Artificial Intelligence in Video Games: Towards a Unified Framework

University of Liège
Department of Electrical Engineering
and Computer Science

Firas Safadi
PhD Dissertation
January 2015

2

Foreword

Four years ago, I started this research with the goal of creating fully autonomous learning agents for real-time strategy games. I began to experiment with a couple of machine learning techniques including reinforcement and imitation learning and eventually had the chance to visit the Inria Saclay research lab in France where I worked on the MASH project[1] with Olivier Teytaud and implemented techniques like neural networks, random forests and fitted Q iteration. Following my stay in Orsay, I continued exploring machine learning techniques such as cascade correlation, extreme learning machines and reservoir computing. While I was working on the integration of learning in an agent for a real-time strategy game, I came to realize that the main issue with artificial intelligence (AI) in video games was not the lack of learning, but rather the severe limitations and rigidity of its behavior. Indeed, game agents are generally designed to handle only a handful of possible cases effectively or to consider only certain aspects of the game and completely ignore others. This results in agents that seem to lack even the most basic understanding of their environment and end up making staggering mistakes, utterly failing at delivering an impression of intelligence. The interesting bit here was that these mistakes are related to the basics of the environment and have little to do with the specifics of the game (i.e., common-sense mistakes). Since many video games share the same type of environment, this issue should have been apparent in most video games, and sure enough it was. All the video games I looked at suffered from narrow AI, and when a certain aspect did seem to be appreciated by the AI in one game, it was not in others. I then understood that these virtual environments are a lot more complex than they appear to be, and that it was impossible to create AI that would even have a rudimentary understanding of all the different aspects they involve in the scope of a video game project, not because it was difficult to program, but because it would outweigh the entire video game. For one video game, it certainly did not seem worth the effort. Then again, why would it have to be for

[1] `https://secure.mash-project.eu/`

any one video game when many shared the same type of environment? This question was the spark that led me to temporarily put aside my initial project and set out to solve what constituted in my opinion a more pressing problem.

Acknowledgments

I would like to start by thanking Damien Ernst for introducing me to the fascinating world of research. More importantly, I thank him for granting me the freedom to explore the exciting fields of artificial intelligence and video games without restraint. None of this work would have been possible without his guidance and his patience.

I also thank Raphael Fonteneau for his extensive support every step of the way, his invaluable contributions and all the captivating discussions I had the pleasure to have with him.

I thank the entire Systmod research unit as well as other researchers I had the chance to talk to, especially Rodolphe Sepulchre, Quentin Gemine, David Lupien St-Pierre, Francis Maes, Tobias Yung and Benjamin Laugraud. Additionally, I thank the people at the Inria Saclay laboratory in Orsay, where I worked for three months. In particular, I thank Olivier Teytaud for his outstanding explanations, and Jean-Baptiste Hoock and Nataliya Sokolovska for all the help they provided during my stay. I extend my thanks to all the administrative staff too.

I am grateful to the University of Liège for offering me the opportunity to work alongside my research. I thank all the people who took part in creating such a nice working environment, including Benoit Donnet, Gérard Dethier, Thomas Leuther and Marc Frédéric.

I thank Paolo Di Carlo for his continuous participation in my research and his contributions to this document. I also thank Natalie Solodovnikova for her contributions.

Finally, I would like to express my deepest gratitude to my friends and family for their unconditional support.

Abstract

The work presented in this dissertation revolves around the problem of designing artificial intelligence (AI) for video games. This problem becomes increasingly challenging as video games grow in complexity. With modern video games frequently featuring sophisticated and realistic environments, the need for smart and comprehensive agents that understand the various aspects of these environments is pressing. Although machine learning techniques are being successfully applied in a multitude of domains to solve AI problems, they are not yet ready to enable the creation of fully autonomous agent that can reliably learn to understand the environments found in complex video games. Since video game AI is often specifically designed for each game, video game AI tools currently focus on allowing video game developers to quickly and efficiently create specific AI. One issue with this approach is that it does not efficiently exploit the numerous similarities that exist between video games not only of the same genre, but of different genres too, resulting in a difficulty to handle the many aspects of a complex and realistic environment independently for each video game. These similarities, however, exist on a conceptual level. While video games do indeed share a variety of concepts, their interpretations vary from one game to another. Hence, these similarities can only be directly exploited at a conceptual level. Inspired by the human ability to detect analogies between games and apply similar behavior on a conceptual level, this thesis suggests an approach based on the use of a unified conceptual framework to enable the development of conceptual AI which relies on conceptual views and actions to define basic yet reasonable and robust behavior. Because conceptual AI is not tied to any game in particular, it benefits from a continuous development process as opposed to a development that is confined to the scope of a single game project.

Résumé

Le travail présenté dans ce manuscrit porte sur le problème de la conception d'intelligence artificielle (IA) pour les jeux vidéo. Ce problème devient de plus en plus difficile au fur et à mesure que la complexité des jeux vidéo augmente. Comme les jeux vidéo modernes sont souvent caractérisés par des environnements complexes et réalistes, il y a un besoin urgent de concevoir des agents robustes et intelligents capables d'apprécier les divers aspects de ces environnements. Bien que les techniques d'apprentissage automatique soient aujourd'hui utilisées dans de nombreux domaines pour résoudre des problèmes d'IA, celles-ci ne permettent pas encore la conception d'agents autonomes capables d'apprendre à apprécier la complexité des environnements de jeux vidéo de manière fiable. Étant donné que d'habitude l'IA de jeux vidéo est spécifiquement conçue pour chaque jeu, les outils actuels permettent surtout aux développeurs de jeux vidéo de rapidement et efficacement concevoir de l'IA spécifique. Un des problèmes de cette approche est qu'elle n'exploite pas assez les nombreuses similitudes qui existent entre les jeux vidéo, qu'ils soient du même genre ou non, rendant ainsi difficile la gestion indépendante des plusieurs aspects liés à un environnement complexe et réaliste pour chaque jeu vidéo. Ces similitudes existent cependant sur le plan conceptuel. Bien que les jeux vidéo partagent une série de concepts, leurs interprétations varient d'un jeu à l'autre. Par conséquent, ces similitudes ne peuvent être directement exploitées qu'au niveau conceptuel. En s'inspirant de la capacité humaine à détecter des analogies entre différents jeux et appliquer des comportements similaires sur le plan conceptuel, cette thèse propose une approche fondée sur l'utilisation d'un framework conceptuel unifié pour le développement d'IA conceptuelle qui repose sur des états et des actions conceptuels pour définir un comportement basique mais robuste et raisonnable. Une IA conceptuelle n'étant liée à aucun jeu en particulier, elle bénéficie d'un processus de développement continu, contrairement à un développement confiné au seul projet d'un jeu vidéo.

10

Abbreviations

AA	Action-Adventure
AI	Artificial Intelligence
AoE	Area of Effect
AP	Ability Points
ARPG	Action Role-Playing Game
ATB	Active Time Battle
CC	Crowd Control
CDS	Conceptual Data Space
CF	Conceptual Framework
CTF	Capture the Flag
DDA	Dynamic Difficulty Adjustment
DPS	Damage per Second
FPS	First-Person Shooter
HMD	Head-Mounted Display
HP	Hit Points
JRPG	Japanese Role-Playing Game
MMORPG	Massively Multiplayer Online Role-Playing Game
MMORTS	Massively Multiplayer Online Real-Time Strategy
MMOTBR	Massively Multiplayer Online Turn-Based Racing
MOBA	Multiplayer Online Battle Arena
MP	Magic/Mana Points
NES	Nintendo Entertainment System
N64	Nintendo 64
NUI	Natural User Interface
NPC	Non-Player Character

PC	Personal Computer
PS	PlayStation
PSP	PlayStation Portable
PvE	Player versus Environment
PvP	Player versus Player
RPG	Role-Playing Game
RTS	Real-Time Strategy
TBS	Turn-Based Strategy
UMS	Use Map Settings
WRPG	Western Role-Playing Game
XP	Experience Points

List of Mentioned Commercial Video Games

Title	Genre	Developer	Year
Adventure	AA	Atari, Inc.	1979
Aion: The Tower of Eternity	MMORPG	NCsoft	2008
Angry Birds	Puzzle	Rovio Entertainment	2009
Black & White	Strategy	Lionhead Studios	2001
Command & Conquer: Red Alert 3	RTS	EA Los Angeles	2008
Counter-Strike	FPS	Valve Corporation	1999
Crash Bandicoot	Platform	Naughty Dog	1996
Creatures	Platform	Creature Labs	1996
Darkwind: War on Wheels	MMOTBR	Sam Redfern	2007
Dead or Alive 4	Fighting	Team Ninja	2005
Diablo	ARPG	Blizzard North	1996
Diablo II: Lord of Destruction	ARPG	Blizzard North	2001
Donkey Kong	Platform	Nintendo	1981
Doom	FPS	id Software	1993
Dota 2	MOBA	Valve Corporation	2013

Dune II: The Building of a Dynasty	RTS	Westwood Studios	1992
Dungeon Keeper	Strategy	Bullfrog Productions	1997
Dungeon Siege II	RPG	Gas Powered Games	2005
EverQuest	MMORPG	Sony Online Entertainment	1999
F.E.A.R.: First Encounter Assault Recon	FPS	Monolith Productions	2005
FarmVille	Simulation	Zynga	2009
Final Fantasy	RPG	Square	1987
Final Fantasy VII	RPG	Square	1997
Final Fantasy XII	RPG	Square Enix	2006
Final Fantasy XIII-2	RPG	Square Enix	2011
Forza Motorsport 5	Racing	Turn 10 Studios	2013
Half-Life	FPS	Valve Corporation	1998
League of Legends	MOBA	Riot Games	2009
Left 4 Dead	FPS	Turtle Rock Studios	2008
M.U.L.E.	TBS	Ozark Softscape	1983
Mario Kart 64	Racing	Nintendo EAD	1996
Pac-Man	Maze	Namco	1980
Pole Position	Racing	Namco	1982
Pong	Sports	Atari, Inc.	1972
Quake	FPS	id Software	1996
Ragnarok Online	MMORPG	Gravity Corporation	2002
Resident Evil 5	FPS	Capcom	2009
Shattered Galaxy	MMORTS	Kru Interactive	2001
Space Invaders	Shoot 'em up	Taito Corporation	1978

StarCraft	RTS	Blizzard Entertainment	1998
StarCraft: Brood War	RTS	Blizzard Entertainment	1998
StarCraft II: Wings of Liberty	RTS	Blizzard Entertainment	2010
Super Mario 64	Platform	Nintendo EAD	1996
Super Mario Bros.	Platform	Nintendo EAD	1985
Tekken: Dark Resurrection	Fighting	Namco	2006
Tekken 6	Fighting	Bandai Namco Games	2009
Tetris	Puzzle	Alexey Pajitnov	1984
The Legend of Zelda: Ocarina of Time	AA	Nintendo EAD	1998
The Witcher 2: Assassins of Kings	ARPG	CD Projekt RED	2011
Tomb Raider	AA	Core Design	1996
Ultima Online	MMORPG	Origin Systems	1997
Unreal Tournament	FPS	Epic Games	1999
Unreal Tournament 2004	FPS	Epic Games	2004
Virtua Fighter 5	Fighting	Sega	2007
Warcraft: Orcs and Humans	RTS	Blizzard Entertainment	1994
Warcraft III: The Frozen Throne	RTS	Blizzard Entertainment	2003
Watch Dogs	AA	Ubisoft Montreal	2014
Wolfenstein 3D	FPS	id Software	1992
World of Warcraft	MMORPG	Blizzard Entertainment	2004

16

Zanac	Shoot 'em up	Compile	1986

Contents

List of Figures

Chapter 1

Introduction

Video games appeared in human history a few decades ago. Today, they represent one of the most widespread forms of entertainment. They have considerably evolved since their first appearance, as illustrated in Figure 3.8. Many aspects have changed. Video games are now richer and more complex. The environments are more detailed. The ways of interacting with video games are also changing with the advent of natural user interface (NUI) technology, which turns player gesture and speech into game controls, and virtual reality technology, which allows players to be fully immersed in virtual environments. They are increasingly accessible with the extensive integration of powerful processors in mobile devices. Video games are no longer restricted to computers and consoles and are widely available on tablets and smartphones. Their substantial diversity ensures that they generate interest for many different people. They generate enough interest for serious competitions to be organized, such as The International, an annual *Dota 2* Championship. The International 2014 held in Seattle featured a total prize pool of over $10,000,000 and a first place prize of over $5,000,000[1]. [6]

A video game has several facets, including the gameplay, the setting, the story and the interface. Gameplay includes components such as rules, objectives and challenges. The setting includes components such as the context or the universe in which the game takes place. The story includes components such as characters, quests and the plot followed by the game. The interface includes components such as user controls, environment and character design and music, sound effects and voice-overs. Depending on

[1]That number is comparable to the $35,000,000 prize of the 2014 FIFA World Cup [8] considering that The International teams are composed of 5 players while the FIFA World Cup teams are composed of 23 players.

(a) Adventure (1979) **(b)** Watch Dogs (2014)

Figure 1.1: Evolution of video games. At the left is Atari's action-adventure game *Adventure*. At the right is Ubisoft's action-adventure game *Watch Dogs*.

the type of the game, some components can be more developed than others. For example, a video game could focus on storytelling and develop the related components while simplifying gameplay, or it could focus on the mental and physical challenges and develop unique gameplay components without including a story, or it could focus on the interface and feature a detailed and beautiful world with a simple story and generic gameplay. More simply, it is possible to consider only two sides to a video game, the game side and the context side. The game side includes the gameplay and interface facets while the context side includes the setting and story facets. Because this work focuses on the game side, it is important to keep in mind that unless otherwise specified, the scope of discussion is generally limited to the game side.

One of the appealing features of video games is that they allow people to easily interact with rich and complex environments. In a real-time strategy game, players build bases, gather resources, research technologies, produce units and attack and defend against enemies. In a driving or flight simulation game, players can maneuver a vehicle in an environment with realistic physics. Players can break into a secure facility or traverse a minefield in an action-adventure game. Carrying out such tasks could be difficult in real life. Moreover, the fact that these environments can be the result of artistic work adds to the appeal of video games. Compared to other art forms such as novels or films, video games present the advantage of allowing players to interact with the artists' creations. Rather than discovering a fantasy world through the author's description, the author can create the world and let players freely explore it on their own. They could then get to choose how to interact with different characters, resulting in a more personal experience for each of them. Thus, video games can also be

used as an art medium. They can mix both game and art elements to offer fun while expressing creativity.

Typically, players incarnate an entity in the game world such as a character and interact with the environment through it. In most cases, the world is populated with other active entities, such as monsters or non-player characters and, in multiplayer games, with entities controlled by other players. These entities are agents. Agents largely contribute to the complexity of the environment. Agents that are controlled by the game, or agents for short, can be adaptive or nonadaptive. An example of nonadaptive agent is an immobile guard in a platform game who fires a gun in the same direction every two seconds. On the other hand, adaptive agents need to be able to sense the environment and react to it in some way. Albeit basic, Goombas in *Super Mario Bros.*, a popular platform game shown in Figure 1.2, are adaptive agents. Likewise, a fiend that chases its prey in a role-playing game or a computer player in a first-person shooter game are adaptive agents. Of course, the more complex the environment grows, the smarter adaptive agents need to get and the harder it becomes to create them. In this context, artificial intelligence (AI) is used to create desirable behavior, such as making a challenging agent, a realistic agent or a funny agent. Unfortunately, the complexity of the environments has grown faster than that of the adaptive agents, with contemporary games often featuring huge realistic worlds but no agents that understand or deal effectively with all the concepts they involve. Most of the time, creating robust behavior comes at the cost of limiting their interaction with the environment in order to keep the number of possible scenarios manageable. For example, it is possible to create robust behavior for a character by constraining them to a certain region of the world which only involves a small subset of the elements of the game. In other cases where adaptive agents have more freedom, the AI is simplified by following general rules without considering all details of the environment or by focusing on a handful of specific yet frequent situations. This results in rigid AI whose limitations can quickly be exposed by exploiting crucial details it discards or by driving it into unusual situations. In other words, AI may fail to provide a consistent challenge or it may undermine the realistic experience a game is expected to deliver. Note that only the AI related to the game side is considered here and throughout this work. It is also worth noting that AI is used in the context side too to control elements related to the story and setting such as character dialog or story events. This is typically achieved through scripting to allow authors to easily write and modify dialog or events.

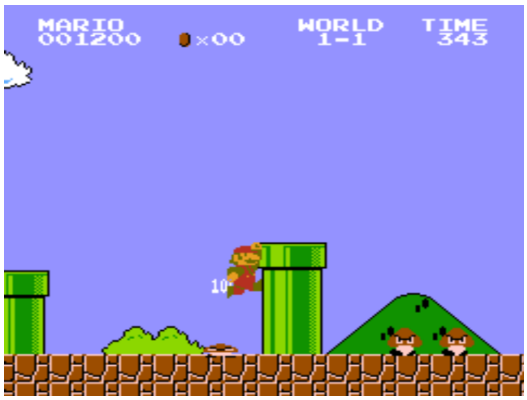


Figure 1.2: Goombas in Super Mario Bros. are basic adaptive agents. Goombas are always moving either left or right and toggle their movement direction every time they bump into an obstacle.

If AI fell behind virtual environments, it is certainly not due to neglect. The importance of AI is reflected by the global efforts put into improving it. Video game developers strive to make better AI for their games. Despite their limitations, adaptive agents such as monsters or non-player characters exhibit increasingly natural behavior in order to blend into their environment as well as possible. They can get angry or be afraid, remember events or people and cooperate with one another to accomplish their goals. Though adaptive agents possess several advantages compared to humans such as being able to always process every detail of the game state, being able to perfectly assess a target distance, being able to compute a firing angle with precision, being able to produce the desired input with no error and never forgetting to check an element or take a particular action, these can be restrained to conceal the machine behind them and give the illusion of life. Machine learning techniques such as reinforcement learning and imitation learning are being explored to create agents that can learn from their mistakes or from human players[2]. AI is also frequently used to improve player experience by automatically tweaking the game according to player performance. This can consist in adapting the difficulty of the game either by modifying the skill level or strength of AI opponents or by generating new content based on player performance. Another example is ranking players by skill and composing even teams to create exciting matchups in competitive multiplayer games.

Since video games are designed for human beings, it is only natural that

[2]See Section 4.3.3.

they focus on their cognitive skills and physical abilities. The richer and more complex a game is, the more skills and abilities it requires. Thus, creating a truly smart and fully autonomous agent for a complex video game can be as challenging as replicating a large part of the complete human intelligence. On the other hand, AI is usually independently designed for each game. This makes it difficult to create thoroughly robust AI because its development is constrained to the scope of an individual game project. Although each video game is unique, they can share a number of concepts depending on their genre. Genres are used to categorize video games according to the way players interact with them as well as their rules. On a conceptual level, video games of the same genre typically feature similar challenges based on the same concepts. These similar challenges then involve common problems for which basic behavior can be defined and applied regardless of the problem instance. For example, in a first-person shooter one-on-one match, players face problems such as weapon selection, opponent position prediction and navigation. An example of first-person shooter one-on-one match is shown in Figure 1.3. Each moment, a player needs to evaluate the situation and switch to the most appropriate weapon, predict where the opponent likely is or is heading and find the best route to get there. All of these problems can be reasoned about on a conceptual level using data such as the rate of fire of a weapon, the current health of the opponent and the location of health packs. These concepts are common to many first-person shooter games and are enough to define effective behavior regardless of the details of their interpretation. Such solutions already exist for certain navigation problems for instance and are used across many video games. Moreover, human players can often effortlessly use the experience acquired from one video game in another of the same genre. A player with experience in first-person shooter games will in most cases perform better in a new first-person shooter game than one without any experience and can even perform better than a player with some experience in the new game, indicating that it is possible to apply the behavior learned for one game in another game featuring similar concepts to perform well without knowing the details of the latter. Obviously, when the details are discovered, they can be used to further improve the basic conceptual behavior or even override it. It may therefore be possible to create cross-game AI by identifying and targeting conceptual problems rather than their game-specific instances. Detaching AI or a part of it from the development of video games would remove the project constraints that push developers to limit it and allow it to have a continuous and more thorough design process.

This document contains six primary chapters in addition to the Intro-

Figure 1.3: *Unreal Tournament* by Epic Games. During a match, a player needs to decide which weapon to use depending on the situation. Some examples of weapon properties which can be reasoned about on a conceptual level are whether the weapon fires slow projectiles or instantly hits the target (hitscan), whether it deals area-of-effect damage, its rate of fire and its precision.

duction and Conclusion chapters. Chapter 2 presents some related work and explains how this work positions itself beside it. Chapter 3 outlines the world of video games using a number of video game genre and title examples in order to provide readers with a set of references used as a basis for various discussions. Chapter 4 focuses on the particularities of AI in video games in order to clarify the context as well as the purpose of AI in that context. In Chapter 5, the problem driving this work is briefly described. A development model for video game AI is then presented in Chapter 6. Finally, Chapter 7 includes some applications of the development model to illustrate its deployment. The Conclusion chapter discusses some of the merits of the proposed approach and notes a few perspectives for the extension of this research. The document is ended with an Appendix chapter which includes two articles that do not directly pertain to this contribution but nevertheless played a part in steering the research in this direction.

Below follows a list of publications of some of the work conducted throughout this research:

- **F. Safadi, R. Fonteneau, D. Ernst**, Artificial intelligence design for real-time strategy games, 25th Annual Conference on Neural Information Processing Systems (NIPS 2011), Workshop on Decision Making with Multiple Imperfect Decision Makers, Sierra Nevada, Spain, December 16th, 2011
  Abstract – For the past two decades, real-time strategy (RTS) games have steadily gained in popularity and have become common in video game leagues. Without a doubt one of the most complicated genre,

RTS games are challenging to both human and artificial intelligence. Regrettably, intelligent agents continue to pale in comparison to human players and fail to display seemingly intuitive behavior that even novice players are capable of. Working towards improving the performance of such agents, we present a clear and complete yet generic AI design in this paper.

- **Q. Gemine, F. Safadi, R. Fonteneau, D. Ernst**, Imitative learning for real-time strategy games, 8th IEEE Conference on Computational Intelligence and Games (CIG 2012), Granada, Spain, September 11-14, 2012
  Abstract – Over the past decades, video games have become increasingly popular and complex. Virtual worlds have gone a long way since the first arcades and so have the artificial intelligence (AI) techniques used to control agents in these growing environments. Tasks such as world exploration, constrained pathfinding or team tactics and coordination just to name a few are now default requirements for contemporary video games. However, despite its recent advances, video game AI still lacks the ability to learn. In this paper, we attempt to break the barrier between video game AI and machine learning and propose a generic method allowing real-time strategy (RTS) agents to learn production strategies from a set of recorded games using supervised learning. We test this imitative learning approach on the popular RTS title *StarCraft II: Wings of Liberty* and successfully teach a Terran agent facing a Protoss opponent new production strategies.

- **F. Safadi, R. Fonteneau, D. Ernst**, Artificial Intelligence in Video Games: Towards a Unified Framework, Submitted to the International Journal of Computer Games Technology, August 2014
  Abstract – With modern video games frequently featuring sophisticated and realistic environments, the need for smart and comprehensive agents that understand the various aspects of complex environments is pressing. Since video game AI is often specifically designed for each game, video game AI tools currently focus on allowing video game developers to quickly and efficiently create specific AI. One issue with this approach is that it does not efficiently exploit the numerous similarities that exist between video games not only of the same genre, but of different genres too, resulting in a difficulty to handle the many aspects of a complex environment independently for each video game. Inspired by the human ability to detect analogies between games and apply similar behavior on a conceptual level, this

article suggests an approach based on the use of a unified conceptual framework to enable the development of conceptual AI which relies on conceptual views and actions to define basic yet reasonable and robust behavior. The approach is illustrated using two video games, *Raven* and *StarCraft: Brood War*.

- **F. Safadi, R. Fonteneau, D. Ernst**, Parallel Cascade Correlation: A GPU Implementation, Submitted, August 2014
  Abstract – This paper presents a new implementation of the cascade correlation architecture which leverages the parallel computing capabilities of GPUs. It shows that by combining the inherent parallelization potential of evolutionary algorithms with the caching properties of the cascade correlation architecture, the algorithm can be simplified to a few easily parallelizable operations. It also comes with an open-source CUDA C++ implementation of the resulting genetic cascade correlation algorithm.

Chapter 2

Related Work

Conceptualizing video games is a process which involves abstraction and is similar to many other approaches that share the same goal, namely that of factoring AI in video games. More generally, abstraction makes it possible to create solutions for entire families of problems that are essentially the same when a certain level of detail is omitted. For example, the problem of sorting an array can take different forms depending on the type of elements in the array, but considering an abstract data type and comparison function allows a programmer to write a solution that can sort any type of array. This prevents unnecessary code duplication and helps programmers make use of existing solutions as much as possible so as to minimize development efforts. Another example of widely used abstraction application is hardware abstraction. Physical components in a computer can be seen as abstract devices in order to simplify software development. Different physical components that serve the same purpose, storage for example, can be abstracted into a single abstract storage device type, allowing software developers to write storage applications that work with any kind of storage component. Such a mechanism is used in operating systems such as NetBSD [138] and the Windows NT operating system family [101].

The idea of creating a unified video game AI middleware is not new. The International Game Developers Association (IGDA) launched an Artificial Intelligence Interface Standards Committee (AIISC) in 2002 whose goal was to create a standard AI interface to make it possible to recycle and even outsource AI code [106]. The committee was composed of several groups, each group focusing on a specific issue. There was a group working on world interfacing, one on steering, one on pathfinding, one on finite state machines, one on rule-based systems and one on goal-oriented action planning, though the group working on rule-based systems ended

up being dissolved [106, 107, 108]. Thus, the committee was concerned not only with the creation of a standard communication interface between video games and AI, but with the creation of standard AI as well [152]. It was suggested that establishing AI standards could lead to the creation of specialized AI hardware.

The idea of creating an AI middleware for video games is also discussed in Karlsson [82], where technical issues and approaches for creating such middleware are explored. Among other things, it is argued that when state systems are considered, video game developers require a solution in between simple finite state machines and complex cognitive models. Another interesting argument is that functionality libraries would be more appropriate than comprehensive agent solutions because they provide more flexibility while still allowing agent-based solutions to be created. Here too, the possibility of creating specialized AI hardware was mentioned and a parallel with the impact mainstream graphics acceleration cards had on the evolution of computer graphics was drawn.

An Open AI Standard Interface Specification (OASIS) is proposed in Berndt et al. [47], aiming at making it easier to integrate AI in video games. The OASIS framework is designed to support knowledge representation as well as reasoning and learning and comprises five layers each dealing with different levels of abstraction, such as the object level or the domain level, or providing different services such as access, translation or goal arbitration services. The lower layers are concerned with interacting with the game while the upper layers deal with representing knowledge and reasoning.

Evidently, video game AI middleware can be found in video game engines too. Video game engines such as *Unity* [22], *Unreal Engine* [23], *CryEngine* [4] and *Havok* [12], though it may not be their primary focus, increasingly aim at not only providing building blocks to create realistic virtual environments but realistic agents as well.

Another approach that, albeit not concerned with AI in particular, also shares a similar goal, which is to factor development efforts in the video game industry, is game patterns. Game design patterns allow game developers to document recurring design problems and solutions in such a way that they can be used for different games while helping them understand the design choices involved in developing a game of specific genre. Kreimeier [87] proposes a pattern formalism to help expanding knowledge

about game design. The formalism describes game patterns using four elements. These are the name, the problem, the solution and the consequence. The problem describes the objective and the obstacles that can be encountered as well as the context in which it appears. The solution describes the abstract mechanisms and entities used to solve the problem. As for the consequence, it describes the effect of the design choice on other parts of the development and its costs and benefits.

Björk et al. [49] differentiates between a structural framework which describes game components and game design patterns which describe player interaction while playing. The structural framework includes three categories of components. These are the bounding category, which includes components that are used to describe what activities are allowed or not in the game such as rules and game modes, the temporal category which includes components that are involved in the temporal execution of the game such as actions and events, and the objective category which includes concrete game elements such as players or characters. More details about this framework can be found in Björk and Holopainen [48]. As for game design patterns, they do not include problem and solution elements as they do in Kreimeier [87]. They are described using five elements which are name, description, consequences, using the pattern and relations. The consequences element here focuses more on the characteristics of the pattern rather than its impact on development and other design choices to consider, which is the role of the using the pattern element. The relations element is used to describe relations between patterns, such as subpatterns in patterns and conflicting patterns.

In Olsson et al. [111], design patterns are integrated within a conceptual relationship model which is used to clarify the separation of concerns between game patterns and game mechanics. In that model, game mechanics are derived from game patterns through a contextualization layer whose role is to concretize those patterns. Conversely, new patterns can be extracted from the specific implementation of these game mechanics, which in the model is represented as code.

Also comparable are approaches which focus on solving specific AI issues. It is easy to see why, since these approaches typically aim at providing standard solutions for common AI problems in video games, thereby factoring AI development. For instance, creating models for intelligent video game characters is a widely researched problem for which many approaches have been suggested. Behavior languages aim to provide an

agent design model which makes it possible to define behavior intuitively and factor common processes. Loyall and Bates [95] presents a goal-driven reactive agent architecture which allows events that alter the appropriateness of current behavior to be recognized and reacted to. ABL, a reactive planning language designed for the creation of believable agents which supports multi-character coordination, is described in Mateas and Stern [96] and Mateas and Stern [97].

Situation calculus was suggested as a means of enabling high-level reasoning and control in Funge [69]. It allows the character to see the world as a sequence of situations and understand how it can change from one situation to another under the effect of different actions in order to be able to make decisions and achieve goals. A cognitive modeling language (CML) used to specify behavior outlines for autonomous characters and which employs situation calculus and exploits interval methods to enable characters to generate action plans in highly complex worlds is also proposed in Funge et al. [68], Funge [67].

It was argued in Orkin [113, 114] that real-time planning is a better suited approach than scripting or finite state machines for defining agent behavior as it allows unexpected situations to be handled more naturally. A modular goal-oriented action planning architecture for game agents similar to the one used in Mateas and Stern [96, 97] is presented. The main difference with the ABL language is that a separation is made between implementation and data. With ABL, designers implement the behavior directly. Here, the implementation is done by programmers and designers define behavior using data.

Anderson [32] suggests another language for the design of intelligent characters. The avatar definition language (AvDL) enables the definition of both deterministic and goal directed behavior for virtual entities in general. It was extended by the Simple Entity Annotation Language (SEAL) which allows behavior definitions to be directly embedded in the objects in a virtual world by annotating and enabling characters to exchange information with them [33, 34].

Finally, learning constitutes a different approach which, again, leads to the same goal. By creating agents capable of learning from and adapting to their environment, the issue of designing intelligent video game characters is solved in a more general and reusable way. Video games have drawn extensive interest from the machine learning community in the last

decade and several attempts at integrating learning in video games have been made with varying degrees of success. Some of the methods used are similar to the previously mentioned approaches in that they use abstraction or concepts to deal with the large diversity found in video games. Case-based reasoning techniques generalize game state information to make AI behave more consistently across distinct yet similar configurations. The possibility of using case-based plan recognition to reduce the predictability of real-time strategy computer players is discussed in Cheng and Thawonmas [55]. Aha et al. [28] presents a case learning and plan selection approach used in an agent that learns to win against a number of different AI opponents in *Wargus*. In Ontañón et al. [112], a case based planning framework for real-time strategy games which allows agents to automatically extract behavioral knowledge from annotated expert replays is developed and successfully tested in Wargus as well. More work using Wargus as a test platform includes Weber and Mateas [144] and Weber and Mateas [145] which demonstrate how conceptual neighborhoods can be used for retrieval in case-based reasoning approaches.

