategy (RTS) games.

Turn based strategy games allow all players to consider all actions carefully in turn before choosing an action. Also, turn based strategy games often focus on simulating real armies with great realism and careful planning. RTS games primarily focus on resource gathering, rapid unit handling and base construction and has a stronger focus on combat. Furthermore, RTS games require the players to choose actions continuously. RTS games are described in detail in section 3.4.1 on page 40.

Before playing the actual game all players must decide their allegiance. Most strategy games have several opposing factions. The types of characters, called *units*, and buildings available is determined by the choice of allegiance.

In figure 2.3 on the following page a screenshot from Dune 2, [Studios, 1992], is depicted. At the top of the screen the player's amount of *credits* available for unit and building construction is seen. The large section positioned lower left on the screen is the view of the battlefield, called the view-port. The black shroud is known as *fog of war* and represents areas of the map which are currently outside the view of the player's units or undiscovered territory. The middle right of the

Figure 2.3: Screenshot from Dune II by Westwood Studios from 1992

screen shows the status of the selected unit and its available actions. At the lower right of the screen the *mini map* is depicted. This is a small representation of the battlefield which allows the player to quickly survey the battlefield and position the view-port over interesting events.

Examples of popular RTS games are Starcraft, [Blizzard, 1998], and the Age of Empires series, [Ensemble Studios, 1997]. An example of a turn based strategy game is Civilization III, [Firaxis Games, 2001].

2.2.4 Shooters

Shooters are one of today's most popular genres. The human player controls an avatar in real time in either a first person or third person perspective. First person means the view of the world is presented through the eyes of the avatar whereas third person perspective implies that the player is viewing the avatar from behind. A shooter with a first person perspective is called a first person shooter (FPS).

In a shooter the player is fighting dangerous animals, vicious monsters and tough marines with various advanced weaponry. Most of these games are primarily based on action. Even though action is an important element in this genre developers are increasingly including stories which entangles the player further in the game.

Figure 2.4 on the next page shows a screenshot from an FPS called Counter-Strike, [CSteam, 1999]. This game contains *teams* consisting of human players and participants. Counter-Strike pits a team of counter-terrorists against a team of

Figure 2.4: Screenshot from Half-Life: Counter-Strike by Valve Software and CSteam from 1999

terrorists in rounds of competition won by completing an objective or eliminating the opposing team. Most of the screen is reserved for viewing the landscape where the avatar is situated. The upper left corner shows the radar, which indicates the general direction of the player's teammates. The bottom of the screen shows the chat-window and status of the avatar along with the time remaining on this map. Also depicted on the screen is a pair of hands and a weapon. This represents the weapon currently equipped by the avatar. The cross-hair in the middle of the screen represents the avatar's current aim. The red marker in the middle of the screen above the cross-hair indicates that the avatar has taken or is taking damage from that direction.

Figure 2.5 on the following page shows a screenshot from Tomb Raider: Angel of Darkness, [Design, 2003] – the sixth instalment in the Tomb Raider series, all of which are third person shooters. The image shows the avatar in a gunfight with two monsters.

The list of marketed shooters is long, but some of the most successful today are the Quake series, [ID Software, 1999], and the Unreal series, [Epic Games, 2004].

2.2.5 Role playing games

Role playing games (RPGs) have their roots in the pen and paper role-playing communities. In these communities the role-players often identify themselves

Figure 2.5: Screenshot from Tomb Raider: Angel of Darkness by Core Design from 2003

with a fictional character in a fictional world.

The digital version of this type of game bears resemblance to the adventure genre, but has a higher focus on the avatar's skills. The player is identifying himself with the avatar and information gathering is a very important element in these games. Often the game world is a fictional medieval world which is called fantasy based RPG. The stories in RPGs are *experienced* more than *played* by the player. The story's progress is influenced by the player's choices. Also, many RPGs have an element of combat in them. This allows the player to see how the level of interactivity changes as his avatar gains more abilities and skills.

In figure 2.6 on the next page a screenshot from Eye of the Beholder, [Westwood Studios, 1991], is shown. The player controls a group of characters called a *party*. Their images, names and some iconography describing their equipped items and abilities are shown in the right of the screen. At the bottom of the screen a textual representation of what has happened in the game is seen. The upper left depicts the current view of the party. Two enemies are also depicted with whom there currently is battle. The arrows in the lower left is used for controlling the party's movement.

The RPG genre's most popular games today include both new and old games, since the stories in these are the primary source of player interest. Examples are

Figure 2.6: Screenshot from Eye of the Beholder II by Westwood Studios from 1991

The Elder Scrolls, [Bethesda Softworks, 1994], and the Fallout series, [Interplay, 1997].

A sub-genre called massive multi-player online role playing games (MMORPGs) is currently growing in popularity. In MMORPGs he players inhabit the same virtual world concurrently. This allows for players to form groups and oppose other groups of either players or characters. Examples are World of Warcraft, [Blizzard, 2005], the EverQuest series, [Sony Online Entertainment, 1999], and the danish newcomer Seed, [Runestone, 2005], (in development at time of writing).

2.2.6 Simulators

The simulator genre contains games which simulate a sport, a car race or a whole world. Games in this genre generally simulate a real life concept such as soccer or a town. The degree of realism in each game varies greatly, mainly in the physics simulation and in the actions available. For example, it might be possible to shoot weapons at the opponents in some car simulators.

The following subsections briefly describe three kinds of simulators which have proven to be popular.

Sport simulators

These games simulate, as the name implies, a sport be that soccer, American football, basketball or tennis. Realism and the right “feel” with the ball are important elements in this genre.

Figure 2.7: Screenshot from Madden NFL 2004 by EA Sports from 2004

Figure 2.7 shows a screenshot from Madden NFL 2004, [EA-Sports, 2004]. The quarterback with number 97 has the ball. In the middle of the screen the 10 yard line is highlighted in yellow. At the left side of the image a yellow marker is placed indicating the avatar's current aim with the ball.

Examples of sport simulators are the FIFA-based, [GameFAQs, 1996], and the NBA-based series, [GameFAQs, 1993].

Driving simulators

Games of this type simulate motorised vehicle driving. The "feel" with the vehicle is very important. E.g. how the car slides when turning hard and how the vehicle accelerates on rainy or snowy surfaces.

In figure 2.8 on the next page a screenshot from Grand Turismo, [Polyphony, 2000], is depicted. Gran Turismo is a game about cars where the focus is on tuning and designing a car and then race it against other players. In the screenshot the view is placed within the car. The lower left shows a miniature representation of the track. To the lower right, upper right and upper left different statistics are presented to the player.

Examples of car simulators are the Driver series, [Atari, 2000], and TOCA race driver, [Codemasters, 1998].

Figure 2.8: Screenshot from Gran Turismo II by Polyphony Digital from 2000

World simulators

These games are simulating a city, a theme park or possibly a world. It is usually the player's obligation to control the game objects in such a way that characters populating the game world are happy or such that economic charts improve.

In figure 2.9 on the following page a screenshot from Sim City 4, [Maxis, 2003], is shown. In this game, the player is responsible for managing an entire city – from city planning, emergency response-times to sewer systems. A large section of the screen shows the city to be managed. Together with the elaborate menu system to the left and the ability to zoom and turn the camera the player interacts directly with the city. The bottom right half of the screen shows a lot of statistics necessary to supervise the city. The lower left shows a simple mini-map and controls for the speed of the simulation.

Examples of popular world simulators include the Black & White series, [Lionhead Studios, 2001], and The Sims series, [EA Games, 2000].

2.2.7 Action based games

Action based computer games focus on action. Among others this genre captures the platform genre and one-on-one fighting games. These genres are described below.

Figure 2.9: Screenshot from SimCity 4 by Maxis from 2003

Platform games

This genre places the avatar in an environment where the avatar collects points often in the form of coins, fruit or jewelry. When enough points are collected the player can proceed to the next level. The main player challenge is coordination of the avatar's movement as the levels contain pitfalls and other obstacles.

In figure 2.10 on the next page a screenshot from Super Mario Brothers, [Nintendo, 1985b], is shown. The top of the screen depicts the current score, amount of coins collected, the map the player is in and the time left to complete the map. The avatar is currently in the middle of a jump. The two mushrooms walking on the ground are characters which can damage the avatar. The bricks hanging in mid-air can be either jumped to and from or smashed by jumping up into the them. The brick with the question mark can be jumped into from beneath and contains either coins or upgrades for the avatar to collect.

The Sonic series, [Sonic Team, 1991], and the Mario series, [Nintendo, 1985a], are two of the most popular and successful game series in this genre.

Fighting games

Fighting games consist of a player's avatar fighting an opponent using kicks, punches and various special moves. Each time the avatar or the opponent is hit health is lost. When the health reaches zero the avatar or opponent has lost the round.

Figure 2.10: Screenshot from Super Mario Brothers by Nintendo from 1985

In figure 2.11 on the following page a screenshot from Street Fighter Alpha 2, [Capcom, 1996], is depicted. The top of the screen has two bars indicating the current health of the player and opponent. At the upper middle a number is showing the amount of seconds left in this round. The avatar to the left is currently performing some special move which deals damage to the opponent to the right.

Examples of popular fighting games are the Mortal Kombat series, [Midway, 1992], and the Tekken series, [Namco, 1994].

2.2.8 An example of genre combination

Warcraft III (see section 3.3.4 on page 39) is primarily a real time strategy (RTS) game. However, Warcraft III also possesses traits from the role playing game genre through the use of NPCs, called *Heroes*. The player is not identifying himself with the heroes but these are a vital element in the story and game world. The heroes are able to carry items, gain experience points through fighting and gain magical abilities far better than the RTS characters.

In figure 2.12 on the following page the heroes Jaina and Arthas are fighting enemies inhabiting the game world. Jaina is selected and in the bottom of the screen her experience, inventory and available spells are visualised.

Figure 2.11: Screenshot from Street Fighter Alpha II by Capcom from 1996

Figure 2.12: Screenshot from Warcraft III by Blizzard from 2002

2.3 Computer game quality

What is it that makes a computer game good or worth playing? Is it beautiful environmental sounds, a photo-realistic game world, the interaction possibilities with game objects and characters or a combination of these elements?

Playing computer games is all about having fun and aesthetic considerations can play a major role in the quality of the game. The concept known as *immersion* is a way of measuring the quality of a game. Even though the main focus of this thesis is the Game AI subsystem the following presentation and discussion will introduce computer game quality as a more general concept. This is primarily to motivate the quality introduced with consistent characters in computer games obtained by means of Game AI methods.

The essential aesthetic elements in a computer game are the *narrative*, the *award* and the *gameplay* which are all presented in detail in [Rouse, 2000]. These elements will be described in short while the concept of *immersion* presented in [Taylor, 2002] will be described in higher detail in section 2.3.1 on the next page. It should be noted that these elements should not be seen in isolation nor as the only ones. Some games do not include a narrative but should not be seen as inferior just like some games do not include an award to the player.

The narrative

Some narrative elements in a computer game can be seen as pure background elements which has no direct effect on the game. It could be the implicit reasons why the avatar is behind enemy lines or why the avatar is running through level after level collecting gold rings as in the Sonic series, [Sonic Team, 1991].

The narrative element can also be realised as an interactive story in which the player experiences and occasionally changes the story. It could be that a choice between good and evil is to be made and the choice changes the layout of the following level. Either way the narrative factor can have a heavy impact on the quality of the game depending on whether the player can conceptually understand and *live* the story. A good narrative can increase the quality of a game but a poor narrative can decrease it.

The award

The player award in a computer game is also a major influence on the quality of a game. The most commonly used award is points which are accumulated throughout the game. The player continues to play the game only to increase his amount of points and possibly to obtain a high-score or to obtain *bragging rights*, [Rouse,

2000], over his friends. In the RPG genre these award-points are shaped as *experience points* and make the avatar better in some way – either by increasing the avatar's skills, fighting abilities or alleviation of some tedious task. A similar point-mapping exists in car games. The points are not collected directly but are specified as the time it takes to complete a racing circuit – the quicker the better.

An award can also be conceptualised as a short movie in the story. These movies are shown between levels and present the player for the next step in the narrative which then hopefully involves the player further in the game.

Awards come in many disguises and range from letting the player see the conceptual art of the game, movies which show how the game was created, additional levels and much more.

Gameplay

In [Rouse, 2000], Richard Rouse presents and explains how game-design is carried out – from the early design phases to the final play-testing of the product. Rouse has the following definition of gameplay:

> "The gameplay is the component of computer games that is found in no other art form: interactivity. A game's gameplay is the degree and nature of the interactivity that the game includes, i.e., how players are able to interact with the game-world and how that game-world reacts to the choices players make.", [Rouse, 2000]

The notion of interactivity is a valid definition of the concept of gameplay because gameplay does not exist in books or movies, but exists in games where players can interact with game objects and other players. Rouse's definition is throughout this thesis used as *the* definition of gameplay.

2.3.1 Immersion as a mark of quality

The concept *immersion* examined in [Manovich, 2001] and in [Taylor, 2002] concerns an immersed feeling with a work – be this a painting, a web site, a piece of music or a computer game. The human computer interaction (HCI) branch discusses when the "feeling" of immersion exists and when and how it is broken. These guidelines can be used to create better interaction possibilities which support a feel of immersion. The HCI communities are furthermore concerned with explaining and understanding the atomic parts of the interface to describe its whole.

The aesthetics branch is on the other hand actively seeking to explain the representations of the work rather than dividing it up in smaller and perhaps more

comprehensible parts. A work is understood and viewed distinctively by the individual person and is therefore not purely understood by analytical methods.

Generally speaking, one can be immersed in a work such as a novel or theatrical performance and the reader or spectator can either objectively follow the narrative or subjectively experience it. Immersion can be described as an act upon a work or described as an involvement *within* a work.

These two aspects of immersion can also be applied to computer games as a player can play a game as an act upon input devices i.e. acting upon the game or the player can be immersed within the game. A feeling of immersion raises the interest a player has in a particular game and thereby raises the quality of the game seen from that player's perspective. As such, a game with a high quality should at least not repel the feeling of immersion from the player.

In [Taylor, 2002], Laurie Taylor discusses perspectives and point-of-views in computer games. By means of immersion terminology she discusses gaming experiences for various representations of perspective. She argues that perspective and point-of-view concepts are fundamental issues in computer game immersion and discusses primarily third- and first-person perspectives. Taylor's conclusion is that the third-person view entails immersion in a more situated sense than the first-person counterpart due to peripheral information available to the player in this view.

The notion of immersion should be seen as a general concept in which many sub-concepts such as a narrative or a game's gameplay exist. A good narrative in a game or a consistent game world are seen as contributing factors to the more general concept of immersion. If a game includes good gameplay this also increases the level of immersion a player has within the game. Immersion can be seen as a general impression a game has on a player.

Consistency

Consistency is a major part of Taylor's thesis. She uses the concept of *game space consistency* meaning that the game world needs to uphold some form of consistency for the player to be immersed within it. Consistency is thus a way to obtain immersion within a computer game.

Taylor argues that consistency within a game need not be visually presented, but lack of consistency in game representations can force player immersion breakdowns. Several events can force breakdowns in a computer game and an inconsistency can often result in total breakdown of immersion a player has.

If this feeling is broken the game is no longer seen as an interactive narrative or an intriguing fantasy adventure, but is instead reduced to chunks of graphics, music tracks and transparent character behaviours. Consistency should thus be seen as a way to obtain immersion, but also a way to rate the amount of immersion

achieved. It is also Taylor's point that inconsistency deprives a computer game of the feeling of immersion.

Inconsistencies in computer game graphics are prime examples of where the feeling of immersion can fade. Low polygon count or low resolution on game object models, homogeneous texture mappings and identical objects are all graphically immersion repelling aspects. If the game is in low resolution the player will find it hard to believe that the depicted graphics is real, but this does not mean that the game is of low quality. The player can still be immersed in a low resolution game – it is then other factors such as a good narrative which increase the immersion.

Sound and music is a well known method in films and in theatrical performances for increasing tension or otherwise placing the audience in a specific mood. Sounds and music in computer games can immerse the player, but sounds and music can also be immersion-breaking in a computer game context. Monotonous background music or playing music in a loop can be a source of irritation for the player. But can actually help the player grasp the theme of the game, since the music's uniformity gets the player in the "mood" the game imposes. Singular and monotone digital voices for the game's characters are not realistic and forces the player to mentally withdraw himself from the game.

The game's level-design can also be an important immersion support, but can unfortunately also be the reason why a player cannot immerse himself in the game. If each level has a uniform layout or game objects placed in one location are similar or perhaps identical to game objects placed in another location, the player's immersion will fade. Uniformity in the layout of the game world where only one way through the level exists is known as *goading* or *railroading*. The element of uniformity often helps novice players to complete the level instead of wandering hopelessly around for hours. But this can be a tedious experience for advanced players. Badly designed levels can be disastrous for the player's immersion since each level look alike. Clever level-design can create a realistic game world model for the player and can occasionally be the sole reason for the player's immersion in the game.

Consistent characters in a game world is also an important aspect in a player's ability to immerse himself in a computer game. Even if the game world is realistic the characters' behaviour can have a great influence on how the game world is experienced and understood by the player. The behaviour of the characters should thus be consistent with the game world. This form of consistency is genre dependent since an adventure NPC is very different from an FPS character, because these have different goals. The adventure NPC gives puzzles to the player and unfolds a story while the FPS character opposes and combats the player. Consistency should be seen as the characters' ability to appear as part of the game world. If this appearance is not explicitly present the player might misinterpret the actions

of a character which often leads to a breakdown of immersion.

A consistency measure of an FPS character might be how much challenge the player receives from this character. A character who fights poorly would not be considered consistent, since the player expects challenge from a character assuming the role of a tough veteran marine. A consistent FPS character should at least be able to show signs of combat experience such as running for cover when shot at. How the behaviour of RTS characters affect consistency in RTS games is discussed further in section 3.4.4 on page 46.

The goal of a game's graphics, animations and sounds are to portray an illusion of a consistent game world to the player. As an example let us assume that the game world contains some kind of forest as known from the real world. The developers wish to assure by all means at their disposal that when a player encounters this forest then it actually is a forest. Animations, sounds and overall "look and feel" must convince the player that it is indeed a collection of trees, which constitute a forest. If this forest contains characters i.e. wild life like rabbits animations and sounds are not always enough to accurately portray a rabbit. The behaviour now becomes an important factor in the "look and feel" of the illusionary rabbit so what looks like a rabbit actually *is* a rabbit. How this illusion is obtained is irrelevant. A simple random replay of animation which show a rabbit jumping around might be enough. If the player is required to interact with the rabbit in some way more advanced methods might be required.

Inconsistencies in computer games are occasionally used to present an authentic game world to the player. The concept *lens flare* is a phenomenon caused by the scattering and internal reflection and refraction of bright light in the optical components of complex lens systems, [Wikipedia, 2004]. Even though a lens flare is not experienced when walking around in the park almost all FPS games incorporate this feature to increase visual photo-realism, because this phenomenon occurs in movies. Ironically, this inconsistency often creates a more consistent game experience.

2.3.2 Consistent character behaviour

Now that the notions of narrative, award, gameplay, immersion and consistency have been introduced the discussion how consistent behaviour can improve the quality of a game has a solid foundation. We will present both academical and commercial views on consistent behaviour in games where combat is an essential element of the gameplay. This focus was chosen because we investigate RTS games which has combat as an important element of gameplay.

Academical vs. commercial approaches Lars Lidén, a developer of Half-Life, describes in [Lidén, 2004] that consistent characters should intentionally react such that the player feels superior. Lidén believes that the player is supposed to win since he is the one to be entertained. Lidén suggests that letting the player win should be an intentional act, obtained with Game AI methods, rather than bad design or bad control mechanisms.

Lidén furthermore suggests several approaches to obtain consistent characters even if the opposing characters are doomed to lose in the end. Methods such as a bad aiming ability raise the tension for the player but not if the characters miss every time. An opposing character should at least miss the first time to warn the player of its whereabouts instead of killing the player and leave him utterly confused.

The peculiarity of Lidén's approach is that the goal he tries to obtain can be seen as a way to assist the player in playing the game, but without the player knowing. If the player notices this help the character ceases to appear consistent and a breakdown in immersion is unavoidable.

In [Buro, 2004] and [Buro and Furtak, 2004] Buro and Furtak describes Game AI in an RTS context and advocates strong participants. Buro focuses on giving the player the greatest challenge by means of strong and near-optimal participants. The approach can give the best players a hard challenge and be downgraded to adjust difficulty for novice players. In contrast, it is very hard to create a challenge for advanced players if the design is aimed towards giving only novice players an interesting opponent. Buro states:

> "The main goal behind the AI research [...] is not to increase the entertainment value of RTS games, but rather to create the strongest RTS game AI possible. The former goal is pursued by the commercial games industry, whereas the latter tries to push the cognitive abilities of machines to new levels. Note, however, that increased playing strength can be converted into higher entertainment value by adapting to the player's performance level to keep games challenging.", [Buro, 2004].

Buro furthermore presents and discusses how and why participants in RTS games should incorporate learning mechanisms, planning under uncertainty and spatial and temporal reasoning. Buro is well aware that all of these features cannot be tackled easily, but argues for a gradual improvement of RTS participants since small steps can have enormous effects on the further improvement of these. Buro continues:

> "[...] because current AI systems do not reach human planning, learning, and reasoning levels, machines can at least aid them playing

> RTS games. [...] It should be possible to create AI modules to handle those local combat situations much more efficiently than humans who have to point and click to give them orders. What makes this hybrid AI approach attractive is that now human players can choose their favourite AI plugins. Moreover, players then can concentrate on high-level decisions without being forced to compete with the World's fastest mouse virtuosos in terms of speed.", [Buro, 2004].

In the same paper Buro also mentions that the lack of an academical RTS Game AI competition is a contributing factor to the lack of academical interest in this area. He hopes that his ORTS platform, [Buro, 2002], helps remedy this problem.

Marco van de Wijdeven has, like Lidén, a background in the commercial computer game industry and captures the commercial computer game industry's view of optimal characters' goals nicely:

> "Stated concisely, the challenge is: Creating an agent that can provide a suitable challenge no matter who or what is opposing it." [van de Wijdeven, 2002]

Van de Wijdeven's view does not contradict Lidén's view. If an optimal character is to provide a suitable challenge to novice players modifications could include Lidéns ideas such as intentional misses.

Although the commercial game industry's focus is on increasing the quality of games, the game industry is not exclusively interested in optimising the challenges provided by the characters. The main goal is to create a behaviour which increases the quality of the game. The goal of academic AI research is on the other hand to create near-optimal characters, because such methods may push the cognitive abilities of machines further. Both approaches pursue and is capable of providing consistent character behaviour.

Entertainment value vs. optimality The essential point in this discussion is whether a character should behave reasonable while being fun to play against, as Lidén believes, or the character should be optimised to give the player the hardest challenge possible, as Buro believes.

Either way, the approach taken to increase the performance of the characters should not make them inconsistent. Depending on the actual game the characters should behave in a consistent way. How the consistent behaviour is achieved depends very much on the genre of the game and of course the actual game.

Optimal characters are needed in the domain of RTS games according to Buro, [Buro and Furtak, 2004]. The primary reason is that research results found in the field of RTS Game AI can possibly be extended to real world problems. Issues

such as tactical decision-making can have an enormous effect on army protocols as the digital method can assist army leaders in real tactical battles.

The measure of the character's level of consistency and how fun this character is to oppose is genre specific, but some general issues can be elaborated upon. The optimality of the character is one such issue. The character's optimality regards how difficult the character is to defeat. If an optimal behaviour approach is taken the result is scalable because the approach can be downgraded to ease difficulty for novice players. A near-optimal behaviour is in some genres seen as the only way to obtain character consistency whereas a non-optimal approach would have created an inconsistent behaviour. We believe that the subproblem of small scale combat (SSC) in the area of RTS, see section 3.4.2 on page 45, needs to include a near-optimal solution if the resulting behaviour can be seen as consistent. Such a behaviour helps to portray the illusion that the RTS units are soldiers.

An optimised approach such as a seemingly unbeatable opponent can appear to be a waste of the player's time. This is only half the truth since the player probably already has won against the second-highest difficulty level meaning that the player searches for more challenge from the opponent.

An optimal opponent cannot in some respect decrease a game's quality if a choice of difficulty level exists. On the other hand can an easily fought opponent decrease a game's quality and force the player to quit the game prematurely if this opponent is too easily beaten. An easily fought opponent can however be hours of fun for novice players which first have to learn the game and its controls.

Like everything else in the world, there exists a compromise between creating an easily (non-optimal) or hard (near-optimal) fought opponent. The compromise involves a mix of the targeted audience's flavour, the game world and what objective the gameplay has.

Chapter 3

Game AI problems

"If I only had a brain"
- The Scarecrow

Most commercial computer games published today contain many different modules, which must work together. The physics simulation, graphics pipeline, sound subsystem and Game AI subsystem must all cooperate in delivering a superb experience to the player. In the rest of this thesis the focus is solely on the Game AI subsystem.

In this chapter we will give an introduction to Game AI problems found in the genres introduced in section 2.2 on page 8. Afterwards, the RTS Game AI problems will be defined in detail. For a reminder of terminology used, please consult section 2.1 on page 6.

Literature in this area is relatively hard to come by. The commercial computer game industry is not overly happy for providing concrete examples and detailed explanations of how they solve the various Game AI problems in their products. They often regard these solutions as trade-secrets and as is common with such secrets they only speak reluctantly or in marketing tongues about their methods. For more information about literature in this area, see appendix A on page 124. The concepts we define in the following sections are based on our personal experiences along with notions from the available literature.

When designing a Game AI subsystem for a computer game some questions must be asked. What problems are to be solved and how advanced need the Game AI methods be to solve these in a timely manner? No player will be immersed in a game if the characters act so slowly that the game grinds to a halt. The Game AI algorithms must therefore be able to run fast enough while providing the characters with consistent behaviours.

In Chess a well defined time limit exists. Special designed super computers are used to play near-optimally within the given time limit. In real time games the demands of the subsystem are even higher. The Game AI subsystem must react to

player input with little or no delay. Of course this also means that the reaction may not necessarily be the optimal one. This does not matter as long as the reaction of the characters is consistent.

3.1 Information available to characters

In [Champandard, 2003] Alex J. Champandard presents an approach to creating consistent participants and characters in shooters. Champandard believes that consistent characters ought to be fully embodied and situated in the game world. This implies that in a given situation the character must not have access to any information the player cannot also access in the same situation – such as looking through walls. In other words the character must not cheat. Creating a consistent character within these restrictions is what Champandard deals with in his book.

In [McLean, 2004] Alex McLean describes a method for handling situations in which characters hunt the player. Requirements for his method includes full access to information about the entire game world, the absolute location of the player and the absolute location of all other hunting characters. Using this information the method enables the characters to hunt the player in a consistent way. The method could be seen as cheating, because characters are able to obtain the location of the player without ever having line of sight to him.

Champandard and McLean both strive to create consistent behaviour for the characters, but their methods differ as seen above. Champandard allows the characters to access the same information as the player and nothing else whereas McLean allows the characters to access more information than the player. Their methods both achieve the wanted behaviour.

This is a prime example of the computer game industry's focus on the end result. The computer game industry provide characters with consistent behaviour but are indifferent to how it is achieved. The commercial game industry is not concerned about whether their consistent characters exhibit true intelligence. It is sufficient if the characters behave consistently in the game world. Therefore, the used methods can be seen as tricks with the sole purpose of upholding the illusion of intelligent characters. Michael Buro writes about this method when applied to RTS games:

> "Also, we acknowledge that commercial RTS game AI often cheats to compensate for its lack of sophistication. Tricks of the trade include map revealing and faster resource gathering. The resulting AI systems may outperform human players and may even create challenging encounters, but they do not advance our understanding of how to create intelligent entities.", [Buro, 2004]

Having access to more information than the player is just another trick used by the commercial game industry in their hunt for consistent behaviour in their games.

3.2 Game AI problems in genres

The requirements of the Game AI subsystem in a game differ much from genre to genre. There is no need for advanced algorithms for language understanding, [Callan, 2003, Chap. 18 & 19] and [Nilsson, 1998, Chap. 24], in action based games, since the player does not need to talk to the characters. Likewise there is no need for combat strategies, [Champandard, 2003, Chap. 34 & 44], in a simple 2D platform game. Thus the natural and intuitive way to create the required behaviour is to analyse the game and the game world in detail and thereafter decide which Game AI methods to use.

Many Game AI problems such as pathfinding, [Patel, 2004], and camera positioning, [Carlisle, 2004], are common to many games regardless of genre and will therefore not be covered here. For further information about these problems and a general introduction to common Game AI methods, see [Rabin, 2004b] and [Rabin, 2004c].

The following sections present an overview of how consistent behaviour is achieved in different genres. The list does not cover all Game AI problems but tries to exemplify how this goal is reached.

3.2.1 Adventure

In the adventure game genre the quality of the non-player characters (NPCs) is determined by the NPCs' ability to conform to the story being discovered through the game.

The NPCs in adventure games are in most cases scripted. Since the player is following a static story line there is usually no need for adaptiveness nor flexibility in the NPC behaviour. Whether the NPCs are consistent depends entirely upon the quality of their scripting.

Early adventure games were textual and the only means of input was by typing in the commands on the keyboard in some almost-natural language. In figure 3.1 on the next page a screenshot from Zork I, [Infocom, 1980] is shown. Zork I was one of the earliest adventure games and as illustrated in the screenshot the natural language parser was not optimal and the error messages sometimes only added to the confusion.

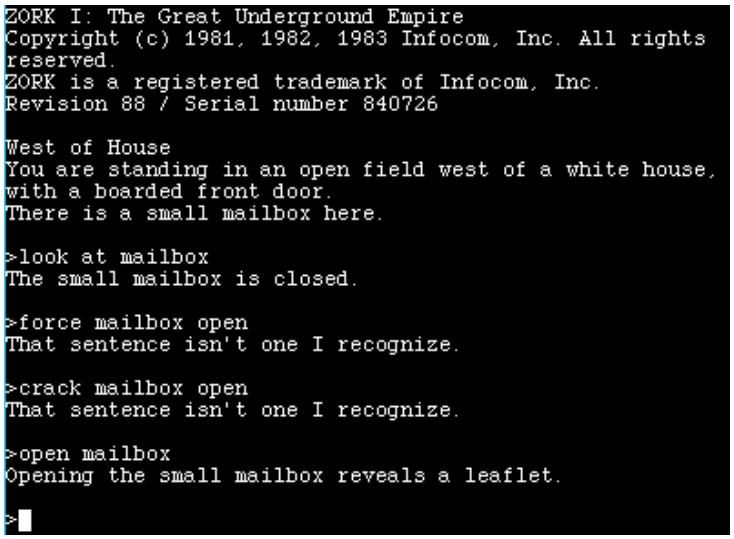

Figure 3.1: Screenshot from Zork I: The Great Underground Empire by Infocom from 1980

3.2.2 Board games

The consistency of a participant in the board game genre depends entirely on how well it plays the game. The better the participant plays the game, the more consistent it is perceived.

To enable a participant to play a board game well, variants of game tree models, see section 5.3 on page 60, are often used.

Many board games are subject to academic interest groups who use these games as testing grounds for machine learning, [Mitchell, 1997], and artificial intelligence algorithms, [Nilsson, 1998], since the problem of winning the actual game is well defined. An example of this is David Fogel's *Blondie24*-algorithm, [Fogel, 2002], which plays Checkers using a neural network method. Blondie24 was a huge success and performed better than 99% of the registered Checkers players at the website where it was the tested.

3.2.3 Strategy games

In the strategy game genre the consistency of the behaviour is far-reaching since the Game AI subsystem acts on many levels. This ranges from simple character behaviour to participant planning tasks. All units available to the player are considered characters and as such their behaviour ought to be consistent. This implies that units representing soldiers should be seen as such by the player.

The consistency of a participant in an RTS game implies that it should show

a high level knowledge of the game world and the units' capabilities. According to Buro, [Buro and Furtak, 2004], a participant should be able to among other things do adversarial real-time planning [Callan, 2003, Chap. 9 & 10], spatial and temporal reasoning, [Nilsson, 1998, Chap. 19] and resource management.

The behaviour of units is generally controlled by simple rules and all high level decisions are deferred to the player. The rules only handle local information and provide local actions. The consistency of the unit solely depends on these rules. The Game AI methods used for controlling participants consist of a collection of overall strategies. An overall strategy dictates the base layout and the construction sequence of units and buildings. These strategies also contain group composition and overall tactics for combat in the game world. Among these strategies a random one is often chosen at the beginning of the game. Again the consistency of the participant depends on the quality of the strategies used.

The performance of some rules for unit handling can be can be seen in the movies section on the enclosed CD, see appendix E on page 154. The movies is recorded from the game Warcraft III, [Blizzard, 2002], which is further discussed in 3.3.4 on page 39. The movies illustrate two situations:

Ranged attack formation In this situation a group of ranged units is seen attacking a sole melee unit. The melee unit is instructed to hold position which means that it will not move. The ranged units spend a lot of time positioning themselves in a circle with a radius of their attack range. A more consistent method would be if the first arriving ranged units to move closer to the melee unit. This would allow more ranged units to stand behind these and begin firing sooner.

Ranged movement This situation depicts a group of ranged units moving across the game world. When moving they walk in line – looking like a snake traversing the game world. When reaching the destination they align themselves in a matrix shape with no wasted time in marching around one another. This is consistent with the way real soldiers align.

3.2.4 Shooters

Creating a consistent character in a shooter requires understanding of the game world in which the character must operate. Examples of consistent behaviour from characters in shooters are target selection [Champandard, 2003, Chap. 20], aiming [Champandard, 2003, Chap. 18] and cooperative behaviour, [Orkin, 2004b] and [Reynolds, 2004].

To create a consistent character, finite state automata, [Kozen, 1997, Chap. 3] and [Fu and Houlette, 2004], are often used since these allow easy handling of the state of the character.

For an example of a shooter striving to achieve a consistent character behaviour, see section 3.3.2 on page 37.

3.2.5 Role playing games

In the role playing game (RPG) genre the consistency of the characters lies in their ability to portray their part in the story of the game and their ability to provide the player with consistent battles.

As in the adventure game genre the most used method for controlling character behaviour is scripting. Furthermore, many games employ more or less sophisticated methods to let the NPCs respond to the player's actions in various ways. Examples of this include reputation systems where an NPC's knowledge of a player's previous actions determines how the NPC react to the player whenever they meet.

At Runestone, [Runestone, 2005], the developers are creating a MMORPG called Seed where NPCs have a personality and memory of their encounters with other NPCs and players. Each NPC maintains a list of known players and NPCs. With each list entry a value is associated. This value represents how much this NPC likes and trusts the corresponding player or NPC. Over time this value reverts towards neutral, but new encounters refreshes and updates this value. So if a player is not encountered for a period of time the NPC can forget this player.

3.2.6 Simulators

In the genre of simulators the requirements of the Game AI subsystem often shifts from the requirements of the above genres. Above, the focus is generally on creating consistent behaviour for characters. In this genre more emphasis is put on making the game world consistent through well-constructed support routines.

Sport simulators

In the sport simulator genre a consistent participant should be able to play the sport at hand. A prerequisite for creating consistent behaviour is that the game world e.g. the actual game rules and limits of the sport simulator is well defined.

In sports requiring interaction with a ball such as soccer the ability to anticipate the effects of kicking the ball must be present to create consistent football players. To be able to predict the whereabouts of the ball a method called Dead Reckoning, [Wikipedia, 2005c], can be used. This method is applied to the problem of predicting future locations of game objects in [Laraée, 2004].

Figure 3.2: Screenshot from Transport Tycoon Deluxe by MicroProse from 1995

Driving simulators

In the driving simulator genre a consistent participant driving another car needs to be able to keep the vehicle on the road. To portray the illusion of a professional driver the participant must also drive near-optimally.

To portray a professional driver a set of way-points are associated with each track. The way-points represent a near-optimal route through the track allowing participants to steer towards these. Along with these way-points simple reactions to other cars are encoded. An example of this method is presented in [Manslow, 2004].

World simulators

In world simulators there often are no characters for the player to interact with. Usually there is no opponent either. The game creates problems for the player in terms of traffic jams, fires, earthquakes and so on. In a city-simulator the player's choices for placement of fire brigades, parks and community features all influence whether the characters, i.e. inhabitants of the city want to move to the neighbourhood.

In figure 3.2 a screenshot from Transport Tycoon Deluxe, [MicroProse, 1995], is depicted. Transport Tycoon Deluxe places the player in control of a transport company which manages inter-city transportation of goods and inhabitants. It is the player's duty to build infrastructure between the cities, create and manage bus routes, railroads, trains and airlines – and turn a profit from this. Competing participant-controlled transport companies also inhabit the world in Transport Tycoon. The Game AI subsystem focuses on simulating city and industrial development and controlling the participants. In figure 3.2 a railroad built by a participant

is shown. Clearly, that way of constructing a railroad is inconsistent with the game world as it adds to transportation times and has an increased cost for the company.

3.2.7 Action based games

Platform games

In platform games the emphasis of the Game AI subsystem is on allowing the simple characters of the game to respond to the actions of the player. Some characters are oblivious to the whereabouts of the player's avatar and move in predefined patterns. Other characters break their movement pattern if the avatar comes within a predefined range.

Based on personal experience we find that many platform games use simple stateless behaviour. Recently however, platform games have shifted from 2D side-scrolling games to 3D games in third person view. This shift blurs the distinction between platform games and other genres such as the adventure and shooter genres. This allows the platform games to share Game AI methods with the before mentioned genres.

Fighting games

A consistent participant in a fighting game is able to fight efficiently. Here, controlling the avatar often means applying long input combinations with the correct timing thereby performing special moves. For the participant to predict what moves the player might use a pattern matching algorithm might be applied. Depending on the difficulty setting the participant is able to remember which moves the player used and in what order. Based on these data and on knowledge of available moves, the participant can predict which moves the player are trying to perform and counter them effectively. The participant accesses data from the game to read the input sequences performed by the player.

The player is also able to predict what moves the participant might use as the animations of the avatars include plenty of hints to what the participant is currently attempting. As a reference to pattern matching algorithms in fighting games, see [Dalmau, 2003, Chap. 7].

3.3 Examples of applied Game AI

In the following, four concrete examples of how Game AI problems are solved in commercial computer games are presented.

Figure 3.3: Screenshot from Black And White by Lionhead Studios from 2001.

3.3.1 Black & White

The *Black & White* series, [Lionhead Studios, 2001], are simulators where the player plays a God and battles against other Gods for control, in the form of faith, of the land. The player can guide the inhabitants of the game world to perform various tasks such as harvesting resources and expanding towns. The way the player battles against the opposing Gods is by means of a large pet creature. This creature somewhat becomes the main focus of the game. The creature is the manifestation of the player-controlled God in the game world.

The creature is wandering about the land to satisfy desires such as eating, playing and sleeping. The desire from the start is eating since this is a natural instinct. Eating something which is not satisfying the creature's hunger, e.g. a rock will be saved in the creature's memory so it knows that a rock is not particular tasty. Each desire has an associated intensity and the creature is "intelligently" choosing the highest rated desire. The player can punish or reward decisions made by the creature. Either by smacking it around when bad decisions are conducted or petting it when good decisions are made. In this way the player is granted some control over the creature and this process bears some similarity to teaching a dog tricks.

In figure 3.3 the creature in this case a cow is entertaining some inhabitants. Luckily, it seems that the cow is not interested in eating the inhabitants – either it has been properly trained or it is simply not hungry.

The interesting Game AI part of this creature is its inner workings. The crea-

ture has a decision tree using the ID3 algorithm, [Mitchell, 1997, Chap. 3], and feedback weights with which it evaluates potential decisions. The feedback weights are game specific values such that attacking a friendly town is not encouraged. These values are modified by the player's punishment or reward. This way, the creature learns that eating a town inhabitant and receiving a punishment is not a good solution for satisfying the hunger – even though the inhabitant probably is tasty.

For a more exhaustive description of the creature behaviour in Black & White, see [Wexler, 2002].

3.3.2 No One Lives Forever 2

No One Lives Forever 2 (NOLF 2), [Sierra Entertainment, 2005], is an example of a first person shooter (FPS). A lot of work has gone into imbuing the characters with a "understanding of human concepts" to create more consistent character behaviour. In [Orkin, 2004a], real time decision-making using Game AI methods in complex 3D FPS environments is explained from one of the developer's point of view. Instead of having characters which follow simple rules, the developers worked hard to:

> "[...] make the characters *live* in the environment instead of just standing around waiting for the player to show up. Because you have the option of sneaking around, we felt it was critical that the game had a life of its own." [Hubbard, 2002]

Concepts such as game object ownership, concept dependency, responsibility issues and priority of these, expected state and presence of others were all included in NOLF 2 as important elements of character behaviour. The characters have an "understanding" of these concepts and are able to appear more consistent in the game world. Active game objects which emitted information about their interaction possibilities were implemented. In this way, the characters were able to *see* or *hear* game objects. This, along with the above-mentioned concept-understanding, created character consistency according to [Orkin, 2004a].

To further improve the consistency of the characters the developers created simple techniques to improve the characters' ability to cooperate. A *Blackboard Architecture*, [Orkin, 2004b], was constructed. This allowed the characters to share information about various game objects – whether the objects were being used, destroyed or turned on or off. E.g. a character standing in a room could check the immediate game world for information and decide to sit down and use a computer. Another character walking into the room would now be able to notice that the computer was being used and decide upon other actions, e.g. opening a file cabinet.

The methods used in conjunction created consistent character behaviour according to [Orkin, 2004a]. For further information about methods used for character improvements in NOLF 2, please consult [Orkin, 2004a] and [Orkin, 2004b].

3.3.3 Half-Life

Half-Life, [Sierra Entertainment, 1998], is another FPS game. It uses simple behaviour models for its characters. These behaviours are not by themselves very innovative, but additional tricks ensures "character and combat believability" according to [Lidén, 2004]. The method used maintains the illusion that the player is actually fighting tough and well trained soldiers.

In [Lidén, 2004] Lars Lidén describes the ideas behind the Game AI methods used in Half-Life. He introduces the concept of *Kung Fu*-fighting borrowed from the Kung-Fu movie genre, where only a couple of opposing martial arts combatants are battling at any given time. This creates intense and realistic combat situations for the player.

The basic idea is to only allow a few of the characters to attack the player at any given time. When a non-attacking character is ready to attack, an already attacking character is chosen and instructed to run for or duck behind cover. This creates the illusion that the characters are cooperating which they truly are not. In Half-Life, the number of simultaneously attacking characters was set to two. This proved to be sufficient to increase the combat tension. It was found that players confronted by the scenario did not notice that only two characters were attacking at the same time, but were surprisingly overwhelmed by the illusion of collaborative teamwork.

To further increase character consistency within the game world the developers added speech to the characters. This was implemented because some players did not notice or in some cases misinterpreted the characters' actions. When running for cover a character would yell "Cover me" or "Flanking" and this made the player aware of the character's actions and intentions. Regarding this simple method, Lidén states:

> "Such cues can be highly effective and often have the beneficial side effect that players assume intelligence where none exists.", [Lidén, 2004]

This is another example which shows that for the commercial computer game industry the end always justifies the means.

Figure 3.4: Screenshot from Warcraft III by Blizzard from 2002 depicting the cheating of the participant at the highest difficulty level.

3.3.4 Warcraft III

Warcraft III, [Blizzard, 2002], is one of the most popular real time strategy (RTS) games available. The units belong to one of four factions, Orcs, Humans, Undead or Elves. Units have different strengths and weaknesses depending on the faction. This makes a consistent faction-independent behaviour model complex to derive. Instead static methods, e.g. scripted behaviour are used. The capabilities of the Warcraft III characters and participants are described further in [Gustafsson, 2004].

In [AMAI, 2005] a group of developers have been creating an extension called *AMAI* to the Game AI subsystem of Warcraft III. The main attraction for players lies in the large collection of available rules and scripts for participant or character control, called *profiles* and *scripts* in AMAI.

In the original Warcraft III Game AI the highest difficulty level called *insane* awards participants more resources than it actually harvests. Every time a resource is gathered and returned to the base the participant receives the double of the actual harvested resource. Figure 3.4 shows a screenshot which depicts the early game for a participant playing the Human faction. The important part is the two yellow $+20$ symbols positioned to the upper right of the building located in the centre. The symbols represent gold deposits made by the two small blue men – these are harvester characters. Even though 20 gold is deposited for each trip to the gold mine only 10 gold is removed from the mine. The amount of gold in the mine

Figure 3.5: Screenshot from Warcraft III by Blizzard from 2002 showing a typical game start for the Orcs.

was from the beginning 12500 gold. This behaviour is also shown in a movie on the enclosed cd, see appendix E on page 154. In this way the illusion that the participant plays *insanely* well is maintained even if it actually is not. However, if the player discovers the cheat the consistency of the participant diminishes.

3.4 Real time strategy (RTS) AI

A real time strategy game is as mentioned in section 2.2.3 on page 10 a war simulator wherein players with different allegiances battle. The main Game AI problems in the RTS domain were briefly presented in section 3.2.3 on page 31. In this section we will elaborate upon the essential RTS game concepts and problems in greater detail. A complementary description of real time strategy games can be found in [Wikipedia, 2005e]. The following sections present RTS games in detail. There are many ways of playing an RTS game, but here the focus is on multiplayer battles in which several players battle on equal terms. This way of playing will be referred to as *skirmish*.

3.4.1 RTS games in detail

An RTS skirmish starts with each player choosing an allegiance and agreeing on a scenario to play. A scenario contains a game world called a *map* and a start sit-

uation. The latter includes starting buildings and characters, called *units*, possible starting resources for each player and a background story which can have more or less impact on the actual scenario.

Figure 3.5 on the previous page shows a typical starting situation in a Warcraft III skirmish. Each player starts with five harvester units and a building in which resources are stored.

Map

An RTS map contains deposits of resources such as gold mines and trees. Reactive game objects called *environmental features* such as water, mountains and rocks also exists on the map. A map is often an $n \times m$ matrix of tiles where each tile or matrix-entry can contain one land-unit if the tile allows it. A land-unit can be positioned on grass or sand-tiles, but not on mountain or water-tiles. Airborne units can be placed on any tile since they fly over environmental features and in particular a land-unit and an air-unit can both occupy the same tile.

Information about the environmental features is available to all players before the game starts, but the starting positions of each opponent is not. The whereabouts of enemy units and buildings is hidden from the player until these come within view of a unit. This is represented by *fog of war* which hides enemy units and buildings in the entire map except for areas within view of the controlled units.

Base

Each player must build, control and maintain a base. A base is a collection of buildings and structures. Each building takes up a number of tiles on the map and cannot be repositioned once built. The buildings cost a number of resource points and give the controlling player various advantages.

Each building has some effect which is available when the building has been constructed or when resources are spent on the particular effect. Normally, an effect is gained through the construction of one building, not many. E.g. a *barrack* gives the player the possibility of creating combat units and a player who builds a *blacksmith* gets the possibility of upgrading his units' damage capability and armour rating. Some buildings are also needed to support the existence of units. These buildings create food or shelter for the units and each gives the player the possibility to support additional units. Some buildings act as defensive structures and inflict damage to enemy units within a specified range. Some constructions give the player the possibility of constructing new types of buildings which again gives the player additional advantages.

In figure 3.6 on the following page the interface for the Orc blacksmith in Warcraft 3, called a *War Mill*, is shown. The cursor hovers above an upgrade for

Figure 3.6: Screenshot from Warcraft III by Blizzard from 2002 showing the options available at the Orc War Mill.

melee weapons which provides the portrayed Orc units with increased damage capability. The only way to upgrade damage capabilities of Orc units is through the War Mill.

Units

The units in a skirmish are either resource gatherers, combat units or specialised units. In general the more powerful the unit, the more resources is required to build it. Each unit has several attributes such as a health-score, movement-speed and perhaps special abilities. It is these attributes which decide the unit's overall strength and resource cost.

Resource gatherers gather resources around the map and transport these back to the base allowing the player to spend the resources. Resource gatherers are very ineffective in combat and therefore need protection against enemy combat units.

Combat units can either be of melee-type, ranged-type or siege-type. The melee-type combat unit can only attack land-units and only if the enemy unit stands on an adjacent tile. A ranged unit can attack both flying and land units and from a distance. Ranged units can sustain less damage than their melee counterparts and is often protected by the latter. When attacking, siege-type units inflict damage to all land-units including the friendly ones positioned in the targeted area. Siege-type units are extremely effective against buildings, since buildings generally take up several tiles. Siege units often have a minimum range so letting enemy combat units come close will diminish their capabilities greatly.

Specialised units are units which do not fall into either the gatherer- or combat-category. These can be invisible fast moving units which cannot deal damage in

combat, but can spy on enemy troop logistics and base constructions. Another example is transport units which can carry other units quickly to various destinations. Destinations could be the opposing player's base for a surprise attack or the player's own base for repairs or healing.

Specialised combat units also exists. These can cast magic across the map or incapacitate enemy units for a short duration. Specialised units often have a high resource cost which makes them inapplicable in all but specialised tactics and strategic plans.

Combat

Combat in an RTS skirmish is resolved in real time. The notion of real time is achieved through a sub-division of a wall clock second into several micro-turns called *game cycles*. Thus, all players' units move and attack simultaneously. Units can be divided into groups and be given commands as a group. This is primarily done to ease the player of giving commands to each unit in the group. A move-command can be given to a group and every unit in that group will then move towards the destination.

When a player or participant orders units around the map there is a limited list of orders available. A list of common orders available to single units and groups of units is given below:

Move orders a move to a destination along the shortest path known.

Attack orders a move within attack range of another unit and keep attacking that unit.

Stop orders units to stop whatever order they were doing.

Stand Ground does as the stop order described above. Furthermore, it ensures that the units remain stationary.

Guard orders the units to follow and defend a given target be this another unit, building or environmental feature.

Attack-move orders a move to a given destination and to attack all enemy units encountered along the path.

Patrol orders an continuous attack-move between a list of destinations.

The attack-move order deserves special mention. It allows a player to select a number of units and order them to attack any enemy units they encounter on their way to a destination. The possibility of giving an attack-move order was added to RTS games, because too much time was spent selecting units one at a time and telling them which enemy unit to attack.

Default behaviour for units

How the units carry out their assigned orders is determined by the behaviour associated with those orders. This is referred to as *default behaviour*. Default behaviour for a unit is also needed when no orders are assigned to that unit. E.g. if a player has ordered a unit to move to a certain destination and the unit completes that order how should the unit behave now that no orders are given?

End condition

An RTS skirmish is lost when the player's base is destroyed and all the player's units are killed. Likewise, the game is won when all opposing players' units are killed and their buildings destroyed. However, scenario specific goals may also be the cause of defeat or victory.

3.4.2 Game AI problems in RTS games

This section introduces some of the most important Game AI problems within an RTS skirmish and thus which requirements the Game AI subsystem should meet to provide consistent behaviour.

Several problems in the RTS Game AI domain are already solved by traditional methods such as the problem of finding the shortest path in a game world by using Dijkstra's algorithm, [Goodrich and Tamassia, 1998, Chap. 10.1] and [Main, 1999, Chap. 14.4]. These problems will not be covered here even though their solutions are part of a consistent character behaviour.

Base layout

Base layout concerns the problem of how to construct buildings such that the positioning allows buildings to fulfil their purpose in the game. An example could be a building which must be defended. It could be positioned at the far end of the base such that enemy units must travel further to reach it. Defensive buildings should maximise protection of the base. Building positioning should also allow for quick passage of friendly units. Positioning walls introduces further considerations. An attempt at solving these problems using graph-algorithms is described in [Grimani, 2004].

Resource management

The problem of resource management involves which and when resources should be gathered and spent. The Game AI method responsible for resource management needs information about the amount of resources available on the map, an estimate

of how much can be gathered given a time frame and heuristics estimating the military and technological needs of the participant. Given this information the Game AI method provides a handling of resources.

Map analysis

The map is analysed to find places of interest such as strategic key locations. Locations with many resources, good positions for bases and positions which are suited for ambushes are all of interest to a participant. This Game AI method is also responsible for studying the map for static information such as where mountains and water-deposits are located. Dynamic information such as enemy unit movement and placement of opposing bases must be evaluated in real time, since this information changes continuously. For a presentation of a layered method for handling map analysis, see [Kent, 2004].

High level tactics

When and how to attack the enemy units, how to ambush the enemy units and estimating what offensive capabilities are needed are all parts of high level tactics. This Game AI method should provide solutions to the positioning of several groups of units. Factors to consider are the units' defensive and offensive capabilities so key locations can be protected or enemy groups can be countered. As reference to Game AI methods for high level tactics in different domains, see [Champandard, 2003, Chap. 44 & 45] and [Ramsey, 2004].

Small scale combat (SSC)

An SSC situation consists of several units from more than one faction fighting each other. The problem of positioning the units is called *unit placement* while the problem of maximising damage to enemy units is called *target selection*. The problem of *group decisions* is of more general nature such as when to flee an SSC situation.

Since these concepts are the primary focus in this thesis these are further elaborated upon in the following section.

3.4.3 Subproblems in small scale combat (SSC)

We have identified three sub-problems which we believe to be essential in solving the SSC problem. These are described in the following along with concrete examples of problem instances.

Target selection

Target selection is the problem of how *n* units should target *m* enemy units given a map and the status and capabilities of all units.

E.g. should the ranged units concentrate on firing on a single enemy unit or is it better to let every ranged unit fire at different enemy units? The problem depends on the enemy units' health, the amount of damage which can be inflicted and the types of the enemy units. Positioning of the units is also important, since some of the ranged units might have to move before attacking.

Unit placement

The problem of unit placement is how to position *n* units given a map and positions of *m* enemy units.

E.g. should the ranged units be positioned close to the enemy units as sacrifice thus giving the enemy something to attack while the melee units attack from the side? Is it perhaps better to have formations in which units follow a given pattern such as ranged units move to the left of the enemy units and melee units move to the right?

The quality of the solution depends very much on the target selection method used and thus solving this problem could decrease or improve the solution to the target selection problem.

Group decisions

The problem of group decisions involves how the units as a group behaves.

E.g. should the group attack from the left side, the right side or flee from the combat situation? The quality of the solution depends on how unit placement is performed and how target selection is applied.

3.4.4 Consistent behaviour in SSC situations

As previously mentioned in section 2.3 on page 20 consistent character behaviour means that the characters appear as a part of the game world. In particular, this implies that the RTS units need to follow the rules and theme of the game world. For units to portray consistency the units must be seen as seasoned combatants. In an SSC situation this means that the units involved must cooperate as real soldiers would, i.e. fight efficiently. To obtain this illusion of soldier behaviour a near-optimal solution to handling the SSC situation is needed.

As Buro and Furtak discusses in [Buro and Furtak, 2004] we are also convinced that RTS games are won on high level decisions. Therefore, support routines must be able to assist the player in the handling of units. We believe that by

handling SSC situations as optimally as possible the behaviour of units in an SSC situation becomes consistent, because soldiers in a game world know how to fight given the conditions and limits of this world.

Optimal behaviour in SSC situations is to minimise sustained damage while damage dealt is maximised. This is a balancing act, since maximising the dealt damage requires that the units engage in combat and thereby receive damage. Optimality thus implies that the chance of winning the SSC situation, i.e. killing each enemy unit, is maximised.

As handling of SSC situations can be implemented as a support routine available to players and participants the gameplay (see section 2.3 on page 20) of the RTS game shifts from manual handling of SSC situations to high-level decisions. As a support routine the optimality of the solution to SSC does not decrease the entertainment value of an RTS game, as discussed in section 2.3.2 on page 24, but the gameplay merely shifts focus.

Chapter 4

The Wargus platform

"It's not easy being green"
- Warcraft Grunt

To measure the performance of our solution to small scale combat (SSC) an extensible and commercially comparable RTS game was needed. We chose to use the game engine known as *Stratagus*, [Stratagus, 2004], due to its maturity and reliability. The Stratagus engine delivers a platform in which developers can implement an actual RTS game. Several games exists for Stratagus and we chose the *Wargus* game for our solution.

In the following the story of Wargus will be introduced. Afterwards, the rules and game world of Wargus are presented.

4.1 The Wargus game world

Wargus implements the complete Warcraft II game which means that Wargus fully adopts the story, setting and game world of Warcraft II.

The Warcraft series are set in the fantasy world of Azeroth where mythical creatures like elves, humans, dwarves and orcs roam. The world flows with magic allowing all races to bend reality at their will. Unfortunately, the handling of these powers got out of hand and attracted the attention of powerful demons intending on destroying all life. The demons managed to corrupt the orcs. The orcs now bloodthirsty and crazed had only one purpose in life: to quench their thirst for battle. The demons let the orcish horde run rampant on Azeroth. This is the setting for Warcraft I.

When Warcraft II starts, the orcish horde has almost wiped out all of the human race. A handful surviving humans were able to rally the dwarves and elves to their cause. From this point on it is the player's responsibility to further discover the tale of Azeroth. For the complete tale of Azeroth as told through the games Warcraft I-III, readers are referred to [Blizzard, 2004].