Transfer learning approaches attempt to use the experience learned from some task to improve behavior in other tasks. In Sharma et al. [134], transfer learning is achieved by combining case-based reasoning and reinforcement learning and used to improve performance over successive games against the AI in *MadRTS*. Lee-Urban et al. [90] also uses MadRTS to apply transfer learning using a modular architecture which integrates hierarchical task network (HTN) planning and concept learning. Transfer of structure skills and concepts between disparate tasks using a cognitive architecture is achieved in Shapiro et al. [133].

Although machine learning technology may lead to the creation of a unified AI that can be used across multiple games, it currently suffers from a lack of maturity. Even if some techniques have been successfully applied to a few commercial games, it may take a long time before they are reliable enough to become mainstream. On the other hand, video game engines are commonly used and constitute a more practical approach at factoring game development processes to improve the quality of video games. They are however comprehensive tools which developers need to adopt for the entire game design rather than just their AI. Furthermore, they allow no freedom in the fundamental architecture of the agents they drive.

The approach presented in this article bears the most resemblance to that of creating a unified AI middleware. It is however not an AI middle-

ware, strictly speaking. It makes use of a conceptual framework as the primary component which enables communication between video games and AI, allowing video game developers to use conceptual, game-independent AI in their games at the cost of handling the necessary synchronization between game data and conceptual data. A key difference with previous work is that it makes no assumptions whatsoever on the way AI should be designed, such as imposing an agent model or specific modules. Solutions can be designed for any kind of AI problem and in any way. A clear separation is made between the development of the conceptual framework, that of AI and that of video games. Because AI development is completely separated from the conceptual framework, its adoption should be easier as it leaves complete freedom for AI developers to design and implement AI in whichever way they are accustomed to. Furthermore, the simplicity of the approach made it possible to provide a complete deployment example detailing how an entire video game was rewritten following the proposed design. In addition, the resulting limited conceptual framework prototype was successfully employed to reuse some of the game AI modules in a completely different game.

Chapter 3

Video Games

Contents

Summary

The purpose of this chapter is to familiarize the reader with video games. It is divided into three sections. The first section provides a brief overview of the history and evolution of video games. The second section presents the diversity of the current video game landscape. The third section concludes the chapter with illustrations of some notable modern video game genres so as to supply the reader with a broad knowledge of the challenges players face in contemporary titles.

Figure 3.1: Nimatron (left) and Spacewar! (right). Images retrieved on the Internet from `1939nyworldsfair.com` and `wikipedia.org`.

3.1 Brief History

The first video game instance dates back to 1940. Westinghouse Electric's "Nimatron", designed by Edward U. Condon, was a machine that challenged players at *Nim*, a mathematical game in which players take turns removing elements from distinct sets until none are left. In 1947, Thomas T. Goldsmith Jr. and Estle Ray Mann designed a "cathode ray tube amusement device" where players had to aim and shoot a target in a limited amount of time using knobs and buttons. The U.S. Military ORO (Operations Research Office) designed *Hutspiel* in 1955, a theater-level war game where players Blue and Red, representing NATO and the USSR, faced each other by allocating troops, nuclear weapons and aircraft sorties among sectors and targets. *Tennis for Two*, the predecessor of the famed *Pong* game, was designed in 1958 by William A. Higinbotham using an analog computer together with an oscilloscope display. Four years later, Steve Russell produced the first version of *Spacewar!* in 1962. The game, two moving spaceships each controlled by a player that must shoot the other, quickly became popular and inspired others to write new games. A competition for *Spacewar!* took place in 1972 at Stanford University. That same year the successful arcade table tennis game *Pong* was developed by Atari.

The release of the Atari 2600 in 1977, which featured joysticks and game cartridges, started bringing video games to private consumers. Following the success of *Pong*, the shoot 'em up arcade title *Space Invaders* developed by Taito Corporation in 1978 revitalized the coin-operated en-

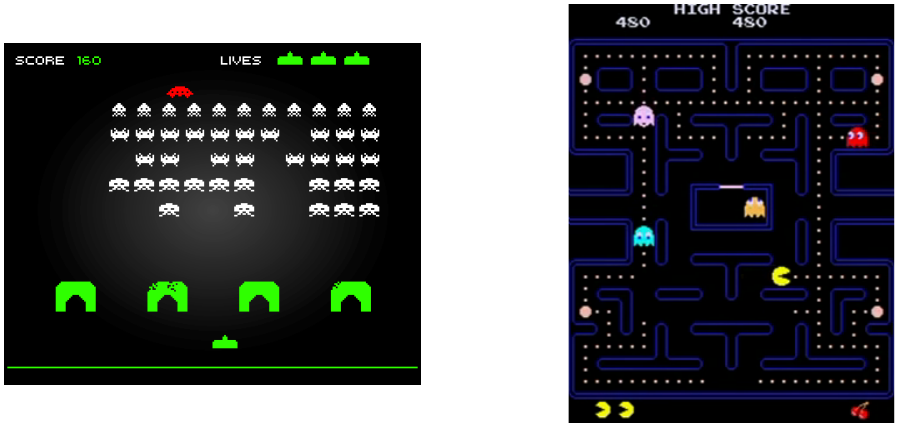


Figure 3.2: Space Invaders (left) and Pac-Man (right).

tertainment business. New genres started to emerge including maze games (*Pac-Man*, Namco, 1980), platform games (*Donkey Kong*, Nintendo, 1981) and racing games (*Pole Position*, Namco, 1982). At that time, in 1981, Atari held a *Space Invaders* championship in the United States in which more than ten thousand people participated. This event helped establish competitive gaming as a major hobby. Later in 1983, Ozark Softscape's *M.U.L.E.*, a multiplayer strategy game, shed some light on multiplayer gaming. *Tetris*, the famous puzzle game, was created by Alexey Pajitnov in 1984. Famicom, better known as the NES (Nintendo Entertainment System), only reached North America in 1985 two years after its original release in Japan in 1983, and a year later in Europe. Together with the release of the Nintendo Game Boy and the Sega Mega Drive (originally released as the Sega Genesis in 1988 in Japan) in 1989, these consoles anchored video games in many homes. It was during this period that popular video game series and franchises such as *Super Mario* (Nintendo, 1985), *The Legend of Zelda* (Nintendo, 1986) and *Final Fantasy* (Square, 1987) started to appear. Shortly afterwards titles such as *Wolfenstein 3D* (id Software, 1992), *Dune II: The Building of a Dynasty* (Westwood Studios, 1992), *Doom* (id Software, 1993) and *Warcraft: Orcs and Humans* (Blizzard Entertainment, 1994) breathed new life in the world of PC gaming.

Around the time the Sega Saturn, the Sony PlayStation and the Nintendo 64 were released in 1994–1996, 3D graphics made their way into many titles and started a race in the pursuit of reality rendering which continues to this day. It was then not long before games like *Quake* (id Software, 1996), *Super Mario 64* (Nintendo EAD, 1996), *Crash Bandicoot* (Naughty Dog, 1996), *Tomb Raider* (Core Design, 1996), *Final Fantasy VII*

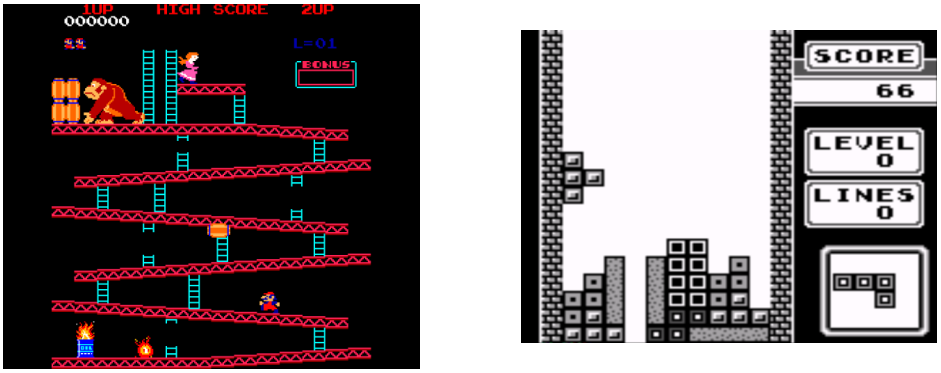


Figure 3.3: Donkey Kong (left) and Tetris (right).

Figure 3.4: Doom (left) and Warcraft: Orcs and Humans (right).

(Square, 1997), *StarCraft* (Blizzard Entertainment, 1998), *Half-Life* (Valve Corporation, 1998) and *The Legend of Zelda: Ocarina of Time* (Nintendo EAD, 1998) forever changed the world of video games along with players' expectations of video game entertainment. In 1997, a *Quake* tournament was held at E3 (Electronic Entertainment Expo) featuring John D. Carmack's[1] turbocharged 1987 Ferrari 328 GTS as the winning prize. By then, most major video game genres had already ripened.

As Internet connectivity started to become mainstream, new forms of large-scale player interaction were imagined and eventually led to the birth of games such as *Ultima Online* (Origin Systems, 1997) and *Shattered Galaxy* (Kru Interactive[2], 2001) and the emergence of the massively multiplayer online role-playing game (MMORPG) and massively multiplayer online real-time strategy (MMORTS) genres, respectively. Many successful community-focused titles followed, including *EverQuest* (Sony Online Entertainment, 1999), *Ragnarok Online* (Gravity Corporation, 2002) and *World of Warcraft* (Blizzard Entertainment, 2004). A few years later, the growth of the mobile

[1] id Software co-founder and 3D graphics innovator.
[2] Formerly Nexon Inc..

Figure 3.5: Super Mario 64 (left) and Tomb Raider (right).

Figure 3.6: Shattered Galaxy (left) and Ragnarok Online (right).

market became an opportunity for new developers to make non graphics-heavy video games for mobile platforms. Games like *Angry Birds* (Rovio Entertainment, 2009) rapidly gained popularity and the trend soon spread to online platforms, such as Facebook, and resulted in games like *FarmVille* (Zynga, 2009).

Meanwhile, in the console market, some developers were focused on the design of more engaging gaming experiences. Motion-sensing controllers such as the Wii Remote (Nintendo, 2006) or the PlayStation Move (Sony Computer Entertainment, 2010) let people interact with video games in more intuitive ways. The Kinect (Microsoft, 2010), a motion capture input device combining an RGB camera and an infrared camera, took a step further in this direction and eliminated the need for a controller. In 2012, Palmer Luckey designed the Oculus Rift, a virtual reality HMD (head-mounted display) which immerses players in virtual worlds.

3.2 The Landscape Today

Video games today are part of mainstream entertainment. Newzoo reports an estimate of 1.2 billion active gamers across the world in 2013. [78] The ISFE (Interactive Software Federation of Europe) 2012 consumer study shows a somewhat even gaming distribution over age and gender in Europe, with 51% of gamers being aged under 35 versus 49% aged 35 and over and a gender breakdown of 55% to 45% in favor of male gamers. [109] This important gaming incidence is in part due to the high diversity of video games.

One of the factors contributing to this diversity is the multitude of platforms on which players can enjoy video games. Video games can be developed for arcades, although these have been limited to a niche market for the last decade, for consoles, like the Wii (Nintendo), the PlayStation 3 (Sony Computer Entertainment) and the Xbox 360 (Microsoft), for PC systems like Microsoft Windows (Microsoft), Mac OS (Apple Inc.) and Linux distributions, for handheld consoles like the Nintendo 3DS (Nintendo) and the PlayStation Vita (Sony Computer Entertainment), for mobile systems like Android (Google), iOS (Apple Inc.) and Windows Phone (Microsoft) as well as for a variety of open standards like ECMAScript[3] and HTML5 or proprietary software like Adobe Flash (Adobe Systems) used on the Internet. Consequently, video games are accessible almost everywhere, be it at home, at work or on the go. This in turn increases the diversity of video games which are developed for different settings, including the living room, the desk or travel, leading to games with varying degrees of input/output sophistication, time consumption and social interaction.

Because of their large number, video games are categorized using genre and theme labels. Genres are used to distinguish games based on their gameplay, while themes are used to characterize the setting or ambiance of a game. Gameplay deals with concepts and abstract game elements such as the way players interact with and the rules of a game. For example, the multiplayer qualifier, which indicates that two or more players can play the game together, is a gameplay feature. The shooter qualifier, which indicates that among the tasks a player needs to complete is aiming and shooting targets, is another example. Conversely, themes are independent of gameplay and serve to depict the form of the objects that implement it. For instance, the concept of player can take the form of a turtle, a pilot or a god in differ-

[3] Better known for one of its implementations named JavaScript.

ent games. Likewise, shooting can be done with arrows, bullets or divine thunderbolts.

The array of genres to choose from is wide. In fact, there are enough genres for them to be categorized as well. Most genres belong to one or several larger families of genres, a genre family being nothing more than a generic genre. Action, adventure and strategy are examples of generic genres to name a few. The action genre usually relies on the player's dexterity to quickly and accurately react to the game. It includes many popular genres such as shooter, fighting and racing games. In the adventure genre, players have to progress through a storyline or explore a world, unlocking new areas or triggering events by acquiring specific items, interacting with objects or NPCs (non-player characters) or solving puzzles. Several genres fall within this category, including genres of action games like action-adventure (AA) games and genres of role-playing games (RPGs) like Japanese RPGs (JRPGs)[4]. Strategy too, which focuses on the player's ability to plan ahead and assess various multi-step scenarios, is a generic genre. It powers a number of genres, including many turn-based genres and board games as well as real-time, fast-paced genres such as real-time strategy (RTS) and multiplayer online battle arena (MOBA). Of course, it is clear that games can belong to multiple genres, and genres can be composed of other genres. One way to visualize this is to think of genres as sets of gameplay features which can be either combined or further specified into new genres. Video games can still be labeled using only the genres which developers, or perhaps players, believe are representative of the essential challenges they offer without preventing them from belonging to the genres of the less dominant features. Some modern genres, like the MMORPG genre, are increasingly rich and complex and draw features from many other genres.

Like genres, themes are numerous and varied. Science fiction, fantasy, post-apocalyptic, space, cyberpunk, mecha and medieval are all examples of themes. Themes can also be specialized and mixed. For example, the cyberpunk and mecha themes are both science fiction themes, while science fantasy mixes elements from both science fiction and fantasy themes. Unlike genres however, themes are not specific to video games and are used in other fields such as literature and cinematography.

[4]Japanese here is not a theme and actually refers to a set of gameplay features which are commonly found in Japanese titles.

As video games are rooted deeper in society, and thanks to the large audience and interest they get, professional gaming is increasingly acknowledged as a serious occupation and esports (electronic sports) are becoming common. Today, there are several international tournaments, such as the World Cyber Games (WCG), the Electronic Sports World Cup (ESWC), the Intel Extreme Masters (IEM) and the League of Legends Championship Series (LCS) World Championship, featuring different genres of competitive match-based games, such as first-person shooter games (e.g., *Quake*, *Counter-Strike*, *Unreal Tournament 2004*), real-time strategy games (e.g., *StarCraft: Brood War*, *Warcraft III: The Frozen Throne*), multiplayer online battle arena games (e.g., *League of Legends*, *Dota 2*) and fighting games (e.g., *Dead or Alive 4*, *Virtua Fighter 5*, *Tekken 6*). The LCS Season 3 World Championship held in 2013 in Los Angeles featured a total prize pool of $2,050,000 and a first place prize of $1,000,000. [11] The participants were recognized by the U.S. government as professional athletes. [10]

3.3 Modern Genres: Illustrations

3.3.1 First-Person Shooter

Although the origin of first-person shooter (FPS) games can be traced back to the 1970s, it wasn't until the early 1990s that they became popular and rose as one of the major genres available today. At the core of the genre, players incarnate a maneuverable avatar with sight in a 3-dimensional world and must destroy targets by aiming and shooting them. Players see the world through the eyes of their avatar in a first-person perspective, hence the qualifier in the genre name. Usually, players have an inventory of various firearms they can switch to at any time during combat. These, along with ammo and power-ups, can be acquired from different locations in the game. FPS games can focus on either a single player experience or a multiplayer experience, or develop both. Player challenges can drastically vary depending on the mode.

In a single player mode, the game often follows a storyline in a way similar to the adventure genre. Players must complete levels or advance in a world by reaching key areas, activating objects or collecting special items. Combat is still the primary purpose, as players continuously encounter enemies trying to halt their progress.

Most of the time, they find themselves in a one-on-many situation and

have to quickly make decisions, like prioritizing targets and switching weapons, to minimize the risk of taking damage. For example, in the presence of strong but slow melee monsters and fast monsters that can easily close in on the player's position, the latter can focus on the fast enemy while moving around and making sure the strong one never gets an opportunity to attack. Also, depending on the target speed and distance, some weapons can be more efficient than others. Other examples include scenarios where players can take cover behind objects or pull some enemies, when there are too many, to a different location and fight them in smaller groups. In many cases, when facing a single enemy, it is because it constitutes a significant threat to the player on its own. A typical instance of this is a boss fight. Boss enemies are much tougher and hit harder than regular enemies. They are found at key locations and, unlike other enemies which can be avoided, must be defeated in order to progress through the storyline. Sometimes, boss enemies are too strong to be defeated in a normal way and players have to figure out tricks to weaken them. For example, players can work out that destroying the lights in a room will severely decrease the boss' accuracy and give them a fighting chance against an opponent who would otherwise quickly put a bullet in their head, or that activating a trigger will release a plasma beam that melts the boss' carapace and expose its vital points. Thus, players have to combine dexterity together with tactical and puzzle-solving skills to overcome the challenges of a single player campaign.

In multiplayer mode, the action commonly takes the form of matches and focuses on competitive play. Examples of match types include deathmatch, where players must all kill each other, team deathmatch, where teams must kill each other, capture the flag (CTF), where teams must capture the opposing team's flag and return it to their base, and domination, where teams have to assume control of different areas on the map.

The simplest match type, the classical one-on-one deathmatch, can already offer interesting challenges. Besides basic combat skills such as aiming and dodging, players also have to know the map and the location of the weapons and power-ups and their respawn rate. This allows players to control the map, making sure that they are always picking up important items as soon as they spawn and denying them to their opponent. Predicting the opponent's position, not in an encounter but when out of sight, is another crucial skill. By accurately modeling the opponent's movement, players can fire projectiles preemptively and deal damage before the enemy is engaged. They can avoid their opponent or guess any weapons or power-ups they

could have acquired too. Incidentally, planning is an integral part of the game. Players can often outsmart their opponent by planning several steps ahead to gain a long-term or tactical advantage. For instance, getting killed to pick up a particular item can prove ultimately rewarding in some cases. Of course, like in most competitions, this is all complemented by strategy. Figuring out the opponent's strengths and weaknesses and exploiting them throughout the competition, like avoiding long-distance encounters against a sharpshooter, is essential for winning. Naturally, team matches offer similar challenges but focus less on individual talent and more on team coordination and tactics.

Unreal Tournament 2004

Unreal Tournament 2004 (UT2004) is a FPS game developed by Epic Games and released in 2004 for PC. The game focuses on competitive play. It features both single player and multiplayer modes and a multitude of match types. The single player mode is essentially a tournament with little story developments. In mutliplayer mode, players can host and join matches over the Internet or local network. Up to 32 players can participate in a match, depending on the match type, the map and the server.

In UT2004, players control an avatar in a bounded 3D world, such as the inside of a building, called a map. They can use the keyboard to move forward or backward, strafe left or right, jump, double jump, dodge, dodge jump, wall dodge, crouch and activate objects. The mouse is used to move the view and aim, as the view is centered on the crosshair, and shoot. Weapons can be fired in two modes, normal and alternate, using the different mouse buttons. When players die, they can automatically respawn at one of the fixed spawn locations in the map after a few seconds.

When they spawn, players start with 100 health points. They lose health as they are hit and die when it reaches zero. Health can be replenished and even increased up to 199 by picking up health packs and vials. Besides health, players can also build up shield points, which max out at 150, by picking up shield packs. Shields can mitigate 50%, 75% or 100% of incoming damage depending on the shield packs picked up.

To deal damage and kill their opponents, players have to pick up weapons and ammo. There are many weapons in UT2004 and no limit to the number of weapons players can carry. Moreover, most weapons have two firing modes, further increasing possibilities for players. For example, the Shield

Figure 3.7: Unreal Tournament 2004. Deathmatch mode (left) and CTF mode (right).

Gun features an alternate fire mode which allows players to block damage, while the two firing modes of the Shock Rifle can be combined, by shooting a Shock Beam into a Shock Ball, to unleash a third powerful area-of-effect attack called the Shock Combo. Weapons can either be picked up from fixed locations in the map or acquired from defeated opponents. Weapons each have various properties such as damage, knockback, fire rate or type (melee, hitscan, projectile, ...).

Aside from weapons, UT2004 also features power-ups which boost player abilities. Unlike weapons, or items in general, picked up power-ups are not stored in the player inventory and have instead an immediate effect on the player's abilities. Health and Shield packs are examples of power-ups. The Double Damage power-up, which causes a player's weapons to deal 200% damage for 30 game seconds, is another one. Players can also get power-ups using adrenaline. Adrenaline is gained by either picking up pills or scoring. When a player's adrenaline is full, a command can be entered to activate a power-up which consumes adrenaline and lasts until it is depleted. Examples of adrenaline power-ups are Invisible, which grants a player invisibility, and Berserk, which grants a player increased rate of fire and knockback.

Finally, UT2004 features a number of different match types, including classic ones such as deathmatch and CTF. Other types include the Bombing Run match, where teams have to move a ball to the opposing team's goal to score points, and the Assault match, where teams take turns in attacking and defending a base and where players get to use vehicles and turrets, such as the Ion Plasma Tank or the Skaarj Space Fighter and the Minigun Turret or the Link Turret. In addition to the different available match types, game rules can further be tweaked using mutators. Mutators can have a big impact on gameplay. An example of mutator that heavily alters gameplay

is the InstaGib mutator, which replaces all weapons and ammo with the Super Shock Rifle, a weapon that instantly kills the target.

3.3.2 Real-Time Strategy

Real-time strategy (RTS) concepts started appearing in the 1980s, but it wasn't until the 1990s that the genre started to have a concrete form and became widespread. The purpose of RTS is to command an army composed of labor and combat forces to earn resources to develop both forces and defeat opponents. RTS games are match-based and involve real-time action, meaning that players have to make decisions and take actions as the game progresses, without any pause time like in other strategy genres such as turn-based strategy. The world is typically presented in a top-down view (e.g., isometric projection for 2-dimensional worlds or perspective projection for 3-dimensional worlds) and players have to control a large number of units by issuing orders to either individual units or groups of units. Because controlling many units in real-time requires dexterity and reactivity, RTS also falls in the action genre.

The world in which a match takes place is called a map. It is often made of tiles, each representing a specific terrain with different properties, such as being traversable or buildable. Some examples of terrain are dirt, cliff, elevated dirt, lava, water and space. In each match, players start with a certain quantity of resources and units and are given a series of objectives to accomplish. In a multiplayer context, the objective is usually to destroy the opponents' forces. Objectives can be more diverse however, especially in a single player setting, where players can be tasked with accumulating resources, defending a building, escorting a unit to a target location or killing a particular unit to name a few. In any case, the starting resources and units are generally not sufficient to accomplish the given objectives and players have to build a larger force in order to win the match.

In order to build additional units, players need resources. These can be gathered from specific locations in the map, in many cases in areas where a new base can be established. Players have then to continuously evaluate their income and match it against their plan to decide whether it should be increased and new resource fields claimed and fortified.

Unit or technological diversity is a key concept in RTS. Normally, players have several unit types to choose from when expanding their forces. Each

unit type has unique properties, such as a weapon type and a movement speed, and is weak or strong against other unit types. Players typically start with a small set of unit types and can unlock other types as they research technologies or acquire specific buildings during the match. Thus, players have to not only decide whether to spend resources to expand their base, their labor force or their combat force, but they also have to decide which technologies should be acquired and what the composition of their combat force should be.

Evidently, players need to know what their opponent's army will look like in order to ready an efficient counter. However, the fog of war in RTS games limits player vision of the map to areas where units have been deployed. This means that in order to assess the opponent's forces and progression, players have to constantly send units into the enemy's territory to temporarily acquire vision over their base. This important scouting also has to be preemptive because units take time to build and it can be difficult to adapt the composition of the army when the enemy threat is imminent.

Players are therefore faced with a series of challenges in a RTS game. They have to seek control of resource fields and develop and maintain an adequate labor force to generate enough income to sustain the required production of combat units, gather intelligence to adapt their strategy to their opponents' and launch assaults and defend against enemy assaults all at the same time. For example, during combat, players have to switch from controlling units in battle to controlling workers in different bases back and forth. Likewise, they have to alternate between making tactical decisions during combat, such as focusing on a specific target or withdrawing to a chokepoint, and making economical or strategic decisions such as establishing a new base, building more workers or prioritizing the production of a particular unit type. In RTS terminology, this heavy multitasking is referred to as macromanagement. Adept RTS players complement their macromanagement skills with micromanagement. Micromanagement deals with the precise control of a unit or a group of units to increase their efficiency. Some examples are using a ranged unit to hit-and-run a melee enemy (i.e., kiting), surrounding an enemy unit and scattering a group of units in the presence of area-of-effect (AoE) enemy fire.

Although RTS is a rather complex genre, some developers have successfully incorporated elements from other genres in order to increase the complexity of their game further. For example, *Dungeon Keeper* (Bullfrog Productions, 1997) featured a Possess Creature spell which allows a player

Figure 3.8: Multiple views in Dungeon Keeper. At the left is the default top-down view. At the right is the first-person perspective view that is enabled when the player possesses a creature.

to get inside a creature's mind and interact with the game in a FPS style, leaving the player's dungeon temporarily unmanaged[5]. *Warcraft III: The Frozen Throne* (Blizzard Entertainment, 2003) is another example, featuring elements of the role-playing genre. Among the various unit types players can build are heroes. Heroes are special unit types that can gain experience points by killing creeps or enemy units and level up to improve their attributes and abilities. *Warcraft III: The Frozen Throne* continues to be played in major competitions such as the World Cyber Games.

StarCraft: Brood War

StarCraft: Brood War (BW) is an expansion pack for *StarCraft*, a RTS game developed by Blizzard Entertainment and released in 1998 for PC. It features a sequel to the campaign in *StarCraft* and brings a number of new units and upgrades to the game. The single player experience revolves around the StarCraft storyline and universe while the multiplayer experience focuses on competitive play. Up to 8 players can face each other in multiplayer.

There are three races in BW, each with an extensive array of distinct units and abilities. At the beginning of a match, players can choose between the Terrans, the Zerg or the Protoss. Although they are balanced, each race is unique, featuring special mechanics and having different strengths and weaknesses. For example, the Terrans can make buildings anywhere and can even move and redeploy their buildings, while the Zerg need Creep, generated by a Hatchery or a Creep Colony, to morph buildings and the

[5]The game actually included a number of AI-powered assistants to help the player with various tasks, so the player could spend extended periods of time in possession mode without leaving the dungeon completely unattended.

Figure 3.9: StarCraft: Brood War. Protoss vs. Zerg (left) and Terran vs. Zerg (right).

Protoss need their buildings to be powered by Pylon energy, available in a field generated by a Pylon.

Matches are played in maps. Maps are 2D worlds made of terrain tiles. Terrain defines the layout of the map. Areas like chokepoints or islands are a result of terrain design. Moreover, special terrain configurations can grant units local advantages. For example, units on high ground benefit from a 30% decrease in accuracy for the attacker. Besides terrain, maps also include resource fields. There are two types of resources in BW, minerals and Vespene gas. Workers can gather minerals directly from mineral fields and Vespene gas from Vespene geysers through a special gas extraction building. Each mineral field contains a limited quantity of minerals and can be entirely consumed. Vespene geysers can be depleted and generate significantly less gas when they are. This forces players to eventually look for more resources in other locations and establish new outposts.

Resources in BW can be used to make units, buildings and upgrades and research abilities. Units and buildings each cost a fixed amount of either minerals or both minerals and Vespene gas and take a certain time to create. In addition to these resources, units also require supplies. Supplies are provided by special units or buildings players have to make in order to build more units. Unlike other resources, supplies are not consumed but held by the units a player owns. When a unit dies, the supplies it holds are released and become available for new units. The number of total supplies a player can have is limited to 200. This limits the number of units a player can control, depending on the types of units owned. For example, large units can require 6 supplies while smaller ones can require 1 or 2 only.

BW is characterized by the diversity of its units, which have many prop-

erties besides their requirements. Hit points, armor, weapon, movement speed, sight range, movement type (ground unit, air unit), nature (organic, mechanical, robotic), size (small, medium, large), energy and available commands and abilities are all examples of unit properties. These properties either are indicated (visible) or can be deduced by the players (hidden). Some of these properties can be improved with upgrades such as Apollo Reactor (Terran), which increases the maximum energy of the Wraith, and Leg Enhancements (Protoss), which increases the movement speed of the Zealot. Like unit types, unit weapons have their own properties, such as damage, cooldown, range and damage type (normal, concussive, explosive). Each damage type has different efficiencies (100%, 50% or 25%) against each unit size (e.g., concussive damage is reduced by 75% against large units). Abilities also have properties, such as target type (none, location, unit, allied unit, organic unit, ...), energy cost, range, damage or effect. Examples of abilities are Lockdown (Terran), which disables a mechanical or robotic unit, Dark Swarm (Zerg), which shields units from ranged attacks in a selected area, and Mind Control (Protoss), which permanently transfers control of an enemy unit to the caster's[6] owner. Basic commands, such as Move, Attack and Gather, can be considered costless abilities.

As a result, most unit types are specialized and work particularly well in specific situations. Players have therefore to carefully determine which unit types, upgrades and abilities they want to unlock and when they want to do it. Because there exist dependencies between the various technologies (e.g., to build an Arbiter Tribunal, a Protoss player needs to have both a Stargate and a Templar Archives), these player decisions can be traced on a tech tree. The tech tree represents the different paths of technological development players can follow during a match.

Given the relatively large number of units players may control throughout a match, the game interface provides unit management features. Players can select a group of units using the mouse and save the selection to recall the group later using keyboard shortcuts. After selecting a unit, they can choose an ability using the keyboard and only use the mouse to designate a target unit or location. This is useful during micromanagement, which can be heavy in BW because unit AI is quite basic. Players can also give units a number of tasks to execute consecutively by queuing commands.

[6] A Dark Archon.

There are a few different match types in BW, such as Team Melee where players can share control of a single army and Greed where players have be first to accumulate a quantity of resources to win the match. However, the Use Map Settings (UMS) mode can be used to play in a map with entirely customized rules. This makes it possible for players to create their own match types. UMS is very popular in both single player and multiplayer modes.

3.3.3 Role-Playing

The role-playing video game (RPG) genre debuted during the 1970s. However, unlike other genres, it was based on the preexisting concepts of tabletop role-playing games. Because of that, different interpretations of the role-playing experience were developed concurrently and this eventually led to the birth of genres like Western RPG (WRPG) and Japanese RPG (JRPG) which were popularized in the 1990s. Although they share the same core concepts of playing characters in a story and using numerical attributes and dice rolls to determine the outcome of actions and events, these genres deliver different experiences. For example, in the JRPG genre which focuses on telling a story, players have to control the protagonists through a fixed storyline, whereas in the WRPG which focuses on making a story, players get to create their own character and have some control over how the story can go. In terms of gameplay, this translates into differences in the type of decisions players have to make throughout the game.

Because the story is a central element in RPGs, players are often challenged in order to progress through the storyline. Challenges, also known as quests in RPGs, are various and include anything from simple world exploration or interaction with NPCs (non-player characters) to more complicated puzzles or missions, such as finding the right combination to open a safe or defeating a powerful enemy. Completing quests rewards players either by unlocking new areas or triggering events or with items required to advance in the story, sometimes called key items. In most cases, RPGs also include optional quests. Players can complete these to earn regular rewards such as money, a potion or a sword, or to explore side stories, such as the history of a secondary character.

Although the story is an important aspect, character growth is the dominant feature in the RPG genre. Characters become stronger as the story

progresses, typically reaching at the end of their journey a level of power orders of magnitude higher than that at which they started, and allow the player to take on more and more powerful and threatening enemies. Since character strength is determined by numerical attributes called stats, character growth mainly consists in improving these stats. This can be achieved in different ways. The level up system is a popular example where characters earn experience points through combat or quests to increase their level. Upon leveling up, characters either see their stats automatically boosted by a fixed or random amount or earn a number of stat points which players can distribute as they see fit, or some combination of both. Another example is the training system where characters improve an attribute individually by exploiting it, like increasing strength by repeatedly swinging a large axe or increasing intelligence by casting spells. Of course, growth does not have to be limited to character stats and players commonly have the possibility of improving other elements such as equipment and abilities. A weapon can have its own stats and be upgraded using materials or a special type of experience points to deal more damage, hit more often or inflict a status effect on the enemy. Likewise, a skill can be upgraded using skill points gained by leveling up, defeating special monsters or completing specific quests. Players have therefore to take into account multiple elements when planning character growth.

Role management, which is inherently linked to character growth, is omnipresent in RPGs. This challenge deals with the decisions that affect character roles in combat. Depending on their stats and abilities, characters can assume different roles in combat. For example, a tank is a character that can draw the attention of the enemy and sustain heavy damage, a healer is a character that can heal and resurrect allies and cure status ailments, a buffer is a character that can boost the stats of its allies by providing them with buffs and auras and a damage dealer is a character than can deal massive damage to the enemy. Several factors are involved in determining the role, or roles, taken by a character in a battle. When it has one, the class of a character, which includes growth parameters and skill base, will predetermine its aptitude at fulfilling a particular role. For instance, a priest will have a hard time taking down monsters as fast as an assassin. Besides class, players can tweak the role of their characters by customizing them using items, and by controlling their growth when the game allows it. For instance, a player can build a priest in a way that maximizes survivability in order to act as a secondary tank or in a way that maximizes damage in order to act as a secondary damage dealer. Characters can also assume roles as a result of arbitrary decisions. For instance, a player can

choose to act as a tank by using a particular subset of skills while another player with the same skill set acts as a damage dealer by using a different subset of skills. This is also easily observed in games where characters don't have a defining class and can instantly switch between different classes in combat. It is thus clear that role management appears both outside and during combat.

Throughout the game, players acquire numerous items which contribute to the growth of their characters. The two most common types of items are equipment and consumables. Equipment refers to anything a player can equip and keep on a character, such as a dagger, a vest, a ring or an item that grants the character an ability. Consumables on the other hand are items like potions, elixirs, grenades or steel ingots which are removed from the player's inventory as soon as they are used. There are many ways players can get their hands on these valuable items. They can be purchased from stores, stolen from enemies, looted after a battle, offered by a NPC for completing a quest or crafted using materials collected from various sources. Equipment is a primary strength factor in RPGs and has to be periodically upgraded in order to keep up with the increasingly powerful enemies encountered throughout the game. Getting the best equipment available at some point in the game can be tedious but often gives players a significant advantage against their opponents.

Naturally, players have to not only make their characters stronger but also test their mettle in combat. Combat is the ultimate purpose of character building in RPGs and players are confronted with the challenge repeatedly in a world filled with foes, whether they are progressing through the storyline or leveling up their characters. Several different combat systems exist, such as turn-based combat where characters take actions one at a time, press-turn battle which is also turn-based but where actions require turn points which can be earned and lost in battle, real-time combat where characters can freely take actions and active time battle where character turns are managed by a real-time clock. Although some systems require players to be reactive and easily fit in the action genre while others don't, they all involve the same base mechanism of planning a couple of character actions while taking into account the opponent's plan. Examples of actions are performing a basic attack with a weapon, casting a spell, using a potion, moving a character, changing a weapon, switching a character with another and changing the class of a character. Combat mechanics make extensive use of dice rolls, or more generally stochastic processes. Stats are injected into formulas and eventually translated into probability dis-

tributions which determine the outcome of actions. For example, when a character attacks an enemy, the character's accuracy stat is combined with the enemy's evasion stat to determine a certain hit probability which is then used to decide whether the attack lands or the attack is avoided. The loot system frequently relies on randomness too, like using a loot table containing probabilities to drop different items for each foe.

Final Fantasy VII

Final Fantasy VII (FFVII) is a RPG developed by Square[7] and released in 1997 for PlayStation. The story of FFVII is that of a party of protagonists trying to save the world from destruction. The goal of the player is thus to guide those characters through their journey, exploring the world, acquiring key items they require to complete their objectives, making them stronger and eliminating the threats they encounter. FFVII also includes a number of minigames, such as the motorcycle and submarine chase games, the ski game and the chocobo race.

There are four main types of view through which the player interacts with the game, each with different controls. The world map, a miniature representation of the planet in 3D, allows the player to travel from place to place using different transportation methods such as walking, driving a buggy, flying an airship or riding a chocobo. In this third-person perspective view, the controller is used to maneuver the party leader or transport and change the camera position. When the player enters a place in the world map, like a city, a town, a cave or a forest, the view changes to a real-size 2D scene with pre-rendered backgrounds[8] where the player can move the party leader and interact with the environment, like talk to a NPC, open a chest, push a button or climb a ladder. The player proceeds from one scene to another as places are composed of several connected scenes, each scene representing a particular area or part of the place. A third type of view is used for battles, which are rendered in 3D and viewed with either an automatic or fixed camera. Because characters and enemies cannot freely move in combat, the controller is only used to navigate a battle menu, enter commands and select targets. The fourth and last main view is the menu, which the player can bring up anytime when out of combat. It is used to prepare for combat and lets the player check the characters and the inventory, change the party formation, change the character equipment or use

[7] Square merged with Enix in 2003 and is now known as Square Enix.
[8] Characters, NPCs and other objects are still rendered in 3D.

Figure 3.10: Final Fantasy VII. Battle (left) and Materia configuration (right).

curative or other non combat-restricted items and abilities. Besides these, each minigame has its own view and controls.

FFVII uses the level up system for character growth. Characters earn experience points (XP) after each battle and their total XP translates into a level between 1 and 99. The XP required to gain a level increases with level, but stronger enemies reward characters with more XP too. When a character levels up, its stats are individually increased by a slightly random amount. Although there are no classes in FFVII, each character has a different set of growth parameters which determine how each stat grows and subsequently the character's inclination towards a certain proficiency, such as using physical attacks or casting magic. Stats can also be boosted either permanently or temporarily with equipment and consumables.

Characters in FFVII have several base stats, such as hit points (HP), strength, spirit and luck as well as additional equipment-related stats which are either entirely determined by weapon or armor stats, such as hit rate or magic evasion, or based on both equipment and base stats, such as attack and defense. Characters are not limited to numerical attributes however and have elemental and status effect attributes which can be altered with equipment. Characters can have an element attached to their attack, such as fire to deal fire damage or earth to deal earth damage, and can take half damage, no damage or even absorb specific elements. They can also inflict or resist certain status effects, such as poison, death or confusion. Like characters, enemies have their own stats and other attributes.

The player inventory can hold many items, including gil[9], weapons,

[9]The currency used in FFVII.

armors, accessories, materias, consumables and key items, which can be acquired in many ways. They can be picked up from the ground, found in a chest, bought from a shop, given by a NPC, won from a battle, stolen or morphed from enemies, or automatically added to the inventory by the game. Characters are always equipped with a weapon and an armor and can optionally wear an accessory. Weapons and armors can in turn be equipped with materias. Weapons have a number of attributes such as attack, bonus stats[10] and power-ups, armors have attributes such as defense, bonus stats and elemental resistances and accessories have attributes like bonus stats, elemental and status effect resistances and other enhancements. Additionally, each weapon or armor has between 0 and 8 materia slots and a certain growth rate which determines how fast materias grow on it. Materia slots can be either linked in pairs or separate. Linked slots allow the player to combine materias.

Abilities in FFVII are granted by materias. The role of a character is thus largely determined by the materias equipped on that character's weapon and armor. Without materias, characters may only perform basic attacks and limit breaks or use items. Limit breaks are special attacks which can only be executed when a character is in a particular state and each character has a unique set of these. There are five types of materias in FFVII. Green materias grant magic abilities such Bolt 3, Cure 2 and Ultima. Yellow materias grant command abilities such as Steal, Deathblow and Manipulate. Red materias grant the ability to summon creatures such as Ifrit, Odin and Bahamut. Purple materias grant auto-abilities and bonuses, such as Cover, Counter Attack and Magic Plus. Finally, blue materias can be paired with other materias, using linked materia slots, to add extra effects, such as All, HP Absorb and Quadra Magic. Like characters, materias level up with ability points (AP) earned from enemies in combat. The total AP a materia receives after a battle depends on the growth rate of the weapon or armor it is equipped on. Examples of growth rates are None, Normal and Double. When a materia levels up, it gains an additional ability. For example, when a level 2 Gravity reaches level 3, it gains the Demi 3 ability, granting its wielder the Demi, Demi 2 and Demi 3 abilities. Each materia has a maximum level and when that level is reached, a new level 1 duplicate of the materia is added to the player's inventory. It is worth noting that, aside from granting abilities, materias can have bonus stats and significantly affect the stats of a character when stacked.

[10] A bonus stat is an increase to a character's base stat.

A player can enter combat in different ways. The most common one is through the random encounter system. In most places where the player maneuvers the party leader, including the world map, there is a random chance at each step to trigger a battle against some enemies which vary according to the area where the battle is triggered. Combat can also be automatically initiated by the game as part of the storyline or by the player running into a visible enemy NPC. FFVII features an active time battle (ATB) system. The ATB system is a turn-based real-time hybrid where characters each have an action bar filled in real-time and can only take an action when their action bar is full. It can be tweaked to be more real-time or more turn-based via the game settings. In battle, the player can issue commands to characters, usually a party of three, as soon as their action bar is full. When a command is entered by the player, it is registered in the action queue and scheduled for execution. The game engine executes actions in the action queue one at a time. Until the action is executed, at which point the character's action bar is reset and starts filling again, that character can no longer take any turns. This means that there can be at most one action per character in the action queue. The latter contains both the actions of the characters, which are controlled by the player, and the actions of the enemies, which are controlled by the game. Like characters, enemies have to wait for their action bar to fill in order to make a move. Stats affect the rate at which an action bar is filled and can result in getting more turns. In addition to the action bar, characters have a limit bar which fills as they receive damage. The higher the damage they receive, the larger the portion filled will be. When the limit bar is full, a limit break becomes available on that character's next turn to unleash. Along with limit breaks, characters can perform various types of actions, such as attacking, casting spells, defending, moving from the front row to the back row or vice versa and using items. Characters and enemies each have two main life pools, HP and magic points (MP). HP is used to sustain damage while MP is consumed to use magic. Both can be replenished using items and abilities and it is crucial to manage the HP and MP pools of the characters efficiently to triumph in battle. The player wins when all enemies have been defeated and likewise loses when all characters have been defeated. Characters or enemies are defeated when they can no longer take any actions, for example when their HP is reduced to zero or when they are petrified.

3.3.4 Action-Adventure

It wasn't long before the adventure genre which emerged in the 1970s was combined with the older action genre to appeal to a broader audience. The resulting action-adventure (AA) genre quickly became a success in the 1980s and today, it encompasses many existing genres. Of course, it remains rather generic despite the mix. The reason AA games are popular is that they offer players both mental and physical challenges. Not only do they include elements from the two genres, but both of them are essential gameplay components and players have to be good at handling both in order to complete the game. It is the fact that both are central that distinguishes AA games from action games with some adventure elements, like some FPS games, and adventure games with some action elements, like some RPGs.

Adventure elements mainly consist in either following a storyline or exploring a world and acquiring and using items and solving puzzles. In a game which focuses on storytelling, players usually face challenges in their order of appearance in the story, often in increasing difficulty, whereas they tend to have more freedom in selecting challenges in a game which focuses on world exploration.

Items are important and can be required to progress through the story, unlock an area or clear a challenge. Players have to manage their inventory carefully and use items when necessary. Examples of inventory management are making sure a character has everything needed before leaving for a long trip, saving a particular item for a boss and using a limited amount of keys to open the right doors. Items have to be acquired in some way, such as being purchased, found or earned.

The puzzles players encounter are purely mathematical sometimes, but in most cases they are practical. Mathematical puzzles are independent from the environment and challenge the player's skill in dealing with numbers and other abstract objects, such as predicting the next number in a series, solving a 3D puzzle and finding a Hamiltonian path in a directed graph. On the other hand, practical puzzles require the player to reason about the environment and the objects in it and understand and exploit properties and mechanics and use logic to deduce complex relationships. Some examples are figuring out that using a fire spell near an ice wall will reveal a hidden passage, that roasting a piece of meat will lure an NPC out and that a switch can be pressed by luring a heavy monster on it to open

a dam and fill a room with water to gain access to an upper level. Other examples can be mixed with action, such as finding out that a boss will do anything to avoid light, that killing a guard silently will not alert those in the adjacent corridor and that running around a monster will make it dizzy.

Action elements remain generic and may test the player's speed and accuracy in a multitude of real-time settings, such as in a dynamic puzzle, in in a competition or in combat. Players have to be reactive and skilled at manipulating controls in order to do things such as aiming while moving, blocking a fast projectile, seizing the chance to attack when an opening appears, jumping at the right time to avoid an obstacle, executing a combo and sneaking past an enemy. They have to be quick-thinking too and take decisions like choosing which platform to move onto in a collapsing building, switching from a moderately dangerous target to a more imminent threat and using an enemy as a shield.

The Legend of Zelda: Ocarina of Time

The Legend of Zelda: Ocarina of Time (OoT) is an AA video game developed by Nintendo EAD[11] and released in 1998 for Nintendo 64. In OoT, the player controls a hero on a quest to fulfill a destiny. The character is maneuvered in a 3D world composed of numerous areas through a third-person perspective view with an automatic camera. The player can move, look around in a first-person view, swim, dive, attack, block, evade, use items and interact with the environment to speak to a NPC, activate an object, jump or climb. In some areas, the hero can ride a horse to travel faster.

Puzzles are encountered throughout the game, in different formats. The player has to collect various items and learn melodies and use them to access new areas or defeat new enemies and progress in the storyline. For example, the player can unlock a sealed door by playing a specific melody or access an underwater dungeon using special boots. Figuring out where to go next and how to gain access to a specific area amounts for a significant part of the gameplay. Another form of puzzle the player repeatedly faces is the dungeon challenge. Dungeons are locations the player has to clear in order to earn essential key items to complete the story. They are made of rooms and corridors which the player progressively gains access to until a boss room, in which a boss must be defeated, is reached. Each dungeon features a unique environment and involves different mechanics.

[11] Nintendo Entertainment Analysis & Development

Figure 3.11: The Legend of Zelda: Ocarine of Time. Combat (left) and Inventory (right).

The player has to find keys to open locked doors and understand the properties of the environment and exploit them to gain access to new rooms, such as adjusting the level of water in a room, moving onto an invisible platform and reflecting sunlight onto a switch to activate it. Combat is part of the process too as these dungeons are filled with enemies which must be overcome in order to advance. Beating the dungeon boss usually involves figuring out a trick, such as throwing a bomb inside the mouth of a monster to stun it, reflecting an enemy's spell with a glass bottle or using a hook to reel the nucleus of an aquatic monster out of a pool. Examples of dungeons are Dodongo's Cavern, the Water Temple and the Shadow Temple.

One of the characteristics of the gameplay in OoT is that the hero can travel through time. The player starts with a young character who eventually travels seven years into the future and acquires the ability to go back and forth the two epochs. The young hero can do things the adult cannot, and vice versa. Using this ability is necessary to complete the game. The player has thus to make links between the present and the future, such as playing a melody from the present in the future, learning a melody in the future to find an item in the present to progress in the future and clearing part of a dungeon in the present and the other in the future.

The player can open the inventory anytime to check key items such as melodies and medallions and dungeon items such as door keys and maps, modify the equipment of the hero and assign different items to the item buttons. The hero can be equipped with a sword, a shield, a tunic and a pair of boots. There are also other pieces of equipment which the player acquires but cannot change, such as the bomb bag and the gauntlets. Depending on the equipment, the hero gains different abilities, such as the ability to resist heat, to dive longer or to lift heavy objects. Other weapons and tools, such as bombs, a bow, a fire spell and a bottle containing a po-

tion, can be assigned to item buttons. Items can be acquired in different ways, such as being purchased from stores using rupees[12], rewarded by NPCs, found in chests and picked up from the ground.

Throughout the game, the hero grows stronger not only as a result of the new equipment, but also through power-ups acquired from various locations. These power-ups improve the hero's abilities permanently and are necessary to face the increasingly stronger enemies trying to halt the hero's advance. Some power-ups are unique and granted by special NPCs while others can be collected by the player. For example, heart pieces and containers, which extend the life bar of the hero, can be found in many different places in the world. Examples of unique power-ups are the double magic power-up, which doubles the hero's magic bar and the double defense power-up, which cuts any damage received by the hero in half.

Action in OoT consists in minigames and other basic challenges such as a fishing game, a horse race and a timed maze, and real-time combat. Combat is an essential gameplay component and thoroughly integrated in the game. It is initiated as soon as the player or an enemy attacks. Enemies are found either roaming in the world like a ghost in a field or lying in wait at specific locations like a dungeon boss. The player can attack and defend in several ways using control combinations, such as performing a roll attack, a sword stab and a sword spin swing and blocking, making a side jump and doing a back flip. In addition to these primary actions, the player can also assign items to the item buttons in order to use them while fighting. Up to 3 items can be set, although the player can reassign the buttons at any time by opening the inventory. Some items can change the view when used. For example, using the bow can switch to a first-person view to allow the player to aim with precision. There are two ways for the player to target an enemy, either by manually controlling the hero or by locking on a target by pressing or holding a special button. When a target is locked-on, the hero is automatically positioned to be facing the target and the player can move closer or farther and turn around the enemy. Any attack made in this state is therefore directed at the target. To defeat an enemy, the player must reduce its health to zero. The life of an enemy is not visible and can be estimated in number of hits required to take it down using a particular weapon or attack. The hero has two main combat resources, a life meter and a magic bar. The hero loses some life with each hit taken. If the life meter is depleted, the hero dies and the game

[12]The currency used in OoT.

is over.[13] The magic bar allows the hero to use attacks infused with magic or magical items, such as executing a charged sword spin attack, shooting an ice-enchanted arrow or using the Lens of Truth, an item which reveals invisible objects. Both life and magic can be refilled by using items such as potions or by picking up power-ups from the ground, such as hearts and magic jars.

3.3.5 Multiplayer Online Battle Arena

Multiplayer online battle arena (MOBA) is a community-designed video game genre which emerged in the early 2000s and was later acknowledged by the industry in the late 2000s. It started in the *StarCraft* community as a custom (UMS) map called Aeon of Strife[14] (AoS). It then moved to the Warcraft III community under the name of Defense of the Ancients (DotA), where it was developed and maintained for several years. The industry eventually picked up its success and embraced the genre, which was renamed to MOBA. MOBA games feature the same type of interface used in RTS games. Players see the world in a top-down view and control units by issuing commands in real-time. Unlike RTS however, players only have one unit to control and cannot produce more units.

A match consists in two teams fighting in a map where each team base is connected to the other by a number of terrain corridors called lanes. In each team base, several CPU-controlled allied units, called minions, spawn periodically to travel along these lanes and attack the enemy base. On each team side, lanes are successively fortified with automatic defense structures called turrets. Only the outermost turret in each lane may be attacked, forcing teams to take down turrets one by one as they advance towards the enemy base. In the enemy base lies a primary structure called the nexus. In order to win, each team has to destroy the enemy nexus. The nexus may only be attacked when at least one lane leading to it has been cleared, or in other words when all turrets in it have been destroyed.

Each player in a team has to choose a hero, the only unit a player controls in a match. When a hero dies in combat, the player has to wait until the hero respawns at the base. Players have different heroes to choose from, each with a unique set of attributes and abilities, such as speed, health points (HP) and mana points (MP) and throwing an ice bolt, becom-

[13] A fairy in a bottle can bring back the hero to life, however.

[14] Like the great Protoss civil war.

ing invisible and restoring HP with each hit. Abilities often have cooldowns[15] and costs. For example, an ability may cost 50 mana points to activate and may be used at most once every 10 seconds. Heroes have basic abilities too which do not have any cost or cooldown, such as moving and attacking. Players can level up these powerful units during a match by defeating enemy minions and heroes. The level up system is similar to the one used in RPGs. Heroes gain experience points by killing enemy units and their attributes and abilities are improved on level up either automatically or manually by the player using attributes and ability points. Leveling up quickly is important as the hero level is a major strength factor and a small difference in level can translate into a significant power gap.

Besides experience, heroes have an inventory and can carry a number of items. Items contribute to the growth of a hero by improving attributes and granting extra abilities. Like hero abilities, item abilities can either be active abilities used by the player, such as removing any movement impairing effects on a hero and teleporting to a target location within a specific radius, or passive abilities, also called auto-abilities, such as slowing the movement speed of nearby enemy units and creating a shield around the hero when the latter's HP drops below a certain point. Items allow players to build their heroes in various ways and adapt to various situations. For example, against a powerful mage opponent, a player may choose to build up the magic resistance and HP of the hero to avoid being killed in one shot. There are also consumable items such as health or temporary power-up potions. Items can be acquired in different ways. They can be purchased from specific locations in the map called stores, combined together to form new, more powerful items or picked-up from the ground.

In order to purchase items, players have to earn gold. Although gold is automatically earned at a low and steady rate throughout the match, the primary source of income is minions. For every enemy minion a hero kills, the player is rewarded with a certain amount of gold. Only the killing blow determines the killer of a minion, and subsequently the rewarded player, if any. Killing minions for gold is called farming. When farming, players have to focus on landing the last hit on as many enemy minions as possible, and in some cases on denying their opponents the last hit on allied minions by finishing them off themselves. Players can also earn gold by defeating enemy heroes and destroying enemy turrets or lose gold when defeated by an opponent. In addition, some items or abilities may affect the rate

[15] Minimum delay between consecutive activations.

at which gold is automatically earned, effectively generating or inhibiting gold. Players can sell items for gold too, although selling a purchased item usually results in a loss.

Team coordination and communication is crucial in MOBA games. Before the match starts, players already have to decide on the hero composition of the team. Different heroes can assume different roles, and leaving some roles unfulfilled can result in a serious disadvantage during team confrontations. For example, crowd control (CC) abilities such as silencing[16], slowing and stunning enemies can have a decisive impact on the outcome of a team fight and if a team chooses heroes with no or little CC potential, they may stand no chance against a team with a better composition. During the match, players are often divided among the different lanes and farming as much as they can while applying pressure on their opponents to prevent them from farming efficiently. Players can exploit temporary advantages to push their lane and take out enemy turrets to allow their minions to advance further in the lane. These advantages can result from enemy mistakes or be gained through coordinated team actions such as drawing enemy attention to a different lane, among other things. Players eventually regroup to launch targeted assaults or defend a particular position, which may initiate a team fight. Team fights usually only last seconds and require excellent team planning, like agreeing on priorities, delegating roles and assigning targets and tasks. Players also have to reactively adapt to the situation as the dynamics of a team fight are such that the outcome of a situation may drastically vary depending on a single action, such as the use of a critical ability which the player may not successfully land on the enemy.

League of Legends

League of Legends (LoL) is a MOBA video game developed by Riot Games and released in 2009 for PC. In LoL, the player incarnates a summoner who has the ability to control champions. Summoners grow with each match they participate in, gaining a certain amount of experience points (XP) and influence points (IP). Each summoner has a level between 1 and 30 which increases as they accumulate XP. Leveling up allows a summoner to gain mastery points, unlock additional rune slots and get access to more summoner spells. As for IP, they can be used to acquire new runes and unlock new champions. There are over a hundred champions to play with

[16] A hero under the effect of silence cannot use active abilities.

in LoL, though players are required to spend IP to permanently unlock each champion. Alternatively, they can use Riot Points purchased from the Riot store with real money.

Masteries grant a summoner's champion bonuses during a match. There are 3 types of masteries in LoL which improve a champion's offense, defense and utility, respectively. Examples of masteries are Weapon Expertise, which grants a champion armor penetration causing attacks to ignore a portion of the target's defense, Tenacious, which reduces the duration of CC effects on a champion, and Greed, which increases the rate of gold earned over time. The number of mastery points a player has is equal to the summoner level, and they can be reallocated at the beginning of a match when teams are choosing their champions. Masteries of each type are organized as a tree shaped by the dependency relationships which exist between them. It is possible to allocate multiple points in some masteries to increase their effect.

Like masteries, runes can be used to improve a champion's efficiency on the battlefield. There are 4 types of runes serving to better the different skills of a champion and different qualities for each rune, with better runes being more expensive than the first ones players can get. Unlike masteries however, runes cannot be readjusted before a match. Instead, summoners have a Runebook composed of rune pages each representing a different rune configuration. During champion selection, players may only choose which rune page they want to use. Rune pages can only be reconfigured outside a match, though more pages can be added to the Runebook using IP. As is the case with masteries, the number of rune slots in a page, and therefore the number of runes a player can use for a match, is equal to the summoner level.

In addition to masteries and runes, a summoner is required to choose 2 summoner spells to use on the battlefield at the beginning of a match. The spells a player can equip depend on the summoner level. These active abilities can be used by a summoner during the match regardless of their champion. They have no cost, but often a long cooldown. Examples of summoner spells are Clairvoyance, which reveals an area of the map for a few seconds, and Exhaust, which lowers an enemy champion's movement speed, attack speed and damage for a short duration.

Of course, each summoner has to select a champion to participate in a match. Champions differ from each other in their attributes and abili-

(a) Spell cast

(b) Item shop

Figure 3.12: League of Legends.

ties and the roles they can assume in a match. They have strengths and weaknesses and are balanced in teams rather than individually. For instance, one champion can easily be countered by another champion while it's more difficult for a team, as long as it is carefully chosen, to be countered by another team. Each champion has four abilities and a signature passive ability. Abilities can be active or passive and can require the player to aim for the target or be automatically directed at it. Champions get one ability point per level which can be used to upgrade one of the four abilities. Their attributes are also improved each level. Champions start at level 1 and max out at level 18. Champions grow stronger throughout the match by leveling up, upgrading abilities and acquiring items. Items can significantly boost a champion's potential in certain areas and are often decisive in an encounter. Players may see the items acquired by any champion, allied or enemy, at any time during the match; and they have to in order to adapt their items to their opponents'.

Gold in LoL is earned automatically over time and by killing enemy units. Farming is the main process of gaining gold, though players can get greater rewards for defeating enemy champions, depending on their current performance. The gold worth of an enemy champion increases with the number of kills scored by the latter since the last death (i.e., the number of consecutive kills) and conversely decreases with the number of deaths suffered by the champion since the last kill (i.e., the number of consecutive deaths). When multiple players contribute in defeating an enemy champion, the champion that registers the last hit scores a kill and receives the reward and the other champions score an assist an receive a bonus reward worth half the kill reward divided among them. Players also receive a gold reward when they destroy an inhibitor or whenever a member of their team destroys an enemy turret.

LoL features several match types each in a different map, though the most popular and defining one is the Summoner's Rift. It is a 5 versus 5 map with 3 lanes and 3 inhibitors and 11 turrets for each team. Between the two team bases lies a jungle, an area filled with neutral monsters which can be killed for extra gold as well as special buffs. Because minions do not travel through the jungle, players have less visibility in this area[17] and need to constantly place wards, special items that grant vision in a certain radius when placed on the map, to stay aware of enemy movement. Players in

[17]Like in RTS games, the fog of war prevents players from seeing areas of a map where they have no units deployed.

LoL can hide in the brushes found in some lanes and in the jungle. When a unit moves onto a brush terrain, it becomes invisible to enemy units outside of the brush. Teams commonly have one player in the top lane, one player in the mid lane, two players in the bottom lane and one player in the jungle. The jungler's main role is to assist the other players in their lane whenever an opportunity arises, such as ambushing an enemy champion who wandered too far from their turret or helping an ally in need. In order to win, each team has to destroy the enemy nexus. At least one inhibitor must be destroyed in order to attack the turrets defending the nexus. The inhibitor is a structure which serves to weaken enemy minions. When the inhibitor of a lane is destroyed, the enemy nexus starts spawning super minions with the regular minions in the lane, and all enemy minions grow stronger. Minions also naturally have their attributes increased periodically during the match, preventing a game from lasting indefinitely.

3.3.6 Massively Multiplayer Online Role-Playing

The massively multiplayer online role-playing video game (MMORPG) genre started appearing in the 1990s as Internet connectivity became more widespread and rose to one of the top genres in popularity in the 2000s. It features classic RPG elements such as character growth and role management and is characterized by a persistent world[18] and a large number[19] of players that can exist in and interact with it simultaneously. As a result, social interaction and evolution are integral to the gameplay and often considered the primary appeal of MMORPGs.

In order to enter the virtual world of an MMORPG, players have to create characters which represent them in the game world. Given the large number of players and the social aspect of gameplay, it is usually possible to customize the appearance of a character in order to allow players to have a unique character design and avoid clones. Players may have to choose a base class or role during character creation or may instead acquire one as a part of the growth system.

Leveling up by defeating enemies or completing quests is a standard way for characters to grow stronger. It allows players to improve their character's natural attributes and abilities and further specialize in a class or role. Growth is not limited to the character level however, as there can

[18] A world which continues to evolve independently of the player's presence.
[19] Thousands.

be a significant difference in power between characters of the same level depending on their equipment. Gear such as weapons and armors serve to complement a character's innate abilities and boost their efficiency in combat. Players have therefore to not only improve their character's talents but also continuously collect new items to upgrade their equipment and maximize their impact on the battlefield.

Players can acquire items from NPCs and other players. For example, friendly NPCs can reward players with items or money for completing quests, while enemy NPCs can drop items for players to loot when defeated. The process of repeatedly killing the same type of enemies to acquire a particular item or money is called farming. High level players often spend considerable amounts of time on farming in order to get their hands on the best equipment available in the game and maximize the advantage they get against their opponents. Players can trade with other players or NPCs, too. Trading is essential in MMORPGs because it can be very hard for players to collect all the items they need on their own. Some items may have incredibly low drop rates for instance. In this case, the probability for a player to acquire multiple rare items only through looting can easily get close to zero. Players therefore turn to several other players who happen to have had luck with the different items they are missing and either buy them in exchange for money or other valuable items.

Understanding the economy of the virtual world is part of the challenge players face in MMORPGs. Carefully assessing the supply and the demand of different goods and even services, such as power-leveling[20], and modeling their evolution allows players to buy or sell items in favorable times to minimize their efforts or maximize their profits. For example, a player can correctly predict that the price of an item is bound to drastically increase because its demand will permanently grow in the days to come and seize the opportunity to stock up on it. Another example is a player or a group of players controlling the price of a particular item by ceaselessly grabbing its instances from other sellers on the market. Wealth often contributes to the power of a character and can even be necessary for a player to be successful.