4.1.1 Scenarios

A typical Wargus game called a *scenario* starts by explaining what the player must do and why. Common main goals in Wargus scenarios include the following:

- **Destroy the opponent**: All players start out almost equally regarding resources and must each build and maintain bases and construct armies. This is a skirmish scenario.
- **Survive for *x* minutes**: Resources are sparse and units are limited. The opponent is considerably stronger and attacks continually. The number of minutes usually ranges from ten to thirty minutes.
- **Destroy or protect a specific asset**: The player must either destroy or protect some asset in the map. Typically this asset is a unit or building. In the case of a unit it usually represents an important NPC in the story.
- **Adventure mode**: The player only has a fixed amount of units usually a handful at his disposal and the map is designed for exploring and adventuring.

Wargus also contains a campaign mode where the player follows a predefined set of scenarios. Together these scenarios constitute a contribution to the tale of Azeroth. In this thesis we solely focus on the skirmish scenarios.

4.1.2 Map

A map in Wargus is a discreet $n \times m$ matrix world where field $(i, j), i \in [0, n-1], j \in [0, m-1]$ can contain one land unit or an environmental feature such as a rock, a tree or some water. Fields with environmental features are always untraversable by land units and in some cases flying units are also unable to traverse the field in question.

4.1.3 Units

The units in Wargus have different abilities and strengths. We have chosen a subset of the many available units in Wargus for inclusion in our solution to the SSC problems. The units in Wargus fall into the three categories, harvesters, combat units and specialised units, as described in section 3.4.1 on page 40. Each type of unit has different actions available. In this thesis only the basic units of type *melee* and type *ranged* will be examined thus ignoring the siege-type units, harvester units, flying units and advanced melee and ranged units.

In Wargus the basic human affiliated melee and ranged units are called *footmen* and *archers* respectively while the orcish counterparts are called *grunts* and *axethrowers*. A footman and a grunt have identical attributes just as archers are identical to axethrowers – only the associated graphics differ.

It should be mentioned that other units in Wargus have other attributes – *Knights* an upgraded footman is faster than footmen and *Elven rangers* an upgraded archer can shoot further than archers. These types are not considered in the method presented.

Unit actions

A unit in Wargus has the possibility of moving in eight different directions, as well as standing still. Figure 4.1 on the next page shows a lone grunt, the *G* depicted, standing in an empty section of a Wargus map. The movement possibilities of this unit is showed. Since no obstacles nor other units are adjacent, the unit has eight possible movement actions.

When a melee unit attacks the targeted enemy unit must be standing in one of the eight adjacent fields, because the melee unit's attack-range equals one field. A ranged unit can attack enemy units from a distance. In Wargus this distance for archers and axethrowers is four fields from the unit's position. Ranged units can also shoot over some environmental features making the local map information extremely important for this type of unit.

As mentioned in section 3.4.1 on page 40 RTS games subdivide a wall time second into several micro-turns called game cycles. Wargus divides a second into 30 game cycles. The time needed to complete an action, the action's time-length, differs in most RTS games and this is also the case in Wargus. A standstill-action takes one game cycle to complete while a movement action takes 16 game cycles. An attack from a footman or grunt takes 26 game cycles while an attack from an archer or axethrower takes 66 game cycles to complete.

It should be noted that damage dealt from attacks is not statically decided. Each time a unit deals damage small fluctuations (randomisations) are incorporated. E.g. melee units deal between two and nine points of damage instead of a constant value.

How actions are handled by the Stratagus engine requires a quick survey. When a unit begins an action an animation-loop depicts the graphics seen on the screen. When a predefined number of game cycles have elapsed in the animation the effect of the action is applied within the engine. This means that the effect is applied before the action is completed. The predefined number is defined in the CCL-scripts on a per-unit type basis.

Figure 4.2 on the next page shows several units about to enter combat. The star depicted in the middle section of the map segment represents an obstacle,

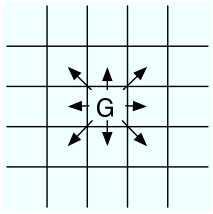

Figure 4.1: The movement possibilities for a lone melee unit

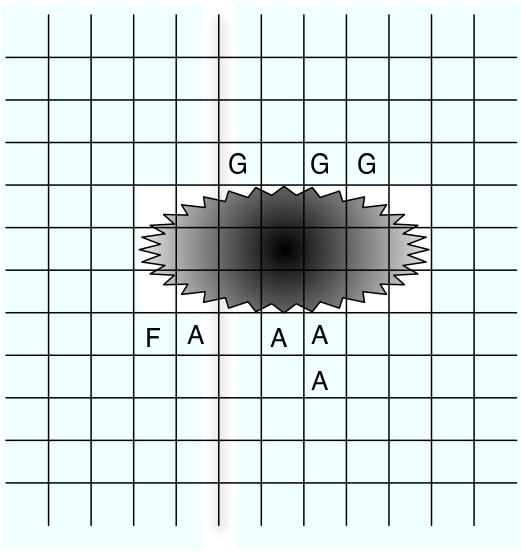

Figure 4.2: A scenario with several units

e.g. a rock, and cannot be traversed. The unit depicted as an *F* is a footman and the *G*s are grunts. Archers are depicted as *A*s. How this actual combat situation ends depends on the tactic used. In this situation the Humans and the Orcs are approximately equally strong.

4.1.4 The built-in participant

The built-in participant in Wargus assumes the role of a human player. The participant has no level of difficulty setting so players cannot adjust the difficulty of this built-in participant. When more challenge is desired the player must choose to fight two or more built-in participants simultaneously.

The built-in participant in Wargus uses the interface provided by the Stratagus engine. The CCL provides an interface for Game AI modification in the form of an order-queue. Each order is executed consecutively by the participant. Examples

of orders are to build a barrack or to train a footman. The scripts filling the order-queue is provided by the developers of Wargus and they contain no variations in the composition of orders. The optimality of the queue thus depends entirely on the order-composition. This method results in similar-looking bases and identical groups of units. The scripts incorporate a trigger mechanism which allows scripts to activate when certain conditions are met. E.g. an attack from an opponent or the completed construction of a building. The activated script then fills the queue with additional orders.

When an SSC situation arises the default behaviour of the units takes over. To avoid a seemingly static group behaviour the Wargus developers use a simple cheat to enable a more consistent behaviour. They increase each participant controlled unit's sight range by two map-fields compared to the player controlled equivalents. When a player encounters participant controlled units these move towards the player's units before they are within sight range. This creates the illusion that the participant units already were moving around while they truly were not.

Chapter 5

Game trees applied to small scale combat

"Under a good general there are no bad soldiers"
- Sun Tzu

The primary goal in this chapter is to present a solution to the problem of creating consistent real time strategy (RTS) unit behaviour in small scale combat (SSC) situations, as examined in section 3.4.3 on page 45. The presented method handles SSC situations near-optimally by means of a timestamped rule-induced game tree.

This chapter starts with an overview of methods for solving SSC situations. Secondly, an introduction to game trees and a definition of timestamped game trees are presented. Then three sections are dedicated to the problems which arise when using game trees. Finally, a discussion of how to measure the optimality of a method for solving SSC is presented.

5.1 Methods for solving SSC

We have investigated the game mechanics of Wargus and no obvious behaviour model resulting in a near-optimal solution to SSC could be identified. Therefore, more complex methods for controlling characters are needed. In this section we will describe the advantages and disadvantages of these methods.

5.1.1 Rule based methods

A rule based method uses rules to govern unit behaviour. The rules normally have access to map information and unit states. Based on this information actions for units are derived. The rules often consist of nested `if - then - else` statements

which combined with information of unit states is equivalent to Finite State Automata, [Kozen, 1997, Chap. 3].

To apply a rule based method to SSC the wanted behaviour is identified and then encoded in the rules. If the wanted behaviour for a unit is to attack the nearest enemy unit a rule implementing this behaviour have to select the nearest enemy unit and attack it.

An advantage of a rule based method is that when the wanted behaviour is identified the rules are quickly both implemented and executed. Additionally, because the rules have full control over the units a wide range of behaviours are achievable.

A disadvantage of the rule based method is that when units encounter a situation not anticipated by the designers of the rules the behaviour is undefined. The optimality of the rules thus depends on the designers' understanding of the problem domain.

5.1.2 Evolutionary based methods

An evolutionary based method uses an Evolutionary Algorithm (EA), [Mitchell, 1997, Chap. 9], [Nilsson, 1998, Chap. 4], [Callan, 2003, Chap. 17] and [Michalewicz and Fogel, 2004], to evolve character behaviour. An EA is a randomised, parallel, hill-climbing algorithm which optimises a predefined fitness function. Deriving a fitness function to describe all nuances of SSC is challenging since SSC contains many elements. Elements such as positioning, damage dealt and received, enemy units' actions and current unit states all influence the optimality of a solution.

To apply an EA to SSC the behaviours of the units have to be parameterised in some way. E.g. if rules describe the behaviour of units an EA could evolve compositions of rules and at the same time also tune the parameters in the rules.

An advantage of using an EA is its ability to locate near-optimal solutions given enough computation time. Additionally, no prior knowledge of how an optimal solution might look is needed to locate a near-optimal solution.

The EA method also has disadvantages. No guarantees for locating a near-optimal solution within a certain timeframe can be given. This means that an EA might run for extensive periods of time and only locate poor solutions. The fitness function raises another problem, because it must rate solutions to the problem domain accurately. Otherwise, good solutions from the EA might result in poor solutions in the actual problem domain. Lastly, an EA algorithm requires extensive tuning of selection, mutation and crossover-methods.

An example of an applied EA method is Per Jefsen's evolutionary method, [Jefsen, 2000], to develop solutions for SSC. In order to do so he implemented his own simple RTS game. In this, the units had a very limited range of actions

and attributes. He used a genetic programming algorithm to develop finite state automatas to control the units. Jefsen concluded that the results were promising. But the game was much too simple and he suggested that incorporating additional actions and types of units would provide more interesting results.

5.1.3 Game tree based methods

A game tree based method models SSC situations by calculating future game situations. This is done as in Chess by examining all possible actions from the current state and estimating their effect.

An advantage is that game tree allows for near-optimal solutions which in the problem of SSC gives consistent character behaviour. Unlike the rule based method this method has the advantage of estimating the outcome of all possible actions before actually choosing an action.

Applying game trees to SSC situations raises three fundamental issues. The size of the game tree is one. The number of game states increases exponentially. Removal of unwanted game states somewhat alleviates this problem. Secondly, the representation of a game state in the tree must be decided upon. Thirdly, it is obvious that if the tree can be fully built the only remaining challenge is to select the optimal path through the tree. Stated concisely: Once the tree is built the problem is to choose the path where the chance of winning is maximised while the opponent's chance is minimised.

We have chosen to investigate the performance and applicability of game trees to handle SSC situations. In the following sections game trees and the attached issues will be examined in detail.

5.2 Game trees

A game tree is a game modelling method used primarily in board games such as Chess and in the Chinese game of Go. A game tree is a data-structure which models a course of a game by looking at possible moves each player can perform each turn. An action corresponds to a change in game state. E.g. an action in Chess is a valid relocation of exactly one of the player's Chess-pieces. The nodes in the game tree represent states in the game and encapsulate relevant game-dependent information. E.g. in Chess each node contains information about where each Chess-piece is located. Whose turn it is to move a piece is implicitly encoded in the tree since each level in the tree directly corresponds to each player's turn. Therefore, this information is not encoded in the nodes.

Formally speaking a game tree is a non-balanced tree where each node n is equivalent to a state in the game. The edges represent transitions between game

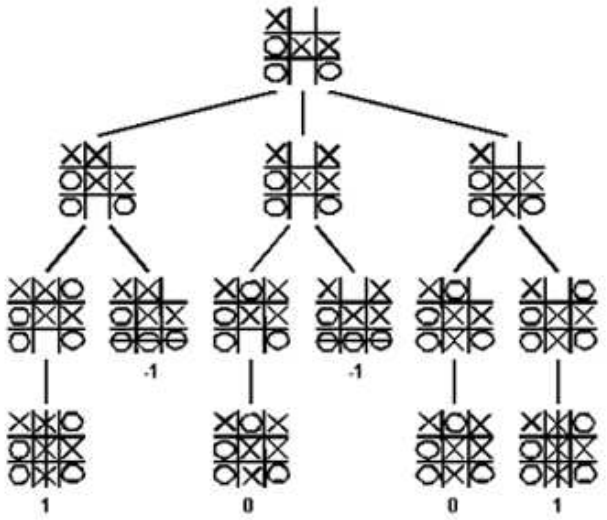

Figure 5.1: An example of a game tree in the game of *Tic Tac Toe*

states. Node n's fanout corresponds to the number of possible moves from the state n represents. The root of the tree r represents the start-state and the nodes in the first level, i.e. r's children represent the states reachable with a single move. The path from the root r to a node n is thus the series of moves which leads to n from r.

Game trees have been used effectively in turn-based board games where players take turn playing one piece at a time. Turn-based games are generally well suited for a game tree method, because each level directly corresponds to a single player's turn to act.

An example game tree used in the game of *Tic Tac Toe* is depicted in figure 5.1. The root of the tree represents the current state of the game and it is the player who places *X*s on the board who is next. Since this player only has three possibilities of placing an *X* the root's fanout is three. Beneath each leaf-node is a number and this number indicate the end-result of the game. In the case of a zero underneath the leaf the game ends in a draw. If the number underneath is an -1 the player who places *X*s has lost. Finally, a win-situation for the *X*-placing player is indicated by an 1. The problem is for the *X*-placing player to choose a move which minimises the chance of losing while maximising the chance of winning. The optimal way to play from this specific state is to select the move which results in the right-most sub-tree. This is optimal, because this move either results in a tie, the left leaf, or in a victory, the right leaf. So independent of what move the opponent makes the *X*-placing player cannot lose. The figure is from [Brockington, 2000].

SSC is somewhat similar to ordinary board games. Both types of games use

a discreet matrix for game world representation. The main difference is that units in RTS games can be moved simultaneously in real time. Furthermore, the actions available to the units have different time lengths associated. Therefore, modifications to the traditional game tree method are needed.

5.2.1 Timestamped game trees

In real-time games an ordinary game tree method is inadequate. The action of the units can have different time lengths. E.g. move *a* could take two game cycles to execute and move *b* could take three game cycles. In ordinary turn-based board games all moves have equal time lengths – in Chess, the time needed to move a Queen equals the time needed to move a Pawn. In real time games some modifications to the game tree method must be thought of. What is needed is a way to represent the time length associated with the actions.

A timestamped game tree encodes the basic information from the ordinary game tree, but also encodes the time needed to move from state *i* to state *j*. This is done by timestamping each node such that node n_i's children, $n_{j1}, n_{j2}, \ldots, n_{jk}$, all have higher timestamps than n_i. The root has due to the above definition the lowest timestamp in the tree.

Table 5.1 shows the actions available to the Wargus units. The first column presents the name of the action and the second column shows the associated abbreviation. The third column shows the number of game cycles associated with the action. Finally the damage of attack-actions is shown. Note that a melee attack has the same abbreviation as a ranged attack.

Name	**Abbreviation**	**Game cycles**	**Damage**
Stand-ground	SG	1	–
Move upper left	UL	16	–
Move upper	UP	16	–
Move upper right	UR	16	–
Move left	LE	16	–
Move right	RI	16	–
Move lower left	LL	16	–
Move lower	LO	16	–
Move lower right	LR	16	–
Melee attack from *x* against *y*	*x* v *y*	26	2 to 9
Ranged attack from *x* against *y*	*x* v *y*	66	3 to 9

Table 5.1: Actions available to Wargus units

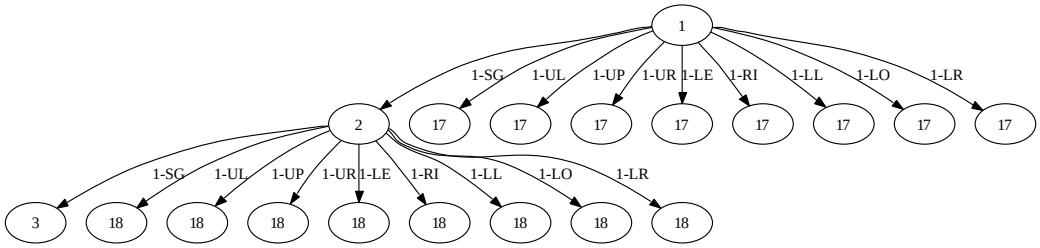

Figure 5.2: A sample timestamped game tree

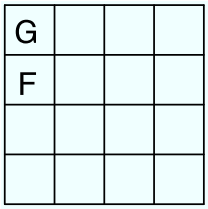

Figure 5.3: The situation, which figure 5.4 and figure 5.5 model

The tree in figure 5.2 is a timestamped game tree built for a single orc grunt with all movement actions available. This is similar to the situation in figure 4.1 on page 51. The tree is built with a lookahead of one game cycle where the actions available in the next game cycle also are included. The numbers inside the nodes are the timestamps. The text on the transitions tells which units are performing actions in this timestamp and gives an abbreviation of the actions in concern. The text always binds to the left transition. The leftmost child of the root models an SG-action for unit 1 whereas the rightmost child models an LR-action also for unit 1. It can be seen from the figure that the grunt has eight movement possibilities and one stand-ground action available in game cycle 1.

Figure 5.3 shows a situation where two opposing units, a grunt and a footman, are adjacent. A sample game tree, where unit 1 (the grunt) and unit 2 (the foot-

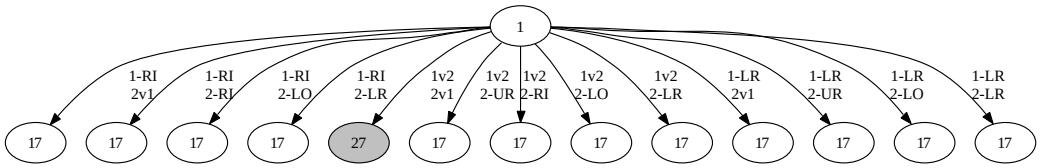

Figure 5.4: Two opposing units ready in the same timestamp

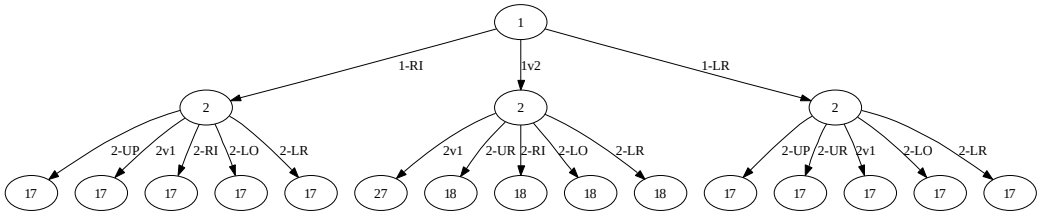

Figure 5.5: Two opposing units ready in different timestamps

man) both are ready to perform actions in game cycle 1 is depicted in figure 5.4 on the preceding page. The fanout of the root becomes the product of actions of both units. Unit 1 is placed in the top left corner of the map in entry *(0,0)* whereas unit 2 is placed just below in entry *(0,1)*. Unit 1 has three possible actions: move right, lower right or attack. Unit 2 has five action possibilities. A total of 15 possible combinations of actions. As figure 5.4 on the previous page shows the total number of children is 13 which could be confusing at first glance. The reason why there are 13 combinations of actions and not 15 is that two of the action combinations are illegal – they conflict. The problem is that two units are ready at the same time and both want to move into the same field. This situation is resolved by letting only one unit move. Figure 5.4 on the preceding page also illustrates the timestamping of nodes. Firstly, the shaded state with timestamp 27 occurs when both units choose to attack. Secondly, the timestamping of those nodes where only one of the units attack are also of importance. These are timestamped with 17, because the movement action is completed before the attack-action and thus the moving unit should be allowed to take action(s) again in game cycle 17 and not wait until game cycle 27.

In figure 5.5 a similar situation to the one in figure 5.3 on the preceding page is shown. The difference from figure 5.4 on the previous page is that unit 1 is ready in game cycle 1 and unit 2 is ready in game cycle 2. It can be seen that in the middle subtree unit 1 has already begun an attack. In game cycle 2 unit 2 is allowed to perform actions. If unit 2 performs a movement action it is ready again in game cycle 18 thereby letting unit 2 begin another action before unit 1 has finished its attack.

Generally, units are not ready to perform actions in the same game cycles. Therefore the kind of tree presented in figure 5.5 is encountered more frequently than the kind of tree shown in figure 5.4 on the previous page.

5.2.2 Issues

As mentioned in section 5.1.3 on page 55 three issues must be handled to apply game trees to SSC situations.

How SSC situations are represented is one of them. The representation of a game state must be precise enough to capture the essential features, e.g. unit states and environmental features. How this is done is examined further in section 5.3.

The size of the game tree is another issue. To get an idea of the size of the game tree, recall that the fanout of a node is calculated as the product of the available actions of the units. Assume that four units ready at game cycle one are involved and that the lookahead is 150 game cycles equivalent to five wall clock seconds in Wargus. If three movement actions are available to each unit the resulting number of leaves is

$$3^{4^{\lceil \frac{150}{16} \rceil}} = 81^{10} \approx 1.21 \times 10^{19} \tag{5.1}$$

This tree can obviously not be fully built within the real time constraints of the game. This issue is elaborated further upon in section 5.4 on page 64.

When the game tree has been constructed the remaining issue is how to assign a numerical value to each node to determine the node's desirability. When this has been done, the actions leading to the subtree minimising the chance of losing and maximising the chance of winning can be chosen. Handling this issue is described in section 5.5 on page 75.

Measuring the quality of solutions to the above three issues is difficult. These depend on each other to deliver a solution to SSC. It is possible to estimate the optimality of the game tree method but this implies only that the three sub-solutions work well together and not whether each one is solved optimally.

5.3 Representation

A node in the timestamped game tree represents an SSC situation – a more or less simplified image of the game state. We will from here on refer to this simplified image as a *snapshot*. A snapshot encapsulates the position of each unit, the map environment and also at which game cycle each unit is ready to perform actions. A snapshot is a 20×20 subset of the map centred around the group of controlled units. The snapshot corresponds to the controlled units' view of the immediate environment. Furthermore, a node contains a *threat matrix* described in section 5.3.1 on the following page which form the basis of the rating method applied as examined in 5.5 on page 75. A sample SSC situation, the corresponding snapshot and threat matrix are shown in figures 5.6 on page 65, 5.7 on page 65 and 5.8 on page 66.

Note that we ignore the fog of war described in section 3.4.1 on page 40 since we implicitly assume that all of the snapshot is within sight-range of the controlled units.

Transitions between two snapshots are the *n* actions, which lead from one snapshot to another. In Chess *n* always equals one because moving two pieces simultaneously is not allowed. In an RTS game *n* can be as large as the number of units considered since units can perform actions simultaneously. Each transition thus models a change in state such as a change in unit positions, change in unit state or a change in the environment. When a transition is created the resulting child node is timestamped with the minimum completion time of all non-completed actions in the path from the root to this node. For details of how the timestamped game tree is constructed, see section C.2.2 on page 138.

5.3.1 Threat matrix

In Chess each piece on the board is given a value which decides its desirability. This value is based on empirical and experimental studies. In [Weeks, 2005] the values of Chess pieces are investigated. These do not change in different Chess situations but are merely used to assess the game state. Thereby a Queen with value nine is not traded directly for a Rook with value five.

In an RTS game context the value of each unit depends very much on the situation at hand. The environment, the actual unit-placement and the composition of units all have a strong influence on the desirability of the individual unit. E.g. if a unit has a low number of hitpoints this is worth less than a more "lively" unit.

A *threat value* (TV) is a value assigned to units. It constitutes the desirability of the unit – the higher the value, the higher the desirability. The value is not statically calculated as in Chess but is based on a number of variable parameters. A *threat matrix* is a 20×20 matrix of TVs derived from a snapshot. Each entry in the snapshot can contain one unit and maps to the same entry in the threat matrix and contains the TV of this unit. Entries in the snapshot with no units contains a default TV of zero. A threat matrix corresponding to the snapshot in figure 5.7 on page 65 is depicted in figure 5.8 on page 66.

5.3.2 Deriving a threat value for units

As mentioned above a threat value (TV) represents a unit's desirability. The higher the TV, the more desirable the unit is perceived. No literature describing the value of RTS units has been found. Therefore, the following is based solely on our understanding of the Wargus game rules.

When is a unit then desirable? A unit which deals damage to enemy units is generally of higher value than one that has to move to deal damage since game

cycles are wasted on moving around. If game cycles are used on movement the game cycles are not used for dealing damage, but moving can in some cases increase the effectiveness of the unit. Recall from section 3.4.4 on page 46 that one of the goals of solutions to SSC situations is to maximise the damage dealt. When no enemy unit can be attacked the number of move actions needed to get within range becomes important. The attributes of a unit also influence its TV, because these represent the flexibility of the unit.

With the above in mind we have identified four different aspects that must be incorporated in the formula for calculating TVs. Again, note that no literature describing how to estimate the threat of units in a RTS game has not been found. The found literature described turnbased equivalents. Therefore, the following is based on our understanding of threat of units in a RTS game and on our empirical testing in the course of this thesis.

1. **The unit's type** A numerical value is introduced for each different type of unit in the game. The value represents the flexibility of the type. The higher this value the more flexible. In Wargus there are several attributes describing the abilities of each type. The following integer attributes have been identified as important and are incorporated in the value:

   AttackRange(u) The maximum Manhattan distance counted in fields from *u*'s position, in which enemy units can be attacked. The higher the attack range, the more useful the unit is.

   DamageDealt(u) A value which serves as the base for the randomised damage dealt by *u* when attacking. The higher the damage dealt, the better.

   Armour(u) When calculating the damage which a unit receives from an attack this value is subtracted. The higher the armour, the better.

   MovementSpeed(u) The speed of a unit. All units included in our solution have the same movement speed. But to incorporate nuances of future unit types this variable was also added. The higher the speed, the better.

   MaxHP(u) The maximum amount of hitpoints *u* can have. The higher hitpoints, the better.

   AttackTimeInGameCycles(u) How many game cycles an attack performed by *u* takes to complete. The higher the value, the slower the attack.

   The factor representing *u*'s type in the TV calculation is $vUT(u)$ and is

calculated as follows:

$$\begin{aligned} vUT(u) &= AttackRange(u) \times \frac{1}{AttackTimeInGameCycles(u)} \times \\ & DamageDealt(u) \times Armour(u) \times MovementSpeed(u) \times \\ & MaxHP(u) \end{aligned} \tag{5.2}$$

2. **Amount of hitpoints** A unit's hitpoints is a very crucial factor. This number indicates how long the unit will be able to contribute to the SSC situation. Meaning, the higher the hitpoints, the better. To capture this aspect the relative amount of hitpoints is calculated as $\frac{CurHP(u)}{MaxHP(u)}$, where $CurHP(u)$ is the actual amount of hitpoints of u.

3. **The placement** A unit having to move to attack an enemy unit is as before mentioned not as desirable as a unit having an enemy unit within its attack range. To capture this aspect the number of movement actions needed to get an enemy within range is incorporated in the TV calculations.

   The $dist(u,v)$ is introduced where u and v are units to indicate the number of movement actions needed for unit u to reach v. This equals the Manhattan distance between units u and v. The nearest enemy unit is located as follows:

$$e(u) = \min_{i \in EnemyUnits} dist(u,i) \tag{5.3}$$

   When the nearest enemy unit is found the factor introducing the placement aspect can be given. The value $DistTCE(u)$ quantifies the notion of either having an enemy unit within range or having to move. The greater the distance to the nearest enemy unit, the higher $DistTCE(u)$ becomes. I.e. the lower the $DistTCE(u)$, the better. Note: $AR(u)$ is a shorthand for $AttackRange(u)$.

$$DistTCE(u) = \begin{cases} 1 & dist(u,e(u)) <= AR(u) \\ dist(u,e(u)) - AR(u) + 1 & \text{otherwise} \end{cases} \tag{5.4}$$

   The reason for not using the $dist(u,v)$ by itself is that the ranged units have an attack range greater than one. If $dist(u,v)$ was used alone the ranged units would be less desirable even if enemy units were within attack range.

4. **The ready time** When calculating the threat matrix of a snapshot, a subset of units are ready to perform actions and others are not. Therefore, it is important to know when a given unit is ready to perform actions. The function

gameCycleTNA(u) is the number of game cycles u requires to complete its current action. This implies that a low *gameCycleTNA*(u) is favourable compared to a high one.

Having introduced the four different aspects the equation for calculating a TV can be presented:

$$TV(u) = \frac{vUT(u)^{\frac{CurHP(u)}{MaxHP(u)}+1}}{\sqrt{DistTCE(u)} \times \sqrt[3]{gameCycleTNA(u)}} \tag{5.5}$$

For a unit's TV we chose $vUT(u)$ as a base value. Since the relative amount of hitpoints of a unit is a very crucial factor we raise the base value to the power of $\frac{CurHP(u)}{MaxHP(u)} + 1$. This ensures a huge reward if the unit has a large amount of hitpoints left. The square root of $DistTCE(u)$ and cubic root of *gameCycleTNA*(u) is introduced, because through empirical testing we found that the importance of these factors were less than linear in their value.