Most MMORPGs include different ways for players to team up. Players may form groups such as parties, alliances and legions. A party is a small, temporary group of players which share a common objective, like complet-

[20]Power-leveling is the process of quickly leveling a low level character with the help of a high level character.

ing a specific quest or clearing a dungeon. Sometimes, several parties may decide to form an alliance to take on a powerful boss monster or a large group of enemies. On the other hand, legions are permanent groups which allow players to forge long-term relationships and establish communities within the game population. Decisions to join or leave a group or invite or kick a player have to be made by players constantly and are critical to success, whether parties or legions are considered.

Player opponents may include either NPCs like monsters or other players, or both. Combat against NPCs is called PvE (player versus environment) while combat against other players is called PvP (player versus player). Combat involving both NPC and player opponents is referred to as PvPvE. Depending on the type of opponents, players in a group may require varying degrees of coordination and planning. NPCs tend to be more predictable than players. A team of experienced players may only need little communication in order to clear a PvE area, whereas facing a team of enemy players can require players to communicate heavily in order to coordinate their actions and work with a consistent strategy.

Aion: The Tower of Eternity

Aion: The Tower of Eternity (Aion) is a MMORPG developed by NCsoft and released in 2008 for PC. It features two player factions, the Elyos and the Asmodians, and a third NPC faction, the Balaur, all relentlessly battling for the domination of several areas. It emphasizes PvPvE but also allows players to focus on PvE or PvP with its vast and rich environment. Players control a character in a 3D world with the ability to fly in some areas. Areas can be disputed, neutral (i.e., no PvP allowed) or completely inaccessible to enemy factions.

Before creating a character, players have to choose whether to be Elyos or Asmodian. They may choose to side with the stronger or weaker faction depending on their current state[21] or simply choose the type of environment they prefer, as each faction has its own unique homeland and character design. The differences are not only aesthetic however, as although players can choose from the same classes regardless of their race, some abilities may vary depending on it. There are currently 10 classes in Aion, each with a unique set of capabilities on the battlefield, such as the equipment they can use, and dozens of active and passive abilities. The current

[21] Naturally, the strength of a faction depends on its players.

maximum level a character can reach is 65.

Players unlock new abilities as their character levels up, either by using skill books which teach abilities to a character permanently or by equipping stigma stones which grant abilities to a character temporarily[22]. Stigma abilities are organized in the form of dependency trees, with more powerful stones requiring several other stones to be equipped. Aion also includes a skill chaining system which regulates how active abilities can be chained. For example, a chain 2 skill may only be used immediately after one of its chain 1 trigger skills has been used. Some chain skills may trigger at a fixed probability rather than every time, requiring players to stay focused and react when an extra chain triggers. Other examples include abilities which can only be used when the character is in a certain state, such as being stunned or stumbled, and abilities which can be repeated multiple times before going on cooldown.

In addition to classes, characters can have professions. Professions allow characters to craft different items using raw or processed materials. Some examples of professions are Armorsmithing, which is used to craft chain and plate armor as well as shields, Tailoring, which is used to craft cloth and leather armor and belts, and Cooking, which is used to craft food, consumable items which temporarily raise a character's efficiency in combat. A player can level up the profession of a character by repeatedly crafting equipment or consumables or simply by processing materials, such as transforming orichalcum ore into orichalcum ingots and then rods which can be used to craft a Dragon Lord Blade. Characters also earn regular XP when crafting in Aion and may level up in a workshop instead of the battlefield.

There are thousands of items in Aion, including equipment and consumables and other types such as materials and quest items. Equipment includes weapons, shields, several pieces of body armor such as headgears and gloves, jewelry such as rings and earrings, wings and stigma stones. Some equipment can be upgraded and socketed with manastones or godstones. Weapons and armors can have an upgrade level between 0 and 15, each additional level further increasing the attributes of the equipment. Manastones can be used to add attribute bonuses on gear, such as increasing the accuracy or the chance to land a critical hit of a character, while godstones can add various effects to weapons, such as a chance of blinding

[22]The character retains the ability as long as the stigma stone is equipped.

(a) Combat (Asmodian)

(b) Quests and character profile (Elyos)

Figure 3.13: Aion: The Tower of Eternity.

the target or dealing extra wind damage with each hit. Consumables include potions, scrolls and food as well as non-combat related items such as skill books and motion manuals[23]. Players can also acquire and use mounts to travel faster in certain areas.

There are three types of temporary groups players can form in Aion. A party can be created to allow at most 6 players to team up. Parties, a maximum of 4, can then be combined into an alliance. Alliances in turn can join forces to form a league. A league may include up to 8 alliances, resulting in 192 players fighting together. Aion also features a legion system. Besides building communities, legions also allow players to control artifacts and fortresses during sieges. Although players can choose to never team up with other players, many quests and NPCs have been designed to challenge groups of varying size, making it difficult or impossible for single players to undertake those challenges on their own. The game also promotes other types of teaming up such as mentoring. A high level player can join and mentor a party of low level characters in order to help them with their quests without disrupting their XP or their loot. Both the mentor and the mentee can be rewarded by the game for working together.

While players can enjoy the standard PvE found in most MMORPGs, such as completing quests in a field or clearing a dungeon[24], they have a number of ways to engage in PvP and PvPvE. For example, rifts allows players from a faction to travel to the land of the opposing faction and ambush them while they level up. Another example is the Dredgion Battleship instance in which an Elyos team competes against an Asmodian team to defeat the captain of the Balaur ship. There are also fortresses in disputed areas which can be conquered by any one of the three factions. Fortresses become vulnerable at specific times called sieges, allowing other factions to attack and attempt to seize their control. Fortresses are heavily guarded and it may take hundreds of players to conquer one. Not only do players receive rewards for conquering fortresses, but maintaining control of these also grants a faction some advantages, such as giving them access to fortress instances and driving down the tax on NPC prices for goods and services. Players are rewarded with a special currency when they defeat opponents from enemy factions which can be traded for PvP optimized gear.

[23]Motion manuals allow players to customize the way their character moves.
[24]Also called an instance.

Chapter 4

AI in Video Games

Contents

Summary

This chapter aims to provide the reader with a bit of insight on the use of artificial intelligence (AI) in video games. It contains three sections. The role of AI in video games is presented in the first section. Then, derived from both context and role, the properties commonly sought for AI in video games are described in the second section. Finally, some of the concepts used in video game design to achieve these properties are briefly explored in the third section. References to other work covering the topic of video game AI are mentioned in a short fourth section.

4.1 The Role of AI in Video Games

Ever since their creation, artificial intelligence (AI) has been used to breathe life in video games. Even the Nimatron, back in 1940, featured an AI to play against humans. Although some definitions of AI attempt to clarify its meaning, the term in the context of video games is loosely used and can take considerably different forms. Video game AI may sometimes not have to exhibit intelligence at all, depending on how strict the adopted definition of intelligence is.

AI may refer to the programmed behavior of a computer player. This behavior can drastically vary in terms of complexity depending on the game. For example, an AI for a Tic-tac-toe game can be expressed by only a few lines of code and amount to little more than lookup operations in a table. On the other hand, an AI for a modern RTS game can require thousands of lines of code and has to deal with multiple problems simultaneously. Furthermore, different games feature problems of different nature. Numbers, shapes, physics or emotions are some of the elements at the basis of the challenges players can face in a video game. A computer player may also need to play and collaborate with or against human players or other computer players in a multiplayer game.

AI can refer to the basic intelligence of a unit or character too. This intelligence defines how a unit or character responds to stimuli from the environment, such as finding its way to a location designated by a commander or fleeing when it is hit by an enemy. This type of AI can have varying degrees of sophistication depending on the unit or character and the game. For example, a RPG can feature normal monsters and elite monsters. Normal monsters may only know how to chase an enemy, while elite monsters could be smarter and able to avoid being lured away from their allies, call allies for help and attack low-health targets or healers first. Newer games also tend to feature more developed basic intelligence thanks to the global increase in computing power of mainstream hardware. For instance, units in *StarCraft II: Wings of Liberty* are smarter than they were in *StarCraft*. When they are gathering, a group of workers in *StarCraft II: Wings of Liberty* automatically balance the load on each mineral field in order to optimize overall efficiency, whereas each worker acts individually and searches for an available mineral field without coordinating with its peers in *StarCraft*. Units in general also have improved pathfinding and obstacle avoidance capabilities.

Some video games feature player assistance AI. This can range from simple auto-aim functions to more complicated processes such as unit management in a RTS game, which allows a player to focus on strategic actions such as scouting or deploying a new base while the computer takes care of controlling units that are not receiving any commands from the player. Player assistance AI can be similar to the one used for computer players, though it is usually selective (i.e., only a subset of the abilities of the computer player are activated) and may include additional routines to avoid interfering with the player's actions or to coordinate between the player and the assistant's actions. An example of player assistance AI is the Auto-battle function in *Final Fantasy XIII-2*, a RPG developed by Square Enix and released in 2011 for PlayStation 3 and Xbox 360. This function can be used to automatically fill a character's action bar in combat (i.e., choose a set of actions to perform with the character's turn depending on the character's current role and the state of the party or the enemy), allowing players to focus on more strategic decisions such as switching a character's role or paying close attention to the enemy's attack pattern.

AI can also be a service provided by the game to improve playing conditions, like matchmaking. Matchmaking services in online games aim to provide players with the ability of being matched with or against players of equal skill and minimize the number of matches where skill disparities are significant enough to prevent a fair match. Another example is dynamic difficulty adjustment, which adjusts the game difficulty in real-time depending on the player's performance to ensure that the player is never overwhelmed by the opponents and vice versa. In some cases, this can include real-time content generation such as spawning different enemies in a FPS or laying down new track sections in a racing game.

In general, all of these applications can be regarded as intelligent agents as long as those are defined as sensing actors in an environment. Multiple environments can be considered in the context of video games. The game world is one environment where actors are virtual entities such as units or characters while the player world is another environment where actors are real entities such as humans or computers. Characters exist within the game world and can sense the environment and act in it. For example, in the game world, a character can detect an enemy when it enters its field of vision and attack it by throwing a fireball on it. Players on the other hand react, in the real world, to stimuli coming from output devices by handling input devices. For example, a human player may see a flash on the screen and move a finger to press a key on the keyboard. In the case of AI assis-

tants, the actor is simply a computer player sharing control with a human player. As for services such as matchmaking, the actor exists, like players, in the player world but is not a player. It could be the game itself or another neutral entity such as an arbiter.

In the end, while AI takes part in several aspects of video games, it essentially allows developers to animate creations in both the virtual world and the real world. These AI-powered entities can directly contribute to the perceived quality of a video game by being smart, challenging, realistic, caring and any other attribute which has a positive impact on the player's perception.

4.2 Properties of AI in Video Games

In order to understand how artificial intelligence (AI) in video games is designed, it is important to consider the contexts of both its development and its deployment. On the deployment side, the context is entertainment. On the development side, the context is business. This affects the AI found in video games in a number of ways.

Though video games can serve purposes other than entertainment such as education, the former remains the dominant segment in the industry. As a result, the prevailing trait of video game AI is its ability to deliver fun to players. The quality of an AI is therefore not measured by how well it can handle a particular task like competing against a player at a card game but rather by how much fun the player gets from its behavior. Alternatively, the AI can be viewed as working on the task of entertaining the player by playing the game instead of playing the game as well as possible.

The properties that can make an AI fun in a video game are numerous and vary according to the player, the game and the application. For a NPC in a RPG, sometimes a good property to have is the ability to uniquely react to many different inputs from the player, giving more value to player choice and creating a more personal and engaging experience for the player. In an AA game, it can be interesting for a human guard the player must defeat to behave in a realistic way, like communicating with its allies or acting emotionally. This allows the player to exploit basic understanding of human behavior in the game. For example, if the guard possesses emotions such as compassion, anger or fear, the player can use tactics such as holding hostage one of the guard's allies to force the guard to surrender, or an-

ticipate an impulsive reaction for the guard such as charging in or fleeing when killing the guard's ally.

For computer players, the apparent lack of common sense is an issue which plagues many video games. For this reason, commonsense thinking is a property that is easily picked up and acknowledged by human players. Computer player actions based on common sense usually provide a powerful impression of intelligence, unlike actions based on raw calculation or brute force which are less likely to impress human players even if they perform better. When the computer player is playing with a human player, accepting input from the human player and prioritizing it over the default behavior can often yield a more friendly and fun-to-play-with AI. When the computer player is playing against the human player, the goal is normally to find the level of difficulty which best entertains the human player. If the computer is too hard to beat, players will give up and stop playing. If the computer is too easy to beat, players will get bored and also stop playing. The optimal entertaining experience tends to lie somewhere around a scenario where both the human and the computer seem to be tied in skill but with the human ending on top most of the time, creating a challenging and exciting match without denying the human player the joy of victory. Another concern to have for computer opponents is fairness with regard to human limitations. Showing no restraint in its capabilities, such as relying on perfect aim or timing, can impact the AI quality negatively as it is impossible for the human player to be better, and even getting close requires a lot of practice. This can result in frustration if the human player feels the computer opponent has an unfair advantage.

Naturally, AI also needs to be consistent with the pace of the genre in terms of speed. For instance, it should be fast and reactive in a real-time combat setting. If the AI takes too long to react, it can seem dull and it might not be able to keep up with the player. On the other hand, it could be too fast for the player to surprise. Likewise, if the AI in a turn-based game requires much more time than the player to make a move, the player may get tired of waiting. Conversely, if it plays instantly, it could put pressure on the player. Thus, AI must be properly tuned in a video game to match pacing.

Because video games are in many cases commercial products, the resources allocated to the development of AI in a project may or may not give developers room for error depending on the importance of AI in the game. For example, AI may not be the selling point of a game that focuses

on a multiplayer experience. It can be necessary to enable a single player mode used to allow players to familiarize themselves with the game before playing against human opponents, but developers may try to minimize the amount of resources used to make it. Even in a single player game, developers may center their efforts around other elements such as graphics or the storyline. Therefore, developers need efficient AI solutions which can be easily integrated, implemented and tested. Although most of these properties are transparent to consumers and do not affect the AI on a functional level, some of them do play a role in its shaping. For instance, the fact that AI is often deterministic can be linked to the necessity for developers to thoroughly test behavior to ensure it works as intended in all scenarios. While designing a non-deterministic AI such as a learning AI can relieve developers from having to cover every possible case, it has the disadvantage of being harder to test and the potential to lead the system into unforeseen problematic states.

It is then clear that one of the difficulties of designing video game AI lies in the fact that it has to be tailored for both the player and the game and, although the game may be static, players vary from one another and also individually change while playing, meaning that the optimum sought by the AI is inherently variable. Hence, developers require ways of crafting malleable AI that can be made to suit a large number of player profiles.

4.3 Meeting the Requirements: Examples

4.3.1 Behavior

As hinted in the previous section, behavior is an important factor of quality for AI in video games. By driving an entity of specific nature in a specific environment, AI must create a believable behavior for the entity to support the global coherence of the setting. In other words, the behavior of the entity must seem natural in its context. For example, if a character is given a reckless personality, its AI should be adjusted accordingly and display a reckless behavior. More and more, developers are designing realistic AI to suit the increasingly realistic video game environments.

In a RPG, it can be fairly easy to give NPCs such as monsters an AI that translates into a suitable behavior. After all, players will not be expecting a grunt to outsmart them. Instead, these common monsters can be given a number of basic tactics and traits sufficient to prevent the player

Figure 4.1: Some NPCs in Dungeon Siege II get angry when a player attacks their leader, turning their attention to that player.

from thinking they are mindless creatures or machines devoid of feelings. *Dungeon Siege II*, a RPG developed by Gas Powered Games and released in 2005 for PC, includes several AI features which make NPC behavior feel natural. Monsters can ambush players and pull them into mobs or call for help. They can retreat and resurrect their fallen companions too. Some of them also have distinctive traits such as hating particular actions. For example, Morden Pikemen hate it when players loot their treasure. A Morden Pikeman will get angry at a player for picking up an item on the ground. When angry at a player, a monster no longer follows its normal behavior and aggressively directs its attacks towards that player. Monsters can get angry for various reasons, including when players cast offensive or healing spells. As for uncommon monsters like boss monsters, more specialized behavior can be implemented when the encounter place is fixed. This limits possibilities and allows developers to fully utilize the environment to create sophisticated behavior without worrying about generalization.

In a FPS, general concepts such as taking cover, dodging or aiming for

vital points can be used in the behavior of enemies in order to create realistic combat. Enemies can also have features like fatigue or curiosity. For example, an enemy could fall asleep during the night or could investigate an unusual sound or object. Another feature which can help forge a natural behavior is memory. An enemy could remember locations such as the place where the player was first encountered or was last seen hiding. If it interrupts a task to execute a new one, it could go back to the task that was interrupted when it completes the new one. In *F.E.A.R.: First Encounter Assault Recon*, a FPS developed by Monolith Productions and released in 2005 for PC, the AI powering enemies features such concepts and can generate behavior based on context. This results in enemies that use the environment effectively, such as finding cover behind nearby objects. The AI includes squad tactics too, allowing enemies to use tactics such as surrounding the player. Shooters often have limits placed on AI accuracy in an effort to mimic realistic aim. While the AI of an enemy could instantly move a crosshair over a player and shoot with perfect accuracy, this ability is usually unrealistic for the enemy. Enemies are more likely to require a short delay to aim and to use movement prediction in order to anticipate where the target will move to, especially when handling projectile weapons.

In a RTS game, unit AI has to be balanced in order to deliver a level of autonomy such that players are involved just enough on the battlefield. If the AI is too simple, unit control will be tedious and will force players to be too involved on the battlefield, leaving them with less opportunities to focus on strategy. If it is too smart, players will seldom need to be involved on the battlefield and will get detached from it, dulling the action component of the game and turning it into plain strategy. Micromanagement can significantly boost the efficiency of units in combat and is often essential to secure victory in a battle. For example, target prioritization is crucial when a player needs to quickly reduce the enemy's firepower or support potential. In *StarCraft II: Wings of Liberty*, a RTS game developed by Blizzard Entertainment and released in 2010 for PC, the AI offers players some comfort in unit control without dimming their role on the battlefield. Examples include automatically queuing units in idle production buildings when the queue command is issued to multiple buildings, automatically designating a worker to carry out a build command when it is issued to multiple workers, preventing a cast command from being executed several times when it is issued to multiple casters (Smartcasting) and idle units automatically moving away from the target location of a build command. Since players frequently handle groups of units in RTS games, AI can also

Figure 4.2: Smartcasting in StarCraft II: Wings of Liberty automatically designates a unit among selected ones to cast the requested ability, allowing players to quickly and efficiently cast many abilities.

be used to appoint leaders to handle group commands. This can simplify these tasks and result in more natural group behavior.

While making a convincing AI for game world entities such as units or characters is usually possible, it is not in many cases for computer players. Computer opponents that serve as an alternative to human opponents fall short of the qualities of a human player, especially when the game features a rich world involving the use of several different concepts. Because their AI is commonly designed by breaking down possibilities into a handful of manageable cases, their behavior is rigid and vulnerable to change. This makes it likely for them to behave in ways humans would not, such as completely ignoring an obvious threat, and quickly shows their total unawareness of basic concepts. Their lack of ability to learn makes them helpless against human players, which are quick to model their opponent and adapt. For example, in a FPS, a human player can easily learn to bait a computer player into an intersection and fire a projectile weapon, like a missile launcher, in time for the projectile to hit the computer player as soon as it appears. A human player can use the same strategy against a computer player to win every time, which is unlikely against a human

player. Not only is their behavior static, computer players are also missing other traits that further distance them from their human counterparts. One example is the perfect consistency of their performance. Computer players are unaffected by external factors such as stress or weariness, which can have a visible impact on the behavior of human players.

4.3.2 Difficulty

Often, AI directly affects the difficulty of the game. Difficulty is a determining factor in the entertainment provided by video games and must be carefully adjusted in order to offer players a reasonable challenge. A reasonable challenge is one the player both acknowledges and feels capable of overcoming. This ensures that players are rewarded with a sense of accomplishment in the case of victory and guilt in the case of defeat, making it less likely for them to give up thinking the game was poorly designed and more likely to want to play again. Moreover, because different players have different levels of skill, a video game must be ready to deliver similar challenges to both weak and strong players. This must be achieved in a way that does not alter the core gameplay. For example, certain mods of *Diablo*, an action-RPG game developed by Blizzard North and released in 1996 for PC, feature difficulty levels where players are forced to fight monsters one by one because they are too powerful. To avoid drawing more than one monster at a time, players have to use equipment which decreases light radius (i.e., the light surrounding the player character and affecting both the visibility and the vision of the character) and navigate carefully amidst the hordes of enemies. This changes the game from a fast-paced action, one-on-many combat type to a less dynamic puzzle-maze type.

Many video games include multiple difficulty levels. These settings allow players to tune the game in order to match their skill and preference. An experienced player looking for a challenge might choose an "Expert" mode whereas a novice player interested in unveiling a storyline without any hassles may choose an "Easy" mode. *The Witcher 2: Assassins of Kings*, an action-RPG developed by CD Projekt RED and released in 2011 for PC and Xbox 360, comes with a tutorial[1] which helps players evaluate their skill level and select an appropriate difficulty setting before starting the main campaign. Depending on the genre, the difficulty of a video game can be adjusted in a number of ways. Making enemies stronger, faster

[1]Enhanced Edition update, released in 2012.

or smarter, adding new types of enemies, maintaining a constant state of danger (i.e., no safety zones), limiting resources such as lives, gold or ammunition, increasing the size of the puzzle, adding time restrictions and denying the player functions such as consulting the map or saving the game are all examples of modifications that increase difficulty. In *Final Fantasy XII*, a RPG developed by Square Enix and released in 2006 for PlayStation 2, some special equipment is hidden in areas where no map is available, making it harder to acquire than regular equipment and requiring players to rely on their memory to map the area. Note that greater difficulty levels are not always targeted at more skilled players. For example, they can simply be designed to follow character growth. In *Diablo*, harder difficulty levels allow players to replay the same dungeons populated with higher level monsters. Players need a high level character with good equipment to face these enemies and therefore have to invest more time leveling up their character and collecting items before succeeding in these difficulties.

Video games may also feature a dynamic difficulty adjustment (DDA) system. Although difficulty may seamlessly increase as the player progresses in a game, DDA allows the game to tune difficulty in real-time to best suit the current state of the player. This can smooth the player's experience in the case of inconsistent performance and can sometimes prevent the player from continuously exploiting a winning strategy. In *Zanac EX*, a shoot 'em up game developed by Compile and released in 1986 for MSX2[2], the enemy AI has a dynamic aggression level which automatically adjusts the quantity and type of ships to send out according to the player's skill and the state of their ship. For example, losing lives decreases the aggression level while equipping a shield type special weapon causes the AI to spawn ships that attack from the sides. *Mario Kart 64*, a racing game developed by Nintendo EAD and released in 1996 for Nintendo 64, automatically increases or decreases the speed of AI-controlled karts depending on their distance to the player in order to avoid a monotonous race where the player would be too far ahead of or behind the other racers. *Unreal Tournament* provides an option to automatically adjust bot skill when playing with bots. When enabled, the bot skill level is dynamically set during a match to the highest value the player manages to win against. *Left 4 Dead*, a FPS developed by Turtle Rock Studios and released in 2008 for PC and Xbox 360, has a special AI called the "Director" (AID). During a game, where a group of players must cooperate to fight against packs of

[2]The second generation of the MSX home computer designed by ASCII Corporation and Microsoft and released in 1983.

Figure 4.3: Adaptive opponents in Mario Kart 64 (left) and Zanac EX (right).

zombie-like enemies, the AID evaluates the performance of each player in real-time and spawns items and enemies at varying locations depending on their situation and skill level. *Resident Evil 5*, a FPS developed by Capcom and released in 2009 for PlayStation 3, Xbox 360 and PC, uses an adaptive difficulty system too. This grading system dynamically tunes the difficulty of the enemies the player is facing by modifying damage multipliers to make them stronger or weaker and by toggling AI functions to make them smarter or dumber.

Another feature frequently found in video games is cheating AI. AI can be allowed to cheat for various reasons. A common reason is to provide a greater challenge to human players when it is too difficult or not currently possible for the computer to defeat a human. For example, computer players are unable to compete against human experts in RTS games using the current AI technology and hardware. A computer player may thus be given unfair advantages such as complete map vision (i.e., no fog of war), increased income from gathered resources and faster production times. *Command & Conquer: Red Alert 3*, a RTS game developed by EA Los Angeles and released in 2008 for PC, features a "Brutal" AI difficulty which resorted to such cheating. The automatic speed adjustment in *Mario Kart 64* is also a form of cheating as it allows AI-controlled karts to move faster than those controlled by human players when closing in on them. Cheating can be used to avoid undesired scenarios too. For instance, if a human player is facing a computer player in a large map in a FPS, roaming for too long without encountering the opponent may cause the human player to lose interest in the match. Tipping off the computer opponent to direct it towards the location of the player can prevent this kind of scenario.

Ultimately, video game AI has to be designed to be easily tunable and

balanced in different settings so as to offer an enjoyable experience regardless of the player's skill level. Additional AI can be used to tune AI behavior as well as other game parameters automatically and refine the player's experience even further.

4.3.3 Learning

One of the shortcomings in video game AI today is its inability to evolve. Although a static behavior may be appropriate for cases with limited interaction, agents that stay longer around human players quickly suffer from this flaw. A NPC following a player character often behaves in a mechanical fashion and fails to portray a realistic being. A computer player facing a human player easily reveals its weaknesses and acts more like a puzzle to solve than a smart adversary. There are a few reasons the video game industry has been reluctant in making use of machine learning techniques. The complexity of modern video game worlds can make it challenging to design a learning agent that can be deployed in an environment where multiple concepts are in play. Even when it is possible to create robust learning agents, the process of training them can be too costly. Furthermore, evolving agents are harder to test and can be problematic to quality assurance (QA) processes. Still, there are some instances of developers applying machine learning to video games, with varying degrees of success.

Creatures is a virtual pet or artificial life game developed by Creature Labs and released in 1996 for PC. In this game, players have to raise and breed small animals called "Norns". These creatures have a brain and can be taught to speak and eat as well as to protect themselves from other creatures. They also have a genome which includes detailed information about their brain, biochemistry and appearance. The underlying learning architecture employs neural networks to model the brains. Besides learning, evolution is also driven by genetic mutations such as point mutations or insertions and deletions which alter genes from generation to generation. Players can thus witness the emergence of new behavior as they interact with the environment.

In *Black & White*, a strategy game developed by Lionhead Studios and released in 2001 for PC, players incarnate a god seeking supreme control over the world. Levels include a number of villages which players must gain control of by performing various miracles, such as moving around objects or beings, making it rain over fields and casting shields and protecting

Figure 4.4: Learning agents in Black & White (left) and Creatures (right).

them from opposing gods. In addition to their direct intervention, players control a large creature that acts on their behalf and can teach it to do their bidding. Reinforcement learning is used to adjust the creature's behavior. After taking an action, the player can reward the creature by stroking it or punish it with a slap. It can also be trained to improve its stats by executing specific tasks. For example, making it carry heavy rocks or trees will increase its strength. Depending on their preference, players can choose to be a caring god with a kind creature and have villagers worship them through love or they can be an oppressive god with an evil creature and have villagers worship them through fear.

Tekken: Dark Resurrection is a fighting game developed by Namco and released in 2006 for PlayStation Portable where two players each control a character and fight each other by performing attacks such as punches and kicks. Players can unleash high-speed combos by executing complex input sequences to quickly deplete their opponent's health bar. The game features a "Ghost Battle" mode where players can record their play and create "ghosts". These AI-controlled fighters use imitation learning to learn behavior from players and can be shared among players as an alternative to the default AI. The game allows players to download and upload ghosts on an online server.

Learning has been applied in online ranking systems too, such as *TrueSkill*. *TrueSkill* is a ranking system developed by Microsoft Research and used by the Xbox Live platform to track player skill levels, provide matchmaking services and create leaderboards. It was launched alongside the Xbox 360 in 2005. These ranking systems need to learn efficiently using as few samples as possible in order to reduce the number of uneven matches arranged, as players rarely enjoy a match they are guaranteed to win or lose. The unsupervised learning approach in *TrueSkill* uses Bayesian infer-

ence at its core to assign skill values to players.

Darkwind: War on Wheels is a massively multiplayer online turn-based racing and vehicular combat game developed by Sam Redfern and released in 2007 for PC. The game also features some strategy and RPG elements and focuses on tactics in circuits and arenas. During a race, players issue orders to move their vehicle and fire at other vehicles using turns. Drivers can either be controlled by human players or by the computer. The game uses genetic algorithms to evolve effective racing lines and optimize the computer drivers.

In *Forza Motorsport 5*, a racing game developed by Turn 10 Studios and released in 2013 for Xbox One, imitation learning is applied on a large scale to power the "Drivatar" technology, which is used at the core of the computer drivers. During each race, the game records player data such as their position or acceleration throughout the race as well as data about other racers and relational data. At the end of the race, the data is uploaded to an online server where the data is processed. Behavior, like faking, is extracted from patterns and applied to the computer drivers. To avoid learning undesired behavior from low quality or bad data, the developers manually review acquired behavior and discard unwanted results.

While learning has the potential to greatly improve the quality of video games, current learning techniques are often impractical as standalone solutions because they require significant amounts of data and processing power to yield interesting results, something individual players are unable to provide. Learning solely based on local player data may easily lead to overfitting, making cloud-based approaches such as the one used in *Forza Motorsport 5* seem more promising.

4.4 AI Techniques for Video Games

Albeit outside the scope of this work, there are many AI techniques used in the video game industry to create game AI efficiently. This subject is already covered extensively by several books that present video game AI in detail. Examples include *Programming Game AI by Example* by Mat Buckland [52], *AI Game Engine Programming* by Brian Schwab [132], *Artificial Intelligence for Games* by Ian Millington and John Funge [102], *Artificial Intelligence: A Modern Approach* by Stuart Russel and Peter Norvig [125] and the *AI Game Programming Wisdom* books by Steve Rabin [120, 121, 122, 123]. More

specific publications that focus on positioning for example also exist, such as work from Remco Straatman [135] and Dave Pottinger[118].

Chapter 5

Problem Statement

Contents

Summary

The goal of this chapter is to present the questions driving this research. It is composed of two parts. The first part introduces the issue of complex environments and AI fragility in video games. Then, the issue of recurring conceptual problems and AI redundancy in video games is briefly described in the second part. Together, these two points outline the problem at the center of this work.

5.1 AI Fragility

As video games continue to feature richer and more complex environments, designing robust AI becomes increasingly challenging. This is easily noticed in modern genres with complex environments, where computer players or even lesser game agents such as NPCs[1] quickly reveal their limitations in dealing with all the game concepts and ultimately fail to cope with the intricacy of the environment. A few examples are mentioned below to illustrate this fact.

In the FPS game *Unreal Tournament 2004*, human players can compete against computer players, or bots, in the single player mode. Several bot skill levels are available, ranging from the weakest Novice bots to the strongest Godlike bots. Regardless of this difficulty setting which can improve the bots' decisions as well as reaction time, aiming and dodging, they can be reliably beaten by any human player taking advantage of their lack of awareness of certain elements of the environment. For example, if a human player followed by a bot takes a lift to an upper level, the bot will continue chasing the human player by taking the lift too, despite the danger involved. All the player has to do is fire a powerful projectile such as a fully-charged BioRifle glob or a trio of rockets or detonate a Shock Combo at the right time to blow the unwary opponent to bits. Another example is the bots' inability to see through transparent surfaces. A human player can wait around a corner with glass windows in the walls to see a bot coming. The bot does not take into account the information revealed through the glass and will simply walk around the corner and be surprised by the player. Even though bots can be very skilled, these kinds of mistakes show how limited the underlying AI is.