Equation (5.5) is normalised to the range of $[0;1]$. If the formula is calculated for an enemy unit the value is negated and the range becomes $[-1;0]$. This implies that an enemy unit with a TV of -1 is more of a threat than an enemy unit with a TV of -0.5.

5.3.3 An SSC example

In figure 5.6 on the following page an SSC situation from Wargus is visualised. The figure shows how the player perceives the situation. Each side has an equal number of melee and ranged units (one melee unit and two ranged units). It can be seen that the grunt which is fighting with the footman has just received four points of damage. It is indicated by the -4 hovering over the grunt.

Figure 5.7 on the next page shows the snapshot of the situation and figure 5.8 on page 66 depicts the associated threat matrix. In figure 5.8 on page 66 each number represents the units' threat values. It can be seen that the orc grunt is of lesser value than the human footman which is due to the amount of hitpoints.

5.4 Pruning

As mentioned in section 5.2.2 on page 60 the large size of the game tree must be handled within the time constraints. In Chess the time constraint imposed is often in terms of minutes generally one or two. In a real time game the time constraints are much more restrictive and a more heavily pruning of the game tree must be incorporated. In Wargus the specific time constraint is $\frac{1}{30}$ second as mentioned

Figure 5.6: An SSC situation in Wargus

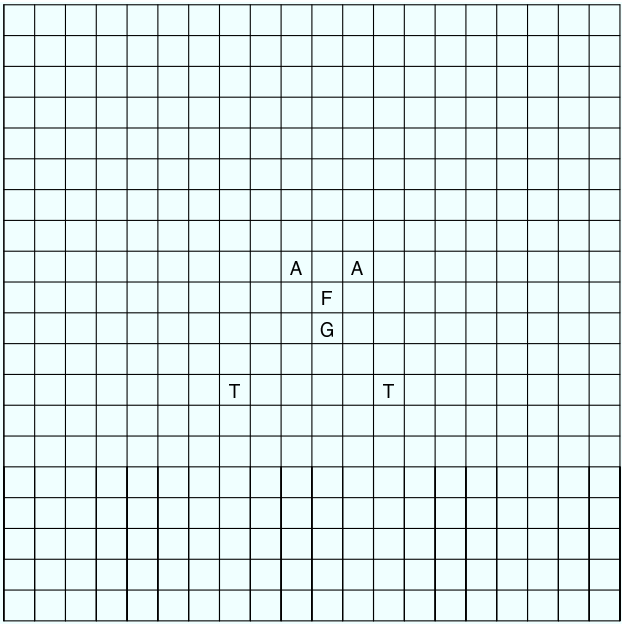

Figure 5.7: The snapshot corresponding to figure 5.6

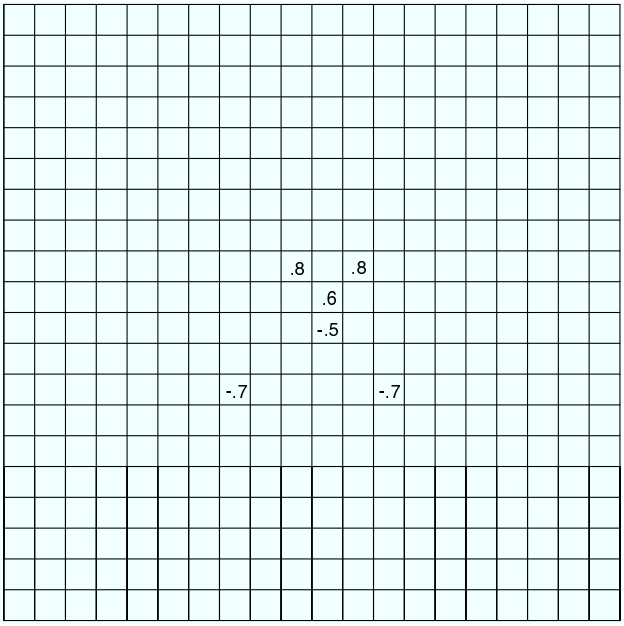

Figure 5.8: The threat matrix derived from the snapshot in figure 5.7

in section 4.1.3 on page 49. However, we focus solely on improving the game quality of Wargus. If the time constraint is violated but not often and not by much the perceived quality of the game does not diminish as the player does not notice these violations.

Traditionally, one way of reducing the size is having a heuristic which decides for each constructed node whether to construct its children. Algorithms such as, *Min-Max*-search and $\alpha - \beta$-search, [Nilsson, 1998, Chap. 12] and [Björnson and Marsland, 2001], are examples of heuristics.

We have chosen a heuristic method which reduces the size of the tree by choosing to construct a subset of a node's children. In section 5.1.1 on page 53 a method for solving SSC using rules to derive actions from the current situation was presented. The general idea to reduce the game tree's size is to use the rule based method as the heuristic for choosing the subset. This allows us to estimate the outcome of each of the available actions. The success of this pruning method relies only on the quality of the rules. Poor rules can prune the desirable game states and result in sub-optimal performance. Therefore, designing the rules must be done very carefully. If done right the less desirable game states will be pruned while the good states will be kept for potential evaluation.

A problem with the rule-based heuristic is that enemy units are beyond control. To model the actions of the enemy units three options are available. First, the span of actions of the enemy units can be untouched. Meaning that the precision

of future game states is very high since the tree models all possibilities. Unfortunately this option is infeasible, because the size of the game tree is not reduced for the enemy units.

Secondly, the actions of the enemy units can be handled with a set of rules. Either the set applied to the controlled units or a separate set. This means that the precision of future game states decreases since it is implicitly assumed that the enemy units follow the rules.

Lastly, all actions of the enemy units can be ignored implying that the enemies always stand still. This is of course very imprecise. In the game these always perform actions. However, since only near-future situations are considered this estimate is partially correct.

Given these options we have chosen to use the third option to model enemy units. This was chosen, because the first option is infeasible. Both the second and third options produce inaccurate future game states. However, the third option decreases the size of the game tree compared to the second.

In the following several rules and sequences hereof will be presented. We have designed the rules with simplicity in mind to allow composition of the rules into sequences. Although all implemented rules and their effect will be described, only a subset of the rules has been incorporated in the rule-sequences as described in section 5.4.3 on page 73.

5.4.1 Rules for game tree pruning

Before the pruning of a game tree node is performed the fanout is the total number of available actions as described in section 5.2.1 on page 57. When selecting the subset of nodes to construct the number of available actions for each unit must be limited. A rule used to select the subset consists of a precondition and an effect. The rule's precondition captures in what situations the rule applies. The effect of a rule is a reduction in the number of available actions for each unit. A rule is said to *click* if the precondition is satisfied.

Since a few of the rules use a concept known as *influence-mapping*, as described in [Sidran, 2003] and [Sweetser, 2004] this concept will be introduced before presenting the set of rules.

Influence mapping

An *influence map* is a strategic perspective typically used by Game AI modules. Conceptually, it provides an overall representation of the environment. It is placed on top of an environment to gather knowledge in a compressed manner and stores game-relevant data. Typically, information about player strength, resources, valuable assets or unit passability is stored.

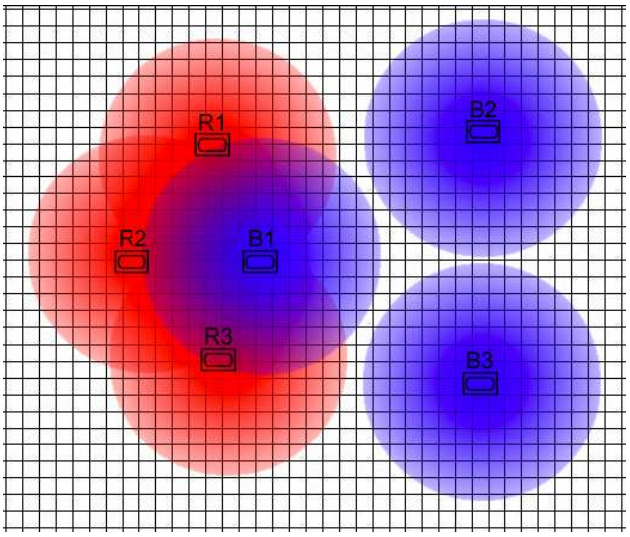

Figure 5.9: A sample influence map

Figure 5.9 from [Sidran, 2004] depicts an influence map, storing information about player strength. The figure contains six units, three blue and three red. Each unit has a value representing its effectiveness or strength. These values are visualised as circles. The colour gradient shows the strength of the units' influence. In the areas where the red units' circles overlap, each unit's strength contributes to the influence in that area.

The information stored in the influence map we use regards player strength. Our influence map subdivides a snapshot into a smaller matrix consisting of *cells*. A cell represents a connected area of the snapshot and stores the number of friendly and enemy units in that specific area. Each cell also stores a value for each player which represents the strength of each player. The strength of a player in a cell influences the value in all other cells. This value is calculated in two steps:

1. **Initialisation** Each influence map cell is initialised with a mapping from snapshot fields to cells. Each cell is then updated to contain the sum of threat values (TVs) of each player's units in the fields within this cell. TVs were defined in section 5.3.2 on page 61. $IV_{p,x,y}$ denotes the initialisation value for the player p in the cell with coordinates (x,y). $IV_{p,x,y}$ is calculated as follows:

$$IV_{p,x,y} = \sum_{u \in PlayerUnitsInCell(p,x,y)} TV(u) \tag{5.6}$$

where $PlayerUnitsInCell(p,x,y)$ is the set of units within the cell with coordinates (x,y) controlled by the player p.

2. **Influence** In the influence step cells influence each other by letting values "flow" from cell to cell. This is done with a diminishing factor λ based on the distance between the two cells. $CV_{p,x,y}$ denotes the final value for player p in the cell with coordinates (x,y). $CV_{p,x,y}$ is calculated as follows.

$$CV_{p,x,y} = IV_{p,x,y} + \sum_{(i,j) \in Cells}^{i \neq x, j \neq y} IV_{p,i,j} \times \lambda^{max(i,j)} \tag{5.7}$$

   where $0 \leq \lambda \leq 1$ and $Cells$ is the set of all cells in the influence map.

For more information concerning influence maps and their applicability with neural networks see [Sweetser, 2004].

5.4.2 Rules

The rules presented in the following reduce the span of actions for a single unit. As previously mentioned each rule has a precondition and an effect. These will be presented along with a description of the rule. This list of rules is devised by the authors and is based on primitive rules used in other RTS games and on our understanding of the Wargus game rules.

No enemy in cell

The *No enemy in cell*-rule applies if there are no enemy units located in the influence map cell of the considered unit. If this rule clicks, the considered unit's span of actions is reduced such that the unit moves towards the influence cell with the highest opponent value. We chose to let this reduction leave three movement possibilities towards the before-mentioned cell.

Precondition No enemy units in the cell of the considered unit.

Effect Move towards the cell with the highest opponent value.

Keep attacking same

The *Keep attacking same*-rule applies if the considered unit has made an attack as its previous action and if the attacked unit is not dead. If this rule clicks, the unit's span of actions is reduced to one – namely attack the same enemy unit again or move towards the enemy unit if it has moved.

Precondition Previous action was an attack and the unit attacked is not dead.

Effect Attack or follow the attacked enemy unit.

Flee

The *Flee*-rule applies if the influence map cell of the considered unit contains a too high opponent value compared to the friendly player value. Too high means that the opponent value is the double of the friendly player value. If this rule clicks, the unit's span of actions is reduced to moving out of the cell and towards the cell with the smallest opponent value. I.e. a reduction to a single action.

Precondition Opponent value in considered unit's cell is twice as large as the friendly player value.

Effect Move out of the cell towards the neighbour cell with the lowest opponent value.

Ranged attack lowest HP enemy

The *Ranged attack lowest HP enemy*-rule applies only to ranged units and only clicks if there are enemy units within the considered unit's attack-range. If this rule clicks, the enemy unit with the least amount of hitpoints will be attacked and the span of actions is reduced to one.

Precondition Considered unit is of ranged type and has enemy units within attack-range.

Effect Attack the enemy unit with the lowest hitpoints.

Ranged attack lowest TV enemy

The *Ranged attack lowest TV enemy*-rule is similar to the *Ranged attack lowest enemy HP*-rule. But instead of choosing the unit with the least amount of hitpoints the unit with the least threat value is chosen.

Precondition Considered unit is of ranged type and has enemy units within attack-range.

Effect Attack the enemy unit with the lowest threat value.

Ranged support

The *Ranged support*-rule allows ranged units to support other units. This rule clicks if an enemy unit within attack-range of the considered ranged unit is adjacent to a friendly unit. The considered unit's span of actions is reduced to one. If multiple enemy units are adjacent to friendly units, one at random is chosen.

Precondition Considered unit is of ranged type. There is an enemy unit within attack-range and it is adjacent to a friendly unit.

Effect Attack the enemy unit adjacent to a friendly unit.

Ranged attack nearest

The *Ranged attack nearest*-rule applies to ranged units. It only clicks if there are enemy units within the considered unit's attack-range. If this rule clicks, the nearest enemy unit is attacked. If multiple enemy units are equally close, a random is chosen.

Precondition Considered unit is of ranged type and has enemy units within attack-range.

Effect Attack the nearest enemy unit.

Ranged attack max *N* enemies

The *Ranged attack max N enemies*-rule applies to ranged units with enemy units within attack-range. If this rule clicks, the considered unit's span of actions is reduced to attacking at most N random enemy units within attack-range.

Precondition Considered unit is of ranged type and has enemy units within attack-range.

Effect Attack at most N enemy units.

Melee units attack lowest HP adjacent enemy

The *Melee units attack lowest HP adjacent enemy*-rule applies to melee units with enemy units adjacent. If this rule clicks, the span of actions is reduced to attacking the enemy unit with the least amount of hitpoints.

Precondition Considered unit is of melee type and is adjacent to enemy units.

Effect Attack the adjacent enemy unit with the least amount of hitpoints.

Melee attack lowest TV adjacent enemy

The *Melee units attack lowest TV adjacent enemy*-rule applies to units of type melee and is similar to the preceding rule. But the adjacent enemy unit with the lowest TV is chosen instead.

Precondition Considered unit is of melee type and is adjacent to enemy units.

Effect Attack the adjacent enemy unit with the lowest threat value.

Remove stand-ground

The *Remove stand-ground*-rule simply removes the considered unit's stand ground action. When this rule clicks, the span of actions is reduced by one.

Precondition None.

Effect Removes the considered unit's stand ground action.

Attack *K* lowest HP enemies

The *Attack K lowest HP enemies*-rule applies if the considered unit has enemy units within its attack-range. If this rule clicks, the considered unit's span of actions is reduced to attacking the *K* enemy units within attack-range with the lowest amount of hitpoints.

Precondition The considered unit has enemy units within attack-range.

Effect Attack the *K* enemy units with the lowest amount of hitpoints.

Attack *K* nearest enemies

The *Attack K nearest enemies*-rule always applies. The considered unit attacks or moves towards the *K* nearest enemy units.

Precondition None.

Effect Attack or move towards the *K* nearest enemy units.

5.4.3 Rules and their sequencing

The rules described in the previous section were designed with simplicity and some degree of predictability in mind. This was done to ease the effort required to compose rules. Compositions of rules become important, because simple rules often do not perform near-optimally. Compositions allow using the rules as building blocks and thereby allow more complex behaviours.

To enable compositions of rules each rule must only reduce the span of actions. This requirement is in order, because rules are applied in sequence. Thus, it is expected that if rule *A* appears before rule *B* in the sequence then rule *B* does not violate the intentions of rule *A*, i.e. rule *B* must not add actions removed by rule *A*.

All rules described in section 5.4.2 on page 69 implements an undo-function which undo's the rule's pruning. This is done to avoid a total exhaustion of actions of the unit. The undo-function works as follows:

1. Before applying a rule to a unit take backup of that unit's actions.
2. Apply the rule.
3. Check if the unit's number of actions is reduced to zero. If so, revert the unit's actions from the backup.

The undo functionality ensures that if rule *A* has been applied before rule *B* then rule *B* can only prune actions left by rule *A* while not exhausting all actions.

In [Davis, 1999] Ian Lane Davis presents an overview of the Game AI system for a game called *Dark Reign*, [Auran, 1997]. He discusses the effectiveness of RTS units in combat situations. He states:

> "[...] in most strategy games, the offensive effectiveness of a unit does not diminish at all with damage until the unit is totally removed from the game. This means it virtually always makes sense to concentrate firepower at one target until it is destroyed and then move on to the next", [Davis, 1999].

This premise also holds for the game of Wargus and with that in mind some sequences of rules will be introduced following this idea.

In the following the sequences of rules we have devised will be presented along with a justification and explanation of each sequence.

Focus fire

Focus fire is a sequence of rules in which primarily ranged units are imposed to focus attacks on the weakest enemy unit within attack-range. The idea behind the *Focus fire*-sequence is to let ranged units concentrate on killing the weakest enemy units instead of damaging many. At the same time the melee units move towards the enemy units and attack these.

Focus fire consists of the following four rules applied in the order given below:

1. **Ranged attack lowest HP enemy**
2. **No enemy in cell**
3. **Attack *K* lowest HP enemies**
4. **Remove stand-ground**

In this sequence the ranged units prioritises attacking enemy units over moving towards them. The *Ranged attack lowest HP enemy*-rule removes all except an attack action if this rule clicks. If so the considered ranged unit has no movement actions available. Since rules cannot add actions the *No enemy in cell*-rule has no effect even if it clicks. The *Attack K lowest HP enemies* obviously has no effect on a ranged unit either. Melee units are guided towards the enemy units and if several units are within attack-range the K weakest are attacked. The *Remove stand-ground*-rule is added because standing still is regardless of the situation almost never optimal.

Attack *K* nearest

This is a sequence guiding units to attack the closest enemy units. The idea behind the *Attack K nearest*-sequence is to minimise the game cycles spent on moving without considering the positioning directly. Instead, the game cycles are spent on dealing damage ignoring any positional advantages presented by the map. Here, K is set to two.

Attack K nearest consists of the following three rules applied in the order given below:

1. **Keep attacking same**
2. **Attack *K* lowest HP enemies**
3. **Remove stand-ground**

Here, the *Keep attacking same*-rule ensures that all units gives attacking the same target higher priority than attacking a new. If a unit is not engaged in combat it is guided towards the K enemy units with the lowest amount of hitpoints.

Ranged assist

Here ranged units acts as support to other friendly units. The idea is that if the melee units are engaged in combat the ranged units can assist in killing the engaged enemy units faster thereby minimising the received damage.

Ranged assist consists of five rules applied in the order given below:

1. **Ranged support**
2. **Keep attacking same**
3. **No enemy in cell**
4. **Attack *K* lowest HP enemies**
5. **Remove stand-ground**

The *Ranged support*-rule ensures that if there is a possibility to support a friendly unit within attack-range then the ranged units will do so. Units not handled by the *Ranged support*-rule behave almost as in the *Attack K nearest*-sequence. Again, *K* is set to two. The difference lies in guidance towards the enemy units. Instead of moving towards the *K* weakest enemy units, the units move towards the influence map cell chosen by the *No enemy in cell*-rule.

5.5 Rating game states

Assessing or estimating how good a game state is a game specific topic. The estimate method evaluating a game state is very hard to design. It requires an extensive insight into the game mechanics. Given a game state the estimate method should return a value which represents the desirability of a state. It should be evident that the precision of the chosen rating method influences the optimality of the game tree method. Also, the speed of the rating method is of interest. The rating method must be able to rate game states in the tree within the time limit.

In order to rate states in a game tree representing an SSC situation we have chosen to use the threat matrix (see section 5.3.1 on page 61) as input. This matrix was chosen because it captures the elements of SSC which we have identified as important. A rating method was designed by hand. However, the interdependency of the elements in SSC is complex making a precise rating method hard to design. Therefore, machine learning methods were also investigated for rating game states. Recall that a threat matrix is of size 20×20 and therefore all rating methods in this project can be seen as functions mapping $\mathbb{R}^{400}[-1,1] \rightarrow \mathbb{R}$.

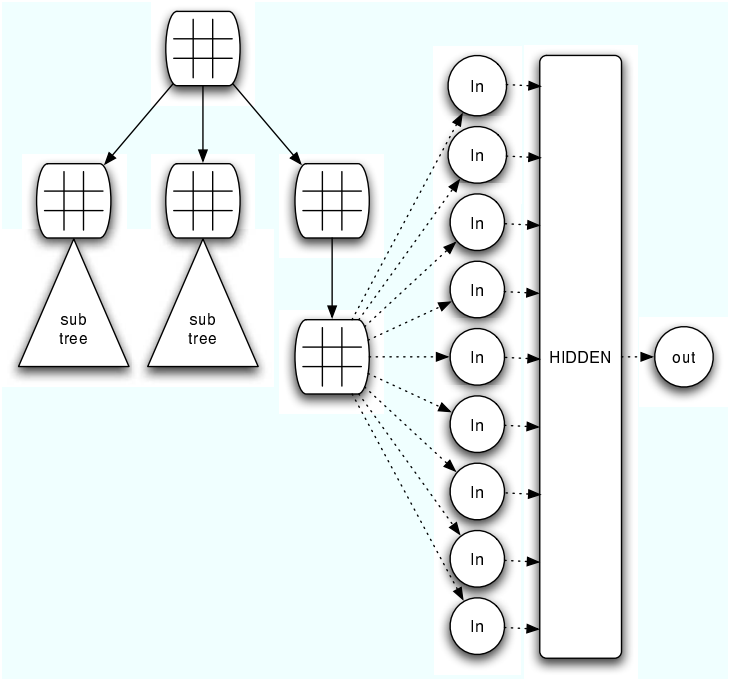

Figure 5.10: A game tree and a sample rating method

In figure 5.10 a game tree and a rating method are shown. Nodes in the tree contain the threat matrix. The rating method shown is a neural network explained in section 5.5.2 on the following page. This figure shows how the threat matrix is mapped via a neural network to a single value.

In the following the handcrafted rating method and the machine learning based methods are presented along with their applicabilities and advantages. Lastly, how to choose actions is discussed.

5.5.1 Handcrafted rating method

As the threat matrix captures many elements of an SSC situation we have designed a simple rating method called *threat matrix rater* (TMR) based on the threat matrix. Recall that TVs for enemy units are negative and TVs for controlled units are positive. The value of a game state is calculated as the sum of all entries in the threat matrix. This rating method uses the overall strength of each side to decide the desirability of each game state. Clearly this method's performance is entirely dependent on the ability of the TVs to accurately portray the situation in the snapshot.

A handcrafted rating method's accuracy depends solely on the skills of the

designer and the designer's understanding of the problem domain. This method is troubled by the fact that it cannot generalise nor adapt to situations not anticipated by the designer. This kind of rating method is often an easy and quick way to rate a situation but high precision is hard to achieve.

The time needed for the TMR to rate a threat matrix is linear in the number of entries in the matrix since the value of each entry must be summed. In this project the amount of summations needed is 400 for each rating.

5.5.2 Machine learning rating methods

A machine learning method is another way to rate game states. Such methods can automatically improve their precision through learning. An advantage of these methods is their ability to generalise from a small set of problem instances with reasonable precision. To obtain the wanted generalisation the machine learning methods need to be trained.

We have chosen to apply two machine learning methods to rate game states. To train these methods a set of handmade game states was designed. The threat matrix of each constructed game state was translated into several equivalent training examples to increase the size of the training set.

In the following the two chosen machine learning methods will be presented.

Neural networks

A neural network (NN) is a learning method which is modelled as the way collections of neurons in the human brain work. An output signal is produced based on several input signals. The NN applied in this project is a fixed fully-connected layered feed-forward network as defined in [Nilsson, 1998] and [Mitchell, 1997, Chap. 4].

We have chosen to use three layers for the NN: an input layer, a hidden layer and an output layer. As input for the NN each entry in the threat matrix is mapped to an input node resulting in 400 nodes in the input layer. We have chosen to use the same size for the hidden layer. As a single value is needed for the rating the output layer contains a single node.

The training of the neural network was performed with the *Resilient Propagation*-algorithm (*RProp*), [Champandard, 2003, Chap. 19], a much used variant of the *Back-Propagation*-algorithm, [Mitchell, 1997, Chap. 4.5.2].

The time needed for the NN to rate a threat matrix is linear in the number of nodes in each layer in the neural network since each input-value must be propagated through the network. In this project the number of calculations is $400 \times 400 \times 1 = 160.000$ for each rating.

The RMSE, [Wikipedia, 2005d], for the NN over the validation set extracted from the training threat matrices is 0.0976.

K nearest neighbour

The *K* nearest neighbour (KNN) is an instance-based learning method, [Mitchell, 1997, Chap. 8]. The KNN applied in this project views each threat matrix as a point in $\mathbb{R}^{400}$ and uses the euclidean distance to determine closest neighbours.

As training the KNN method stores each sample threat matrix for later usage in rating. Rating is done by calculating the euclidean distance of the given threat matrix to each stored point. The rating value is taken as the average of the *K* nearest.

The time needed for the KNN to rate a threat matrix is linear in the number of stored threat matrices and *K*, because the distance between the threat matrix and the stored set must be calculated and the *K* nearest must be found. In this project the number of stored examples is 3600 and *K* is 5. Therefore, $3600 \times 400 \times 5 = 7.200.000$ calculations must be made for each rating. This was empirically tested to be too slow and the number of stored threat matrices was therefore reduced to 360 resulting in $360 \times 400 \times 5 = 720.000$ calculations for each rating which resulted in no perceived violations of the real time constraints.

The RMSE for the KNN over the validation set extracted from the training threat matrices is 0.2562.

5.5.3 Choosing actions for units

A game tree is built in game cycles where at least one unit has no actions assigned. This means that the root node of the game tree represents a situation in which at least one unit is ready to act. When the game tree has been built the action sequence leading from the root to the child with the highest value is assigned to the units in the engine.

The rating methods described in sections 5.5.1 on page 76 and 5.5.2 on the preceding page only rate single states. This means that the rating methods do not account for states which are reachable from the state being rated. To let future states influence the choice of unit actions we have introduced two variants of each rating method:

1. Only rate the children of the root ignoring the future states reachable beyond the children of the root. This is the equivalent of rating all nodes in the tree. But as the rating methods does not explicitly use the reachable states to rate a given node those states can be ignored.

2. Only rate the leaves of the game tree and then assign the average value of the children to each internal node. This allows a child node of the root to indicate an estimated value of the subtree this node represents.

5.6 Measuring the performance of an SSC situation

Recall from section 3.4.4 on page 46 that an optimal solution to SSC leads to consistent behaviour for the involved units. When a method for solving SSC has been implemented a way of measuring the optimality of the method must be available. As optimality leads to consistency all of the created methods for handling SSC situations are evaluated. This provides an indication of which computer controlled method is the most optimal in the designed SSC situations.

As discussed in section 3.4.4 on page 46 human players can use the method as a support routine which automatically carries out combat in SSC situations resulting in a near-optimal handling of these. The goal is not to create an interesting behaviour but to create the most optimal behaviour which leads to a consistent behaviour.

5.6.1 SSC situation value

To evaluate a method solving SSC several SSC situations have been designed, as described in section 7.1.2 on page 90. Each method is evaluated on each designed situation. When two methods oppose each other a measure of performance must be available. Preferably as a numerical value indicating how well a particular method performed. Options for this value include: a binary indicator of whether the given method won or lost, a count of inflicted casualties or a count of the amount of damage inflicted. We have chosen a more expressive value indicating the relative strength of each faction in the situation. We have designed a *Situation Value* (SV) which indicates how good an situation is for a particular game cycle compared to the starting situation. We have chosen to use the number of units and their hitpoints as indicators of how good the situation is. Thus, for a game cycle x we introduce the following factors:

$EUnits_x$ is the number of enemy units at game cycle x.

$FUnits_x$ is the number of friendly units at game cycle x.

eHP_x is the sum of all enemy units' hitpoints at game cycle x.

fHP_x is the sum of all friendly units' hitpoints at game cycle x.

The SV is calculated to indicate that the higher the SV, the better the situation. It is considered bad to lose units and thereby hitpoints. These incidents should reduce the SV. It is also considered bad if the enemy has many units and many hitpoints. Given these considerations we have designed the following equation:

$$SV_x = FUnits_x \times \frac{fHP_x}{fHP_0} - EUnits_x \times \frac{eHP_x}{eHP_0} \tag{5.8}$$

If friendly hitpoints or units are lost the SV is reduced. If enemy units are damaged the SV is not reduced as if no enemy unit was damaged. Note, that due to the calculations involved, the value given to one faction is the negated value of the opposing faction's.

5.6.2 Experiments

In section 5.2.2 on page 60 we identified three issues which had to be handled before the performance of a game tree-based method could be evaluated. Section 5.3 on page 60 presented one solution to the problem of representation. As mentioned in section 5.4.3 on page 73 we designed three rule sequences which we deemed applicable to the game of Wargus. In section 5.5.3 on page 78 six rating methods were presented to the problem of rating game states.

To investigate these different solutions to the issues involved with game tree-based methods we have chosen to evaluate all permutations of these resulting in $1 \times 3 \times 6 = 18$ game tree-based methods. These permutations will be referred to as *experiments* and are presented in section 7.1.1 on page 89.

Chapter 6

Extending the Stratagus engine

"There are no problems, only opportunities"
- Bill Austin

In this chapter we will present the overall scope of our implemented system and describe how our module interfaces with the Stratagus RTS engine. As mentioned in chapter 4 the Stratagus engine was chosen as the platform for evaluations and from now on the Stratagus engine which is written in C will only be referred to as the *engine*. For complete design, implementation details and pseudo-code for the game tree construction, see appendix C on page 129.

For solving the SSC problem the game tree-based method can be seen as a support routine as described in section 3.4.4 on page 46. Therefore, our module which handles the problem of SSC should be designed so replacing or extending this module can be accomplished. As such, the design of this support routine was suited for an object oriented approach. We decided to create a JNI interface, [Sun Microsystems, 2003] for Java v. 1.4.2 since this allowed us to separate the game tree code from the engine while using object oriented facilities.

6.1 The Stratagus engine background

Stratagus, [Stratagus, 2004], is an RTS engine written in C and is released under the GPL, [Free Software Foundation, 1991]. Using what the Stratagus developers call the *Craft Configuration Language* (CCL) a potential game developer can implement an RTS game by specifying the rules of the game and what media files should be used. The CCL is a *Lua*, [Ierusalimschy *et al.*, 2003], interface which exposes some of the Stratagus engine's internal data-structures. The Lua interface allows developers to modify how the engine runs the actual game and how the game is presented to the player graphically. There are some technical limits to this interface which have their roots in the 10 year long development history of the Stratagus engine.

Wargus, [Wargus, 2004], is a set of CCL-scripts for Stratagus that enable end-users to play the famous Warcraft I and II, [Blizzard, 1995]. The Wargus project does not distribute the media files associated with Warcraft II, but includes a conversion utility which takes the original Warcraft II (including expansion) CDs and converts the media on these CDs to a format Stratagus can read. The Wargus project is maintained by the Stratagus developers and this implies that all new releases of Stratagus are accompanied by a simultaneous release of Wargus.

Stratagus and Wargus started their lives as an application called *FreeCraft*. It was an Open-Source game which allowed people to play Warcraft I and II on other platforms than Windows provided the player had the original CDs from Blizzard. Blizzard's legal department struck down on this project due to the flamboyant use of the *Craft* name in the project. From the ashes of FreeCraft rose the Stratagus engine and the Wargus project. The inherent limits in the CCL of Stratagus stems from this history as it primarily was designed to play the Wargus game. Since then, the Stratagus developers have focused on generalising the CCL and thereby giving new game developers more freedom to deviate from the way Wargus is played.

6.2 The Stratagus engine

Recall from section 3.4 on page 40 that we focus on multi-player network games called skirmishes. To reach the goal of creating a support routine enabling both participants and players to use it we needed to investigate how the engine handles network games and unit control.

6.2.1 Communication protocol

For network communications the engine uses a P2P protocol based on the UDP-protocol, [Stallings, 2000, Chap. 17.4] and [Coulouris *et al.*, 2001, Chap. 3.4.6 & 4.2.3]. To minimise the network traffic, all clients in a networked game maintain a complete state of the game. The only traffic occurring on the network is the orders assigned by the players. Due to this, all clients which participate in a networked game maintain the complete game world state internally. An approach like this has several implications. E.g. the computations required for the participant are performed on all clients simultaneously and must be guaranteed to reach the same conclusions on different hardware. More specifically, when a unit has no assigned order and something changes in the unit's environment the default behaviour as previously described in section 3.4.4 on page 46 makes the unit perform some action. No network traffic occurs with this event as the change happens on all clients and these all reach the same end-result.

Randomisations in the Stratagus engine is a noteworthy issue. To avoid the network traffic associated with agreeing on a seed to the random number generator the engine uses a static seed which never changes. This implies that if a player issues the exact same orders at the exact same time in two different games on the same map, the two games evolve exactly alike. This is because the estimated damage and other factors which rely on randomisations always result in the same.

6.2.2 Unit control mechanisms

To enable our game tree-based module to control individual units we investigated how the engine handles units and their assigned orders. The following methods which handles units in networked games was found:

SendCommandMove Accepts a pointer to a unit which should be moved and a pair of coordinates to move to. The path to the target location is calculated by the pathfinding algorithm, the A*-algorithm [Wikipedia, 2005a], incorporated in the engine. Technically, this method changes the state of the unit to a *move* state which makes the control loop of the engine move the unit according to the rules of the actual game.

SendCommandStandGround This method takes a pointer to a unit. This unit's state changes to the *standground* state. When trying to override the default behaviour of the units in Wargus this is the method used to make them stand still.

SendCommandAttack This method accepts two pointers to units. One is the aggressor and the other is the target. Also accepted is a pair of coordinates which indicate whether the attack is an ordinary attack on an enemy unit or an attack move command. In the case of a zero unit pointer this is interpreted as an attack move command. This method is thus used to issue attack actions against enemy units or to issue attack-move actions which move the unit to the specified location and engages all enemies encountered along the way.

The engine implements an order queue for each unit. Meaning that a player can issue several orders to a unit and this unit will then complete these orders in turn. All the above methods also accept an integer indicating whether or not the order being assigned should empty the order queue and insert the current order as the next to perform. We always flush the queue, because we derive actions continually.

Several other methods for assigning orders to units exist in the engine. These control other aspects of the units e.g. following other units, constructing buildings or harvesting resources. We do not use these in our module for handling SSC so these will not be presented here.

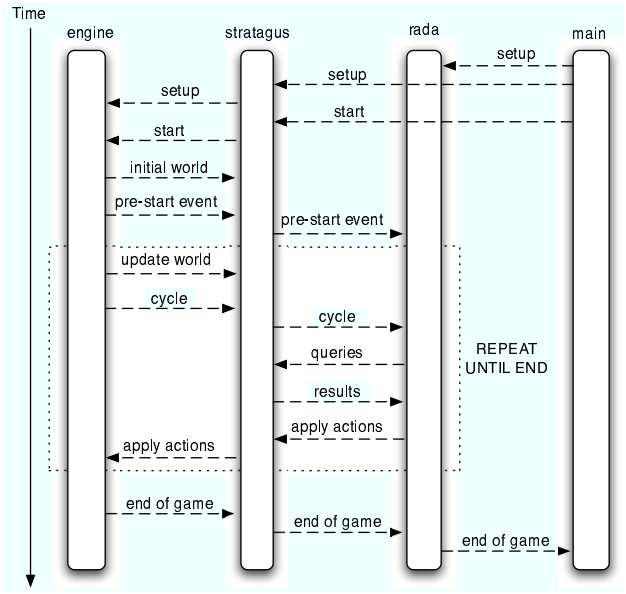

Figure 6.1: A callgraph depicting the overall communication between the engine and our module

6.3 Integration with the engine

For realising our game tree-based module we implemented an automatic propagation of much data and changes herein from the engine to our module. The propagation is done once per game cycle and once the engine is initialised the propagation happens relatively fast, because only changes are propagated. Changes in non-static map-topology such as forests harvested were ignored while the static map-topology such as gold-mines and rocks was included. A set of callbacks was inserted in the control loop of the engine. These send events back to our module accordingly.

6.3.1 Execution path

In figure 6.1 a simplified call graph is depicted and shows the nature of our integration with the engine. The four modules depicted are the engine and three Java packages called `stratagus`, `rada` and `main`. The `stratagus` package encapsulates the engine in an easily accessible Java representation along with a set of methods needed for interaction with the engine. The `rada` package contains two essential elements; a set of auxiliary classes allowing for game tree construction and a set of classes which contains variants of the game tree-based methods. The `main` package is solely responsible for configuring the two Java packages.

As depicted in the figure the only package to ever communicate directly with the engine is the stratagus package. This was done to separate our module from the engine. All communication between the rada package and the engine thus goes through the stratagus package.

All evaluations follow the same pattern. The main package configures the rada and stratagus packages. The stratagus package then configures the engine to suit the needs of the main component – including informing the engine which objects to use for callback targets. The main component sends a start signal to the stratagus package which is propagated to the engine. The engine then propagates the world-model to the stratagus package and sends an event which indicates that the engine is about to start. This event is propagated to the rada package which allows our code to perform any needed pre-game computations on the map – such as map analysis, see section 3.4.2 on page 45. When these computations are complete the engine resumes execution and initiates the game. For each game cycle the engine updates the world representation and sends an event to the stratagus package which propagates this information on to the rada package. The rada package then queries the stratagus package for all data needed to perform the computations at hand. The calculated actions are then applied to the engine. This cycle of update/query/apply continues until a game result has been reached which results in the end-of-game event being propagated all the way back to the main component.

6.3.2 The C to Java link

We have implemented a JNI, [Sun Microsystems, 2003], module which is embedded in the Stratagus engine. It contains several references to JNI data structures allowing the engine to access fields and call methods in the Java virtual machine. The C code references objects from the stratagus Java package. These objects are used to notify the stratagus package of newly created units, updating the location of existing units, removing dead units, managing controlled groups of units and so on.

The Stratagus package contains several methods with the native keyword. Meaning that their implementation is given in C but that the method is callable from Java. The noteworthy of these methods are the unit manipulation methods called `attack`, `move` and `standground` which directly translate their parameters to suitable parameters for their C counterparts as described in section 6.2.2 on page 83.

6.4 Java Packages

Our primary java packages are the `stratagus` and `rada` packages. These packages are described briefly in the following. Full design and implementation details of both java packages are given in appendix C on page 129 along with the game tree construction-algorithms.

6.4.1 Stratagus Java package

The Stratagus package encapsulates the engine enabling future users to use the Wargus game as a testbed for Game AI modules. Furthermore, the JNI module and the modifications we have made to the engine completely hide the engine-specific details of network and unit control from the user. Therefore, this package enables users to quickly implement and test a Game AI module without worrying about engine-specific issues.

The Stratagus package contains an interface which must be implemented by its user. This interface provides the callbacks needed by the engine to propagate events to the implementing object.

6.4.2 Rada Java package

This package[1] uses the stratagus package described above to handle SSC situations. In this package we have created an `experiment` framework which enables fast implementation and testing of methods for handling SSC situations. This is realised by means of an experiment interface and an experiment factory. The experiment interface contains several methods. Some are designed for the game tree-based experiments and others are general purpose. The framework handles among other things the setup of the stratagus package and the recording of situation values (SVs). We have implemented all experiments using this framework. Another part of this experiment framework includes the three rating methods we have used in this project. Two of these rating methods were implemented as a part of the experiment framework. The final rating method, the neural network, was included in the package using the Joone, [Joone, 2005], framework which provided the necessary functionality for rating the game tree nodes.

The package furthermore contains many supporting classes enabling us to handle the game tree construction efficiently. Among these are fast accessor methods to units and fast derivation of game tree child nodes. The game tree algorithm constructs the tree to a specified depth and assign orders to each unit. The game tree construction-algorithm is capable of assigning orders to units at any given game

[1] The name, rada, is more or less randomly chosen.

cycle x as long as x is included in the tree. Meaning, the tree can be built y game cycles ahead and orders to units can be assigned x game cycles ahead. The only requirement is that $x < y$.

To handle the rule sequences used for pruning of nodes we have created a standard way of implementing these realised by means of a rule interface. A factory-class, as described in [Gamma *et al.*, 1994, Chap. 3], was implemented as a way of statically instantiating the rules described in section 5.4 on page 64.

Chapter 7

Results

"Success is getting what you want
happiness is wanting what you get"
- Dave Gardner

In this chapter the setup for the evaluation of the game tree method introduced in chapter 5 on page 53 is presented along with the obtained results. To automate the evaluation of our method we have chosen to create several maps. Each contains one group of human units and one group of orc units positioned in a small scale combat (SSC) situation. The situation value (SV) of these situations is then measured as described in section 5.6 on page 79.

Firstly, the setup for our evaluations is presented. Secondly, the results are presented along with a discussion. Thirdly, a subset of the performed evaluations is recorded as movies and these will be presented accompanied by a discussion on the perceived execution time of our method. Lastly, the tables referenced from this chapter is presented.

7.1 Setup

To measure the different variants of our game tree method we have chosen to evaluate a number of experiments. Recall from section 5.6.2 on page 80 that we have identified 18 variants of the game tree-based methods. These variants will be described in the following section and will be used as basis for measuring the game tree-based method against the built-in methods of Wargus.

All evaluations have been performed on two 2.4 GHz Pentium 4s with 512 MB RAM. We noticed no slowdowns in the game while using the game tree methods. This hardware is adequate for running the method in real time.

The selected experiments are described in the following. Afterwards the SSC situations wherein the evaluation takes place are presented.

7.1.1 Experiments

The following experiments which implement different variants of the game tree method have been chosen for evaluation. These experiments were chosen to portray and measure the overall performance of several rating methods and rule sequences against the Wargus built-in handling of SSC situations.

KNNGTall-experiment uses the *K* nearest neighbour as rating method. It only rates the children of the root.

KNNGTavg-experiment uses the *K* nearest neighbour as rating method. It rates the leaves in the tree. For each internal node it assigns the average of this node's childrens' value.

NNGTall-experiment uses the neural net as rating method. It only rates the children of the root.

NNGTavg-experiment uses the neural net as rating method. It rates the leaves in the tree. For each internal node it assigns the average of this node's childrens' value.

TVGTall-experiment uses the threat matrix rater (TMR) as rating method. It only rates the children of the root.

TVGTavg-experiment uses the TMR as rating method. It rates the leaves in the tree. For each internal node it assigns the average of this node's childrens' value.

The different rating methods were described in section 5.5 on page 75. Each of the game tree-based experiments above are each run with the three different rule sequences for pruning the tree. These were described in section 5.4.3 on page 73 and are the *Focus Fire*, *AttackKNearest* and the *Ranged Assist*-sequences.

We have chosen to hold the performance of our method up against two experiments. These are called the *built-in* methods as they only use simple actions available from the engine. They simulate the way Wargus handles units in SSC situations. These are therefore applicable for investigating whether the game tree-based methods can improve the current handling of SSC situations.

AttackNearest-experiment orders each controlled unit to attack the nearest enemy unit.

AttackMove-experiment orders the controlled units to attack-move towards a calculated centre of the group of enemy units.

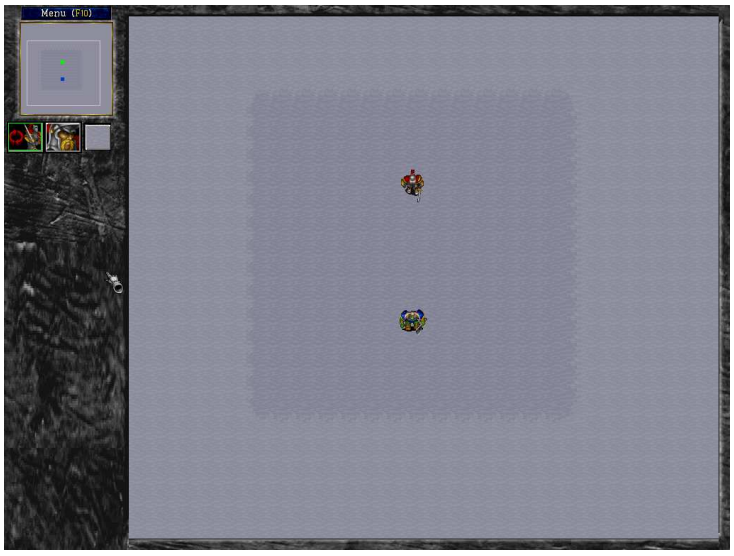

Figure 7.1: A simple SSC situation with two opposing melee units called *Scen1vs1*

7.1.2 SSC situations

All of the designed SSC situations have been created with the map editor provided by the engine and each is depicted in figure 7.1 to 7.5. Each of the SSC situations were placed in different maps. On these maps all experiments fought against the built-in experiments. In the following each situation will be described. Recall from section 5.6 on page 79 that we use the situation value (SV) to evaluate the performance of an experiment. We record the SV at the end of an evaluation as this value depicts the strength of the winning faction.

Scen1vs1 In figure 7.1 a very simple SSC situation is shown. Each side controls only one melee-unit. No environment features exist. This situation is used primarily as a test situation to determine whether each experiment is able to perform well in very simple situations. Since each side is equally strong and the situation is very simple the winner is expected to be decided randomly. The end SV is expected to be approximately 0 in this situation. This situation is referred to as *Scen1vs1* in the results.

Scen3vs2 Figure 7.2 on the next page shows another SSC situation. Here two human footmen combat three orc grunts. No environment features are placed on the map. The orc faction is very strong compared to the human faction. In this situation the winner is expected to be the Orcs since the human faction is outnumbered. Therefore, the SV of the orc faction should clearly

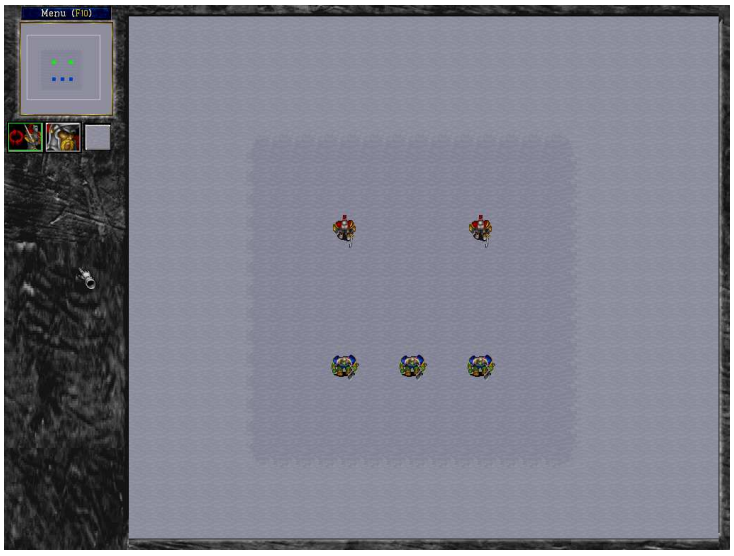

Figure 7.2: Two melee units opposing three melee units called *Scen3vs2*

indicate that the human faction loses. This map is referred to as *Scen3vs2* in the results.

Scen7vs7 Figure 7.3 on the following page shows an SSC situation typical of those encountered in an RTS game. The two groups of units are of equal size. Each faction controls four melee units and three ranged units. The environment contains no environmental features. This situation is expected to be complex enough to show the diversity of the different methods for solving SSC. Therefore, the winner is more dependent on the optimality of the method used than the situations previously described. This map is referred to as *Scen7vs7* in the results.

archer-ambush In figure 7.4 on the next page yet another SSC situation is depicted. Each group of units consists of two melee units and four ranged units. The environment consists of small narrow paths between the two groups. The ranged units have high importance, because if each narrow path is blocked by a melee unit the ranged units can attack from a distance while being out of reach from the enemy melee units. As above, this situation is expected to be complex enough to show the diversity of the methods used for SSC. This map is referred to as *archer-ambush* in the results.

Captured In figure 7.5 on page 93 the last SSC situation is shown. In this situation the human faction controls four melee units and one ranged unit. The orc faction controls three melee units and two ranged units. The human faction

Figure 7.3: Two squads with four melee and three ranged units each called *Scen7vs7*

Figure 7.4: Two squads with two melee and four ranged units each called *archer-ambush*

Figure 7.5: Three melee and two ranged units surrounded by four melee units and one ranged unit called *Captured*

is stronger than the orc faction. The orc units are surrounded by the human units. The winner is expected to be the Humans and the end SV for the human faction should indicate this. This map is referred to as *captured* in the results.

We consider the *Scen7vs7* and *archers-ambush* to be the most interesting situations as they are *fair* to both factions. The SV is 0 at the beginning of the situations and none of the groups have any map specific advantages. The situations called *Captured* and *Scen3vs2* are on the other hand *unfair* situations. The starting SV of *Scen3vs2* for the Orcs is 1, thus the Orcs are the strongest faction at the start of the situation. In *Captured* the starting SV is 0 but the composition of the groups and start positions favour the Humans.

7.2 Questions

We wish to answer the following questions in order to discuss the obtained results.

1. Do any of the experiments perform erratically in *Scen1vs1*? I.e. does the SV measured at the end of the situation provide a clear winner?

2. Do any of the experiments when controlling the orc units lose in *Scen3vs2*? I.e. was the SV negative for the Orcs?

3. In *Scen3vs2* do any of the experiments win when controlling the human units? I.e. was the SV positive for the Humans?

4. Do any of the experiments when controlling the human units lose on *Captured*? I.e. was the SV negative for the Humans?

5. In *Captured* do any of the experiments win when controlling the orc units? I.e. was the SV positive for the Orcs?

6. Which built-in experiment is the best in the designed SSC situations?

7. Do any of our game tree-based experiments outperform the built-in experiments in SSC situations?

8. Which rule sequence performs best?

9. Which rating method performs best?

Questions 1, 2, 3, 4 and 5 above serve as tests since answers to these question should be no. If for a given experiment the answer is yes to any of the five questions this specific experiment deserves greater analysis.

To evaluate the game tree-based experiments we need to know which built-in experiment is the most optimal. Question 6 should provide this information.

We expect that we can answer yes to question 7 making the extra work involved with a game tree-based method a worthwhile investment. This would mean that the built-in methods were outperformed by the game tree-based methods.

The answers to questions 8 and 9 are of interest when examining the areas where more work could be conducted.

To provide a foundation to answer the above questions we have chosen to compare each game tree-based experiments to each of the built-in experiments. The built-in experiments are also compared to each other to determine the optimal one. Each experiment combination is evaluated in every designed SSC situation. We record the end SV in the game cycle where a winner has been found, i.e. only one faction remains on the map. All collected results can be seen in appendix D on page 144.

Recall from section 6.2 on page 82 that the random number generator in the engine always returns the same sequence of random numbers. Due to this fact we have chosen to run each evaluation only once and record the result, because running the evaluations multiple times would provide identical results.

7.3 Presentation of results

Here, we will answer the nine questions formulated previously. All collected results can be seen in appendix D on page 144.

1. As seen in tables 7.1 on page 100 and 7.2 on page 101 the end SV of all evaluations on *Scen1vs1* is approximately 0 as expected. Thus, none of the experiments perform erratically on *Scen1vs1*.

2. In table 7.3 on page 102 it is shown that all experiments win when they control the orc units on *Scen3vs2*. This was expected since this SSC situation favours the orc side.

3. Table 7.4 on page 103 tells that all experiments lose when controlling the human units on *Scen3vs2*. This was also expected.

4. Table 7.5 on page 104 shows that all experiments win when controlling the human units on *Captured*. It was also expected.

5. Table 7.6 on page 105 tells that all experiments lose when controlling the orc units on *Captured*, as expected.

6. Recall from section 7.1.1 on page 89 that we evaluate the optimality of an experiment by comparing it to the built-in experiments. In table 7.7 on page 105 the results of evaluating the built-in experiments are shown. Based on the average SV we conclude that the *AttackMove*-experiment is the overall most optimal built-in experiment. As the *AttackMove*-experiment performs better than the *AttackNearest*-experiment in the fair scenarios this experiment is the most interesting when evaluating the game tree-based methods in these scenarios. When looking at the unfair scenarios we are interested in the *AttackNearest*-experiment.

7. Tables 7.8 on page 106 and 7.9 on page 107 show the results of the game tree-based experiments against the two built-in experiments. Table 7.8 on page 106 shows the game tree experiments vs. the *AttackMove* experiment. Table 7.9 on page 107 shows the game tree experiments vs. the *AttackNearest* experiment.

   From table 7.8 on page 106 it can be seen that the experiment performing best overall including in the unfair situations is the *KNNGTall* experiment with the *RangedAssist* rule sequence. The *TVGTall* experiment with the *AttackKNearest* rule sequence performs best in the fair situations though.

   Table 7.9 on page 107 shows that experiment *NNGTall* with the *AttackKNearest* rule sequence performs best overall including the fair situations.

In the unfair situations the *KNNGTavg* experiment with the *RangedAssist* rule sequence performs best.

If both tables are taken into consideration it can be seen that the *TVGTall* experiment with the *AttackKNearest* rule sequence overall averagely outperforms both built-in experiments. The average is $\frac{0,105+0,049}{2} = 0,077$. Even though the *NNGTall* experiment with the *AttackKNearest* rule sequence loses to the *AttackMove*-experiment its overall performance is quite impressive with an average of $\frac{0,236-0,153}{2} = 0,042$.

8. Table 7.10 on page 107 shows the performance of the rule sequences averaged. It can be seen that the *AttackKNearest* rule sequence overall performs best. This rule sequence also performs best in the fair situations. But the *RangedAssist* rule sequence performs best in the unfair situations.

9. Table 7.11 on page 108 reveals the performance of the rating methods averaged. It can be seen that the best rating method overall and also in the fair situations is the threat matrix rating method which only rates the immediate children of the root (experiment *TVGTall*). In the unfair situations the *K* nearest neighbour rating method which averages the values of the internal nodes (experiment *KNNGTavg*) performs best.

The answers to questions 7, 8 and 9 raise a few issues.

Firstly, determining the optimal strategy in a SSC situation depends on the opposing player's strategy, because the game tree experiments do not perform equally well when opposing the built-in experiments.

Secondly, choosing the best rule sequence among the designed is difficult since the sequence performing best in the fair situations is not the best in the unfair situations. Meaning, a rule sequence depends on the actual situation as expected.

Thirdly, choosing the best rating method is non-trivial. The performance of designed indicate that the *TMR* performs well in fair situations and the *KNN* performs best in the unfair situations. Again, the rating method depends on the complexity of the situation.

7.4 Presentation of movies

Recall from section 5.4 on page 64 that the time constraint for our method is $\frac{1}{30}$ seconds. This is the guideline ensuring that the player does not notice a slowdown in the game caused by the game tree method.

The recorded movies are available on the enclosed CD. These are described in the movies section on the CD. The movies have some graphical flaws resulting

from very technical linker and compiler problems of the engine and the Mac OS X 10.4 operating system.

We have recorded four movies. These illustrate how the choice of rating method and rule sequence influences the end-result of an evaluation. No slowdowns in either of the movies can be seen.

1. *KNNallAKNasHumansVSAttackMoveOnScen7vs7.mov* shows the course of an evaluation on *Scen7vs7*. The *KNNGTall* experiment with the *Attack-KNearest* rule sequence as Humans fights the *AttackMove* experiment which plays Orcs. Notice there are no slowdowns in the game.

   From the table D.2 in appendix D.2 on page 146 we see that the end SV for this evaluation is $-0,128$. Meaning that the game tree based method lost.

2. *TVavgAKNasHumansVSAttackMoveOnScen7vs7.mov* shows almost the same evaluation as above. The difference is that the *TVGTavg* experiment is used instead of the *KNNGTall* experiment. Again, notice there are no slowdowns.

   From table D.2 in appendix D.2 on page 146 we see that the end SV for this evaluation is $0,119$. Meaning that the game tree based method won.

3. *TVallAKNasOrcsVSAttackNearestOnScen7vs7.mov* shows the course of an evaluation on *Scen7vs7* where the *TVGTall* experiment with the *Attack-KNearest* rule sequence as Orcs fights the *AttackNearest* experiment playing Humans. Again, there are no visual slowdowns.

   From table D.11 in appendix D.2 on page 146 we see that the end SV of this evaluation is $0,85$. Meaning that the game tree based method won quite big.

4. *TVallAKNasOrcsVSAttackNearestOnScen7vs7.mov* shows almost the same evaluation as above. The difference is that the *FocusFire* rule sequence is used instead of the *AttackKNearest* rule sequence. No slowdowns can be seen.

   From table D.10 in appendix D.2 on page 146 we see that the end SV of this evaluation is $-0,925$, Meaning that the game tree based method lost.

Items 1 and 2 illustrate the difference in the end result when choosing different rating methods. In this case choosing the right rating method is therefore a crucial factor. This choice decides whether the game tree-based method wins or loses.

Items 3 and 4 above show the difference in the end result when choosing different rule sequences. As with item 1 and 2 this choice of rule sequence determined which side won.