In the MOBA game *League of Legends*, human players can participate in a "Co-op vs. AI" match to team up against bots. There are two difficulty settings for bots, Beginner and Intermediate. Both share a number of critical shortcomings, such as poor threat assessment. Bots will make aggressive plays even when their opponents' damage potential largely exceeds theirs, resulting in losing trades. Bots also ignore threats that are on the way when returning to base. Human players waiting close to the enemy base in a lane are completely ignored by the bots returning to the base who simply walk past them, usually without ever making it back home. This rigid return

[1] Unlike computer players, NPCs often interact with the environment in a limited way and therefore their AI does not have to involve the entire complexity of the game.

behavior results in additional issues. If the return routine triggers during a duel between a bot and a player, for instance because the bot's health drops below a certain point, and the player's health is low enough to be finished off by a single blow, the bot will not deal it and will instead run away. These shortcomings show again that the bot AI is rather limited.

The RTS game *StarCraft: Brood War* also features bots and a mode to play against them called "Melee". The difficulty setting cannot be configured by the player for this mode, though various different levels can be used in a custom map played in UMS mode, including an Insane level. Here too, these settings do little to palliate the severe flaws from which the bots suffer. For example, a human player can send a worker at the beginning of a match against a bot to land a hit on one of its workers and run away, causing the bot to send them all to chase the intruder and leaving it with no income. The player can then draw circles with the worker and the bot will relentlessly chase it until it eventually gives up, though the same tactic can be used repeatedly to prevent the bot from doing anything. A similar trick can also be used to lead the bot's army away from an enemy base to launch an assault on it, or away from the player's base in case the bot is the one attacking. Another problem is that bots do not protect their resources. A human player can send workers to gather from the mineral fields located in the enemy base and even build a refinery on its Vespene geyser and the bot will simply build its own refinery in a remote base. Once again, the fragility of the AI is easily exposed.

Diablo II: Lord of Destruction (LoD) is a RPG developed by Blizzard Entertainment and released in 2001 for PC. An interesting feature in LoD is that players can hire a mercenary to fight alongside their character. Unfortunately, the AI driving the behavior of mercenaries is very limited, often resulting in giving the player a harder time trying to deal with it rather than making their life easier. For example, a mercenary lets itself be surrounded and killed with relative ease. It also does not discern between enemies in any way, meaning that it could be attacking[2] a monster which is immune to physical damage and which does not even deal any damage while being under attack by a more dangerous and vulnerable monster. Mercenaries also ignore status effects. If they are under the effect of the Iron Maiden curse, which reflects physical damage back to the attacker, they will continue to attack with no hesitation and die. Note that resurrecting a mercenary requires the player to return to the nearest town and costs a

[2]Using physical attacks.

Figure 5.1: Mercenary AI in Diablo II: Lord of Destruction. The desert mercenary (poisoned) easily gets surrounded by Maw Fiends.

significant amount of gold, which can make their poor performance quite frustrating.

All of these examples illustrate issues that are not specific to their respective games. Similar issues can be found in similar games, and all of them lead to the same observation, which is that developers are unable to design robust AI for complex games. This problem is important because when AI fails to display the realistic behavior required to match the realistic environment in which it operates, it creates inconsistencies in the mind of the player and undermines the convincing experience that the game is attempting to deliver. If guards can forget their friends after the player kills them and continue patrolling as if nothing was going on, it does not matter how realistic the environment looks because the overall experience is not realistic anyway. It is therefore interesting to ask the following question. What could be done to enable the design of robust AI for modern, increasingly complex genres?

5.2 AI Redundancy

It is worth noting that none of the issues described in the previous section are irremediable. In fact, most of them could easily be fixed. The reason

they exist is because development resources are limited. Developers need to create the AI in the span of a game project and may not have enough time to consider such scenarios or handle them. It is also interesting to note that these issues involve elements that are generic and not specific to the game. The danger of taking a lift after an opponent is not unique to UT2004, the income loss induced by sending workers to chase a harmless enemy is not unique to BW and the threat of getting surrounded for a NPC is not unique to LoD. Instead, these apply to the entire FPS, RTS and RPG genres respectively, and some even apply to multiple genres.

The fact that games share concepts through genres also means that AI in video games of the same genre is likely to face similar challenges involving the concepts they have in common. Since each video game usually has its own AI which is developed specifically for it, this means that there can potentially be a lot of redundancy in AI from game to game to handle the same problems related to the same concepts. Of course, this does not prevent games of the same genre to differ significantly from one another. However, it is not unreasonable to imagine that at least part of the AI could be factored under the form of generic, conceptual AI, which could be complemented with specific AI if necessary.

By detaching the development of such AI from the development of the game and targeting it at recurring conceptual problems, it can become possible to create more robust solutions to these problems and use them across multiple games, thus taking a step towards enabling the design of robust AI for complex genres. This could result in access for game developers to better AI without the hassle of designing it. For players, it could result in more convincing AI with less common-sense mistakes that cannot be justified by a skill level setting. The question that can be asked is then the following. How could conceptual, cross-game AI be designed and used?

Chapter 6

Towards a Unified Framework

Contents

Summary

This chapter presents a development model for video game AI based on the use of a unified conceptual framework. The first section introduces the reason behind unification. Section two proposes conceptualization as a means to achieve unification. The third section discusses the design of conceptual AI while the fourth section discusses conceptual problems. Section five then focuses on the integration of conceptual AI in video games.

6.1 Why unify?

Originally, the wish for a unification of artificial intelligence (AI) development stems from a number of contemplations. The foremost observation is that AI in video games often displays serious shortcomings. Especially in complex genres, players are used to watching their virtual peers commit unforgivable mistakes. The second observation, following from the first, consists in realizing that creating a robust AI is a very consuming task. Either too many cases need to be exhaustively covered or the various concepts involved in the game must be thoroughly implanted in the AI engine. This is not only time consuming, but can also pose an intellectual challenge because it requires concepts to be extracted from the game, clearly defined, organized and properly handled by the AI. Most of the time, budget constraints or profitability measures prevent video game developers from traveling down this path, although it would not be surprising that their in-house designs could gradually converge towards the application of conceptual solutions.

One more relevant observation is the fact that video games are categorized in genres. Genres are used to group video games that share a number of concepts. The more specific the genre, the larger the number of shared concepts becomes. With concepts being shared heavily, it is natural to believe that a lot of AI could be factored using common elements. This belief is reinforced by a fourth observation, which is a human player's success in applying the same policy in different games. As they play, humans tend to build, in their mind, policies for the different types of environments and challenges encountered. These policies can then be used in any game featuring similar elements. This means that these policies need not be learned individually for every game and are instead simply completed or refined throughout the player's entire gaming experience. Of course, additional specialized knowledge is assembled and stored for each game and used to tweak these policies or even override them in order to improve performance in individual games.

Together, these observations hint at a unified AI core. Indeed, not only does the idea of factoring AI seem natural, its execution also calls for a centralization of effort in order to create a single, robust kernel which would be perfected over time, not unlike the evolution of a human player's collection of policies formed throughout their gaming experience.

6.2 Conceptualize and Conquer

Despite their apparent diversity, video games share concepts extensively. Thus, creating AI that operates solely on concepts should allow developers to use it for multiple games. This raises an important question however, namely that of the availability of a conceptual interpretation of video games. In reality, for AI to handle conceptual objects, it must have access to a conceptual view of game data during runtime.

When humans play a video game, they use their faculty of abstraction to detect analogies between the game and others they have played in the past. Abstraction in this context can be seen as a process of discarding details and extracting features from raw data. By recalling previous instances of the same conceptual case, the experience acquired from the other games is generalized and transformed into a conceptual policy (i.e., conceptualized). For example, a player could have learned in a RPG to use ranged attacks on an enemy while staying out of its reach. Later, in a RTS game, that player may be faced with the same conceptual situation with a ranged unit and an enemy. If, at that time, the concept of kiting isn't clearly established in the player's mind, they may remember the experience acquired in the RPG and realize that they are facing a similar situation: control over an entity with a ranged attack and the ability to move and the presence of an enemy. The player will thereby conceptualize the technique learned in the RPG and attempt to apply it in the RTS game. On the other hand, if the player is familiar with the concept of kiting, a simple abstraction of the situation will lead to the retrieval of the conceptual policy associated with it, without requiring the recall of previous instances and associated experiences and their conceptualization.

Note that kiting can be defined using only concepts, such as distance, attack range and movement. Distance can have several distinct interpretations, for example yards, tiles or hops. Attack range can be a spell range, a firearm range or a gravity range. Walking, driving and teleporting are all different forms of movement. Kiting itself being a concept, it is clear that concepts can be used to define other concepts. In fact, in order to define conceptual policies, different types of concepts are necessary, such as objects, relationships, conditions and actions. Weapon, enmity, mobility[1] and hiding are all examples of concepts.

[1]The condition of being mobile.

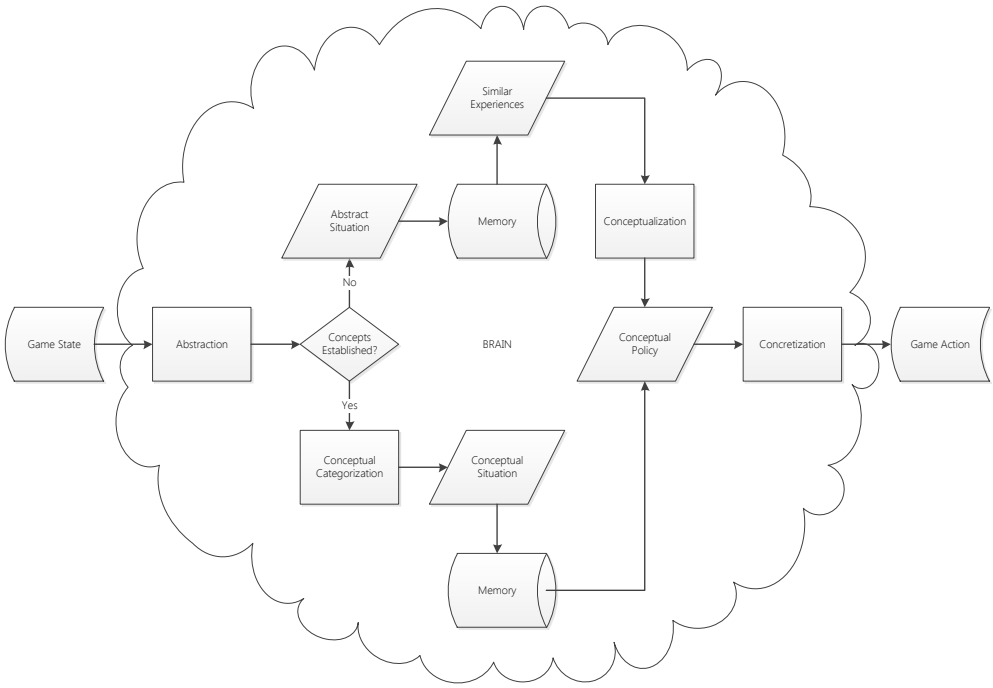


Figure 6.1: Possible process of human decision making in a video game using conceptual policies, as described above. If memory queries don't yield any results, a concrete policy is computed in real-time using other cognitive faculties such as logic or emotion.

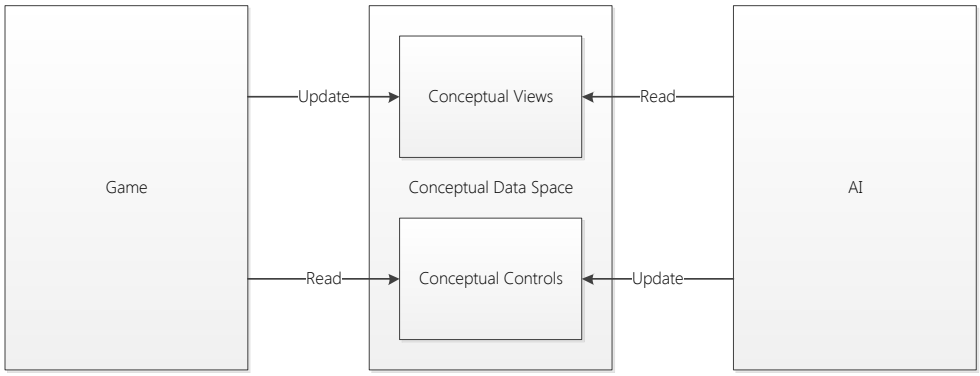


Figure 6.2: Basic architecture of a video game using conceptual AI. The game maintains a conceptual view of its internal state. A conceptual view is the projection of a part of the game state into conceptual space. Based on this conceptual data, the AI controls an agent in the game by issuing conceptual commands, which the game translates back into game actions.

According to the process shown in Figure 6.1, conceptual AI, that is AI which operates entirely on concepts, could be used in video games under the premise that three requirements are met. These would be:

1. the ability to translate game states into conceptual states,
2. the ability to translate conceptual actions into game actions,
3. and the ability to define conceptual policies[2].

Though the third requirement raises no immediate questions, the other two appear more problematic, as translating states and actions needs to be done in real-time and there currently exists no reliable replacement for the human faculty of abstraction. It follows from the latter assertion that this translation must be manually programmed at the time of development. This means that the game developer must have access to a library of concepts during development and write code to provide access at runtime to both conceptual views and conceptual controls of the game for the AI to work with. Using such a process, both the real-time availability and the quality conditions of the translation are satisfied.

As hinted in Figure 6.2, rather than translating game states into conceptual states discretely, it is easier to simply maintain a conceptual state

[2]A conceptual policy maps conceptual states to conceptual actions.

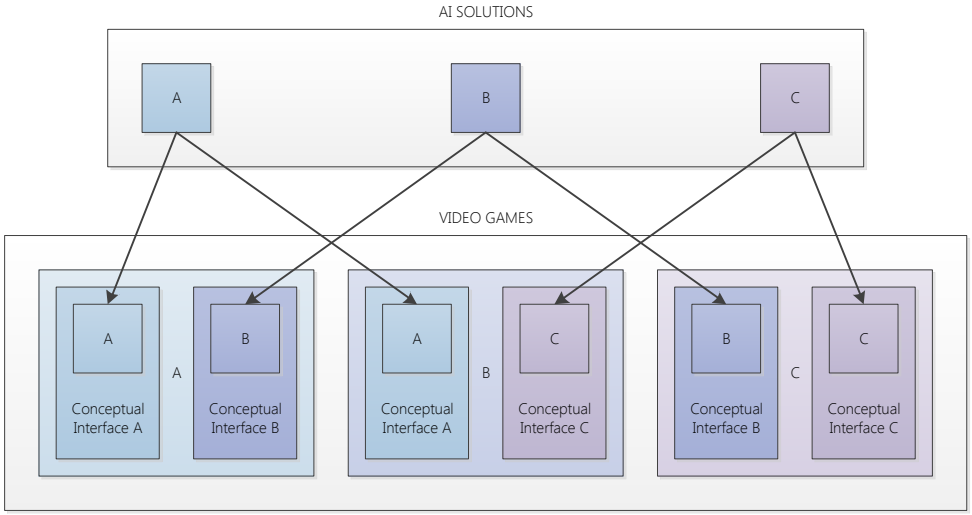


Figure 6.3: Using the same AI in multiple games. AI A can run in games A and B because both implement the conceptual interface A it requires. A conceptual interface is a set of conceptual views and controls.

in the conceptual data space (CDS). In other words, the conceptual state is synchronized with the game state. Every change in the game state, such as object creation, modification or destruction, is directly propagated to the conceptual state. Note that there is no dynamic whatsoever in the CDS. A change in the CDS can only be caused by a change on the game side, wherein the game engine lies.

Obviously, this design calls for a unified conceptual framework (CF). That is, different developers would use the same conceptual libraries. This would allow each of them to use any AI written using this unique framework. For example, a single AI could drive agents in different games featuring conceptually similar environments, such as a FPS arena. This is illustrated in Figure 6.3.

From a responsibility standpoint, the design clearly distinguishes three actors:

1. the game developers,
2. the AI developers,
3. and the CF developers.

The responsibilities of game developers include deciding which AI they need and adding code to their game to maintain in the CDS the conceptual views required by the AI as well as implementing the conceptual control interfaces it uses to command game agents. Thus, game developers neither need to worry about designing AI nor conceptualizing games. Instead, they only need to establish the links between their particular interpretation of a concept and the concept itself.

On the opposite side, AI developers can write conceptual AI without worrying about any particular game. Using only conceptual elements, they define the behavior of all sorts of agents. They also need to specify the requirements for each AI in terms of conceptual views and controls.

Finally, the role of CF developers is to extract concepts from games (i.e., conceptualize) and write libraries to create and interact with these concepts. This includes writing the interfaces used by game developers to create and maintain conceptual views and by AI developers to access these views and control agents.

Because the CF should be unique and is the central component with which both game developers and AI developers interact, it should be developed using an open-source and extensible model. This would allow experienced developers from different organizations and backgrounds to collaborate and quickly produce a rich and accessible framework. Incidentally, it would allow game developers to write their own AI while extending the framework with any missing concepts.

6.3 Designing Conceptual AI

From a technical perspective, writing conceptual AI is similar to writing regular AI. That is, developers are free to design their AI any way they see fit. Conceptual AI does not require a specific form. The only difference between conceptual AI and regular AI is that the former is restricted to the use of conceptual data. Rather than operating on concrete objects, such as knights, lightning guns or fireball spells, it deals with concepts such as melee tanking units, long-range hitscan weapons and typed area-of-effect damage projectile abilities. Likewise, actions involve conceptual objects and properties instead of concrete game elements and can consist in producing an anti-air unit or equipping a damage reduction accessory. This difference is illustrated in Figures 6.4 and 6.5.

```
1  void handle_enemy(pc_t& enemy)
2  {
3    ...
4
5    if (enemy.type() == pc_t::cleric || enemy.type() == pc_t::
         sorcerer || enemy.type() == pc_t::ranger)
6      queue_action(use_skill(Skill::root, enemy));
7
8    queue_action(attack(enemy));
9
10   ...
11 }
```

Figure 6.4: Fortress Defender combat code snippet

```
1  void handle_enemy(pc_t& enemy)
2  {
3    ...
4
5    if (enemy.ranged() && can_impair_movement())
6      queue_action(use_skill(get_skill(SkillType::disable_move)
           , enemy));
7
8    queue_action(attack(enemy));
9
10   ...
11 }
```

Figure 6.5: Conceptual combat code snippet

Figure 6.4 shows an excerpt from the combat code of a Fortress Defender, a melee NPC in a RPG. A Fortress Defender can immobilize enemies, a useful ability against ranged opponents who might attempt to kite it. Before commanding the NPC to attack an encountered enemy, the code checks whether the type of opponent is one of those who use a ranged weapon and starts by using its immobilization ability if it is the case.

Figure 6.5 shows a possible conceptualization of the same code. Note how the design remains identical and the only change consists in replacing game elements with conceptual ones. As a result, the new code mimics a more conceptual reasoning. In order to prevent a ranged enemy from kiting the melee NPC, the latter checks whether a movement-impairing ability is available and uses it on the target before moving towards it. Whether the

actual ability turns out to slow, immobilize or completely stun the opponent holds little significance as long as the conceptual objective of preventing it from kiting the NPC is accomplished. Although this requires developers to think in a more abstract way, they do retain the freedom of designing their AI however they are accustomed to.

Despite this technical similarity, the idea of conceptualizing video games suggests looking at AI in a more problem-driven way. There are two obvious reasons. First, conceptual AI does not target any game in particular, meaning that it should not be defined as a complete solution for an entire game. Second, with the various interpretation details omitted, AI developers can more easily identify the conceptual problems that are common to games of different genres and target the base problems first rather than their combinations in order to leverage the full factoring potential of conceptualization. The idea of solving the base conceptual problems and combining conceptual solutions is illustrated in Figure 6.6.

Besides combining them, it can be necessary to establish dependencies between solutions. An AI module may rely on data computed by another module and require it to be running to function properly. For example, an ability planner module could require a target selection module by planning abilities for a unit or character according to its current target. This can be transparent to game developers when the solutions with dependencies are combined together into a larger solution. When they are not however, game developers need to know whether an AI module they plan on using has any dependencies in order to take into account the conceptual interfaces required by those dependencies. This means that AI developers have to specify not only the conceptual interface an AI solution uses, but also those required by its dependencies. Dependencies in combined and individual AI solutions are illustrated in Figure 6.7.

It can be argued that problems are actual video game elements. The difference between them and other elements such as objects is that they are rarely defined explicitly. They might be in games where the rules are simple enough to be listed exhaustively in a complete description of the problem the player is facing, but often in video games the rules are complex and numerous and a complete definition of the problems players must face would be difficult to not only write, but also read an understand. Instead, a description of the game based on features such as genres, environments or missions convey the problems awaiting players in a more intuitive way. With such implicit definitions, there can be many ways of breaking down

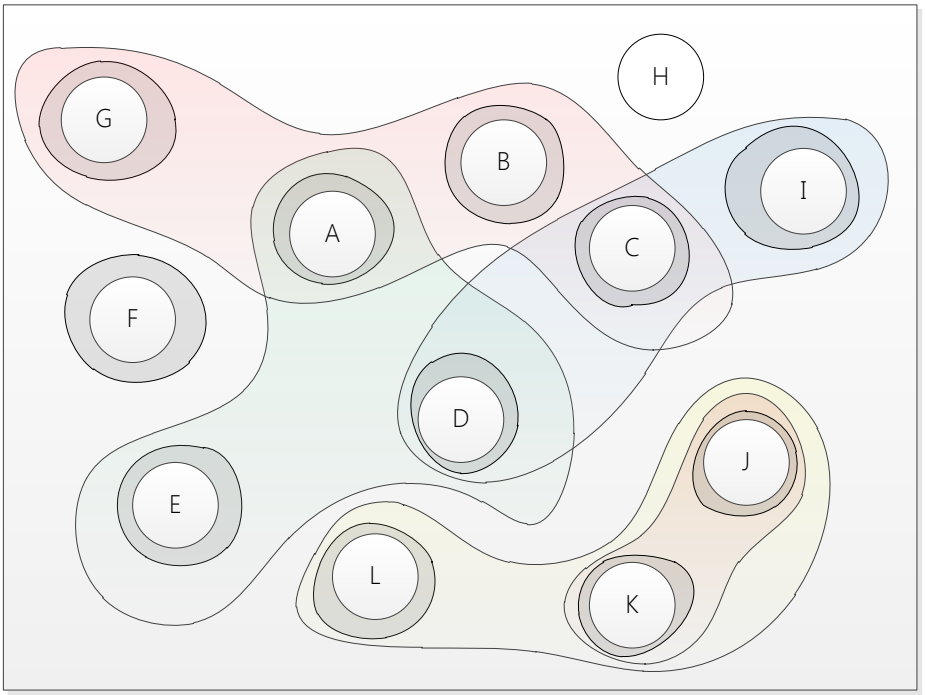


Figure 6.6: Conceptual problems (circles) and solutions (irregular forms). Instead of looking at the whole ADE and CDI problems found in two different games and solving them directly, solving problem D twice in the process, it is more interesting to identify the individual problems A, C, D, E and I and solve them once first. A solution based on those of the individual problems can then be developed for each game without having to solve them again.

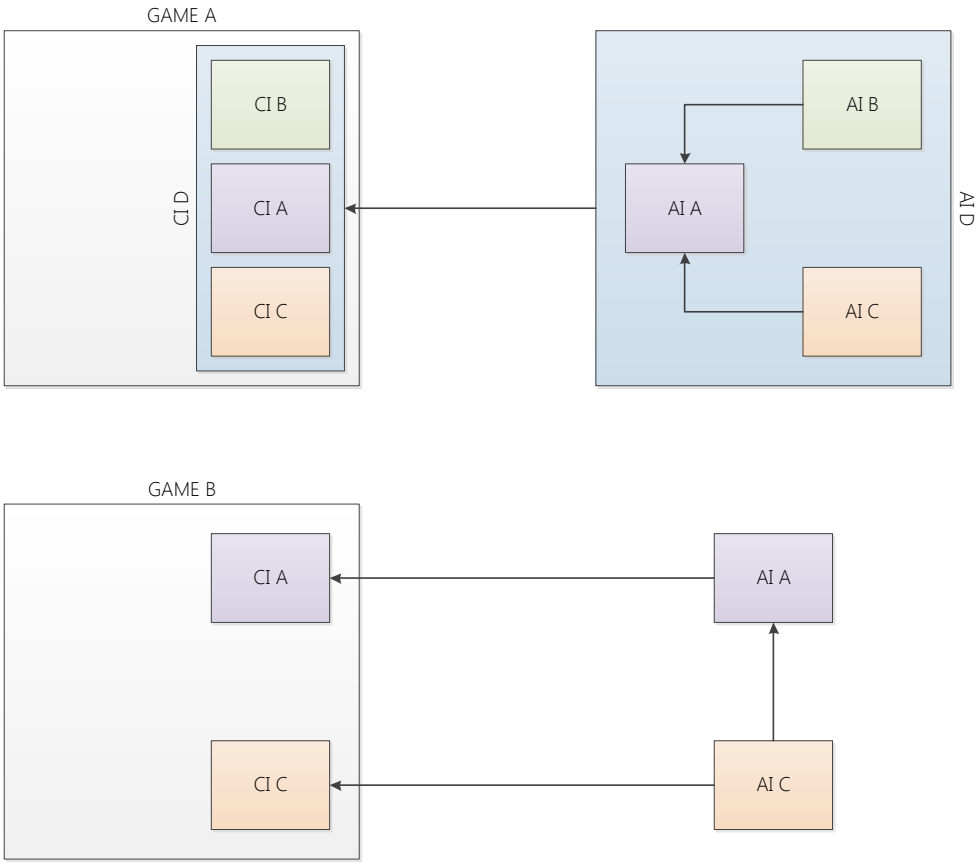


Figure 6.7: AI dependency in combined and individual solutions. Arrows represent requirement. A combination of AI solutions (AI D) has its own conceptual interface (CI D) which includes those of its components, making AI dependency transparent to game developers. In the case of separate AI solutions, a dependency (AI C requires AI A) translates into an additional conceptual interface (CI A) for game developers to provide.

video games into conceptual problems. Different AI developers might consider different problems and compositions. There are no right or wrong configurations of conceptual problems, though some may allow developers to produce AI more efficiently than others, just like the concepts making up the CF. It was suggested that the CF should be developed using an open-source model to quickly cover the numerous existing concepts through collaboration and ease the addition of new ones. The same suggestion could be made for conceptual problems. If conceptual problems are listed and organized in the CF, AI developers can focus on solving conceptual problems instead of identifying them. As with concepts, as conceptual problems are identified and included in the CF, they become part of the AI developers' toolkit and allow them to better design their solutions. This task can be added to the responsibilities of CF developers, though since AI developers are the ones facing these conceptual problems and dealing with their hidden intricacies, they are likely to detect similarities between solutions to seemingly distinct problems, and in extension similarities between problems, and could collaborate with CF developers to restructure problems or contribute to the CF directly. Similarly, game developers deal with the details of the explicit elements and may have valuable contributions to make to the CF. In a way, CF developers can include both game and AI developers who could be assuming a double role either as direct contributors or as external collaborators. Such an organization together with the idea of breaking down video games into conceptual problems and using these as targets for conceptual AI is shown in Figure 6.8. The AI used in a video game could thus be described as solutions to elementary or composite conceptual problems.

6.4 Identifying Conceptual Problems

Conceptual problems are the heart of this video game AI development model. Indeed, it would serve little purpose to conceptualize video games if the resulting concepts could not be used to identify problems that are common to multiple games. Problem recurrence in video games is the raison d'être of such a model and why factoring video game AI is worth pursuing. The amount of factoring that can be achieved depends on how well recurring problems are isolated in video games not only of the same genre, but of any genre. This could be used as a measure of the efficiency of the model, as could be the amount of redundancy in AI solutions to disjoint problems. Clearly identifying and organizing conceptual problems is there-

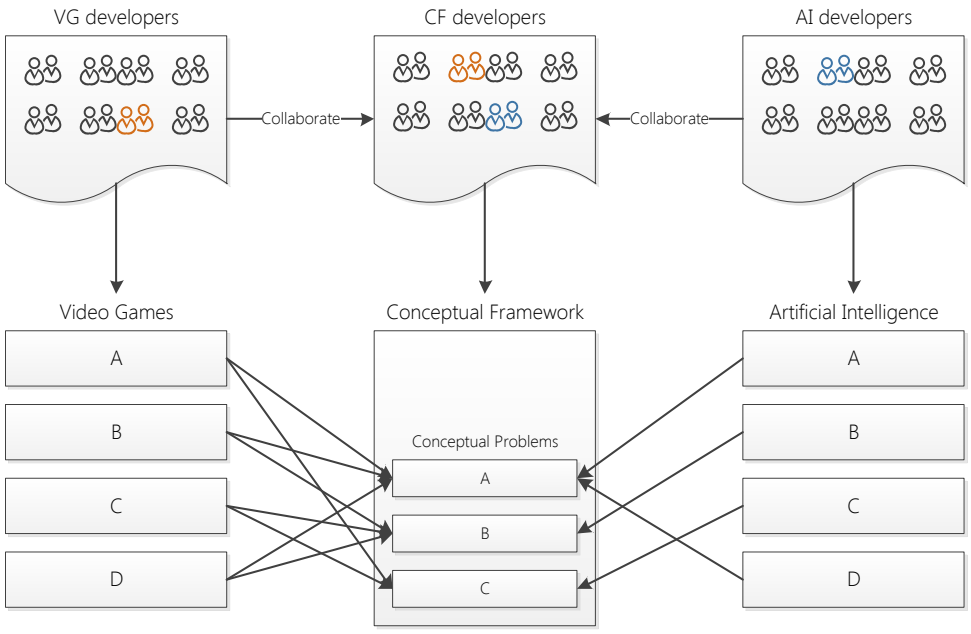


Figure 6.8: Collaboration between developers. Some game and AI developers, possibly large organizations or pioneers, also help developing the CF. Others only use it. Conceptual problems (CP) are listed and organized in the CF. A conceptual problem can be included in multiple games (CP B is included in VG B, C and D), and can have multiple solutions (AI A and D are two different solutions for CP A).

fore a crucial dimension of this development model.

Problems and their solutions can either be elementary or composite. Elementary problems are problems whose decomposition into lesser problems would not result in any AI being factored. They are the building blocks of composite problems. The latter combine several problems, elementary or composite, into a single package to be handled by a complete AI solution. For example, an agent for a FPS arena deathmatch can be seen as a solution to the problem of control of a character in that setting. This problem could be decomposed into smaller problems which can be found in different settings such as real-time combat and navigation.

Navigation is a popular and well-studied problem found in many video games. Navigation in a virtual world often involves pathfinding. Common definitions as well as optimal solutions already exist for pathfinding problems. Examples include the A* search algorithm, which solves the single-pair shortest path problem[3], and Dijkstra's algorithm, which solves the single-source shortest path problem[4]. Although standard implementations can be found in developer frameworks and toolboxes, it is not unusual for developers to commit to their own implementation for environment-based customization.

A problem decomposition is often reflected in the AI design of a video game. For example, the AI in a RTS game may be divided into two main components. One component would deal with the problem of unit behavior and define behavior for units in different states such as being idle or following specific orders. This AI component could in turn include subcomponents for subproblems such as pathfinding. Defining autonomous unit behavior involves elements such as the design of unit response to a threat, an attack or the presence of an ally and is a problem that can be found in other games such as RPGs and FPSs. The other main component would deal with the problem of playing the RTS game to make it possible for a human player to face opponents without requiring other human players. This component could be organized in a number of modules to deal with the various tasks a player has to handle in a RTS game. A strategy manager can handle decisions such as when to attack and which units to produce. A production manager can handle construction tasks and assign workers to mining squads. A combat manager can designate assault lo-

[3]Find the least-cost path between a source node and a destination node in a graph.
[4]Find the least-cost path between a root node and all other nodes in a graph.

cations and assign military units to combat squads. Squad managers can handle tasks such as unit formations and coordination and target selection. These AI components can provide insight on the different conceptual problems they attempt to solve and their organization. Coordination between a group of units to select a common target or distribute targets among units and maneuver units can be included in the larger problem of real-time combat which is not exclusive to the RTS genre. On the other hand, production-related decisions could be taken based on generic data such as total air firepower or total ground armor, making it possible for the same conceptual policy to be used for any RTS game providing a conceptual view through which such data can be computed.