7.5 Discussion

As suspected and described in section 5.2.2 on page 60 the solutions to all of the issues presented in that section influence the optimality of a game tree-based method. Especially the choice of rating method and rule sequence influence the end-result, as illustrated in the previous section.

The only requirement of representing a game state in a node is that it must provide sufficient information for the rating methods and rule sequences to perform well. The questions answered in section 7.3 on page 95 indicate that our representation contains enough information, because we are able to outperform the built-in experiments.

From table 7.9 on page 107 it can be seen that the *AttackKNearest* rule sequence almost always outperforms the *AttackNearest* experiment regardless of rating method. This leads us to conclude that it is more favourable to estimate the outcome of attacking several enemy unit than just attacking the nearest. This is an interesting fact. It means that it is worthwhile to consider the K nearest enemy units before actually choosing one to attack.

We have so far introduced our method as a support routine by enabling both players and participants to use the method. Restricting the availability to participants would allow current games to handle SSC situations near-optimally while not changing the gameplay for the player. This restriction enables the game-developers to adjust the difficulty level by changing rule sequences, rating methods or the amount of game cycles predicted by the game tree.

7.5.1 A problem with the game tree method

Figure 7.6 on the following page shows a situation where the game tree method performs erratically. It shows the *KNNGTall* experiment with the *FocusFire* rule sequence as the Orcs in the situation called *Scen3vs2* opposing an experiment where all units stand still. Recall that the Orcs are expected to win in this situation. The game tree method does win as expected but it takes a lot longer than necessary. Figure 7.6 on the next page shows the grunt labelled with a 1 as the only one attacking the enemy footman. The grunts labelled 2 and 3 do nothing. They just stand still behind grunt 1. This is clearly not an optimal handling of the SSC situation as grunts 2 and 3 ought to assist grunt 1 in the attack. Recall from section 5.4.3 on page 73 that the *FocusFire* rule sequence uses the *Attack K lowest HP enemies* rule for melee type units. The erratic behaviour occurs, because this rule sequence ensures that units attack or move towards the K enemy units with the lowest amount of hitpoints. There is only one enemy unit in the depicted situation so the rule constraints grunt 2 and 3 to move closer to the footman allowing them to attack. As we do not have a dynamic path-finding algorithm the

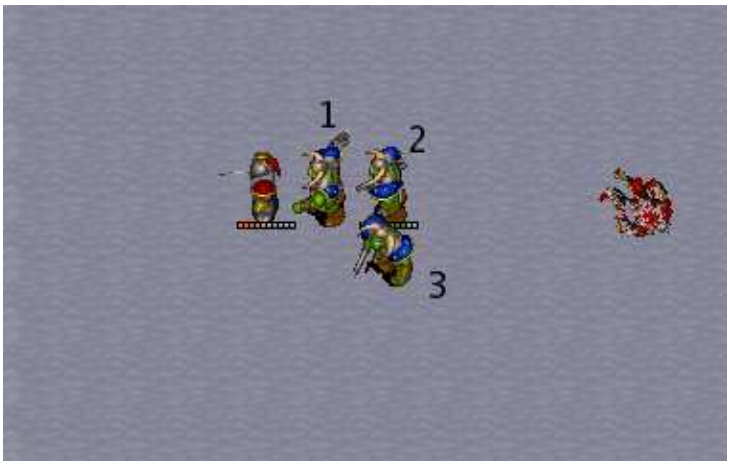

Figure 7.6: A problem with our game tree-based method. Here illustrated by the *KNNGTall* experiment with *FocusFire* rule sequence as the Orcs

rule chooses one adjacent tile which decreases the distance to the target by one. This leaves only one action available to both grunts but as grunt 1 stands in the field chosen by the rule for the grunts to move to grunt 2 and 3 are not allowed to move and thus just stand still.

Situations as this show the importance of designing the rules and the rule sequences correctly. We, the designers of the rules and the sequences, had not foreseen the situation depicted and did not incorporate any handling of such situations. If more work, thought and time was put into the making of the rule sequences such erratic behaviour could be overcome.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	0,033	0,033
TVGTall w. AKN	0,033	0,033
TVGTall w. RA	0,033	0,033
KNNGTall w. FF	0,033	0,033
KNNGTall w. AKN	0,033	0,033
KNNGTall w. RA	0,033	0,033
NNGTall w. FF	0,033	0,033
NNGTall w. AKN	0,033	0,033
NNGTall w. RA	0,033	0,033
TVGTavg w. FF	0,033	0,033
TVGTavg w. AKN	0,033	0,033
TVGTavg w. RA	0,033	0,033
KNNGTavg w. FF	0,033	0,033
KNNGTavg w. AKN	0,033	0,033
KNNGTavg w. RA	0,033	0,033
NNGTavg w. FF	0,033	0,033
NNGTavg w. AKN	0,033	0,033
NNGTavg w. RA	0,033	0,033
AttackNearest	0,033	0,033
AttackMove	-0,033	0,033

Table 7.1: Situation *Scen1vs1*, Human side. FF is *FocusFire*, AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are approximately zero.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	-0,033	0,033
TVGTall w. AKN	-0,033	0,033
TVGTall w. RA	-0,033	0,033
KNNGTall w. FF	-0,033	0,033
KNNGTall w. AKN	-0,033	0,033
KNNGTall w. RA	-0,033	0,033
NNGTall w. FF	-0,033	0,033
NNGTall w. AKN	-0,033	0,033
NNGTall w. RA	-0,033	0,033
TVGTavg w. FF	-0,033	0,033
TVGTavg w. AKN	-0,033	0,033
TVGTavg w. RA	-0,033	0,033
KNNGTavg w. FF	-0,033	0,033
KNNGTavg w. AKN	-0,033	0,033
KNNGTavg w. RA	-0,033	0,033
NNGTavg w. FF	-0,033	0,033
NNGTavg w. AKN	-0,033	0,033
NNGTavg w. RA	-0,033	0,033
AttackNearest	-0,033	0,033
AttackMove	-0,033	-0,033

Table 7.2: *Scen1vs1,* Orc side. FF is *FocusFire,* AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are approximately zero.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	1,717	0,900
TVGTall w. AKN	1,883	0,944
TVGTall w. RA	1,717	1,033
KNNGTall w. FF	1,767	1,033
KNNGTall w. AKN	1,800	1,450
KNNGTall w. RA	1,717	1,550
NNGTall w. FF	1,189	1,967
NNGTall w. AKN	1,733	1,467
NNGTall w. RA	1,356	1,967
TVGTavg w. FF	1,783	1,950
TVGTavg w. AKN	1,800	1,533
TVGTavg w. RA	1,767	1,517
KNNGTavg w. FF	1,517	1,133
KNNGTavg w. AKN	1,800	1,033
KNNGTavg w. RA	1,733	0,900
NNGTavg w. FF	1,883	1,289
NNGTavg w. AKN	1,767	1,467
NNGTavg w. RA	1,683	1,750
AttackNearest	1,733	1,033
AttackMove	1,850	1,517

Table 7.3: *Scen3vs2*, Orc side. FF is *FocusFire*, AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are positive meaning that the Orcs won as expected.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	-1,256	-1,233
TVGTall w. AKN	-1,967	-1,967
TVGTall w. RA	-1,356	-1,800
KNNGTall w. FF	-1,256	-1,222
KNNGTall w. AKN	-1,967	-2,017
KNNGTall w. RA	-1,322	-2,017
NNGTall w. FF	-1,967	-1,933
NNGTall w. AKN	-1,917	-2,017
NNGTall w. RA	-1,322	-2,017
TVGTavg w. FF	-1,967	-2,017
TVGTavg w. AKN	-1,967	-2,017
TVGTavg w. RA	-1,967	-2,017
KNNGTavg w. FF	-1,967	-1,233
KNNGTavg w. AKN	-1,967	-2,017
KNNGTavg w. RA	-1,256	-2,017
NNGTavg w. FF	-1,967	-1,233
NNGTavg w. AKN	-1,356	-2,017
NNGTavg w. RA	-1,967	-1,222
AttackNearest	-1,733	-1,850
AttackMove	-1,033	-1,517

Table 7.4: *Scen3vs2*, Human side. FF is *FocusFire*, AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are negative meaning that the Humans lost as expected.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	0,436	1,671
TVGTall w. AKN	1,271	1,189
TVGTall w. RA	0,500	1,018
KNNGTall w. FF	0,193	1,039
KNNGTall w. AKN	0,464	1,039
KNNGTall w. RA	0,739	1,671
NNGTall w. FF	0,514	1,168
NNGTall w. AKN	1,039	1,600
NNGTall w. RA	0,407	1,200
TVGTavg w. FF	0,421	0,664
TVGTavg w. AKN	0,507	1,061
TVGTavg w. RA	0,393	0,536
KNNGTavg w. FF	0,479	1,671
KNNGTavg w. AKN	0,943	1,514
KNNGTavg w. RA	0,607	1,671
NNGTavg w. FF	0,250	1,157
NNGTavg w. AKN	0,179	1,136
NNGTavg w. RA	0,429	1,629
AttackNearest	1,179	1,571
AttackMove	1,471	1,686

Table 7.5: *Captured*, Human side. FF is *FocusFire*, AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are positive meaning that the Humans won as expected.

Game tree-based experiment	AttackNearest	AttackMove
TVGTall w. FF	-1,029	-1,829
TVGTall w. AKN	-1,343	-1,629
TVGTall w. RA	-0,911	-1,543
KNNGTall w. FF	-1,671	-1,471
KNNGTall w. AKN	-0,657	-1,700
KNNGTall w. RA	-0,921	-0,679
NNGTall w. FF	-1,254	-1,339
NNGTall w. AKN	-0,657	-1,857
NNGTall w. RA	-1,393	-1,093
TVGTavg w. FF	-1,757	-1,296
TVGTavg w. AKN	-1,039	-1,643
TVGTavg w. RA	-0,514	-1,232
KNNGTavg w. FF	-1,136	-1,529
KNNGTavg w. AKN	-1,050	-1,200
KNNGTavg w. RA	-0,586	-1,318
NNGTavg w. FF	-0,793	-1,829
NNGTavg w. AKN	-1,061	-1,771
NNGTavg w. RA	-1,814	-1,657
AttackNearest	-1,179	-1,471
AttackMove	-1,571	-1,686

Table 7.6: *Captured*, Orc side. FF is *FocusFire*, AKN is *AttackKNearest* and RA is *RangedAssist*. The important fact in this table is that all values are negative meaning that the Orcs lost as expected.

Built-in experiment	SV	Fair	Unfair
AttackMove	0,060	0.313	-0.179
AttackNearest	-0,060	-0.313	0.179

Table 7.7: End situation value of AttackMove and AttackNearest vs. each other (Averaged over sides and situations). The important fact in this table is that the AttackMove performs slightly better than the AttackNearest on the average.

Game tree-based experiment	SV	Fair	Unfair
TVGTall w. FF	0,013	0,134	-0,123
TVGTall w. AKN	0,049	0,458	-0,366
TVGTall w. RA	-0,081	0,123	-0,323
KNNGTall w. FF	0,039	0,225	-0,155
KNNGTall w. AKN	-0,149	-0,045	-0,307
KNNGTall w. RA	0,067	-0,002	0,131
NNGTall w. FF	-0,186	-0,400	-0,034
NNGTall w. AKN	-0,153	-0,160	-0,202
NNGTall w. RA	-0,099	-0,254	0,014
TVGTavg w. FF	-0,059	0,024	-0,175
TVGTavg w. AKN	-0,191	-0,180	-0,267
TVGTavg w. RA	-0,196	-0,160	-0,299
KNNGTavg w. FF	-0,094	-0,240	-0,011
KNNGTavg w. AKN	-0,175	-0,243	-0,167
KNNGTavg w. RA	-0,189	-0,252	-0,191
NNGTavg w. FF	-0,165	-0,234	-0,154
NNGTavg w. AKN	-0,297	-0,389	-0,296
NNGTavg w. RA	-0,201	-0,594	0,125

Table 7.8: End SV result of game tree-based experiments vs. AttackMove (Averaged over sides and situations). The important fact in this table is that TVGTall and KNNGTall outperforms AttackMove on average in two out of three situations.

Game tree-based experiment	SV	Fair	Unfair
TVGTall w. FF	-0,189	-0,44	-0,033
TVGTall w. AKN	0.105	0,302	-0,039
TVGTall w. RA	-0.132	-0,319	-0,013
KNNGTall w. FF	-0.431	-0,835	-0,242
KNNGTall w. AKN	0.028	0,160	-0,090
KNNGTall w. RA	-0.320	-0,852	0,053
NNGTall w. FF	-0.281	-0,322	-0,380
NNGTall w. AKN	0.236	0,539	0,050
NNGTall w. RA	-0.078	0,043	-0,238
TVGTavg w. FF	-0.127	0,062	-0,380
TVGTavg w. AKN	-0.178	-0,269	-0,175
TVGTavg w. RA	-0.259	-0,568	-0,080
KNNGTavg w. FF	-0.421	-0,788	-0,264
KNNGTavg w. AKN	0.062	0,224	-0,069
KNNGTavg w. RA	0.140	0,226	0,125
NNGTavg w. FF	-0.128	-0,162	-0,157
NNGTavg w. AKN	0.051	0,245	-0,118
NNGTavg w. RA	-0.285	-0,296	-0,417

Table 7.9: End SV result of game tree-based experiments vs. AttackNearest (Averaged over sides and situations). The important fact in this table is that KNNGTavg outperforms the AttackNearest on average in two out of three situations.

Rule sequence	SV	Fair	Unfair
FocusFire	-0,165	-0,248	-0,174
AttackKNearest	-0,043	0,053	-0,170
RangedAssist	-0,159	-0,280	-0,093

Table 7.10: End SV result of rule sequences vs. the built-in experiments (Averaged over sides, situations and opposing experiments). The important fact in this table is that the AttackKNearest rule-sequence performs better than the two other rule-sequences.

Rating method	SV	Fair	Unfair
TVGTall	-0,039	0,043	-0,149
KNNGTall	-0,127	-0,225	-0,102
NNGTall	-0,086	-0,092	-0,132
TVGTavg	-0,161	-0,181	-0,229
KNNGTavg	-0,135	-0,254	-0,093
NNGTavg	-0,160	-0,238	-0,170

Table 7.11: End SV result of rating methods vs. the built-in experiments (Averaged over sides, situations and opposing experiments). The important fact this table shows is that the TVGTall rating method performs overall best.

Chapter 8

Future work

"Choice. The problem is choice"
- Neo

In this chapter we wish to portray the nature of enhancements which could be made in our project. We will describe the reason for the enhancements and propose ways to obtain these.

8.1 Engine enhancements

When designing a system for real time use several optimisations may be needed in order to uphold the real time constraints. Even though some optimisations were implemented in this project additional contributions could be made to make the system even more resilient to the time limit imposed. The primary motivations for the optimisations are that the game tree can be built to a deeper level and that more game states can be considered in the model.

As our project is written in Java and integrated with the engine as described in section 6 on page 81 a possible optimisation would be to integrate the game tree and its support functions within the engine. This optimisation would make the data-propagation between C and Java unnecessary and would thus result in a quicker solution. Also, an exact representation would be available at all times and no considerations about what to propagate would be needed. Some of the code should be redesigned due to the object hierarchy in the project and C's non-object oriented facilities.

8.2 Machine learning accuracy

When using machine learning techniques as described in section 5.5 on page 75 their accuracy can always be improved. As we train the machine learning algo-

rithms with handmade training examples and associated values the accuracy of the algorithms depend solely upon the quality of the examples. A more thorough analysis of the problem domain would very likely increase the quality of the training examples and thereby increase the accuracy of the rating methods. In section 5.5.3 on page 78 we also described how to choose actions. Other methods for choosing game tree nodes and assign values to the internal nodes could be investigated.

The threat values constituting the threat matrix are calculated by a handmade equation as described in section 5.3 on page 60. It could be that we left out some essential features and therefore threat value does not capture the overall value of a unit. We chose the parameters involved due to apparent importance. But whether these are the most important features is left unknown. If a more expressing equation could be found the obtained results could also be improved.

8.3 Improving evaluation methods

The scenario value (SV) described in section 5.6 on page 79 measures the course of an SSC situation. The SV does not incorporate the number of game cycles spent in the situation. Incorporation of game cycles would allow us to assign a lower SV to the erratic behaviour described in section 7.5.1 on page 98 thereby not rating these evaluations as high as they currently are. How this incorporation could be achieved is not obvious to us.

The set of situations produced in the engine's editor-module was made to be able to test our game tree method and to let it battle against the other implemented experiments. By extending the set of SSC situations one could discover whether the end-results of the simulations would produce equally good results on these as well. Also, it would be interesting to implement our method as an actual support routine in Wargus thereby making the game tree method available to the players. It would be interesting to see whether human players would find our method applicable to handle SSC situations.

8.4 Improving the integration with Wargus

Recall from section 5.3 on page 60 that fog of war was ignored. We assumed implicitly that when the tree is about to be built, an actual combat situation is already in process or just about to be. This is partially incorrect. Enemy units not yet seen by the controlled units must not influence the decisions made. The inclusion of fog of war can be achieved by letting the fields under the fog of war be unavailable until these are within sight-range of a friendly unit. Available fields in the snapshot would thus be the fields in sight. Some memory of previously

seen enemy units would have to be incorporated as it would be illogical to ignore enemy units just seen.

The code handling the units implicitly assumes that the controlled units are either of type melee or type ranged. In these two types there are a lot of different units. We chose two for each side which were identical in attributes to make testing and correctness of the algorithms easier. Additionally, the possibility of including flying units was ignored. Since flying units have the same action possibilities as either the melee land unit or the ranged land unit extending the code to include these could be accomplished.

Allowing additional actions for the units such as the ability to cast magic or use siege weaponry upon a target area is technically hard to achieve. The magic or siege attacks are often in effect for a prolonged time. Units standing in the target zone are affected by the magic- or siege-damage for a while. The model cannot currently handle this effect though the design could be extended to model this. The prolonged effects could be modelled as attacks appearing in the tree at intervals. Many of the effects give x points of damage every y'th second. An attack node could be inserted every y'th game cycle and estimate damage to the units within the area of effect accordingly.

8.5 Game tree extensions

As described in section 7.5.1 on page 98 we do not use the path-finding algorithm in the engine. Instead we statically calculate the shortest path from all tiles to all tiles. An extension of the model would naturally be to incorporate the handling of non-static paths. A dynamic pathfinding-algorithm would be needed to handle other units obstructing the path. This algorithm would need to be extremely fast since the shortest paths change dynamically during the calculation of the tree. Some of the experienced problems with the game tree-based method were due to the model's inability to handle dynamic shortest paths. The A* algorithm in the engine could if redesigned be used to give these paths. But whether this would violate the real time constraint is currently unknown.

Another future work topic is the fairly inflexibility of the game tree construction, because the number of game cycles to evaluate is specified statically. Another approach could be to incorporate a detection of time spent in game tree construction allowing it to stop if no more time is available. This would of course introduce some problems. E.g. if the time limit is exceeded and units still exist with no orders. This problem should then be solved reasonably so no units would stand around doing nothing.

As described in section 5.4 on page 64 we model the enemy units as standing still in the game tree. This is imprecise and in all SSC situations the enemy units

move around and attack. The problem is that in the current model the future states are very imprecise. A different method for pruning the tree or a less restrictive model for the enemy unit's behaviour could be investigated.

By designing near-optimal rules and sets of rules the game tree could only consider near-optimal states. Finding near-optimal rules are of course a game specific topic. This problem depends on the actual engine, how complex the environment is and the interdependency of the rules. Sequences of rules are equally hard to design. The order is a very important factor for the success of the rule sequence. We designed a few sample rule sequences by hand as previously described in section 5.4.3 on page 73 which at the time of creation seemed like good and simple strategies. In a further investigation of the area of SSC a good starting point would be to extend this rule set and possibly find better rule sequences. The rule system of our game tree based method could also be extended to include better debugging facilities. This would let us detect the erratic behaviour described in section 7.5 on page 98 and handle this before it appears in the evaluations.

Chapter 9

Conclusion

"I may not have gone where I intended to go
but I think I have ended up where I intended to be"
- Douglas Adams

In this thesis we defined the problem of small scale combat (SSC) situations in real time strategy (RTS) games. We investigated the area of Game AI to provide a basis for understanding the methods used for solving the SSC problem.

We presented an overview of several computer game genres and argued that consistent behaviour of characters in a game world is a contributing factor to the overall quality of a computer game. We examined several behavioural models in different computer game genres. We discussed whether the used methods resulted in consistent character behaviour. Based on these investigations we concluded that in SSC situations an optimal unit behaviour results in a consistent unit behaviour.

We presented different methods for handling SSC situations along with a discussion involving the advantages and disadvantages of these methods. Based on this examination we chose a game tree model as the method for handling SSC situations.

Investigating the game tree method applied to RTS games further, we introduced the notion of timestamped game trees to handle the varying time lengths of actions and concurrent actions. Three issues were identified and handled. The first issue concerning the representation of the tree was handled by modelling a SSC situation as a simplified snapshot of the game state. The second issue regarding the size of the tree was handled by letting sequences of rules reduce the fanout of each node. The last issue involving which states to choose after the tree had been constructed was handled by letting different algorithms rate the game tree nodes. A handcrafted rating method and two machine learning algorithms were designed to perform this rating. As input to the rating methods we introduced the threat matrix which contained all units threat values. The threat value formula was designed to capture units' desirability in a single value.

To evaluate the performance of the game tree-based methods the Open Source RTS game of Wargus was chosen as platform. Wargus was chosen because it is of a commercial comparable quality and we had access to the source code. Furthermore, Wargus contains simple rules for handling SSC situations. These built-in rules were opposed to the game tree variations to investigate which method performed best.

The results obtained show that the performance of the game tree-based methods relies on the actual pruning and rating methods, as expected. Nevertheless, the game tree method outperformed the built-in methods of Wargus. This fact suggests that a game tree method which respects the real time constraints can actually perform better than methods currently used by the commercial computer game industry.

If an optimal rule sequence can be devised by analysing the rules of the targeted game the game tree model can be used to give players the option of autonomously handling SSC situations effectively. This lets the player shift his focus to the high level decisions needed to win in an RTS game. Furthermore, if the game tree method is used solely by the computer controlled opponents the challenge posed by the opponents could be adjusted by using different rating methods or different rule sequences without succumbing to letting the opponent cheat.

The game tree method is not without its limitations. Much work is needed to implement the game tree and to fit the rules and the rating method to the actual game. Also, as rule sequences are used to prune the game tree this method is also sensitive to the ability of the rule designers to understand the game at hand.

To conclude, we found that game trees can be used as an alternative way of obtaining consistent behaviour in SSC situations. But more work could be made in order to increase the performance of the game tree method in this domain.

Bibliography

[Abandonia, 2005] Abandonia. Home of abandonware DOS games, 2005. http://www.abandonia.com. Visited 18. May 2005.

[Adams and Mendler, 2002] David Adams and Michael Mendler. Automated Generation of Dungeons for Computer Games. http://www.dcs.shef.ac.uk/teaching/eproj/ug2002/pdf/u9da.pdf. Visited 21. May, 2002.

[AMAI, 2005] AMAI. Advanced Melee AI, 2005. http://amai.wc3campaigns.com/. Visited 5. July 2005.

[Atari, 2000] Atari. Driver, 2000. http://www.atari.com/driv3r/. Visited 21. June 2005.

[Auran, 1997] Auran. Dark Reign, 1997. http://www.auran.com/games/darkreign/default.htm. Visited 23. August 2005.

[Bethesda Softworks, 1994] Bethesda Softworks. The Elder Scrolls, 1994. http://www.elderscrolls.com/. Visited 20. May 2005.

[Björnson and Marsland, 2001] Yngvi Björnson and Tony A. Marsland. Multi-cut α-β pruning in game-tree search. *Theoretical Computer Science*, 252:177–196, 2001.

[Blizzard, 1995] Blizzard. Warcraft II, 1995. http://www.blizzard.com/war2bne/. Visited 18. May 2005.

[Blizzard, 1998] Blizzard. Starcraft, 1998. http://www.blizzard.com/starcraft/. Visited 31. May 2005.

[Blizzard, 2000] Blizzard. Diablo II, 2000. http://www.blizzard.com/diablo2/. Visited 18. May 2005.

[Blizzard, 2002] Blizzard. Warcraft III, 2002. http://www.blizzard.com/war3/. Visited 31. May 2005.

[Blizzard, 2004] Blizzard. The Story of Warcraft, 2004. http://www.worldofwarcraft.com/info/story/. Visited 22. April 2005.

[Blizzard, 2005] Blizzard. World of Warcraft, 2005. http://www.worldofwarcraft.com. Visited 20. May 2005.

[Brockington, 2000] Mark Brockington. Pawn Captures Wyvern: How Computer Chess Can Improve Your Pathfinding, 2000. http://www.gamasutra.com/features/20000626/brockington_01.htm. Visited 8. July 2005.

[Buckland, 2005] Mat Buckland. AI-Junkie, 2005. http://www.ai-junkie.com/. Visited 19. July 2005.

[Buro and Furtak, 2004] Michael Buro and Timothy M. Furtak. RTS Games and Real-Time AI Research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS), Arlington VA*, 2004.

[Buro, 2002] Michael Buro. ORTS: a hack-free RTS game environment. In *Proceedings of the International Computers and Games Conference*, 2002.

[Buro, 2004] Michael Buro. Call for AI Research in RTS Games. In *Proceedings of the AAAI-04 workshop on AI in games, San Jose*, 2004.

[Callan, 2003] Rob Callan. *Artificial Intelligence*. Palgrave Macmillan, 2003.

[Capcom, 1996] Capcom. Street Fighter Alpha II, 1996. http://www.gamefaqs.com/coinop/arcade/data/583633.html. Visited 16. May 2005.

[Carlisle, 2004] Phil Carlisle. An AI Approach to Creating an Intelligent Camera System. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Champandard, 2003] Alex J. Champandard. *AI Game Development - Synthethic Creatures with Learning and Reactive Behaviours*. New Riders, 2003.

[Codemasters, 1998] Codemasters. TOCA race driver II, 1998. http://www.codemasters.com/tocaracedriver2/index.php?territory=EnglishUSA. Visited 21. June 2005.

[Coulouris *et al.*, 2001] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed System – Concepts and Design*. Addison Wesley, 2001.

[CSteam, 1999] CSteam. Counterstrike, 1999. http://counterstrike.sierra.com/. Visited 16. May 2005.

[Dalmau, 2003] Daniel Sanchez-Crespo Dalmau. *Core Techniques and Algorithms in Game Programming*. New Riders, 2003.

[Davis, 1999] Ian Lane Davis. Strategies for Strategy Game AI. *AAAI Spring Symposium Technical Report SS-99-02*, 1999.

[Design, 2003] Core Design. Tomb Raider: Angel of Darkness, 2003. http://www.tombraiderchronicles.com/tng/. Visited 22. June 2005.

[EA Games, 2000] EA Games. The Sims, 2000. thesims.ea.com/. Visited 21. June 2005.

[EA-Sports, 2004] EA-Sports. Madden NFL 2004, 2004. http://www.easports.com/games/madden2004/home.jsp. Visited 16. May 2005.

[Ensemble Studios, 1997] Ensemble Studios. Age of Empires, 1997. http://www.ensemblestudios.com/aoe.htm. Visited 31. May 2005.

[Epic Games, 2004] Epic Games. Unreal, 2004. http://www.unreal.com/. Visited 20. June 2005.

[Firaxis Games, 2001] Firaxis Games. Civilization III, 2001. http://www.civ3.com/. Visited 14. July 2005.

[Fogel, 2002] David Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann Publishers, 2002.

[Free Software Foundation, 1991] Free Software Foundation. GNU General Public License version 2, 1991. http://www.gnu.org/copyleft/gpl.html. Visited 22. April 2005.

[Free Software Foundation, 1999] Free Software Foundation. GNU Lesser General Public License version 2.1, 1999. http://www.gnu.org/copyleft/lesser.html. Visited 27. July 2005.

[Fu and Houlette, 2004] Dan Fu and Ryan Houlette. The Ultimate Guide to FSMs in Games. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[GameFAQs, 1993] GameFAQs. NBA games, 1993. http://www.gamefaqs.com/search/index.html?game=NBA&x=0&y=0. Visited 21. June 2005.

[GameFAQs, 1996] GameFAQs. FIFA, 1996. http://www.gamefaqs.com/search/index.html?game=fifa&x=0&y=0. Visited 21. June 2005.

[GameFAQs, 2005] GameFAQs. Video game FAQs, 2005. http://www.gamefaqs.com. Visited 18. May 2005.

[GameSpot, 2005] GameSpot, 2005. http://www.gamespot.com. Visited 7. July 2005.

[Gamma *et al.*, 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Goodrich and Tamassia, 1998] Michael T. Goodrich and Roberto Tamassia. *Data Structure and Algorithms in Java*. John Wiley and Sons, 1998.

[Grimani, 2004] Mario Grimani. Wall Building for RTS Games. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Gustafsson, 2004] Tommi Gustafsson. Warcraft III AIs, 2004. http://users.tkk.fi/~tgustafs/wc3ai.html. Visited 6. July 2005.

[Hubbard, 2002] Craigh Hubbard. PC Gamer September article. *Imagine Media*, 2002.

[ID Software, 1999] ID Software. Quake III Arena, 1999. http://www.idsoftware.com/games/quake/quake3-arena/. Visited 18. May 2005.

[Ierusalimschy *et al.*, 2003] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua 5.0 Reference Manual. *Technical Report MCC-14/03, PUC-Rio*, 2003. http://www.lua.org. Visited 22. April 2005.

[IGN, 2005] IGN, 2005. http://www.ign.com. Visited 7. July 2005.

[Infocom, 1980] Infocom. Zork I: The Great Underground Empire, 1980. http://en.wikipedia.org/wiki/Zork_I. Visited 23. June 2005.

[Interplay, 1997] Interplay. Fallout, 1997. http://www.nma-fallout.com/. Visited 20. May 2005.

[Jefsen, 2000] Per Jefsen. RTS Artificial Intelligence - An evolutionary approach. Master's thesis, DAIMI, Aarhus University, 2000.

[Joone, 2005] Joone. Java Object Oriented Neural Engine, 2005. http://jooneworld.com/index.html. Visited 27. July 2005.

[Kent, 2004] Tom Kent. Multi-Tiered AI Layers and Terrain Analysis for RTS Games. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Kozen, 1997] Dexter C. Kozen. *Automata and Computability*. Springer Verlag, 1997.

[Laraée, 2004] François Dominic Laraée. Dead Reckoning in Sports and Strategy games. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Lidén, 2004] Lars Lidén. Artificial Stupidity: The Art of Intentional Mistakes. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Lionhead Studios, 2001] Lionhead Studios. Black and White, 2001. `http://www.lionhead.com/bw/index.html`. Visited 21. June 2005.

[LucasArts, 1987] LucasArts. Maniac Mansion, 1987. `http://www.if-legends.org/\%7Eadventure/LucasArts.html\#Maniac_Mansion`. Visited 23. May 2005.

[LucasArts, 1990] LucasArts. Monkey Island, 1990. `http://www.worldofmi.com/thegames/monkey1/index.php`. Visited 16. May 2005.

[Main, 1999] Michael Main. *Data Structures & Other Objects Using Java*. Addison Wesley, 1999.

[Manovich, 2001] Lev Manovich. *The Language of New Media*. MIT Press, 2001.

[Manslow, 2004] John Manslow. Fast and Efficient Approximation of Racing Lines. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Mathiassen *et al.*, 2001] Lars Mathiassen, Andreas Munk Madsen, Peter Axel Nielsen, and Jan Stage. *Objektorienteret Analyse og Design*. Forlaget Marko, 2001.

[Maxis, 2003] Maxis. SimCity 4, 2003. `http://simcity.ea.com/about/simcity4/overview.php`. Visited 16. May 2005.

[McLean, 2004] Alex McLean. Hunting down the Player in a convincing manner. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Media Research Group, 2004] Media Research Group. Grand Master Chess, 2004. `http://www.alawar.com/games/chess/`. Visited 16. May 2005.

[Michalewicz and Fogel, 2004] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2004.

[MicroProse, 1995] MicroProse. Transport Tycoon Deluxe, 1995. `http://www.tycoongames.net/introduction.html`. Visited 30. June 2005.

[Midway, 1992] Midway. Mortal Kombat, 1992. `http://en.wikipedia.org/wiki/Mortal_Kombat`. Visited 21. June 2005.

[Mitchell, 1997] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[Namco, 1994] Namco. Tekken, 1994. http://en.wikipedia.org/wiki/Tekken. Visited 21. June 2005.

[Nareyek, 2001] Alexander Nareyek. Review: Intelligent Agent for Computer Games. *Computers and Games – Second International Conference – CG 2000*, pages 414–422, 2001.

[Nilsson, 1998] Nils J. Nilsson. *Artificial Intelligence A New Synthesis*. Morgan Kaufmann Publishers Inc., 1998.

[Nintendo, 1985a] Nintendo. Mario, 1985. http://en.wikipedia.org/wiki/List_of_Mario_games. Visited 21. June 2005.

[Nintendo, 1985b] Nintendo. Super Mario Brothers, 1985. http://www.classicgaming.com/tmk/smb.shtml. Visited 16. May 2005.

[Orkin, 2004a] Jeff Orkin. Constaining Autonomous Character Behaviour with Human Concepts. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Orkin, 2004b] Jeff Orkin. Simple Techniques for Coordinated Behaviour. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Paramount, 2005] Paramount. Tomb Raider, 2005. http://www.tombraidermovie.com/. Visited 26. July 2005.

[Patel, 2004] Amit J. Patel. Pathfinding, 2004. http://theory.stanford.edu/~amitp/GameProgramming/. Visited 2. May 2005.

[Polyphony, 2000] Polyphony. Gran Turismo II, 2000. http://www.gamefaqs.com/console/psx/data/197469.html. Visited 16. May 2005.

[Rabin, 2004a] Steve Rabin, editor. *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Rabin, 2004b] Steve Rabin. Common Game AI Techniques. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Rabin, 2004c] Steve Rabin. Promising Game AI Techniques. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Ramsey, 2004] Michael Ramsey. Designing a Multi-Tiered AI Framework. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Reynolds, 2004] John Reynolds. Team Member AI in an FPS. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Rouse, 2000] Richard Rouse. *Game Design Theory and Practice*. Wordware Publishing Inc., 2000.

[Runestone, 2005] Runestone. Seed, 2005. http://www.seedthegame.com. Visited 26. July 2005.

[Sidran, 2003] Ezra Sidran. *The Current State of Human-Level Artificial Intelligence in Computer Simulations and Wargames*. PhD thesis, University of Iowa, 2003.

[Sidran, 2004] Ezra Sidran. A Calculated Strategy: Readings directed towards the creation of a strategic artificial intelligence. *Readings for Research. University of Iowa*, 2004.

[Sierra Entertainment, 1998] Sierra Entertainment. Half Life, 1998. http://www.sierra.com/product.do?gamePlatformId=180. Visited 5. July 2005.

[Sierra Entertainment, 2005] Sierra Entertainment. No one lives forever II, 2005. http://nolf2.sierra.com/site.html. Visited 18. May 2005.

[Sierra, 1987] Sierra. Leisure Suit Larry, 1987. http://www.if-legends.org/~adventure/Sierra_On-Line,_Inc/Larry.html. Visited 23. May 2005.

[Sonic Team, 1991] Sonic Team. Sonic the Hedgehog, 1991. http://en.wikipedia.org/wiki/List_of_games_featuring_Sonic_the_Hedgehog. Visited 21. June 2005.

[Sony Online Entertainment, 1999] Sony Online Entertainment. EverQuest, 1999. http://eqlive.station.sony.com/. Visted 20. June 2005.

[Stallings, 2000] William Stallings. *Data & Computer Communications*. Prentice Hall, 2000.

[Stratagus, 2004] Stratagus. The Stratagus Engine v. 2.1, 2004. http://stratagus.sf.net. Visited 22. April 2005.

[Studios, 1992] Westwood Studios. Dune II, 1992. http://www.flashback-aw.net/games.php?GameID=38. Visited 16. May 2005.

[Sun Microsystems, 2003] Sun Microsystems. Java Native Interface, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html. Visited 18. July 2005.

[Sweetser, 2004] Penny Sweetser. Strategic Decision-Making with Neural Networks and Influence Maps. In *AI Game Programming Wisdom 2*. Charles River Media, Inc., 2004.

[Taylor, 2002] Laurie N. Taylor. Video Games: Perspective, Point-of-View, and Immersion. Master's thesis, Graduate Art School, University of Florida, 2002.

[TV2, 2005] TV2. TV2 spil, 2005. http://spil.tv2.dk/. Visited 23. May 2005.

[van de Wijdeven, 2002] Marco van de Wijdeven. Game Trees in Realtime Games, 2002. http://ai-depot.com/GameAI/GameTree.html. Visited 21. May 2005.

[Wargus, 2004] Wargus. Wargus v. 2.1, 2004. http://wargus.sf.net. Visited 22. April 2005.

[Weeks, 2005] Mark Weeks. Relative Value of Chess Pieces, 2005. http://chess.about.com/library/ble23pvl.htm. Visited 11. July 2005.

[Westwood Studios, 1991] Westwood Studios. Eye of the Beholder II, 1991. http://www.abandonia.com/games/176/Eye_of_the_Beholder_2.htm. Visited 16. May 2005.

[Wexler, 2002] James Wexler. Artifical Intelligence in Games. 2002.

[Wikipedia, 2004] Wikipedia. Lens Flare Definition, 2004. http://en.wikipedia.org/wiki/Lens_flare. Visited 28. April 2005.

[Wikipedia, 2005a] Wikipedia. A* search algorithm, 2005. http://en.wikipedia.org/wiki/A-star_search_algorithm. Visited 2. May 2005.

[Wikipedia, 2005b] Wikipedia. Computer and video game genres, 2005. http://en.wikipedia.org/wiki/Computer_and_video_game_genres. Visited 12. May 2005.

[Wikipedia, 2005c] Wikipedia. Dead Reckoning, 2005. http://en.wikipedia.org/wiki/Dead_reckoning. Visited 28. June 2005.

[Wikipedia, 2005d] Wikipedia. Mean squared error, 2005. http://en.wikipedia.org/wiki/Mean_squared_error. Visited 27. July 2005.

[Wikipedia, 2005e] Wikipedia. Real-time strategy, 2005. http://en.wikipedia.org/wiki/Real_time_strategy. Visited 21. July 2005.

[Wikipedia, 2005f] Wikipedia. Wikipedia, 2005. http://en.wikipedia.org/wiki/Wikipedia. Visited 13. May 2005.

[Wikipedia, 2005g] Wikipedia. Wikipedia: Neutral point of view, 2005. http://en.wikipedia.org/wiki/Wikipedia:Neutral_point_of_view. Visited 15. May 2005.

[Wolf, 2002] Mark J. P. Wolf, editor. *The Medium of the Video Game*, chapter 6. University of Texas Press, 2002.

[Woodcock, 2005] Steven M. Woodcock. The Game AI Page, 2005. http://www.gameai.com/. Visited 19. July 2005.

[Yahoo, 2005] Yahoo. Yahoo Games, 2005. http://games.yahoo.com/. Visited 18. May 2005.

Appendix A

A note on search for literature

We have been extensively searching for literature describing the application of methods from soft computing and game theory to commercial computer games. This effort was however largely fruitless. Most literature found was from people within the game industry describing by example how to improve the game and the experience for the human player. There is little effort to describe the theoretic foundations for applying these techniques to the commercial computer game domain.

Aside from papers we also found several websites among others [Woodcock, 2005] and [Buckland, 2005] maintained by people from the commercial computer game industry regarding Game AI.

The situation is improving thanks to among others the work of Michael Buro and his team at the University of Alberta. They actively call for a research agenda in real time strategy games [Buro, 2004]. They are however mostly focused on creating an open platform, [Buro, 2002], for testing Game AI in RTS games.

Books We also found a few books covering Game AI development. They do a thorough job at explaining by examples how to develop Game AI methods for different types of games. In [Champandard, 2003] Alex Champandard develops a consistent opponent for first person shooters and explains all methodologies used in the process. In [Rabin, 2004a] a collection of articles concerning the topic of Game AI written by people from the commercial computer game industry is presented.

Older games As a part of this thesis we also researched the games which in retrospect defined many of the computer game genres (see section 2.2 on page 8) of today. Official information about these games is hard to come by as the companies which created these vanish or devote their official homepages to newer games.

As sources of information to older games we found fan-based websites dedicated gaming sites such as [GameFAQs, 2005] or the so-called *abandonware* sites such as [Abandonia, 2005].

Wikipedia.org The online encyclopedia called Wikipedia, [Wikipedia, 2005f], is a source of much information on the Internet. Anyone can edit any entry and this is why so many topics exist on the site but is also why its credibility can be questioned.

The people behind Wikipedia a group called Wikimedia are well aware of this issue. They write:

> "Wikipedia's status as a reference work has been controversial. It has received praise for being free, editable, and covering a wide range of topics. It has been criticized for a perceived lack of accountability and authority when compared with traditional encyclopedias, systemic biases, and deficiencies in some topics.", [Wikipedia, 2005f].

To remedy this, Wikipedia has adopted an official policy, [Wikipedia, 2005g], which asserts that Wikipedia articles must have a *neutral point of view*. We acknowledge that Wikipedia might be biased and use references to Wikipedia with care – as a way of getting background information for introductory purposes and cross-references where possible.

Appendix B

Summary of diary

We have been writing a diary during the making of this thesis where notes of design- and implementation-progress were kept. The individual entries which were updated on a daily basis contains design choices, implementation details regarding non-thesis relevant information, the current progress of the thesis-report, notes and general thoughts. We have loosely followed an iterative sequence of process so design, implementation, game-engine examinations and testing tasks were alternated throughout the thesis-work as described in [Mathiassen *et al.*, 2001].

During the course of this project several seminars were held where this project was presented. Constructive criticism and many good solution-models came from these meetings where both associate professors and students alike were prepared to comment on design and general issues.

A diary-extract is presented below giving a general overview of the diary.

September '04 - November '04 Early work with the general domain of RTS games was conducted and a choice of game-engine, Stratagus, was decided upon. General preliminary design-issues were discussed on more or less general levels.

In this early phase the game-engine was investigated to understand whether it was technically possible to incorporate a Game AI module as a disjunct client. As such it was found that extending the engine was technically more plausible than extending the network protocols substantially.

Early implementation work based on the language of C++ was started but was abandoned due to technical limits of the engine in collaboration with C++.

November '04 - January '05 The general design issues were exemplified and design-work became more focused in solving the problems within SSC. An overview of an RTS opponent was created and identifying areas of responsibility was conducted. As a result several packages were identified and designed.

The game-engine was studied in greater detail to find other applicable approaches than the disjunct client-model.

Implementation based on Java using the JNI-interface was started and within days the code reached the level of the C++ implementation.

January ’05 - March ’05 The design progress moved towards a optimal solution-model for SSC. Several discussions with Associate Professor Thiemo Krink were conducted and it was during these discussions a game tree method was considered and afterwards decided upon.

Much work was needed to incorporate JNI in the engine and the overall work with the link between Java and the engine was performed during this period of time.

Implementation regarding the game tree method was started and many discussions involving learning algorithms primarily a neural network-solution into the model were conducted. An implementation consisting of an experiment framework was started with the objective of engine experimentation. This framework was created in such a way that testing-code and individual experiment executions could be incorporated easily.

March ’05 - April ’05 With the general design-issues in place much of the work done during this period was used in implementation and design phases of the actual code.

The engine was now and then referenced to test various situations on which our code was based.

The implementation reached new heights as the set of rules was implemented. This was to allow experiments in real simulations within the engine. A parser which was used in context with training the neural network, timing of the individual runs and testing of the code was created. Several frameworks, Log4J and the NetBeans profiler were included in the project to allow more testing and timing capabilities. These were incorporated to allow a collaboration with our present code.

April ’05 - May ’05 Design work in this period was reduced to designing test-suites (unit-tests and integration-tests), final design of the neural network-model, evaluation of the timing-runs and general design-layout changes.

The primary implementation work was done in the experiment framework and in the timing of experiment- and test-cases. Implementation work regarding loading and storing training examples was made to the parser written earlier. Implementation and incorporation of the Joone-framework was also conducted during this period of time.

For those modules in our project, which deserved special attention, software reviews were made to secure the correctness of these.

The thesis report was started during the start of this period and several layout options were considered.

May ’05 - June ’05 The design process was extended to include the design of additional rules which were needed in the pruning of the game tree. Design of two alternative pruning methods KNN and TVGT was made and some minor design changes were also carried out. Training examples were created by hand and multiple ways of generating a greater number of these were discussed. Several rule-compositions were discussed to give the units a more consistent behaviour.

During May implementation work was primarily made in extending the set of rules and testing this. The two alternative pruning methods were also implemented and tested in several scenarios in Stratagus. Methods for generating additional training examples based on the hand-made ones were implemented and tested. The rule sequences were implemented and tested using an integration test-suite.

Several chapters in the thesis report were written as preliminary versions.

June ’05 - August ’05 Design-work in this phase of the project was only used to discuss and re-design parts of the thesis-report.

Some code was created in the start of this period which could be used to collect and merge various findings from the evaluation-runs. A module was implemented to automate the data collection and merge the results.

The thesis report was finalised during this phase of the project.

Appendix C

Design and Implementation

In this appendix we will present the design and implementation details of our system. Recall that the communication between our module and the Stratagus engine was examined in chapter 6 and will therefore not be covered here.

C.1 Design

This section describes the design details of the game tree method. The primary design choice was to create a plugin system for the engine. This design would allow some players to use the game tree module for SSC situations while letting other players use the standard client. I.e. the players would be able to choose between several plugin modules and pick the one best suited for their playing style as discussed in section 2.3.2 on page 24. As already examined in chapter 6 on page 81 the method favoured an object oriented approach and in this light we implemented three packages which allowed code-separation from the engine along with an extensible and flexible design model. These packages will be examined in detail in the following sections.

C.1.1 Java packages

The package called `stratagus` was created solely for interfacing between our module and the engine. Our implementation for the game tree-based methods called `rada` uses this package for all functions related to the engine. Following, each package and the associations between the classes in these will be examined.

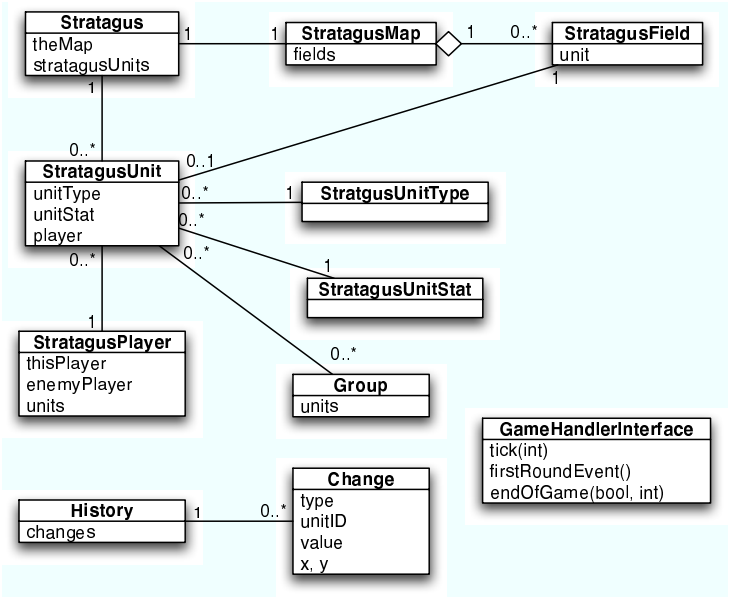

Figure C.1: The UML diagram of the Stratagus package

Stratagus package

In the following, the classes which constitute the `stratagus` package are described. The associated UML-diagram is depicted in figure C.1. All of the classes with the prefix Stratagus are used to explicitly represent the data we need from the engine whereas the History, Change and GameHandlerInterface classes represent non-engine dependent information.

Stratagus Stratagus is the main class of the Stratagus package. It implements all of the required functions which the JNI code uses. This class furthermore contains the scenario's map and all units on this.

StratagusMap The StratagusMap class is a representation of the map in the scenario. It contains a collection of fields which is associated with the map. This class also contains methods for distance-calculations so units may quickly access distance-information.

StratagusField The StratagusField class represents the individual entries on the map. This class contains information about the field's location on the map and whether a unit or a environment feature is positioned in this location.

Group The Group class represents a group of units which a single player controls in the engine. This object is a placeholder for a collection of units and offers little functionality but accessor-methods.

StratagusUnit The StratagusUnit is an essential class since it represents a unit in the engine. It contains a large amount of attributes although some important attributes are contained in the StratagusUnitType- and StratagusUnitStat-classes instead. This is because these either are static for the individual types of units (melee or ranged) or depend on upgrades such as an increase in armour or weapon-damage.

StratagusUnitType The StratagusUnitType class contains information about those attributes which are shared for all units of that specific type.

StratagusUnitStat The StratagusUnitStat class contains information about those attributes which have been upgraded by technological advances.

StratagusPlayer The StratagusPlayer class represents the player controlling the units. This class should not been seen as an actual player but merely a way of identifying who controls which units. The class contains information about what faction is played and which units are in the player's control.

Change and History The Change and History classes are used to record a complete run of a simulation. A Change-object represents an essential single change in a unit-state. The History class contains a collection of Change-objects and the functionality of loading and storing these.

GameHandlerInterface The GameHandlerInterface class is the interface all applications are required to implement and pass on to the Stratagus class to enable callbacks from the engine.

Rada package

The `rada` package contains several components including an experiment framework which can be used for data collection and testing purposes. A game tree component also is included along with the set of rules. Each component is described in turn below.

In figure C.3 on page 134 the `rada` package is depicted. The `stratagus` component described earlier is also shown in this figure. This is to emphasise that

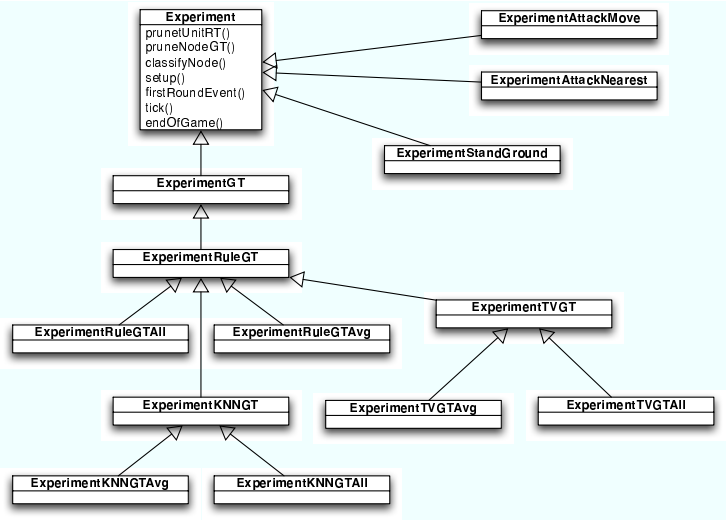

Figure C.2: The UML diagram of the Experiments component

communication is solely transmitted between the `stratagus` package and the `rada` package and not between the `rada` package and the Stratagus engine.

Firstly, the Experiments-component depicted in figure C.2 will be described and following this description the remaining classes of the `rada` component shown in figure C.3 on page 134 will be elaborated upon.

Experiment framework

The classes which are included in the experiment framework are described in the following and the UML associated which depicts the overall design of the experiment framework is shown in figure C.2. It should be noted that several experiment classes are not depicted nor described since these only served in testing various engine-dependent aspects.

Experiment Experiment is an interface and represents a general experiment which all experiments in the framework implement. It contains methods which aim towards a game tree model since this was the primary focus of this thesis.

ExperimentAttackMove The ExperimentAttackMove class represents the experiment in which an attack-move command examined in section 3.4.1 on page 40 is given to each controlled unit. The destination of the attack-move is calculated as the centre of the enemy group. This command is given to each controlled unit every 50'th game cycle. From here on the resulting behaviour is determined solely by

the attack-move order and the engine's default behaviour as described in section 3.4.4. This way of conducting combat is very similar to the way human players do since players generally select several units and then perform an attack-move command towards the enemy forces.

ExperimentAttackNearest The ExperimentAttackNearest class represents the experiment in which each controlled unit is ordered to attack the nearest enemy unit. If multiple units are equally close a random one of these is chosen. This order is given to each unit every 10'th game cycle. If the path to the nearest enemy unit is blocked by obstacles the controlled unit will automatically find a way around these. If another unit comes closer this unit will be attacked instead, i.e. no memory of who engages who is included.

ExperimentStandGround The ExperimentStandGround class orders each controlled unit to stand ground as described in 3.4.1. This stand ground order is given to each unit every 10'th game cycle. No movement actions will be performed. Due to the default behaviour a stand ground'ed unit will only attack if the enemy stands in front of it.

ExperimentGT This class is the most general experiment which uses a game tree model. It is the methods within this class which construct the game tree and apply the chosen actions found in the game tree to the engine. Every experiment which uses some form of game tree extends this class and overrides certain methods such as how rating on both internal and leaf nodes is accomplished.

ExperimentRuleGT The ExperimentRuleGT class is a super class for several experiments. This class extends the ExperimentGT with the option of using rules to prune the game tree as described in section 5.4. Every subclass overrides the rating method used whereas this class uses a neural network to evaluate the game tree nodes. Nodes are rated by rating all leaves and taking the minimum value and assigning this to the parent node. Thus, all subclasses use the same game tree construction algorithm and only vary in the rating method and how the internal nodes are rated.

The ExperimentRuleGTAvg subclass also uses a neural network to rate nodes but rates each internal node with the average of its children.

The ExperimentRuleGTAll subclasses the same rating method as above but only rates the child-nodes of the root to derive actions.

ExperimentTVGT The ExperimentTVGT class uses the *threat matrix rater* (TMR) rating method as previously described in section 5.5.1. The TMR uses the

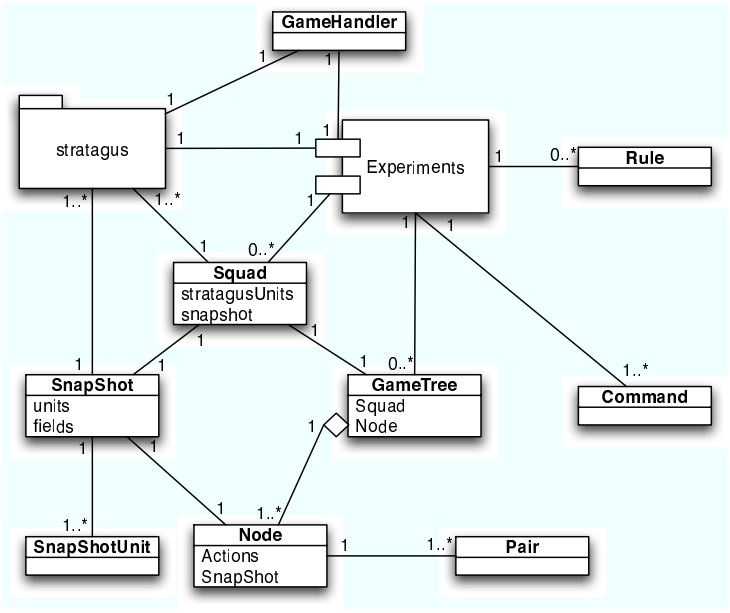

Figure C.3: The UML diagram of the rada package

sum of the threat matrix to decide the related node's value. In this experiment all game tree leaves are rated according to the sum whereas internal nodes are given the minimum value of their children.

The ExperimentTVGTAvg subclass also uses the TMR as rating method but rate all internal nodes with the average of the children.

The ExperimentTVGTAll experiment also uses the TMR rating method but only rate the the immediate children of the root.

ExperimentKNNGT The ExperimentKNNGT class uses a *K* nearest neighbour (KNN) algorithm as rating method. This experiment rates the leaves and assigns to a node the minimum of its children.

The ExperimentKNNGTAvg subclass also uses the KNN rating method to rate each leaf and each internal node is given the average value of its children.

Finally, the class ExperimentKNNGTAll uses the KNN method but only rates the immediate children of the root.

Rada package continued

GameHandler The GameHandler class is an actual implementation of the GameHandlerInterface from the `stratagus` package. This class organises the

propagation of events to experiments. The GameHandler is also responsible for outputting results from the experiments.

Squad The Squad class encapsulates a collection of units and is capable of creating a snapshot centred around the units.

SnapShot The SnapShot class represents the controlled units and their immediate environment. This class contains methods for updating the state of the units and is furthermore responsible for detecting and handling conflicts in unit-actions. A SnapShot-object is part of a node in the game tree.

SnapShotUnit The SnapShotUnit class represents a unit in our model. It contains many of the attributes found in the StratagusUnit class in the `stratagus` package but has additional methods for calculating its threat value and distances to the nearest enemies. Lastly, a collection of methods is available for removing certain actions which have been deemed by the rules or the actual game's state to be illegal.

GameTree The GameTree class is our implementation of a timestamped game tree. It is responsible for building the tree to a certain depth and to build the tree correctly according to included rules.

Node The Node class is our implementation of a node in the game tree. The class does not contain much functionality but acts as an object for holding references to a SnapShot-object. This class also contains a representation of the actions which lead to this node from its parent. This information is obtained by using a collection of Pair-objects.