More conceptual problems could be derived from these AI components. The real-time combat problem is a complex recurring problem found in many different games and may incorporate problems such as role management, equipment tuning, positioning, target selection and ability planning. Some of these conceptual problems are briefly described below.

6.4.1 Role Management

Role management in combat is a recurring problem in video games. Role management deals with the distribution of responsibilities, such as damaging, tanking, healing and disabling, among a group of units or characters fighting together. Roles can be determined based on several factors, including unit type or character class, attributes and abilities, equipment and items, unit or character state and even player skill or preference. Without targeting any specific game, it is possible to define effective policies for role management using conceptual data only. The data can be static like a sorted list of role proficiencies indicating in order which roles a unit or character is inherently suited for. Such information can be used by the AI to assign roles in a group of units of different type in combat. Dynamic data can also be used to control roles in battle, like current hit points[5], passive damage reduction against a typed attack and available abilities of a unit. For instance, these can be used together to estimate the current tanking capacity for units of the same type. Naturally, the interpretation of these concepts varies from one game to another. Yet a conceptual policy remains effective in any case.

[5]The amount of damage a unit can withstand.

In a RPG, if a party is composed of a gladiator, an assassin and two clerics, the gladiator may assume the role of tank while a cleric assumes the role of healer and both the assassin and the other cleric assume the role of damage dealers. This distribution can vary significantly however. For example, the gladiator may be very well equipped and manage to assume the double role of tank and damage dealer, or conversely, the assassin may be dealing too much damage and become the target. If the tank dies, the healer may become the target[6] and assume both the role of tank and healer. In this case, the other cleric may switch to a healer role because the tanking cleric could get disabled by the enemy or simply because the lack of defense compared to a gladiator could cause the damage received to increase drastically, making two healers necessary to sustain enemy attacks. Roles can thus be attributed during combat depending on character affinities and on current state data too.

A similar reasoning process can be used for units in a RTS game. In a squad composed of knights, sorcerers and priests, knights will be assuming the role of tanks and fighting at the frontlines, while priests would be positioned behind them and followed by the sorcerers. Sorcerers would thus be launching spells from afar while knights prevent enemy units from getting to them and priests heal both injured knights and sorcerers. Even among knights, some might be more suited for tanking than others depending on their state. Heavily injured knights should not be tanking lest they not survive a powerful attack. They should instead move back and wait for priests to heal them while using any long range abilities they might have. Unit state includes not only attributes such as current hit points but also status effects and available abilities. Abilities can significantly impact the tanking capacity of a unit. Abilities could create a powerful shield around a unit, drastically increase the health regeneration of a unit or even render a unit completely invulnerable for a short amount of time. Likewise, healing and damage dealing capacities can vary depending on available abilities. The healing or damage dealing capacity of a unit may be severely reduced for some time if the unit possesses powerful but high-cooldown abilities which have been used recently. If the knights fall, either priests stay at the front and become the tanks or they move to the back and let the sorcerers tank depending on who of the two has the higher tanking capacity. Again, conceptual data can be used to generate operating rules to dynamically assign roles among units.

[6]Healing often increases the aggression level of a monster towards the healer, sometimes more than damaging the monster would.

Figure 6.9 shows a conceptual AI function which can be used to determine the primary tank in a group. The primary tank is usually the unit or character that engages the enemy and is more likely to initiate a battle. Figure 6.10 details a possible implementation of the scoring function. It estimates the total amount of damage a unit could withstand based on its hit points and the overall damage reduction factor it could benefit from that can be expected during the battle given the abilities of both sides. A damage reduction factor is just one way of conceptualizing defensive attributes such as armor or evasion. The `dmgred_abilities` function could create a list of available damage reduction abilities and average their effects. For each ability, the amount of reduction it contributes to the average can be estimated using the reduction factor it adds, the duration of the effect, the cooldown of the ability as well as its cast time. In the case of conflicting abilities (i.e., abilities whose effects override each other), the average reduction bonus could be estimated by spreading the abilities over the cooldown period of the one with the strongest effect. The `dmgamp_abilities` function could work with damage amplification abilities in a similar way. It could also take into account the unit's resistance to status effects.

Any form of distribution of responsibilities between units or characters fighting together can be considered role management. Role management does not assume any objective in particular. Depending on the goal of the group, different distribution strategies can be devised. The problem of role management in combat can therefore be described as follows. Given an objective, two or more units or characters and a set of roles, define a policy which dynamically assigns a number of roles to each unit or character during combat in a way which makes the completion of the objective more likely than it would be if units or characters each assumed all responsibilities individually. An example of objective is defeating an enemy unit. Roles do not have to include multiple responsibilities. They can be simple and represent specific responsibilities such as acting as a decoy or baiting the enemy.

6.4.2 Ability Planning

Another common problem in video games is ability planning. Units or characters may possess several abilities which can be used during combat. For instance, a wizard can have an ice needle spell which inflicts water dam-

```
1  void set_tank(UnitList& grp)
2  {
3    //Get a list of the enemies the group is fighting
4    UnitList enemies = get_nearby_threats(grp);
5
6    Unit* toughest = NULL;
7    double score = 0;
8
9    //For each unit in the group, estimate its toughness
       against the enemy
10   UnitList::iterator u;
11   for (u = grp.begin(); u != grp.end(); ++u)
12   {
13     double cs = score_tanking(*u, grp, enemies);
14     if (cs > score)
15     {
16       toughest = *u;
17       score = cs;
18     }
19   }
20
21   //Assign the role of tank to the toughest unit in the group
22   if (score > 0)
23     set_role(toughest, Role::tank);
24 }
```

Figure 6.9: Primary tank designation. This function could be used to determine which unit or character should engage the enemy.

```
1  double score_tanking(Unit* u, UnitList& grp, UnitList&
       enemies)
2  {
3    //Set the base score to the current unit hit points
4    double score = u->hitpts();
5
6    //Get primary damage type of enemy
7    DamageType dt = get_primary_dtype(enemies);
8
9    //Get current damage reduction of the unit
10   double dr = u->dmgred(dt);
11
12   //Factor in average reduction bonus from ally abilities
13   dr += dmgred_abilities(u, grp, dt);
14
15   //Factor in average amplification bonus from enemy
         abilities
16   dr -= dmgamp_abilities(u, enemies, dt);
17
18   if (dr >= 1.0)
19     return numeric_limits<double>::infinity();
20
21   //Estimate effective hit points
22   score *= 1.0/(1.0 - dr);
23
24   return score;
25 }
```

Figure 6.10: Tanking capacity estimation. This function could be used to evaluate how fit of a tank a unit or character is.

age on an enemy and slows it for a short duration, a mana shield spell which temporarily causes damage received to reduce mana points instead of health points and a dodge skill which can be used to perform a quick sidestep to evade an attack. Each of these abilities is likely to have a cost such as consuming a certain amount of mana points or ability points and a cooldown to limit its use. Units or characters thus need to plan abilities according to their objective to know when and in what order they should be used. As with role management, both static and dynamic data can serve in planning abilities. For example, if the enemy's class specializes in damage dealing, disabling abilities or protective abilities could take precedence over damaging abilities because its damage potential may be dangerously high. However, if the enemy's currently equipped weapon is known to be weak or its powerful abilities are known to be on cooldown, the use of protective abilities may be unnecessary.

Although abilities can be numerous, the number of ability types is often limited. These may include movement abilities, damaging abilities, protective abilities, curative abilities, enhancing abilities, weakening abilities and disabling abilities. Evidently, it is possible for an ability to belong to multiple categories. Abilities can be described in a generic way using various conceptual properties such as damage dealt, travel distance, conceptual attribute modification such as increasing hit points, effect duration, conceptual attribute cost such as action point cost, and cooldown duration. Abilities could also be linked together for chaining, such as using an ability to temporarily unlock another. Ability planning can then be achieved without considering the materialization of the abilities in a particular world. Even special abilities used under certain conditions, such as a boss attack that is executed when the hit points of the boss fall under a specific threshold, can be handled by conceptual policies. For instance, a powerful special ability of a boss monster can be unavailable until a condition is met. At that point, a policy that scans abilities and selects the most powerful one available would automatically result in the use of the special ability. If the ability must be used only once, a long cooldown can stop subsequent uses assuming cooldowns are reset if the boss exits combat.[7]

Abilities can be planned according to some goal. For example, the goal could be to maximize the amount of damage dealt over a long period of time, also called damage per second (DPS). Maximizing DPS involves de-

[7]This is to ensure that the boss can use the special ability again in a new battle in case its opponents are defeated or run away.

termining a rotation of the most powerful abilities with minimum downtime[8]. Conversely, the goal could be maximizing the amount of damage dealt over a short period of time, or dealing as much damage as possible in the shortest amount of time, also called burst damage. A burst plan is compared to a DPS plan in Figure 6.11. While the burst strategy (Plan A) obviously deals more damage at the beginning, it is clear that the DPS strategy (Plan B) results in more damage over the entire period. The DPS plan orders long-cooldown abilities in a way that avoids simultaneous cooldown resets because these powerful abilities need to be used as soon as they are ready to make the most out of them, which is not possible if multiple ones become ready at the same time. It also avoids the downtime between the two consecutive uses of the purple ability in Plan A by better interleaving its casts throughout the time period. This leads to a higher output overall. Note that the burst strategy eventually converges towards the the DPS strategy.

When combat is largely based on abilities, predicting and taking into account enemy abilities becomes crucial for effective ability planning. If a lethal enemy attack is predicted, a unit or character can use a protective ability such as casting a shield just before the attack is launched to nullify its effect. Alternatively, it can use a disabling ability to prevent the enemy from using the ability or interrupt it. Known enemy abilities could be evaluated in order to predict the enemy's likely course of action and plan abilities accordingly. Just like role management, ability planning can be dealt with by defining interesting conceptual policies for various frequently encountered objectives.

In Figure 6.12, the DPS of an ability chain is estimated by adding up the damage and duration of each ability in the chain. Ability chains can be useful to represent linked abilities, for example when an ability can only be activated after another. They can also be used to generate different versions of the same ability in cases where using an ability after a specific one alters the attributes of the ability. If activating ability Y after X increases the damage of Y by 100% or reduces its use time by 50%, X and Y may be interesting from a DPS standpoint in cases where they otherwise are not when considered individually. The attribute values of Y can then be different from their default ones depending on the chain in which they appear. Of course, this function only estimates a theoretical damage and is more useful to generate all-purpose ability rotations than to plan abilities against

[8]A state where all useful abilities are on cooldown.

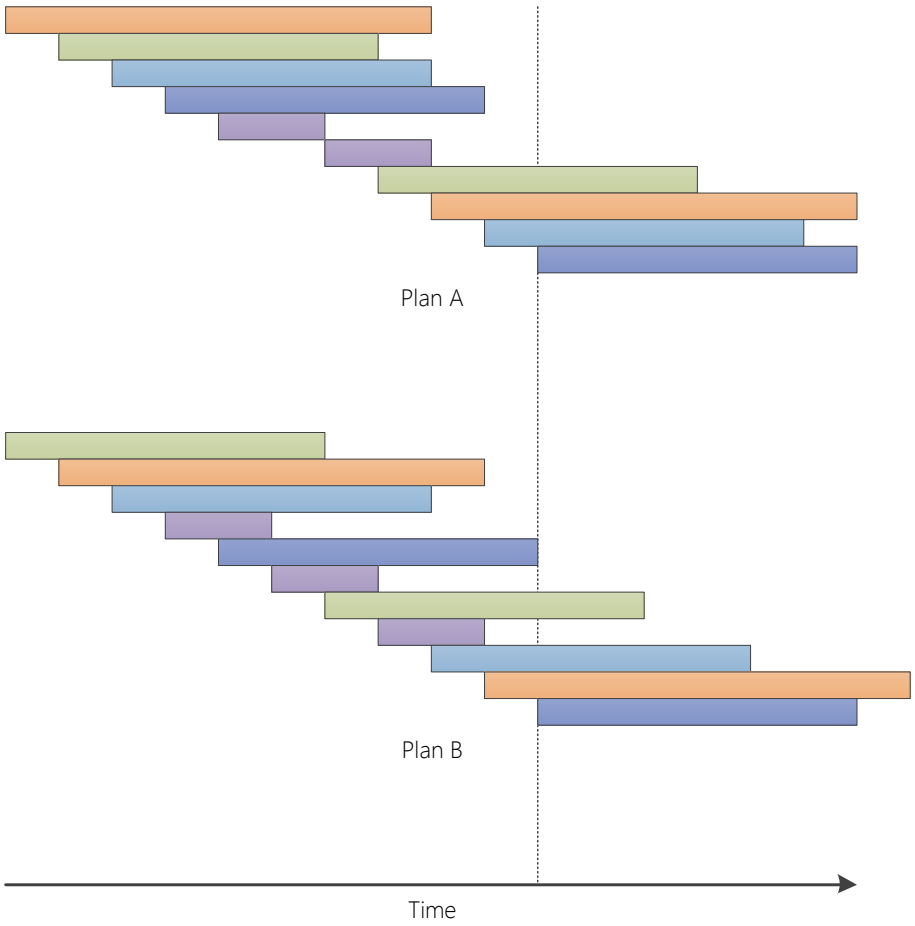


Figure 6.11: Ability planning using a burst strategy (Plan A) and a DPS strategy (Plan B). Rectangles represent cooldown periods of abilities. Each color corresponds to a different ability. Cast time is represented by a delay between the use of two consecutive abilities.

```
1  double calc_dps(AbilityChain& ac)
2  {
3    double dmg = 0;
4    double dur = 0;
5
6    AbilityChain::iterator a;
7
8    //Add up the damage and duration of each ability in the
       chain
9    for (a = ac.begin(); a != ac.end(); ++a)
10   {
11     dmg += (*a)->damage();
12     dur += (*a)->usetime();
13   }
14
15   //DPS = total damage / total execution time
16   if (zero(dmg))
17     return 0;
18
19   if (zero(dur))
20     return numeric_limits<double>::infinity();
21
22   return dmg/dur;
23 }
```

Figure 6.12: DPS estimation of an ability chain. This function can be useful for creating optimal DPS plans.

a specific enemy. DPS can be more accurately estimated by factoring in the attributes and status effects of both the user and the target. If the target is very resistant against a particular type of damage, powerful abilities of this type may be outranked by less powerful ones dealing a different type of damage. The attributes or the status effects of the user can also affect the effectiveness of different abilities in different ways. One ability may have a high base damage value but gain nothing from the strength of the user, while another ability may have a low base damage but greatly benefit from the strength attribute and end up out-damaging the former. Use time can also vary depending on the user's attributes. Note that the use time corresponds to the total time during which the user is busy using an ability and cannot use another. Some abilities may involve both a cast time (i.e., a phase where the user channels energy without the ability being activated) and an activation duration (i.e., the time between the activation of the ability and the time the user goes back to an idle state). This function does not calculate other costs either. If abilities cost ability points or mana points to use in combat, these additional costs can be estimated for the chain together with the time cost since they usually cannot be ignored during a battle.

The concept of abilities is used in several genres. They usually correspond to actions that can be taken in addition to base actions, such as moving, at a cost. Given an objective and a set of abilities, the problem of ability planning is to produce a sequence of abilities which leads to the completion of the objective. Note that the set of abilities does not have to belong to a single entity. Like in role management, the objective can be fairly abstract and common, such as running away, disabling an enemy or protecting an ally.

6.4.3 Positioning

A frequently encountered problem in video games is positioning in the context of combat. Maneuverable units or characters have to continuously adjust their position according to their role or plan. A character whose role is to defend other characters will move to a position from which it can cover most of its allies from enemy attacks. An archer will attempt to stay outside the range of its enemies but close enough to reach them. A warrior with strong melee AoE attacks must move to the center of a group of enemies so as to hit as many of them as possible with each attack. An assassin may need to stick to the back of an enemy in order to maximize its damage. A

specialized unit with poor defense could remain behind its allies in order to easily retreat in case it becomes targeted. This kind of behavior results from conceptual reasoning and needs not be specific to any one game.

While navigation deals with the problem of traveling from one position to another, positioning is more concerned with finding new positions to move to. New positions can be explicitly designated for a unit or character or they could be implicitly selected by adjusting movement forces. For example, a unit may need to step outside the range of an enemy tower by moving to a specific position, or it could avoid bumping into a wall while chasing another unit by adding a force that is normal to the direction of the wall to its steering forces instead of selecting a position to move to. When positions are explicitly calculated, navigation may be involved to reach target positions. This can lead to a dependency between solutions to positioning problems and solutions to navigation problems.

Figure 6.13 shows a function which moves a unit out of the attack range of a group of enemies. For each enemy, it creates a circular area based on the enemy's attack range and centered on its predicted position. The latter is simply calculated by adding the enemy's current velocity to its position. This function ignores enemies that are faster than the unit because even if the unit is currently outside their range, it would eventually fall and remain within their reach. This could be delayed however. A list of immediate threats is thus created and used to compute a force to direct the unit away from the center of threats as quickly as possible. Note that this code does not differentiate between threats. It can be improved by weighting each position in the calculation of the center according to an estimation of the danger the threat represents. The more dangerous the threat, the larger the weight can be. This would cause the unit to avoid pressing threats with higher priority. This function could be used for kiting.

The code in Figure 6.14 shows how a straight line projectile can be dodged by a unit. A ray is created from the current position of the projectile and used to determine whether a collision with the unit is imminent. If this is the case, the unit is instructed to move sideways to avoid collision. The bounding radius of the projectile as well as that of the unit are used to determine the distance which must be traveled. The side on which the unit moves depends on its relative position vis-à-vis the projectile course. Of course, this function does not take into account the speed of the projectile and could therefore be better. If the projectile is slow compared to the unit, the movement could be delayed. On the other hand, if it is too fast,

```
1  void stay_safe(Unit* u, UnitList* enemies)
2  {
3    UnitList threats;
4
5    UnitList::iterator e;
6
7    //Iterate on enemies to detect immediate threats
8    for (e = enemies.begin(); e != enemies.end(); ++e)
9    {
10     //Ignore enemies that can't be outrun
11     if (u->maxspeed() > (*e)->maxspeed() &&
12       distance(u->position(), (*e)->position() + (*e)->
             velocity()) <= (*e)->maxrange())
13       threats.add(*e);
14   }
15
16   //Get the center of the threats
17   Vector c = center(threats);
18
19   //If the unit is located at the center, drop one of the
       threats
20   if (c == u->position())
21     c = center(remove_weakest(threats));
22
23   //Create a force that pulls the unit away from the center
24   Vector dir = u->position() - c;
25
26   //Add a steering force of maximum magnitude
27   u->addforce(dir*u->maxforce()/dir.norm());
28 }
```

Figure 6.13: Avoiding enemy attacks by staying out of range. This function can be used for kiting.

dodging may be impossible and the unit would not need to waste any time trying to do that.

Clearly, both code examples presented above follow a purely conceptual reasoning and could apply to a multitude of video games. They operate solely on conceptual objects and properties such as units, positions, velocities, steering forces and distances. Creating a comprehensive collection of general policies to deal with positioning problems can be time-consuming, making it unlikely to be profitable for a video game developer. When the solutions are conceptual and target all video games however, they may become profitable, providing incentive for AI developers to undertake the challenge.

Like role management and ability planning, positioning exists within the context of an objective. It is possible to design conceptual yet effective positioning policies for generic objectives such as maximizing damage dealt or minimizing damage received. Given an objective, the problem of positioning is to control the movement of a maneuverable entity in a way which serves the completion of the objective. Note that objectives could automatically be derived from roles. Depending on the space and the type of movement, different positioning problems could be considered. For example, it may be more interesting to consider 2D positioning and 3D positioning separately than to consider a single multi-dimensional positioning problem.

6.5 Integrating Conceptual AI in Video Games

Since conceptual AI is designed independently from games, an integration mechanism is necessary for it to be used by game developers. Game developers must be able to choose and connect AI solutions to a game. This is achieved by registering AI controllers with conceptual objects. To assign control, partial or complete, of an entity in the game to a particular AI, the corresponding controller must be instantiated and registered with the projection of the entity in CDS. The AI then controls the conceptual entity, effectively controlling the entity in the game. For example, a game developer could use two AI solutions for a racing game, one for controlling computer opponents on the tracks and another for dynamically adjusting the difficulty of a race to the player's performance. Each time a computer opponent is added to the race, a new instance of the driving AI is created and registered with its conceptual projection. As for the difficulty AI, it

```
1  void dodge_projectile(Unit* u, Projectile* p)
2  {
3    //Create a ray for the projectile course
4    Ray r(p->position(), p->velocity());
5
6    //Get a list of objects intersecting the ray
7    ObjectList is = intersection(r, p->radius());
8
9    //Only dodge if u is the first object to intersect the ray
10   if (is.front() == u)
11   {
12     //Is u exactly on the projectile course?
13     if (r.passthru(u->position()))
14     {
15       //Move perpendicularly by a distance equal to the sum
             of bounding radiuses
16       u->move(u->position() + r.normal()*(p->radius() + u->
             radius()));
17       return;
18     }
19
20     //Project the unit position on the projectile course
21     Vector pr = r->project(u->position());
22
23     //Get a normal to the projectile course with a norm equal
           to the distance between the unit position and its
          projection
24     Vector mv = u->position() - pr;
25
26     //Rescale it to the width of the intersection
27     mv *= (p->radius() - (mv.norm() - u->radius()))/mv.norm()
          ;
28
29     //Follow the normal to avoid collision
30     u->move(u->position() + mv);
31   }
32 }
```

Figure 6.14: Dodging a straight line projectile. This function assumes that the projectile is not penetrating.

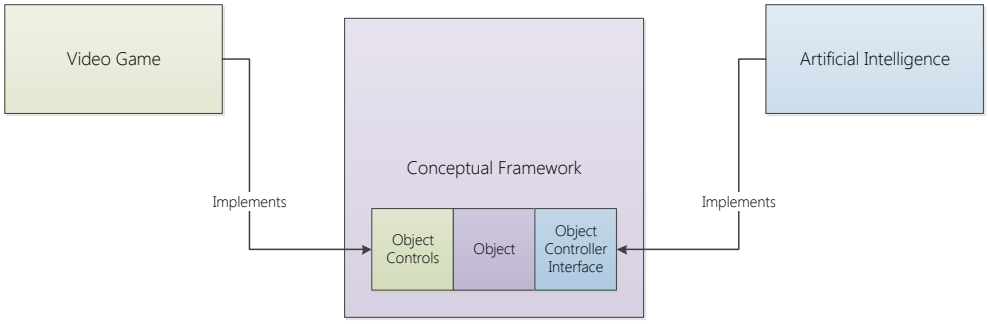


Figure 6.15: Conceptual controls and controller interface. Both are defined by the CF developers. Conceptual controls have to be implemented by game developers while the controller interface has to be implemented by AI developers.

can be created at the beginning of the race and registered with a real-time player performance evaluation object.

For each controllable conceptual object defined by the CF developers, a controller interface is defined together with it. This interface describes functions the AI must implement in order to be able to properly assume control over the conceptual object. These are not to be confused with the conceptual controls, also defined by the CF developers, which the AI can use to control the conceptual object and which are implemented by the game developers. Figure 6.15 illustrates the distinction.

It is possible for multiple controllers to share control of the same object. For example, a NPC could be controlled by different AI solutions depending on its state. It may have a sophisticated combat AI which kicks in only when the NPC enters a combat state and otherwise remains on standby, while a different AI is used when the NPC is in an idle state to make it roam, wander or rest. Multiple controllers however may lead to conflict in cases with overlapping control. One way to resolve conflicts is for AI controllers to have a table indicating a priority level for each conceptual control. Conceptual control calls would then be issued by controllers with their respective priorities and queued for arbitration. Of course, when multiple AI controllers are integrated into a complete solution, this issue can be handled by the author of the solution in whatever way they may choose and only the complete controller can be required to provide a priority table for conceptual controls.

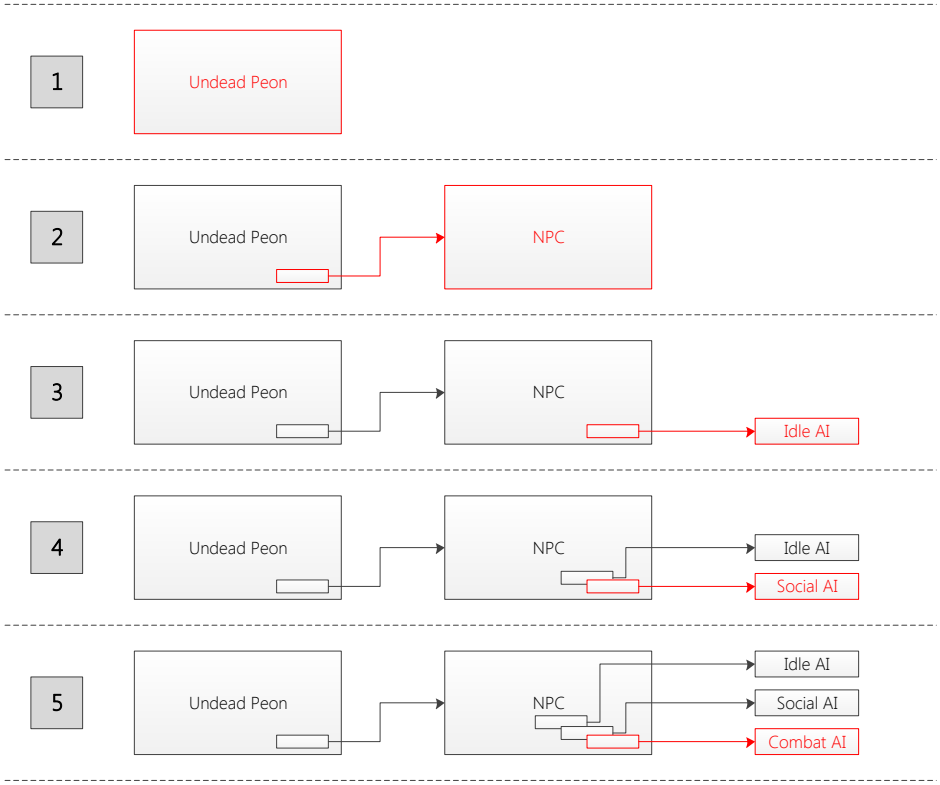


Figure 6.16: Registering multiple controllers with a conceptual object. Depending on its state, the Undead Peon is controlled by one of the three AI solutions.

Figure 6.16 shows how multiple controllers can be registered with a conceptual object. First, an object in the game, an Undead Peon, is created. Following this, its projection in CDS, a NPC, is created and linked to the Undead Peon. Finally, several independent AI controllers, one for generating idle behavior when the Undead Peon is idle, another for generating social behavior when the Undead Peon is around other Undead Peons and other types of NPCs and another for generating combat behavior when the Undead Peon is facing enemies, are created and registered with the NPC in CDS. In this case, there is no overlap in the control of the NPC by the different AI solutions. Using this registration mechanism, an AI controller can also verify that its dependencies are running and access them via the conceptual object.

Examples of functions found in controller interfaces are an update function and event handlers. An update function is used to update the internal state of the AI and can be called every game cycle or at an arbitrarily set update rate. This function is illustrated in Figure 6.17. Note how the NPC in CDS has no internal state update cycle. This is because there is no dynamic in the CDS. Objects in CDS are projections of game objects and are only modified as a result of a change in game objects. Event handlers are used to notify AI controllers of game events, such as a unit being killed by another. When an event occurs in the game, a conceptual projection is fired at the projection of the involved game object. The events that can involve a conceptual object are determined by the CF developers and used to create the controller interface. An AI controller does not necessarily need to handle all events. This is obvious for partial controllers. Therefore, it is possible for AI controllers to ignore some events. Event handlers are illustrated in Figure 6.18. Other examples are functions for suspending and resuming the controller.

When game developers link AI solutions to their games, they can either link them statically at build time or load them dynamically at runtime. Loading AI at runtime makes it easier to test different AI solutions and can also allow players to hook their own AI to the game. Typically, the AI would be running within the video game process, though it can be interesting to consider separating their execution. Deploying the AI in a separate process means it can run on a different machine. The latter could be optimized for AI processing or it could even be on the Internet, making it possible for AI developers to offer AI as a service. A multi-process design can easily be imagined, as shown in Figure 6.19.

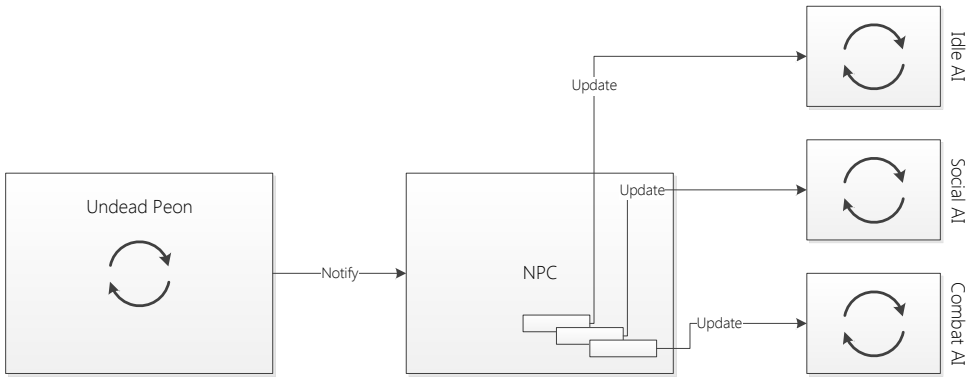


Figure 6.17: Updating the internal state of AI controllers when game objects update theirs. Note how objects in CDS do not have an update cycle.

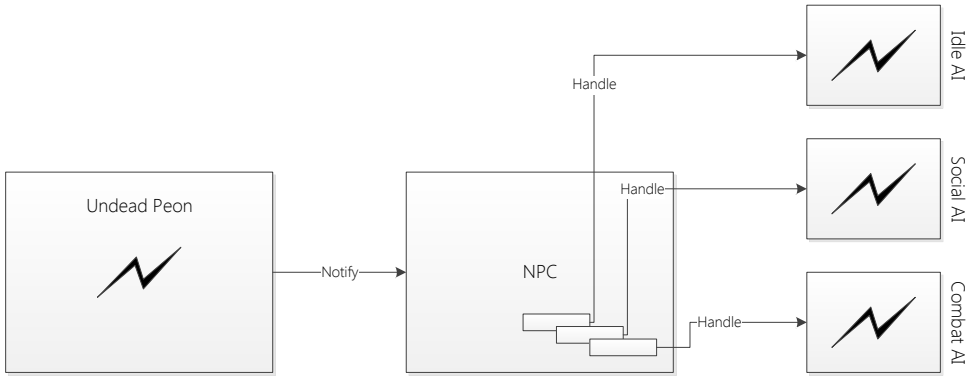


Figure 6.18: Event handling by AI controllers. Game events are projected into CDS before being pushed to AI controllers.