Pair Pair is a class which encapsulates information about important changes in the game state. A collection of Pair-objects is used to derive a child-node from its parent.

Rule The Rule class is an abstract class which is used as a placeholder for a rule's specific priority. The class contains two methods the pruneRT- and pruneGT-method which an actual implementation of a rule should override. These methods are used in the calculations to derive a node's fanout.

A factory-class described in [Gamma *et al.*, 1994, Chap. 3] was implemented called RuleFactory. This static class is used for instantiating rules described in section 5.4 and is used from the class responsible for setting up the Java packages.

Command The Command class contains actions applicable to units in the engine. This class is used exclusively by the ExperimentGT-class and its subclasses.

C.2 Implementation

In this section we wish to present the implementation details of the most important methods in our module. We also aim to present a detailed view of how we construct game trees and how we apply rules to these.

Rating methods

As a handcrafted rating method the threat matrix rater (TMR) was implemented. The TMR simply iterates through the threat matrix and sums the entries.

To implement the neural network, a large Java project called *Joone*, [Joone, 2005], was used. Joone is released under the LGPL, [Free Software Foundation, 1999], license and provided the necessary training and experimentation functionality which were needed to rate the game tree nodes. Each input node in the network is mapped to an entry in the threat matrix. The output of the network is a single value which represents the state's value.

The *K* nearest neighbour algorithm was also implemented as rating method. When rating a node the KNN algorithm iterates linearly through every stored training threat matrix and calculates the average of these *K*. *K* was chosen to 5 in all experiments.

C.2.1 Datastructures and methods

In this section we aim to describe the different methods of the most interesting of our classes.

StratagusMap

As previously mentioned in section C.1.1 on page 129 the StratagusMap class is responsible for representing the map as the engine views it. This is done through a double array of fields-objects. The most important methods are the following two:

calculateDistances We are often interested in knowing the length of the shortest path between two fields on the map. To obtain this information, we run a Dijkstra's shortest path algorithm on all fields of the map before starting the game. This method is responsible for exactly that. As a map contains

at least 32×32 tiles this computation takes some time. We have incorporated the loading and saving of precomputed data for each map so Dijkstra's algorithm only runs once per map and this saved information is then only loaded on subsequent runs.

getDistance This method accepts two pairs of coordinates and returns the length of the shortest path between these.

SnapShotUnit

A SnapShotUnit instance is a direct representation of a StratagusUnit tailored for inclusion in the game tree calculations. It contains the unique identifier of the StratagusUnit unit it represents along with the important attributes of the StratagusUnit. A SnapShotUnit knows in what game cycle it is ready to perform actions. An instance of SnapShotUnit also maintains an array of available actions and if the unit is of the ranged type a list of ranged attack actions is also included. The methods of interest in a SnapShotUnit are the following four:

estimateDamage When a SnapShotUnit attacks a unit, we need a way of estimating the amount of damage the engine will actually apply if the action was performed in the engine. This method calculates this value. The method takes the target unit as parameter and returns the estimated amount of damage. Note that we use the Java random number generator and not the generator in the engine. This implies that this method is not accurate.

calculateTV A SnapShotUnit is able to calculate its own threat value as described in section 5.3 on page 60 and calculateTV returns this value.

getMapDistanceToUnit This method simply queries the static shortest path information in the map and returns the number of tiles in the shortest path between this unit and the unit given as an argument.

getNumberOfActions A simple and efficient way of obtaining the number of actions currently available to the unit.

SnapShot

To enable efficient access to the SnapShotUnit instances contained in a SnapShot instance we have three datastructures available depending on the nature of the access. These were implemented due to the heavy time constraints involved and are as follows:

To enable a quick lookup of a specific unit given the unit's unique identifier a hash table using Java's HashMap instance contains a reference to all units in the

snapshot. This hash table uses the unit identifiers as keys so efficient lookups can be made.

If quick access to the immediate vicinity of a specific unit is required, i.e. the position is known we have an array representation of the units where the neighbourhood of a given unit can be efficiently searched.

Lastly, a priority queue contains all units sorted by the game cycle in which they are ready to perform actions.

A SnapShot instance contains the following methods of interest:

getRatingInput This method returns an array of doubles with 400 entries. Each entry is uniquely mapped to one of the fields in the SnapShot and contains the threat value for the unit occupying that field or zero as a default value. This method iterates over the array representation of the units.

getReadyUnits For easy access to the units ready to perform actions in the game cycle represented by the snapshot, the getReadyUnits method returns an array containing these. As the ready times of units do not change during the life of a single snapshot this method stores the result of the first call to getReadyUnits in an internal variable. Subsequent calls to getReadyUnits on the same instance of a SnapShot then returns the precalculated result. The priority queue allows this method to perform its calculations quickly.

updateSnapShot When instantiating a new SnapShot we need to update the information in the new instance. This is achieved through the updateSnapShot method. This method accepts a linked list of actions and updates the game state by applying the effect of each action sequentially.

removeIllegalMoveAction When a unit performs an action the performed action might influence the movement actions available to other units. This method takes as arguments a unit and a linked list of actions to be performed by other units in the snapshot. The method iterates through the available actions of the unit and removes all actions which would conflict with at least one action in the linked list.

setUnitActions This method takes a unit as argument and assigns the unit its available movement actions as determined by the unit's placement on the map. No other factors than placement on the map and static environment features are handled by this method.

C.2.2 Game tree construction

The GameTree class is responsible for constructing the timestamped game tree. The pseudo-code seen in algorithm 1 on page 140 depicts our buildTree method

which builds the tree to a user-defined depth.

The buildTree method first checks if the argument node does not contain units from both sides or if the depth given has been reached. In this case a leaf node is identified and a rating of the node is performed.

In the general case the Buildtree-method first assigns the basic experiment-independent allowed actions such as not moving outside the map and not moving into static environment features to each unit. Following, the unit-actions are pruned according to the associated experiment and its rules if any. A sorting of the units based on the amount of actions is then performed. Afterwards, the children of the node are calculated. The method is then called recursively for each of the children and the best rated child maximumWeightChild is stored.

It should be noted that in our game tree model the effect of an action is applied immediately and then the unit performing the action will be ready to perform another action when exactly enough game cycles have passed in the model. Meaning, a node which models an attack reduces the hitpoints of the receiving unit immediately and not when the action is completed. This is a discretisation compared to the way it is simulated in the engine, because the action's effect lies implicit in the transition and not during the transition as in the engine.

The pseudo-code of calculateChildren-method is depicted in Algorithm 2. This method takes as input a tree node, a list of units and a list of actions and creates the node's fanout, i.e. the children of the node as output.

The method first checks if the list of units is empty and if so a child is created as a deep copy of the actual node. An update of the child is then performed as well as updating the timestamp of the child. The child-node is then added to the children-list of the node.

In the general case the unit with the least amount of actions available is removed from the list of units and experiment-dependent actions are pruned and then actions which conflicts with other units' earlier assigned actions are also removed. Following the removal of actions each available action is appended to a list and the list is then deep copied. The calculateChildren-method is then called recursively. If all actions for a unit were removed the unit receives a standGround-action and the method is then called recursively as above.

Recursion tree construction

When the game tree is being built just before the children of a node are calculated the units are sorted by their available actions. The reason for this is of optimisation reasons since the number of children of a node is the same independent of the order of the units. Put another way, the leaves and the number of layers in the recursion tree is always the same regardless of which unit is branched on first, secondly and so on. The leaves constitute the permutations of available actions

Algorithm 1: The buildTree algorithm

Input: The number of steps to look ahead, depth, and the root node, node
Output: A timestamped game tree, evolved from the input node

```
begin
    if node does not contain units from both sides or
    node.timestamp > depth then
        Experiment.rateNode(node);
        return ;
    end
    assignActionsToReadyUnits(node.readyUnits);
    Experiment.pruneNode(node);
    sortUnitsByNumberOfAvailableActions(node.readyUnits);
    calculateChildren(node, a copy of node.readyUnits, empty list);
    val ← −∞ ;
    foreach c in node.children do
        buildTree(depth, c);
        if c.weight > val then
            val ← c.weight;
            node.maximumWeightChild ← c;
        end
    end
    Experiment.rateNode(node);
end
```

and each layer refers to a specific unit. It should be clear that nodes with a large number of children should be placed as far down the recursion tree as possible for minimising internal recursion tree nodes.

Figures C.4 on page 142 and C.5 on page 142 shows two equally expressing trees for three units. In figure C.4 on page 142 the unit with three actions available is chosen to be branched on first, then the unit with two actions and finally the unit with one action available. Opposed to this a similar situation is shown in figure C.5 on page 142 where the same units exist and have the same actions available. The only difference is that the units in figure C.5 on page 142 are sorted based on their available actions before recursing.

Since the number of leaves in the recursion tree is the same only the internal nodes can be minimised. This minimisation can be accomplished by inserting the unit with the fewest actions first.

Algorithm 2: The calculateChildren algorithm

Input: A node, node, a list of units, unitList, and a list of actions, actionList
Output: A list of nodes, node.children, constituting the children of the input node

```
begin
    if unitList is empty then
        child ← deep copy of node ;
        updateNodeFromActions(child, actionList);
        child.timestamp ← min_{u∈child.units} u.ready;
        Add child to node.children;
        return ;
    end
    u ← removeFirstUnit(unitList);
    Experiment.pruneUnitActions(u, actionList);
    removeIllegalUnitActions(u, actionList);
    if u.actions is not empty then
        foreach a in u.actions do
            addActionToEndOfList(actionList, a);
            aList ← deep copy of unitList ;
            calculateChildren(node, aList, actionList);
            removeActionAtEndOfList(actionList);
        end
    else
        a ← standGround action;
        addActionToEndOfList(actionList, a);
        aList ← deep copy of unitList ;
        calculateChildren(node, aList, actionList);
        removeActionAtEndOfList(actionList);
    end
end
```

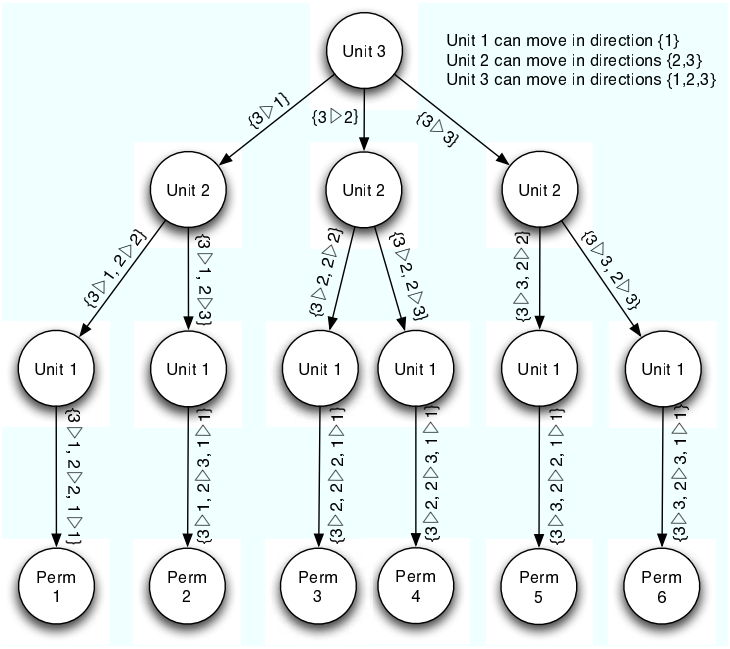

Figure C.4: The full recursion graph

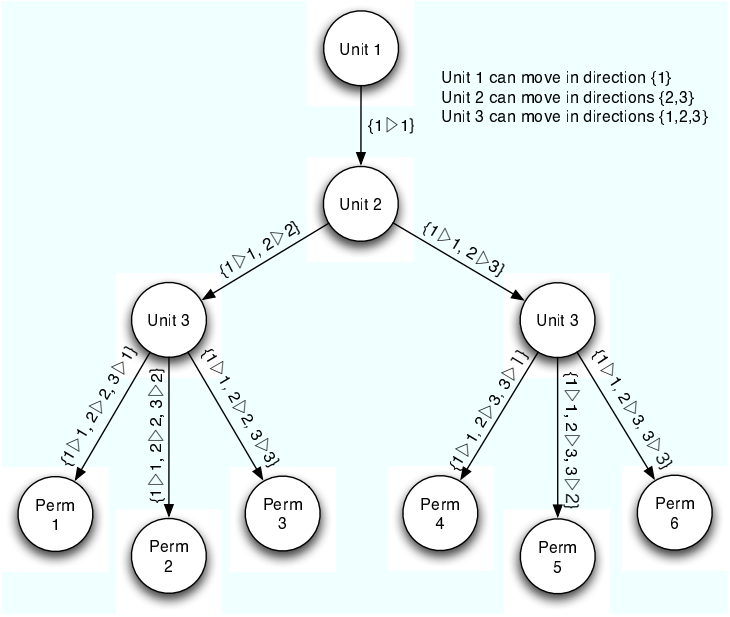

Figure C.5: The reduced recursion graph

Applying choice

When the game tree has been built and the nodes have been rated the task at hand is to give commands to the units in the engine. From the root node the maximumWeightChild node is chosen and the actions which led to this node are inserted into a queue of commands. This queue only contains the actions of the controlled units and not the actions of non-controlled units. When the queue is filled with the maximumWeightChild's actions the commands are propagated to the units in the engine.

However, the implemented game tree and the command queue can handle cases much more complex cases than just choosing the immediate best child-node of the root. The game tree is as an argument given the amount of game cycles to which the tree should be built. The number of game cycles is also given to the method which fills the queue of commands. As such, the tree can be built to game cycle y and the queue can be filled to game cycle x where the only requirement is that $x < y$. Due to the uncertainty involved in the model such as opponent modelling and randomisations we chose to only assign commands as the immediate maximumWeightChild of the root dictated. One could by simply changing an argument value calculate a long list of commands but due to the uncertainty involved the model would hastily become incorrect.

Appendix D

Results

In this appendix all tables generated from the evaluation runs described in section 7.2 are presented. Firstly, we will describe how the tables should be read and then present the tables.

D.1 Reading the tables

Tables C.1 through C.16 show the results of all evaluations performed.

Tables C.1 through C.3 show the results of the game tree-based experiments playing as Humans against the *AttackMove*-experiment in all designed situations.

Tables C.4 through C.6 show the results of the game tree-based experiments playing as Humans against the *AttackNearest*-experiment in all designed situations.

Tables C.7 through C.9 show the results of the game tree-based experiments playing as Orcs against the *AttackMove*-experiment in all designed situations.

Tables C.10 through C.12 show the results of the game tree-based experiments playing as Orcs against the *AttackNearest*-experiment in all designed situations.

Tables C.13 and C.14 show the results of the *AttackNearest*-experiment as both Orcs and Humans against the built-in experiments.

Tables C.15 and C.16 show the results of the *AttackMove*-experiment as both Orcs and Humans against each of the built-in experiments.

Tables C.17 through C.20 show the results in tables C.1 through C.12 averaged over the rule sequences.

Tables C.17 and C.18 show the results of the game tree-based experiments playing as Humans against the built-in experiments, averaged over rule sequences.

Tables C.19 and C.20 show the results of the game tree-based experiments playing as Orcs against the built-in experiments, averaged over rule sequences.

Tables C.21 and C.22 show the results of the game tree-based experiments against the built-in experiments, averaged over controlling both Humans and Orcs and the rule sequences.

Tables C.23 through C.26 show the results in tables C.1 through C.12 averaged over rating methods.

Tables C.23 and C.24 show the results of the rule sequences playing as Humans against the built-in experiments, averaged over rating methods.

Tables C.25 and C.26 show the results of the rule sequences playing as Orcs against the built-in experiments, averaged over rating methods.

Tables C.27 and C.28 show the results of the rule sequences against the built-in experiments, averaged over controlling both Humans and Orcs and the rating methods.

Table C.29 shows the results of the built-in experiments playing against each other, averaged over controlling both the Humans and the Orcs.

D.2 The tables

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,428	0,033	0,221	-1,233	1,671
KNNGTall	-0,506	0,033	-0,118	-1,222	1,039
NNGTall	-1,267	0,033	0,218	-1,933	1,168
TVGTavg	-0,528	0,033	-0,118	-2,017	0,664
KNNGTavg	-1,1	0,033	-0,121	-1,233	1,671
NNGTavg	-0,35	0,033	-0,118	-1,233	1,157

Table D.1: End result of game trees with FocusFire against the AttackMove experiment (GT as Humans.).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,467	0,033	0,379	-1,967	1,189
KNNGTall	-0,25	0,033	-0,557	-2,017	1,039
NNGTall	-0,128	0,033	-0,307	-2,017	1,6
TVGTavg	0,119	0,033	0,586	-2,017	1,061
KNNGTavg	-1,322	0,033	0,629	-2,017	1,514
NNGTavg	-0,792	0,033	-0,443	-2,017	1,136

Table D.2: End result of game trees with AttackKNearest against the AttackMove experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,161	0,033	0,086	-1,8	1,018
KNNGTall	-1,478	0,033	0,386	-2,017	1,671
NNGTall	-1,075	0,033	0,214	-2,017	1,2
TVGTavg	-0,875	0,033	0,386	-2,017	0,536
KNNGTavg	-0,842	0,033	-0,132	-2,017	1,671
NNGTavg	-0,9	0,033	-0,632	-1,222	1,629

Table D.3: End result of game trees with RangedAssist against the AttackMove experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,036	0,033	0,136	-1,256	0,436
KNNGTall	-1,444	0,033	0,136	-1,256	0,193
NNGTall	-0,081	0,033	0,189	-1,967	0,514
TVGTavg	0,494	0,033	0,136	-1,967	0,421
KNNGTavg	-1,042	0,033	0,207	-1,917	0,479
NNGTavg	-0,147	0,033	0,207	-1,967	0,25

Table D.4: End result of game trees with FocusFire against the AttackNearest experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,517	0,033	0,879	-1,967	1,271
KNNGTall	-0,097	0,033	0,125	-1,967	0,464
NNGTall	0,1	0,033	1,257	-1,917	1,039
TVGTavg	-0,108	0,033	-0,45	-1,967	0,507
KNNGTavg	0,139	0,033	0,429	-1,967	0,943
NNGTavg	0,8	0,033	0,65	-1,356	0,179

Table D.5: End result of game trees with AttackKNearest against the Attack-Nearest experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,378	0,033	0,114	-1,356	0,5
KNNGTall	-0,858	0,033	-0,464	-1,322	0,739
NNGTall	-0,128	0,033	0,436	-1,322	0,407
TVGTavg	-1,167	0,033	-0,471	-1,967	0,393
KNNGTavg	0,108	0,033	-0,371	-1,256	0,607
NNGTavg	-0,4	0,033	0,054	-1,967	0,429

Table D.6: End result of game trees with RangedAssist against the AttackNearest experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,092	0,033	0,65	0,9	-1,829
KNNGTall	0,875	0,033	0,65	1,033	-1,471
NNGTall	-1,2	0,033	0,65	1,967	-1,339
TVGTavg	0,092	0,033	0,65	1,95	-1,296
KNNGTavg	-0,389	0,033	0,65	1,133	-1,529
NNGTavg	-1,117	0,033	0,65	1,289	-1,829

Table D.7: End result of game trees with FocusFire against the AttackMove experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,117	0,033	0,868	0,944	-1,629
KNNGTall	0,111	0,033	0,514	1,45	-1,7
NNGTall	-0,133	0,033	-0,071	1,467	-1,857
TVGTavg	-0,983	0,033	-0,443	1,533	-1,643
KNNGTavg	0,128	0,033	-0,407	1,033	-1,2
NNGTavg	-0,125	0,033	-0,196	1,467	-1,771

Table D.8: End result of game trees with AttackKNearest against the AttackMove experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,317	0,033	0,25	1,033	-1,543
KNNGTall	0,522	0,033	0,564	1,55	-0,679
NNGTall	-0,394	0,033	0,239	1,967	-1,093
TVGTavg	-0,389	0,033	0,239	1,517	-1,232
KNNGTavg	-0,278	0,033	0,246	0,9	-1,318
NNGTavg	-1,083	0,033	0,239	1,75	-1,657

Table D.9: End result of game trees with RangedAssist against the AttackMove experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,925	-0,033	-1,007	1,717	-1,029
KNNGTall	-1,025	-0,033	-1,007	1,767	-1,671
NNGTall	-0,389	-0,033	-1,007	1,189	-1,254
TVGTavg	0,625	-0,033	-1,007	1,783	-1,757
KNNGTavg	-1,311	-0,033	-1,007	1,517	-1,136
NNGTavg	0,3	-0,033	-1,007	1,883	-0,793

Table D.10: End result of game trees with FocusFire against the AttackNearest experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,85	-0,033	-1,039	1,883	-1,343
KNNGTall	0,517	-0,033	0,093	1,8	-0,657
NNGTall	0,708	-0,033	0,093	1,733	-0,657
TVGTavg	0,383	-0,033	-0,9	1,8	-1,039
KNNGTavg	0,708	-0,033	-0,379	1,8	-1,05
NNGTavg	0,592	-0,033	-1,061	1,767	-1,061

Table D.11: End result of game trees with AttackKNearest against the Attack-Nearest experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,511	-0,033	-0,5	1,717	-0,911
KNNGTall	-1,544	-0,033	-0,543	1,717	-0,921
NNGTall	0,817	-0,033	-0,954	1,356	-1,393
TVGTavg	-0,089	-0,033	-0,543	1,767	-0,514
KNNGTavg	-0,097	-0,033	-0,543	1,733	-0,586
NNGTavg	-0,339	-0,033	-0,5	1,683	-1,814

Table D.12: End result of game trees with RangedAssist against the AttackNearest experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
AttackMove	0,289	0,033	0,996	-1,85	1,571
AttackNearest	-1,017	0,033	0,771	-1,733	1,179

Table D.13: End result of AttackNearest vs. AttackNearest, and AttackMove (AttackNearest as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
AttackMove	0,106	0,033	-0,143	1,033	-1,471
AttackNearest	1,017	-0,033	-0,771	1,733	-1,179

Table D.14: End result of AttackNearest vs. AttackNearest, and AttackMove (AttackNearest as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
AttackMove	0,128	0,033	-0,121	-1,517	1,686
AttackNearest	-0,106	-0,033	0,143	-1,033	1,471

Table D.15: End result of AttackMove vs. AttackNearest and AttackMove (AttackMove as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
AttackMove	-0,128	-0,033	0,121	1,517	-1,686
AttackNearest	-0,289	-0,033	-0,996	1,85	-1,571

Table D.16: End result of AttackMove vs. AttackNearest and AttackMove (AttackMove as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,041	0,033	0,229	-1,667	1,293
KNNGTall	-0,744	0,033	-0,096	-1,752	1,25
NNGTall	-0,823	0,033	0,042	-1,989	1,323
TVGTavg	-0,428	0,033	0,285	-2,017	0,754
KNNGTavg	-1,088	0,033	0,125	-1,756	1,619
NNGTavg	-0,681	0,033	-0,398	-1,491	1,307

Table D.17: End result of game trees averaged over rules against the AttackMove experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,058	0,033	0,376	-1,526	0,736
KNNGTall	-0,8	0,033	-0,068	-1,515	0,465
NNGTall	-0,036	0,033	0,627	-1,735	0,654
TVGTavg	-0,26	0,033	-0,262	-1,967	0,44
KNNGTavg	-0,265	0,033	0,088	-1,713	0,676
NNGTavg	0,084	0,033	0,304	-1,763	0,286

Table D.18: End result of game trees averaged over rules against the AttackNearest experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,175	0,033	0,589	0,959	-1,667
KNNGTall	0,503	0,033	0,576	1,344	-1,283
NNGTall	-0,576	0,033	0,273	1,8	-1,43
TVGTavg	-0,427	0,033	0,149	1,667	-1,39
KNNGTavg	-0,18	0,033	0,163	1,022	-1,349
NNGTavg	-0,775	0,033	0,231	1,502	-1,752

Table D.19: End result of game trees averaged over rules against the AttackMove experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,195	-0,033	-0,849	1,772	-1,094
KNNGTall	-0,684	-0,033	-0,486	1,761	-1,083
NNGTall	0,379	-0,033	-0,623	1,426	-1,101
TVGTavg	0,306	-0,033	-0,817	1,783	-1,104
KNNGTavg	-0,233	-0,033	-0,643	1,683	-0,924
NNGTavg	0,184	-0,033	-0,856	1,778	-1,223

Table D.20: End result of game trees averaged over rules against the AttackNearest experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	0,067	0,033	0,409	-0,354	-0,187
KNNGTall	-0,121	0,033	0,24	-0,204	-0,017
NNGTall	-0,7	0,033	0,157	-0,094	-0,054
TVGTavg	-0,427	0,033	0,217	-0,175	-0,318
KNNGTavg	-0,634	0,033	0,144	-0,367	0,135
NNGTavg	-0,728	0,033	-0,083	0,006	-0,223

Table D.21: End result of game trees averaged over rules against the AttackMove experiment (Also averaged over sides).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
TVGTall	-0,069	-0	-0,236	0,123	-0,179
KNNGTall	-0,742	-0	-0,277	0,123	-0,309
NNGTall	0,171	-0	0,002	-0,155	-0,224
TVGTavg	0,023	-0	-0,539	-0,092	-0,332
KNNGTavg	-0,249	-0	-0,277	-0,015	-0,124
NNGTavg	0,134	-0	-0,276	0,007	-0,468

Table D.22: End result of game trees averaged over rules against the AttackNearest experiment (Also averaged over sides).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,696	0,033	-0,006	-1,479	1,229
AttackKNearest	-0,318	0,033	0,048	-2,008	1,257
RangedAssist	-0,888	0,033	0,051	-1,848	1,288

Table D.23: End result of rule sequences averaged over rating methods against the AttackMove experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,364	0,033	0,168	-1,721	0,382
AttackKNearest	0,225	0,033	0,482	-1,856	0,734
RangedAssist	-0,47	0,033	-0,117	-1,531	0,512

Table D.24: End result of rule sequences averaged over rating methods against the AttackNearest experiment (GT as Humans).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,275	0,033	0,65	1,379	-1,549
AttackKNearest	-0,148	0,033	0,044	1,316	-1,633
RangedAssist	-0,218	0,033	0,296	1,453	-1,254

Table D.25: End result of rule sequences averaged over rating methods against the AttackMove experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,454	-0,033	-1,007	1,643	-1,273
AttackKNearest	0,626	-0,033	-0,532	1,797	-0,968
RangedAssist	-0,294	-0,033	-0,597	1,662	-1,023

Table D.26: End result of rule sequences averaged over rating methods against the AttackNearest experiment (GT as Orcs).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,485	0,033	0,322	-0,05	-0,16
AttackKNearest	-0,233	0,033	0,046	-0,346	-0,188
RangedAssist	-0,553	0,033	0,174	-0,198	0,017

Table D.27: End result of rule sequences averaged over rating methods against the AttackMove experiment (Also averaged over sides).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
FocusFire	-0,409	-0	-0,419	-0,039	-0,446
AttackKNearest	0,426	-0	-0,025	-0,03	-0,117
RangedAssist	-0,382	-0	-0,357	0,065	-0,255

Table D.28: End result of rule sequences averaged over rating methods against the AttackNearest experiment (Also averaged over sides).

	Scen7vs7	Scen1vs1	archer-ambush	Scen3vs2	captured
AttackMove	0,198	0,033	0,427	-0,409	0,050
AttackNearest	-0,198	-0,033	-0,427	0.409	-0,050

Table D.29: End result of AttackMove and AttackNearest vs. each other (Averaged over sides).

Appendix E

Contents of the enclosed CD

We have chosen a web-page layout for browsing the data on the enclosed CD. In the root of the CD there is a file called `index.html` which displays the entry page for the data on the CD.

The movies included on the CD are recorded with Snapz Pro[1] and edited with Apple Quicktime 7 Pro[2]. Therefore, the viewing the movies require the quicktime player.

There are the following sections on the CD:

Graphs This section shows the graphs of all performed evaluations. Each graph shows the situation value (SV) for each 25 game cycle.

Movies The movies section contains links to and a description of all recorded movies.

Software This section contains links to Quicktime Player installers for Windows and Mac OS X. Note that the Windows installer bundles the latest iTunes music player.

Stratagus The source code of the Stratagus engine, the Wargus game and our module is available in this section along with a guide to installing, compiling and running this project. Note that we have included the media files from Warcraft II and expansion for use with the Wargus game. These media files are copyright of Blizzard. We do not know whether further distribution of these media files is legal but the reader should be aware of this issue.

[1] `http://www.ambrosiasw.com/utilities/snapzprox/`
[2] `http://www.apple.com/dk/quicktime/pro/`