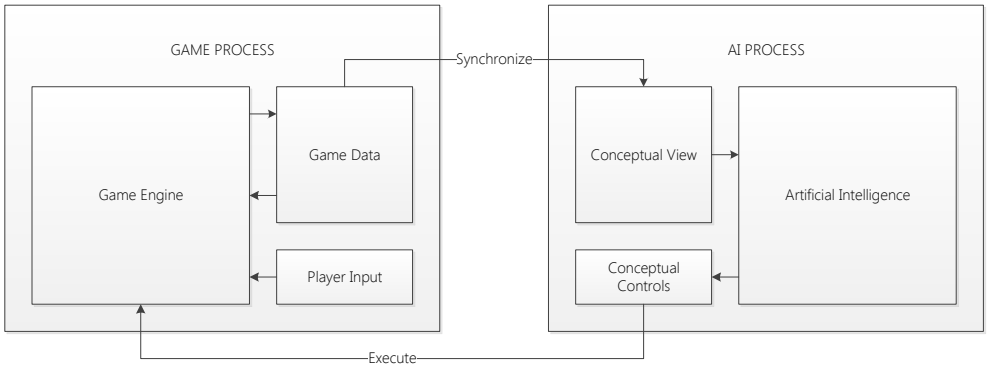


Figure 6.19: Running AI in a separate process. Synchronizing a conceptual view with game data requires an inter-process communication mechanism such as sockets or remote procedure call (RPC) systems. The mechanism is also required for using conceptual controls.

Chapter 7

Applications

Contents

Summary

This chapter includes some applications of the development model presented in the previous chapter. It contains two sections. The first section describes a design experiment conducted on an open-source video game in order to concretize the idea of introducing a conceptual layer between the game and the

AI. The second section then describes a second experiment which makes use of the resulting code base to integrate a simple conceptual AI in two different games.

7.1 Graven: A Design Experiment

7.1.1 Description

The Graven experiment consists in rebuilding an open-source video game called *Raven* according to the design presented in the previous chapter.[1] Namely, the AI is separated from the game and a conceptual layer is added in between. The AI is adapted to interact with the conceptual layer rather than directly with the game and the latter is modified to maintain a conceptual view in memory and use the conceptual AI. Albeit basic, Raven involves enough concepts to use as a decent specimen for conducting experiments relating to the deployment and use of a CF. The goal of the experiment is twofold:

1. Concretize the design architecture as well as key processes in a working example.

2. Obtain a code base to use as a limited prototype for testing conceptual AI in multiple games.

Note that the Graven experiment does not directly aim at demonstrating the efficiency of conceptual AI.

7.1.2 Raven

Raven is an open-source game written by Mat Buckland. A detailed presentation of the game as well as the code can be found in *Programming Game AI by Example* [52] where it is used to demonstrate a number of AI techniques such as path planning, goal-driven behavior and fuzzy logic. It is a single-player, top-down 2D shooter featuring a deathmatch mode.

Maps are made of walls and doors and define spawn locations for players as well as items. When players die, they randomly respawn at one of the fixed spawn locations. Items also respawn at fixed time intervals after they are picked up. There are two types of items in Raven, weapons and health packs. Three weapons can be picked up. These are the Shotgun, the Rocket Launcher and the Railgun. A fourth weapon, the Blaster, is automatically included in every player's inventory at spawn time.

[1] See Figure 6.2.

Each weapon is characterized by a unique set of features such as a firing rate and the maximum quantity of ammunition that can be carried for it. The Blaster is a basic projectile weapon with unlimited ammo. The Shotgun is a hitscan weapon which fires several pellets that spread out. The Rocket Launcher is a projectile weapon which fires rockets that deal AoE damage when they explode either on impact or after traveling a certain distance. The Railgun is a hitscan weapon which fires penetrating slugs that are only stopped by walls. Players can pick up weapons they already have. In that case, only the additional ammo is added to their inventory.

Initially, a default number of bots are spawned depending on the map. Bots can then be added to and removed from the game. The player can possess one of the existing bots to participate in a match. The left and right mouse buttons can be used to fire and move respectively, while numbers on the keyboard can be used to switch weapons. Despite their adorable look, these bots will compute the shortest paths to navigate the map, avoid walls, pick up ammo and health when needed, estimate their opponent's position to aim projectiles properly, use the most appropriate weapon depending on the situation, remember where they last saw or heard an opponent, chase or run away from an opponent, perform evasive maneuvers and, of course, kill. A preview of the game is shown in Figures 7.1 and 7.2.

The world in a Raven game is essentially composed of a map, bots and projectiles. The map is composed of walls and includes a navigation graph used for pathfinding as well as triggers. Triggers are used to define item pick up locations as well as temporary sound sources. This composition is illustrated in Figure 7.3. The bot AI is primarily made of 6 interdependent modules, as shown in Figure 7.4. The brain module handles abstract goals and takes decisions such as attacking the current target or retrieving an item. The steering module manages steering forces resulting from multiple simultaneous behaviors such as avoiding a wall while seeking an enemy. The path planner module handles navigation requests by computing paths between nodes in the navigation graph. The sensory memory module keeps track of all the information the bot senses and remembers, such as visible enemies, hidden enemies and gunshot sound locations. The target selection module is used to select a target among a group of enemies. Finally, the weapon system module handles aiming and shooting and also includes per-weapon specific modules to evaluate the desirability of each weapon given the current situation.

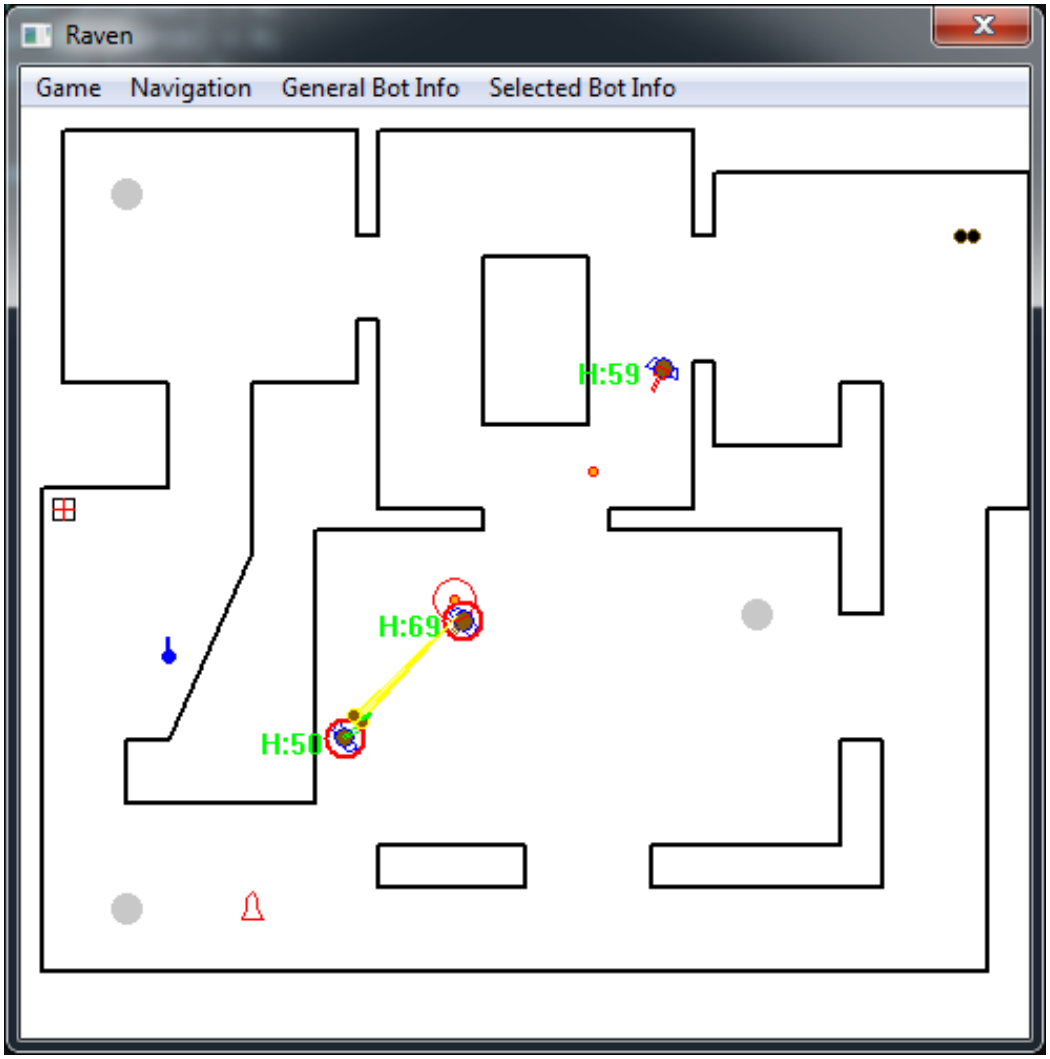


Figure 7.1: Screenshot taken from the Raven game. Player spawn locations are drawn in gray. On the top right corner is a Shotgun in black. At the bottom is a Rocket Launcher. At the left are a Railgun and a health pack. Each bot has its current hit points drawn next to it.

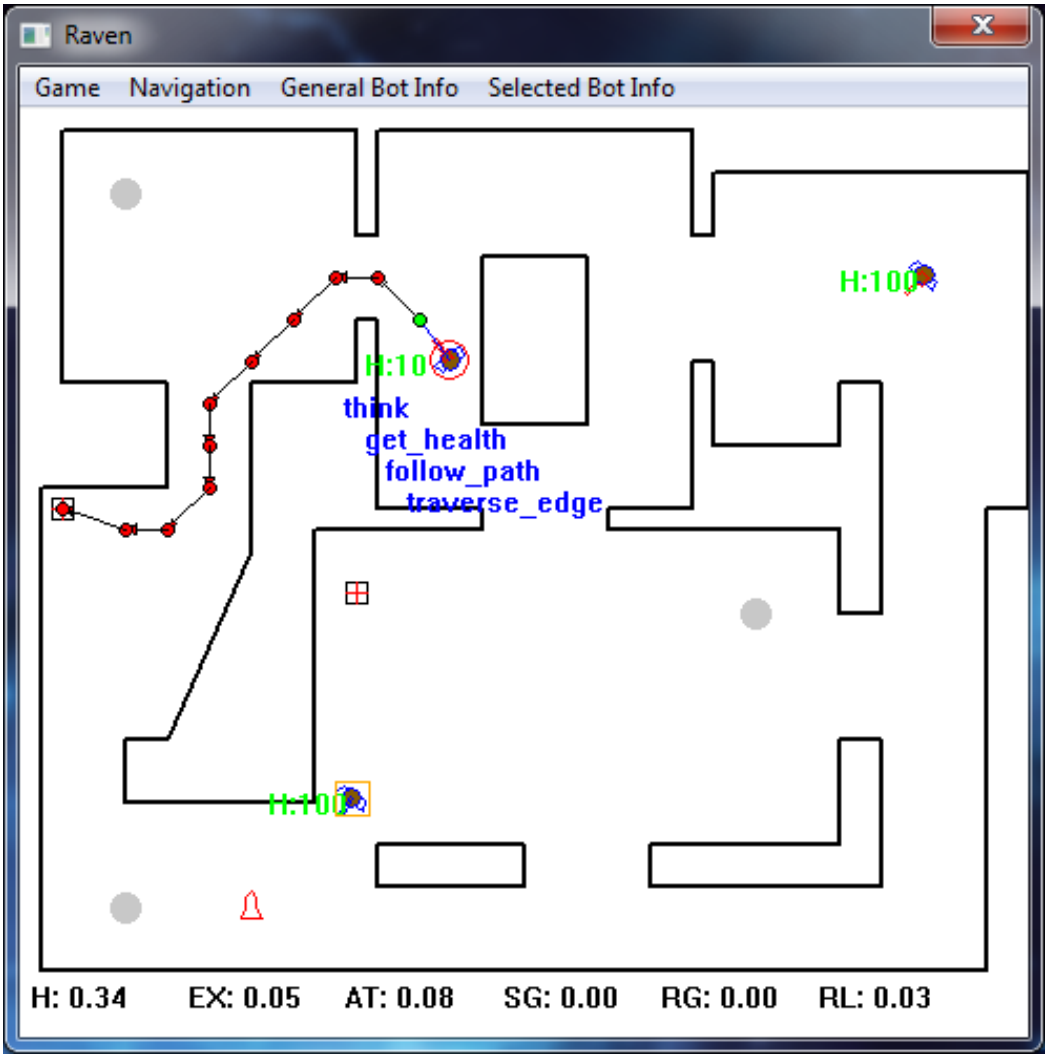


Figure 7.2: The AI information of a selected bot in Raven. Are shown are the goal stack, the path the bot is currently following, the current target of the bot (shown as a colored square around another bot) as well as a number of numerical desirabilities which indicate how important some of the actions the bot is thinking about are. From left to right, these are getting health, exploring, attacking the current target, getting a Shotgun, getting a Railgun and getting a Rocket Launcher.

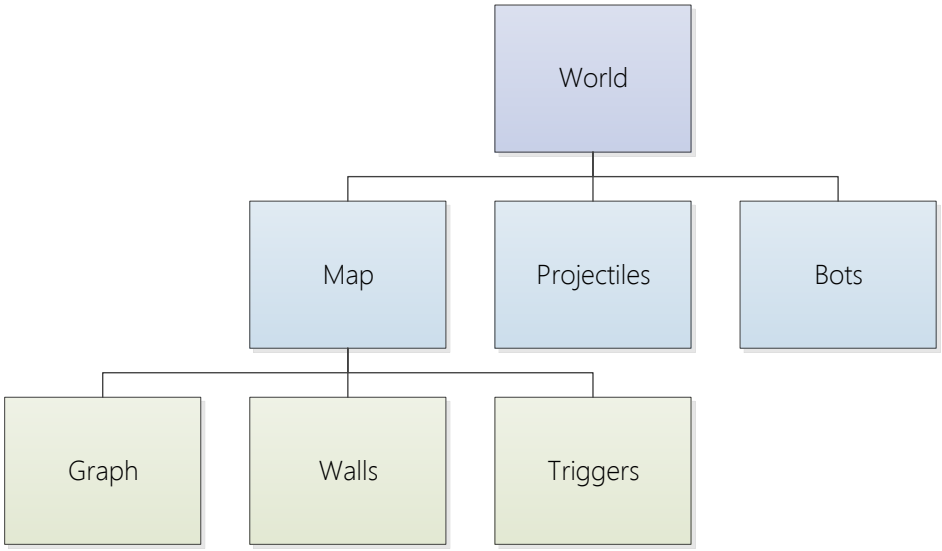


Figure 7.3: Overview of the Raven world composition.

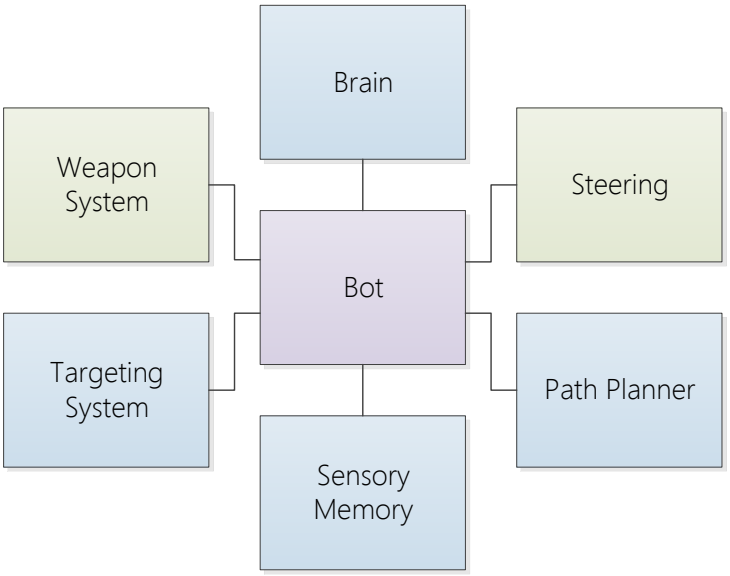


Figure 7.4: Overview of the Raven bot AI structure. Concrete actions such as firing a weapon or applying a steering force are taken by the green modules.

7.1.3 Overview of the Code Structure

The code structure in Graven comprises five categories of components:

1. the Raven classes,
2. the conceptual view classes,
3. the conceptual AI classes,
4. the conceptual controls,
5. and the Raven control classes.

The Raven classes are the game classes and an adaptation of the original code where all the AI components are removed and code to synchronize the conceptual view with the game state is added. The second category is a library of objects representing concepts corresponding to the Raven objects. The conceptual AI classes are a modification of the original AI code in which the AI is made to interact with the conceptual layer rather than the game. The fourth category includes a set of conceptual controls used by the conceptual AI to control bots. Finally, the Raven control classes implement these conceptual controls. Note that from a design perspective, the conceptual controls belong in the conceptual layer classes and their implementation in the game classes. They are separated in the code structure for the purpose of clarity.

7.1.4 Conceptualization

Raven is primarily composed of generic elements, as can be seen in Figure 7.3. A 2-dimensional world, projectiles or walls are concepts commonly found in many video games. The added conceptual layer thus largely consists of clones of the objects in Raven. Unlike their Raven counterpart however, conceptual objects are entirely static and do not update their own state. Instead, their state is only modified as a result of a modification on the game side. This is illustrated in Figure 7.5.

In Figure 7.5, the `Raven_Weapon` class declares a `ShootAt` function which is used to fire the weapon and which is implemented by each of the four Raven weapon classes. It also defines an `IncrementRounds` function which is used to update the number of rounds left for the weapon when a bot picks up additional ammo. In the corresponding `CptWeapon` class, the `ShootAt` function has been removed, and the `IncrementRounds` function

```
1  class Raven_Weapon
2  {
3    ...
4
5    //this discharges a projectile from the weapon at the given
         target position (provided the weapon is ready to be
        discharged... every weapon has its own rate of fire)
6    virtual void  ShootAt(Vector2D pos) = 0;
7
8    void IncrementRounds(int num)
9    {
10     m_iNumRoundsLeft+=num;
11     Clamp(m_iNumRoundsLeft, 0, m_iMaxRoundsCarried);
12
13     //Synchronize rounds in CDS
14     GetCptWeapon()->SetNumRoundsLeft(m_iNumRoundsLeft);
15   }
16
17   ...
18 };
19
20 class CptWeapon
21 {
22   ...
23
24   //Removed
25   //virtual void  ShootAt(Vector2D pos) = 0;
26
27   void SetNumRoundsLeft(int n)
28   {
29     m_iNumRoundsLeft = n;
30   }
31
32   ...
33 };
```

Figure 7.5: Modifications in the conceptual copy of a Raven class. No game behavior is defined in CDS, which hosts nothing more than a projection of the game state.

```
1  //Applies a steering force to a bot
2  void ApplyForce(int agent_id, Vector2D force);
3
4  //Rotates the facing direction of a bot
5  void RotateToward(int agent_id, Vector2D position);
6
7  //Switches the equipped weapon of a bot
8  void EquipWeapon(int agent_id, int weapon_type);
9
10 //Fires the equipped weapon of a bot
11 void ShootAt(int agent_id, Vector2D position);
```

Figure 7.6: Conceptual controls used by the AI in Graven. In order, these are used by the conceptual AI to send commands to apply a steering force to a bot, to rotate a bot toward a certain position, to switch the currently equipped weapon of a bot and to fire a bot's weapon at a given position. Together, these conceptual controls are sufficient to replicate the intricate behavior from the original code.

has been replaced with a `SetNumRoundsLeft` function which can be used by the game to update the number of rounds left for the weapon in CDS. The synchronization process is detailed in a subsequent section.

Four conceptual controls have been defined. These are used by the conceptual AI to control the bots in Graven and are shown in Figure 7.6. The `ApplyForce` function can be used to apply a steering force to a bot and control its movement. The `RotateToward` function can be used to rotate a bot and control the direction of its field of view. The `EquipWeapon` function can be used to switch a bot's weapon to any of those it holds in its inventory. Lastly, The `ShootAt` function can be used to fire a bot's equipped weapon. These conceptual controls can be applied to a `CptMvAgent2D` object, the conceptual projection of a Raven bot in CDS. They are implemented game-side.

On the AI side, a `CptMvAgent2D` represents a controllable object and therefore the class comes with a controller interface. For an AI to be recognized as a valid controller by the game, it has to implement this interface. The interface is shown in Figure 7.7. It includes six functions. The `KilledBy_Handler` function is called whenever a bot is killed by an opponent and allows the controller to retrieve information about the killer. The `HeardSound_Handler` function is called when a bot hears a gunshot and can be used by the AI to find the origin of the sound. The `BotRemoved_Handler` function is called when a player removes a bot from the game via the main

```
1  class CptMvAgent2D_Controller
2  {
3  protected:
4    CptMvAgent2D* m_pOwner;
5
6  public:
7    virtual ~CptMvAgent2D_Controller() {}
8
9    //Called when a bot is killed by an opponent
10   virtual void KilledBy_Handler(CptMvAgent2D* attacker) = 0;
11
12   //Called when a bot hears a gunshot
13   virtual void HeardSound_Handler(CptMvAgent2D* source) = 0;
14
15   //Called when the player removes a bot from the game
16   virtual void BotRemoved_Handler(CptMvAgent2D* bot) = 0;
17
18   //Called when the player takes control of a bot
19   virtual void Suspend() = 0;
20
21   //Called when the player hands back control to a bot
22   virtual void Resume() = 0;
23
24   //Called every game update cycle
25   virtual void Update() = 0;
26 };
```

Figure 7.7: The controller interface of a `CptMvAgent2D`. These functions are used by the `CptMvAgent2D` class to relay events to the AI.

menu and can be used to notify other bots that the removed bot no longer exists. The `Suspend` and `Resume` functions serve to temporarily disable the controller when a bot is possessed by the player. The last `Update` function is used to allow the AI to update its state every game cycle.

Functionally, the AI in Graven is the same as the original Raven AI. It slightly differs in its structure however. In Raven, the `Raven_WeaponSystem` class serves as a weapon inventory and handles weapon switching and also aiming and shooting, whereas weapon selection and aiming and shooting are separated in Graven. The central AI module through which other AI modules interact is the `CptBot` class. It resembles the original `Raven_Bot` class, though there are two significant differences. One, it interacts solely with the conceptual layer instead of the game. Two, it does not host any game state data such as current position and velocity, which is found in the

```
1  void Raven_Bot::Spawn(Vector2D pos)
2  {
3    ...
4
5    //Direct modification: sync!
6    m_iHealth = m_iMaxHealth;
7    cpt->SetHealth(m_iHealth);
8
9    //Function call: don't sync, already done in function
       definition!
10   SetAlive();
11
12   //Different class: don't sync, WeaponSystem has its own
       sync code!
13   m_pWeaponSys->Initialize();
14
15   ...
16 }
```

Figure 7.8: Conceptual data synchronization in Graven. Synchronization code in a class is added whenever its members are modified directly.

`CptMvAgent2D` it controls. The AI state is thus clearly separated from the game state.

7.1.5 Creating a Conceptual View

The following process is used to synchronize the conceptual view with the Raven game state. For each class representing an object in the Raven game which has some projection in the CDS, a pointer to an object of the corresponding conceptual class is added to its data members. Then, following each statement that directly[2] modifies a member of the class, a second operation is added to update the conceptual object accordingly. The conceptual object is created at the beginning of the class constructor and destroyed at the end of its destructor. By confining the synchronization code of an object to its class, its synchronization is done only once and never mixed with that of other objects. This idea is illustrated in Figure 7.8.

One problem with this technique is that it cannot be used directly with virtual classes because, even if they have corresponding conceptual classes, they do not represent actual objects with an independent projection in the CDS. The projection of a virtual class only exists as a part of the projection

[2]Without calling a function.

of a concrete class (i.e., a conceptual concrete class) and can only be accessed through this conceptual concrete class. A remedy for this problem is using a pure virtual getter implemented by its concrete subclasses, as shown in Figure 7.9. This involves another problem however, since virtual functions cannot be called in the constructor.[3] This is solved by moving the synchronization code in the constructor into an additional `sync` function for each class. This applies even to concrete classes. The `sync` function in a subclass always starts by calling the `sync` function of its superclass, ensuring that the synchronization code of an object remains confined within its class definition. A call to the `sync` function is added immediately after the creation of a conceptual object in the constructor of a concrete class, effectively executing the synchronization code of all its superclass constructors.

In order to properly synchronize certain template classes in Raven, it is necessary to use additional data type parameters to accommodate conceptual data types associated with the base parameters. For example, the `Trigger_Respawning` template class in Raven takes an entity type parameter which determines the type of game object that can activate the trigger. The class `Trigger_WeaponGiver` which extends `Trigger_Respawning` uses a `Raven_Bot` as parameter. However, its conceptual projection, a `Cpt TriggerWeaponGiver`, requires a `CptMvAgent2D` parameter. For this reason, the `Trigger_Respawning` class takes two parameters in Graven, one for the game data type and one for the corresponding conceptual data type.

7.1.6 Registering the Conceptual AI

The `CptBot` class implements the `CptMvAgent2D_Controller` interface and provides the AI functionality of the original Raven. The `CptMvAgent2D` class defines an `AddController` function which can be used by the game to register `CptMvAgent2D_Controller` objects with its instances. All registered controllers are updated and notified through the `CptMvAgent2D` instance. This is shown in Figure 7.10.

A `DMController` module can be used to instantiate and register `CptBot` objects without exposing the class to the game. Figure 7.11 shows how a controller is registered in the constructor of the `Raven_Bot` class. After creating and synchronizing a corresponding `CptMvAgent2D`, the `Register DMController` function is used to relegate the control of the bot to the

[3] In C++, the virtual table of an object is only initialized after its construction is complete.

```
1  class MovingEntity : public BaseGameEntity
2  {
3    ...
4
5    //Virtual accessor - Retrieves the conceptual projection of
         this entity
6    virtual CptMvEntity2D* GetCptMvEntity2D() const = 0;
7
8    ...
9
10   void SetVelocity(const Vector2D& NewVel)
11   {
12     m_vVelocity = NewVel;
13
14     //Velocity changed, update conceptual data
15     GetCptMvEntity2D()->SetVelocity(m_vVelocity);
16   }
17
18   ...
19 }
20
21 class Raven_Bot : public MovingEntity
22 {
23 protected:
24
25   //The conceptual projection
26   CptMvAgent2D* cpt;
27
28   ...
29
30 public:
31
32   //Returns the entire conceptual projection of this bot
33   CptMvAgent2D* GetCptMvAgent2D() const { return cpt; }
34
35   //Returns the conceptual projection of the MovingEntity
       part of this bot
36   CptMvEntity2D* GetCptMvEntity2D() const { return cpt; }
37
38   //Returns the conceptual projection of the BaseGameEntity
       part of this bot
39   CptEntity2D* GetCptEntity2D() const { return cpt; }
40
41   ...
42 }
```

Figure 7.9: Synchronization with virtual classes. The virtual class `MovingEntity` uses a pure virtual getter implemented by its concrete subclass `Raven_Bot` for its synchronization code.

```
1  class CptMvAgent2D : public CptMvEntity2D
2  {
3  private:
4
5    //List of registered controllers
6    std::list<CptMvAgent2D_Controller*> controllers;
7
8    ...
9
10 public:
11
12   //Registers a new controller
13   void AddController(CptMvAgent2D_Controller* c)
14   {
15     controllers.push_back(c);
16   }
17
18   //Notifies controllers that a bot has been removed from the
         game
19   void BotRemoved(CptMvAgent2D* bot)
20   {
21     std::list<CptMvAgent2D_Controller*>::iterator it;
22     for (it = controllers.begin(); it != controllers.end();
          ++it)
23     {
24       (*it)->BotRemoved_Handler(bot);
25     }
26   }
27
28   ...
29 }
```

Figure 7.10: Controller management in the `CptMvAgent2D` class.

```
1  Raven_Bot::Raven_Bot(Raven_Game* world,Vector2D pos) : ...
2  {
3    //Create the conceptual projection
4    cpt = new CptMvAgent2D(world->GetCptWorld2D());
5
6    //Synchronize initialization
7    sync();
8
9    ...
10
11   //Instantiate and register a DMController
12   RegisterDMController(cpt);
13 }
```

Figure 7.11: Conceptual AI registration in Graven. The `Register DMController` function is defined in the `DMController` module and is used to instantiate the `CptBot` class.

conceptual AI.

7.2 Using the Graven Targeting AI in StarCraft

7.2.1 Description

Following the Graven experiment which produced a limited CF prototype as well as a number of conceptual AI solutions, a second experiment was conducted to assess the work involved in using a simple conceptual AI solution in different games. Two games were used in this experiment, *Raven* and *StarCraft: Brood War* (BW). Albeit very different, these two games share a common conceptual problem, namely target selection. Target selection in combat deals with deciding which enemy should be targeted in the presence of multiple ones. In Raven, a bot may face multiple opponents at the same time. Likewise in BW, a unit may face multiple enemy units on the battlefield. This experiment consists in using the same solution to this targeting problem in both Raven and BW, resulting in having the exact same code drive the targeting behavior of both bots in Raven and military units in BW.

7.2.2 StarCraft and The Brood War API

Although BW is not open-source, hackers have long been able to tamper with the game process by breaking into its memory space. Eventually,

a development framework was built on top of this hacking. The Brood War Application Programming Interface (BWAPI) is an open source C++ framework which allows AI developers to create custom agents by providing them with means to access the game state and issue commands. More information regarding the features and the design of the API can be found on the project's web page.[4]

7.2.3 Targeting in Graven

The targeting system module in Graven, `CptTargetingSystem`, is used by the main AI module `CptBot`. To function, it requires another module, the sensory memory module `CptSensoryMemory`, which determines which enemies the bot currently senses. The targeting system works by setting an internal target variable which the bot module can read to find out which enemy it should aim at.

The original AI selects targets based on their distance to the bot and prioritizes closer enemies. It was modified to instead select targets based on their health and prioritize weaker enemies, a more interesting strategy for this experiment because the default unit AI in BW also uses distance as the primary factor in target selection. The main module function is shown in Figure 7.12. The vision update function in the sensory module is shown in Figure 7.13.

7.2.4 Completing the Graven Conceptual Layer

In terms of conceptual view, the requirements of the targeting module include those of its dependencies (i.e., the sensory memory module). The solution requirements can quickly be determined by looking at Figures 7.12 and 7.13. It requires a 2D world with the list of targetable entities that exist in it as well as a list of vision-blocking obstacles such as walls typically defined in a map. The entities must have their position, facing direction, field of view and health attributes synchronized. All of these concepts are already defined in the conceptual layer used in Graven.

In addition to those, BW involves three more concepts which are not present in Raven and which need to be defined. First, the concept of entity ownership is required to specify the player a unit belongs to. In Raven, a player is associated with a single bot. In BW, a player is associated with

[4] `https://code.google.com/p/bwapi/`

```
1  void CptTargetingSystem::Update()
2  {
3    int LowestHPSoFar = MaxInt;
4    m_pCurrentTarget = 0;
5
6    //grab a list of all the opponents the owner can sense
7    std::list<CptMvAgent2D*> SensedBots;
8    SensedBots = m_pOwner->GetSensoryMem()->
       GetListOfRecentlySensedOpponents();
9
10   std::list<CptMvAgent2D*>::const_iterator curBot =
       SensedBots.begin();
11   for (curBot; curBot != SensedBots.end(); ++curBot)
12   {
13     //make sure the bot is alive and that it is not the owner
14     if ((*curBot)->isAlive() && (*curBot != m_pOwner->
         GetAgent()))
15     {
16       int hp = (*curBot)->Health();
17
18       if (hp < LowestHPSoFar)
19       {
20         LowestHPSoFar = hp;
21         m_pCurrentTarget = *curBot;
22       }
23     }
24   }
25 }
```

Figure 7.12: Modified target selection in Graven. Health is compared instead of distance.

```
1  void CptSensoryMemory::UpdateVision()
2  {
3    //for each bot in the world test to see if it is visible to the owner of
         this class
4    const std::list<CptMvAgent2D*>& bots = m_pOwner->GetWorld()->GetAllBots();
5    std::list<CptMvAgent2D*>::const_iterator curBot;
6    for (curBot = bots.begin(); curBot != bots.end(); ++curBot)
7    {
8      //make sure the bot being examined is not this bot
9      if (m_pOwner->GetAgent() != *curBot)
10     {
11       //make sure it is part of the memory map
12       MakeNewRecordIfNotAlreadyPresent(*curBot);
13
14       //get a reference to this bot's data
15       CptMemoryRecord& info = m_MemoryMap[*curBot];
16
17       //test if there is LOS between bots
18       if (m_pOwner->GetWorld()->isLOSOkay(m_pOwner->GetAgent()->Pos(), (*
             curBot)->Pos()))
19       {
20         info.bShootable = true;
21
22         //test if the bot is within FOV
23         if (isSecondInFOVOfFirst(m_pOwner->GetAgent()->Pos(), m_pOwner->
               GetAgent()->Facing(), (*curBot)->Pos(), m_pOwner->GetAgent()->
               FieldOfView()))
24         {
25           info.fTimeLastSensed = Clock->GetCurrentTime();
26           info.vLastSensedPosition = (*curBot)->Pos();
27           info.fTimeLastVisible = Clock->GetCurrentTime();
28
29           if (info.bWithinFOV == false)
30           {
31             info.bWithinFOV = true;
32             info.fTimeBecameVisible = info.fTimeLastSensed;
33           }
34         }
35
36         else
37         {
38           info.bWithinFOV = false;
39         }
40       }
41
42       else
43       {
44         info.bShootable = false;
45         info.bWithinFOV = false;
46       }
47     }
48   }//next bot
49 }
```

Figure 7.13: Vision update in the Graven sensory memory module.

multiple units. Therefore, an owner property is required for units to differentiate between allies and enemies. The second concept is that of sight range. In Raven, a bot has a 180 degree field of view but its vision range is only limited by obstacles. In BW, a unit has a 360 degree field of view but can only see up to a certain radius. A sight range property is thus required. The third concept is the plane. The world in BW is two-dimensional but there are ground and air units. Ground units are not always able to attack air units and vice versa. A property to indicate the plane in which a unit exists and which planes it can target is thus needed. As a result, five new members are added to the `CptMvAgent2D` class, a player ID, a sight range, a plane flag and two plane reach flags. Note that the sensory memory module is slightly modified to take into account this information, though this has no impact on its functionality in Raven.

As far as conceptual controls are concerned, the aiming and shooting controls in Graven are not necessary for BW. When a unit in BW is given an order to attack another unit, the target only needs to be within firing range to be automatically attacked continuously. Only one conceptual control, an attack command, is required for this experiment and added to the conceptual framework.

7.2.5 Integrating the targeting AI in StarCraft

In order to use the targeting AI from Graven in BW, there are a few tasks that need to be completed. These are:

1. adding code to the game to maintain in memory a conceptual view including the elements mentioned above,

2. implementing the attack conceptual control,

3. and creating an AI solution which makes use of the targeting AI to control units.

Conceptual View

The conceptual view is maintained using 3 callback functions provided by the BWAPI, the `onStart` function which is called at the beginning of a BW game, the `onEnd` function which is called at the end of the game and the `onFrame` function which is called every game frame. The code added in

```
1  void GravenAIModule::onStart()
2  {
3    ...
4
5    //Create 2D world
6    cptWorld = new CptWorld2D();
7
8    //Add an empty map
9    cptWorld->pSetMap(new CptMap2D());
10
11   //Set map dimensions
12   cptWorld->GetMap()->pSetSizeX(Broodwar->mapWidth() * 32);
13   cptWorld->GetMap()->pSetSizeY(Broodwar->mapHeight() * 32);
14 }
```

Figure 7.14: Conceptual view code in the `onStart` callback function. Map dimensions in BW are given in build tiles, each build tile representing a 32 by 32 area.

```
1  void GravenAIModule::onEnd(bool isWinner)
2  {
3    ...
4
5    //Destroy world
6    delete cptWorld;
7  }
```

Figure 7.15: Conceptual view code in the `onEnd` callback function. The conceptual world destructor also destroys associated objects.

each of these functions is shown in Figures 7.14, 7.15 and 7.16 respectively. The `syncUnit` function is shown in Figure 7.17.

Because the source code of BW is not available, the synchronization process is different from the one used in the Graven experiment. Every game cycle, the game state is scanned and new (or destroyed) units are added to (or removed from) the conceptual view and the states in CDS are synchronized with unit states in the game.

Conceptual Controls

The `Attack` conceptual control is easily implemented using the basic `attack` command players can give to units in BW. The implementation is shown in Figure 7.18.

```
1  void GravenAIModule::onFrame()
2  {
3    ...
4
5    //For each unit visible to the player
6    Unitset units = Broodwar->getAllUnits();
7    for (Unitset::iterator u = units.begin(); u != units.end(); ++u)
8    {
9
10     //Ignore neutral units which include mineral fields and critters
11     if (u->getPlayer()->isNeutral())
12       continue;
13
14     //Get the projection of the unit in CDS
15     CptMvAgent2D* cptUnit = dynamic_cast<CptMvAgent2D*>(cptEntityMgr->
           GetEntityFromID(u->getID()));
16
17     //Projection found, synchronize state and update controllers
18     if (cptUnit)
19     {
20       syncUnit(cptUnit, *u);
21       cptUnit->Update();
22     }
23
24     //Projection not found, create one
25     else if (u->exists() && u->isCompleted())
26     {
27       cptUnit = new CptMvAgent2D(cptWorld);
28       syncUnit(cptUnit, *u);
29       cptWorld->pAddBot(cptUnit);
30       cptEntityMgr->RegisterEntity(cptUnit);
31
32       //If the unit can attack, register the targeting AI
33       if (u->getPlayer() == Broodwar->self() && u->canAttack())
34       {
35         RegisterDMController(cptUnit);
36       }
37     }
38   }
39
40   //Remove projections of units that no longer exist in the game
41   std::list<CptMvAgent2D*> cptUnits = cptWorld->GetAllBots();
42   for (std::list<CptMvAgent2D*>::iterator c = cptUnits.begin(); c != cptUnits
         .end(); ++c)
43   {
44     if (!Broodwar->getUnit((*c)->ID()) || !Broodwar->getUnit((*c)->ID())->
           exists())
45     {
46       cptEntityMgr->RemoveEntity(*c);
47       cptWorld->pRemoveBot((*c)->ID());
48     }
49   }
50 }
```

Figure 7.16: Conceptual view code in the `onFrame` callback function. The `RegisterDMController` creates a `CptBot`, which uses the Graven targeting AI to attack enemies, and adds it to the list of controllers of the `CptMvAgent2D`.

```
1  void GravenAIModule::syncUnit(CptMvAgent2D* u, Unit unit)
2  {
3    //Synchronize CptEntity2D attributes
4    u->SetID(unit->getID());
5    u->SetEntityType(cpttype_bot);
6    u->SetScale(1);
7    u->SetBRadius(MAX(unit->getType().height() / 2, unit->
       getType().width() / 2));
8    u->SetPos(Vector2D(unit->getPosition().x, unit->getPosition
       ().y));
9
10   //Synchronize CptMvEntity2D attributes
11   u->SetHeading(Vector2D(unit->getVelocityX(), unit->
       getVelocityY()));
12   u->SetVelocity(Vector2D(unit->getVelocityX(), unit->
       getVelocityY()));
13   u->SetMass(1);
14   u->SetMaxSpeed(unit->getType().topSpeed());
15   u->SetMaxTurnRate(unit->getType().turnRadius());
16   u->SetMaxForce(unit->getType().acceleration());
17
18   //Synchronize CptMvAgent2D attributes
19   u->SetMaxHealth(unit->getType().maxHitPoints() + unit->
       getType().maxShields());
20   u->SetHealth(unit->getHitPoints() + unit->getShields());
21   u->SetScore(unit->getKillCount());
22   u->SetPossessed(false);
23   u->SetFieldOfView(360);
24   u->Face(Vector2D(unit->getVelocityX(), unit->getVelocityY()
       ));
25   u->SetWorld(this->cptWorld);
26   u->SetStatus(unit->exists() ? CptMvAgent2D::alive :
       CptMvAgent2D::dead);
27   u->SetPlayer(unit->getPlayer()->getID());
28   u->SetSightRange(unit->getType().sightRange());
29   u->SetPlane(unit->isFlying());
30   u->SetAirReach(unit->getType().airWeapon() != WeaponTypes::
       None);
31   u->SetGroundReach(unit->getType().groundWeapon() !=
       WeaponTypes::None);
32 }
```

Figure 7.17: Synchronizing conceptual unit state. Some attributes are not required by the targeting AI and only serve as illustrations.

```
1  void Attack(int agent_id, int target_id)
2  {
3    Unit u = Broodwar->getUnit(agent_id);
4    Unit v = Broodwar->getUnit(target_id);
5
6    // Don't attack under explicit move orders
7    if (u->getOrder().getID() == Orders::Move)
8      return;
9
10   // Already attacking that target
11   if (u->getLastCommand().getTarget() != NULL && u->
         getLastCommand().getTarget()->getID() == target_id)
12     return;
13
14   if (v->getType().isFlyer())
15   {
16     if (u->getType().airWeapon() != WeaponTypes::None)
17       u->attack(v);
18     return;
19   }
20
21   else
22   {
23     if (u->getType().groundWeapon() != WeaponTypes::None && u
           ->exists())
24       u->attack(v);
25     return;
26   }
27 }
```

Figure 7.18: Implementation of the `Attack` conceptual control in BW. Because the targeting AI only selects targets the unit can attack, the test to see whether the unit is flying could be discarded.

```
1  void CptBot::Update()
2  {
3    //if the bot is under AI control but not scripted
4    if (!GetAgent()->isPossessed())
5    {
6      //examine all the opponents in the bots sensory memory
           and select one
7      //to be the current target
8      if (m_pTargetSelectionRegulator->isReady())
9      {
10       m_pTargSys->Update();
11     }
12
13     //update the sensory memory with any visual stimulus
14     if (m_pVisionUpdateRegulator->isReady())
15     {
16       m_pSensoryMem->UpdateVision();
17     }
18
19     //Attack
20     if (m_pAttackRegulator->isReady() && m_pTargSys->
           isTargetPresent())
21     {
22       Attack(m_pOwner->ID(), m_pTargSys->GetTarget()->ID());
23     }
24   }
25 }
```

Figure 7.19: The update function of the `CptBot` class. The function uses the `Attack` conceptual control to issue commands to the units in the game.

Conceptual AI

For units capable of attacking, an attack AI is added to the list of controllers of their projection using the `RegisterDMController` function. This function instantiates the `CptBot` class, which is similar to the one in Graven but which has been modified to only use the sensory memory and targeting system modules. The update function of the `CptBot` module is shown in Figure 7.19. Note that the sensory memory module only registers reachable enemy units. Allied units are ignored.

7.2.6 Results

The same targeting AI was successfully used in both Raven and BW, as shown in Figures 7.20 and 7.21. Unsurprisingly, the CF prototype[5] built from Raven, a very simple 2D shooter, had to be slightly extended for this experiment. Even so, the effort required to integrate the Graven targeting AI in BW was minimal. Of course, the AI was minimal too. This shows however that the work involved in creating conceptual AI that can be used in different games does not have to grow significantly with the number of games it can be applied to and that when a conceptual problem is clearly identified, it can be solved independently of the game it appears in.

Obviously, though it may not have been the goal of the experiment, the modified unit AI performs better in combat than the original one for ranged units, since it uses a better a strategy. In the presence of enemies, the original unit AI acquires a target by randomly selecting one within firing range. The modified unit AI on the other hand selects among targets within its sight radius the one with the lowest health. Because the sight range of a ranged unit is often close to its firing range, this behavior is similar to the original one in the sense that the unit does not move to reach a target when another target that is already in firing range exists. The behavior is therefore close but the unit does target weak enemies first in order to reduce their firepower as fast as possible. Moreover, setting a short memory span in the `CptSensoryMemory` class prevents units from remembering runaway targets for too long and starting to look for them. This helps maintain similarity between the original and modified unit AI. That way, the original unit behavior is maintained, making it harder for players to notice any difference other than the improved targeting strategy. Needless to say, the targeting AI remains completely unchanged. Note that modifying the sensory module to pick up targets that are within firing range rather than sight range makes the strategy work for melee units as well.

The modified unit AI was tested using 10 battles of 5 Terran Ghosts versus 5 Terran Ghosts, one group being controlled by the modified unit AI and the other by the original unit AI. Ghosts are ranged ground units. The group with the modified unit AI won every battle. The number of Ghosts lost during each battle is reported in Figure 7.22.

[5] More specifically the `CptMvAgent2D` class.

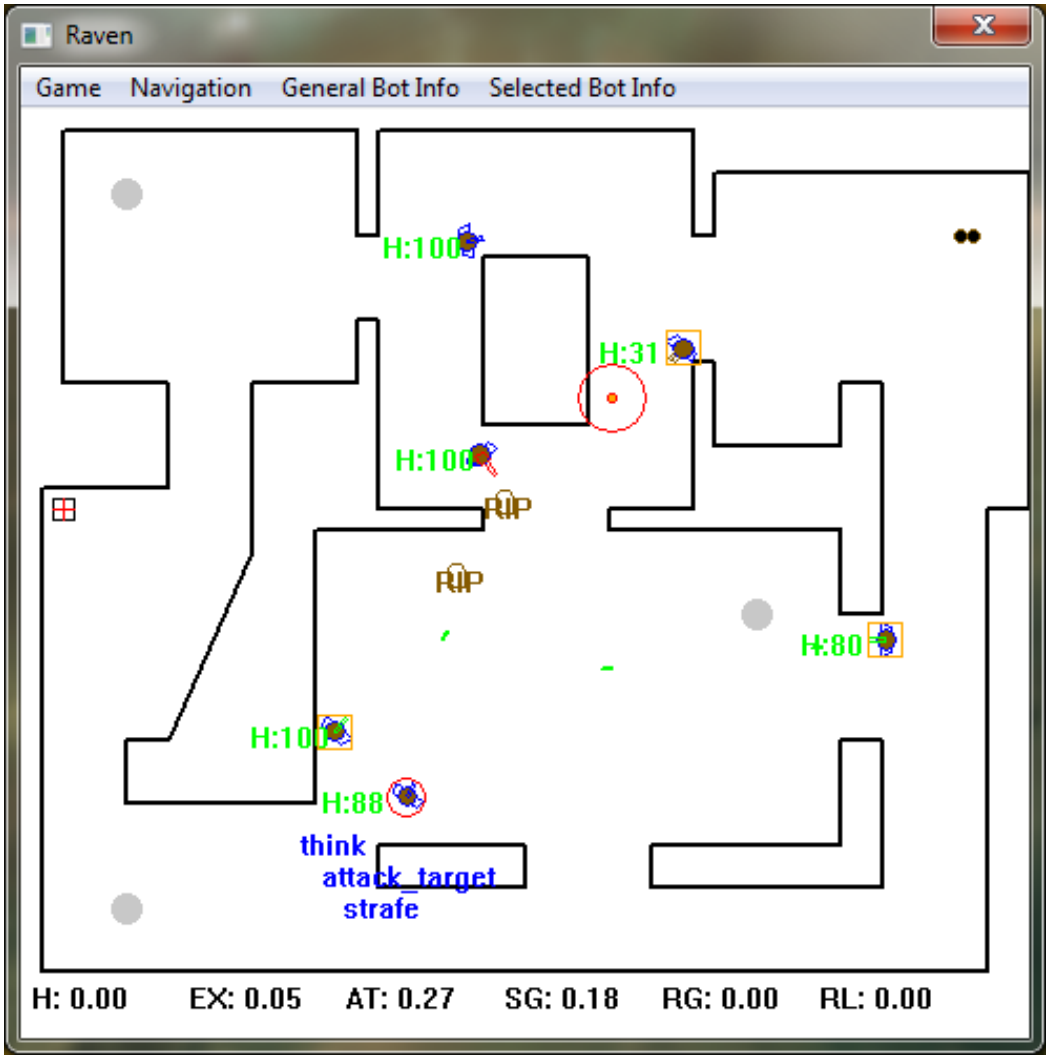


Figure 7.20: Raven with the modified targeting AI. The selected bot can be seen aiming at the enemy with low health (31), instead of the one close to it.

Figure 7.21: StarCraft: Brood War with the modified unit AI. The selected Goliaths are prioritizing Dragoons instead of the Archons in front of them because of their lower health.

Battle	1	2	3	4	5	6	7	8	9	10
Units lost Modified AI	3	3	1	3	2	3	3	4	4	3
Units lost Original AI	5	5	5	5	5	5	5	5	5	5

Figure 7.22: Units lost in each battle for each group with the modified and original unit AI.

Chapter 8

Conclusion

The main contribution of this research is an approach for the development of AI for video games[1] based on the use of a unified conceptual framework to create a conceptual layer between the game and the AI. This approach is inspired by an interpretation of human behavior. Human players have the ability to detect analogies between games and generalize, or conceptualize, the knowledge acquired in one game and apply it in another. By conceptualizing video games and asking game developers to create conceptual views of their games using a unified framework, it becomes possible to create solutions for common conceptual problems and use them across multiple video games. Developing solutions for conceptual problems rather than specific video games means that AI design is no longer confined to the scope of individual game projects and can be more efficiently refined over time. Such conceptual AI can then serve as a core engine for driving agents in a variety of video games which can be complemented by game developers specifically for each game. This would both reduce AI redundancy and facilitate the development of robust AI.

Such an approach can result in a number of advantages for game developers. First, it means that they no longer need to spend a lot of resources to design robust game AI unless they want to and can simply use existing AI solutions. Even though they have to add code for the creation of conceptual views, not having to worry about game AI can result in significant cuts in development time. For example, they would not even need to plan for coordination mechanisms between multiple agents in the game. Moreover, they do not need to use conceptual AI for all tasks. They can select the problems they want to handle using conceptual AI and use regular AI for

[1]The AI referred to here is game related and does not include context related AI as specified at the beginning of this work.

other tasks. Story and environment related AI, which this approach does not apply to, can be designed using existing tools and techniques, such as scripting engines and behavior trees, which make it easy to implement specific behavior. In addition, the continuous development of conceptual AI is likely to yield better quality solutions over time than what can be achieved through independent game projects. It may also be that clearly identifying and organizing the conceptual problems that make up the challenges offered by video games could allow game developers to compose new challenges more easily.

Since this approach allows AI development to progress independently of video games, it could lead to the birth of a new game AI business. AI developers could compete to create the best AI solutions and commercialize them or they could collaborate to design a solid open-source AI core which would be perfected over time. Additionally, machine learning techniques would be more straightforward to apply with a unified conceptual representation of game elements. These techniques can be used to learn specialized behavior for each game which can enhance the basic generic behavior. This is similar to the way humans tune their generic experience as they learn specific data about a video game they are playing to improve their performance in that particular game.

With an open-source unified conceptual framework, incentive for both game developers and AI developers to contribute to the development of the framework and the conceptualization of video games would exist. AI developers would benefit from a better conceptual framework because it would help factor AI better and allow more efficient AI development, resulting in better quality AI which benefits game developers directly when they integrate it in their games to create smarter, more challenging and more realistic agents.

Because the conceptual layer constitutes a sort of middleware, a new version of the conceptual framework may not be compatible with AI developed prior to its update. Even if it is, legacy AI may require an update in order to benefit from the improved conceptual framework. Another disadvantage of the approach is that it requires more computational resources in order to maintain a conceptual view in memory during runtime, though this may not represent a major obstacle with mainstream hardware featuring increasingly more processing cores and system memory. Other issues may also arise from the separation of AI from video games. Indeed, game developers could lose some control over the components of their games

and subsequently over the ability to balance them. For instance, it may be necessary to design new mechanisms to allow game developers to retain control over the difficulty of the game and adjust the skill level of their agents. Furthermore, although machine learning techniques such as imitation learning could benefit from a larger learning set as a unified conceptual representation would give them access to data from many games, they would require a translation process to project human actions into conceptual data space since, unlike AI actions, those are not conceptual. In other words, without a translation process, conceptual game states could only be linked to concrete game actions.

Though an implementation of the approach was presented to illustrate some applications, alternative implementations can easily be imagined. For example, even if the AI code was compiled alongside the game code in Graven, it was designed to be independent. AI modules can be compiled independently from game code and either linked to the game statically or dynamically loaded at runtime. An implementation using the latter option would benefit from easier testing of different AI solutions. When deployed, it would allow players to switch between different solutions too. This may not be desirable however, as untested solutions may result in unexpected behavior. A security mechanism could be added to prevent the game from loading unverified AI modules.

Perhaps the most exciting extension to this research would be a study of the world of conceptual problems found in video games. Both the video game industry and the scientific community would benefit from tools for describing and organizing problems using a set of convenient standards. This would help better categorize and hierarchically structure problems and result in a clearer view and understanding of the complexity of video games.

Appendix

Contents

Imitative Learning for Real-Time Strategy Games

Quentin Gemine, Firas Safadi, Raphaël Fonteneau
and Damien Ernst

Abstract

Over the past decades, video games have become increasingly popular and complex. Virtual worlds have gone a long way since the first arcades and so have the artificial intelligence (AI) techniques used to control agents in these growing environments. Tasks such as world exploration, constrained pathfinding or team tactics and coordination just to name a few are now default requirements for contemporary video games. However, despite its recent advances, video game AI still lacks the ability to learn. In this paper, we attempt to break the barrier between video game AI and machine learning and propose a generic method allowing real-time strategy (RTS) agents to learn production strategies from a set of recorded games using supervised learning. We test this imitative learning approach on the popular RTS title StarCraft II® and successfully teach a Terran agent facing a Protoss opponent new production strategies.

1 Introduction

Video games started emerging roughly 40 years ago. Their purpose is to bring entertainment to the people by immersing them in virtual worlds. The rules governing a virtual world and dictating how players can interact with objects or with one another are referred to as game mechanics. The first video games were very simple: small 2-dimensional discrete space, less than a dozen mechanics and one or two players at most. Today, video games feature large 3-dimensional spaces, hundreds of mechanics and allow numerous players and agents to play together. Among the wide variety

of genres, real-time strategy (RTS), portrayed by games like Dune II (Westwood Studies, 1992), Warcraft (Blizzard Entertainment, 1994), Command & Conquer (Westwood Studios, 1995) or StarCraft (Blizzard Entertainment, 1998), provides one of the most complex environments overall. The multitude of tasks and objects involved as well as the highly dynamic environment result in extremely large and diverging state and action spaces. This renders the design of autonomous agents difficult. Currently, most approaches largely rely on generic triggers. Generic triggers aim at catching general situations such as being under attack with no consideration to the details of the attack (i.e., location, number of enemies, ...). These methods are easy to implement and allow agents to adopt a robust albeit non-optimal behavior in the sense that agents will not fall into a state for which no trigger is activated, or in other words a state where no action is taken. Unfortunately, this type of agent will often discard crucial context elements and fail to display the natural and intuitive behavior we may expect. Additionally, while players get more familiar with the game mechanics and improve their skills and devise new strategies, agents do not change and eventually become obsolete. This evolutionary requirement is critical for performance in RTS games where the pool of possible strategies is so large that it is impossible to estimate optimal behavior at the time of development. Although it is common to increase difficulty by granting agents an unfair advantage, this approach seldom results in entertainment and either fails to deliver the sought-after challenge or ultimately leads to player frustration.

Because the various facets of the RTS genre constitute very distinct problems, several learning technologies would be required to grant agents the ability to learn on all aspects of the game. In this work, we focus on the production problem. Namely, we deal with how an agent takes production-related decisions such as building a structure or researching a technology. We propose a generic method to teach an agent production strategies from a set of recorded games using supervised learning. We chose StarCraft II as our testing environment. Today, StarCraft II, Blizzard Entertainment's successor to genre patriarch StarCraft, is one of the top selling RTS games. Featuring a full-fledged game editor, it is the ideal platform to assess this new breed of learning agents. Our approach is validated on the particular scenario of a one-on-one, Terran versus Protoss matchup type. The created agent architecture comprises both a dynamically learned production model based on multiple neural networks as well as a simple scripted combat handler.

The paper is structured as follows. Section 2 briefly covers some related work. Section 3 details the core mechanics characterizing the RTS genre. Section 4 and 5 present the learning problem and the proposed solution, respectively. Section 6 discusses experimental results and, finally, Section 7 concludes and highlights future lines of work.

2 Related Work

Lately, video games have attracted substantial research work, be it for the purpose of developing new technologies to boost entertainment and replay value or simply because modern video games have become an alternate, low-cost yet rich environment for assessing machine learning algorithms.

Roughly, we could distinguish 2 goals in video game AI research. Some work aims at creating agents with properties that make them more fun to play with such as human-like behavior [Umarov et al., 2012, Togelius et al., 2011]. Competitions like BotPrize or the Turing test track of the Mario AI Championship focus on this goal. It is usually attempted on games for which agents capable of challenging skilled human players already exist and is necessary because, often, agents manage to rival human players due to unfair advantages: instant reaction time, perfect aim, etc. These features increase performance at the cost of frustrating human opponents. For more complicated games, agents stand no chance against skilled human players and improving their performance takes priority. Hence, performance similar to what humans can achieve can be seen as a prerequisite to entertainment. Indeed, we believe that facing a too weak or too strong opponent is not usually entertaining. This concept is illustrated in Figure 1. In either case, video game AI research advances towards the ultimate goal of mimicking human intelligence. It was in fact suggested that human-level AI can be pursued directly in these new virtual environments [Laird and van Michale Lent, 2000].

The problem of human-like agent behavior has been tackled in first-person shooter (FPS) games, most notably the popular and now open-source game Quake II, using imitative learning. Using clustering by vector quantization to organize recorded game data and several neural networks, more natural movement behavior as well as switching between movement and aim was achieved in Quake II [Bauckhage et al., 2003]. Human-like behavior was also approached using dedicated neural networks for handling weapon switching, aiming and firing [Gorman and Humphrys, 2007]. Fur-

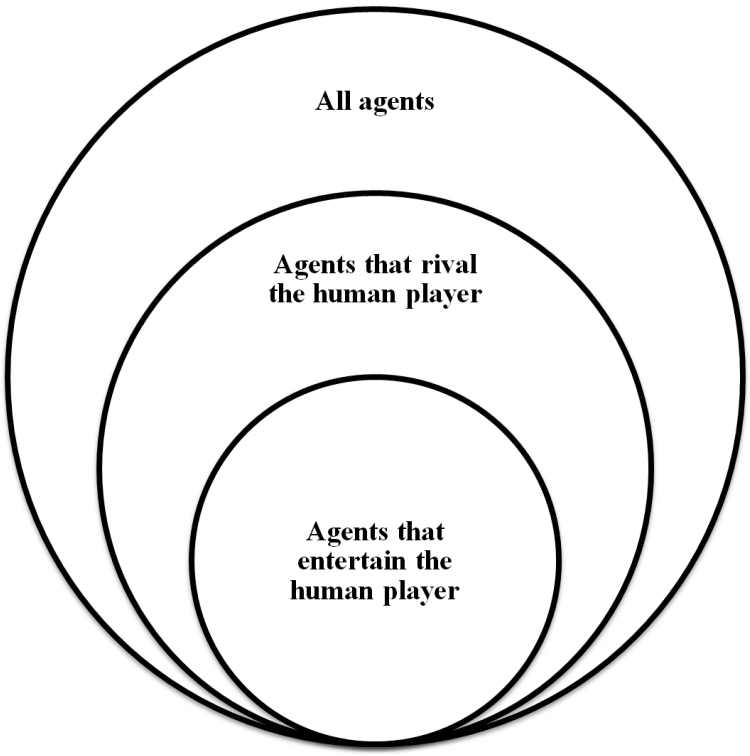


Figure 1: Agent set structure for a video game

ther work discussed the possibility of learning from humans at all levels of the game, including strategy, tactics and reactions [Thurau et al., 2004].

While human-like agent behavior was being pursued, others were more concerned with performance issues in genres like real-time strategy (RTS) where the action space is too large to be thoroughly exploited by generic triggers. Classifiers based on neural networks, Bayesian networks and action trees assisted by quality threshold clustering were successfully used to predict enemy strategies in StarCraft [Frandsen et al., 2010]. Case-based reasoning has also been employed to identify strategic situations in Wargus, an open-source Warcraft II clone [Ontañón et al., 2007, Aha et al., 2005, Weber and Mateas, 2009a]. Other works resorted to data mining and evolutionary methods for strategy planning and generation [Weber and Mateas, 2009b, Ponsen et al., 2006]. Non-learning agents were also proposed [McCoy and Mateas, 2008]. By clearly identifying and organizing tasks, architectures allowing incremental learning integration at different levels were developed [Safadi et al., 2011].

Although several different learning algorithms were applied in RTS environments, few were actually used to dictate agent behavior directly. In this paper, we use imitative learning to teach a StarCraft II agent to autonomously pass production orders. The created agent building, unit and technology production is entirely governed by the learning algorithm and does not involve any scripting.

3 Real-Time Strategy

In a typical RTS game, players confront each other on a specific map. The map is essentially defined by a combination of terrain configuration and resource fields. Once the game starts, players must simultaneously and continuously acquire resources and build units in order to destroy their opponents. Depending on the technologies they choose to develop, players gain access to different unit types each with specific attributes and abilities. Because units can be very effective against others based on their type, players have to constantly monitor their opponents and determine the combination of units which can best counter the enemy's composition. This reconnaissance task is referred to as scouting and is necessary because of the "fog of war", which denies visibility to players over areas where they have no units deployed.

Often, several races are available for the players to choose from. Each race possesses its own units and technologies and is characterized by a unique play style. This further adds to the richness of the environment and multiplies mechanics. For example, in StarCraft II players can choose between the Terrans, masters of survivability, the Zerg, an alien race with massive swarms, or the Protoss, a psychically advanced humanoid species.

Clearly, players are constantly faced with a multitude of decisions to make. They must manage economy, production, reconnaissance and combat all at the same time. They must decide whether the current income is sufficient or new resource fields should be claimed, they must continuously gather information on the enemy and produce units and develop technologies that best match their strategies. Additionally, they must swiftly and efficiently handle units in combat.

When more than two players are involved, new diplomacy mechanics are introduced. Players may form and break alliances as they see fit. Allies have the ability to share resources and even control over units, bringing additional management elements to the game.

Finally, modern RTS games take the complexity a step further by mixing in role-playing game (RPG) mechanics. Warcraft III, a RTS title also developed by Blizzard Entertainment™, implements this concept. Besides regular unit types, heroes can be produced. Heroes are similar to RPG characters in that they can gain experience points by killing critters or enemy units to level up. Leveling up improves their base attributes and grants them skill points which can be used to upgrade their special abilities.

With hundreds of units to control and dozens of different unit types and special abilities, it becomes clear that the RTS genre features one of the most complex environments overall.

4 Problem Statement

The problem of learning production strategies in a RTS game can be formalized as follows.

Consider a fixed player u. A world vector $\boldsymbol{w} \in \mathcal{W}$ is a vector describing the entire world at a particular time in the game. An observation vector $\boldsymbol{o} \in \mathcal{O}$ is the projection of $\boldsymbol{w}$ over an observation space $\mathcal{O}$ describing the

part of the world perceivable by player u. We define a state vector $\boldsymbol{s} \in \mathcal{S}$ as the projection of $\boldsymbol{o}$ over a space $\mathcal{S}$ by selecting variables deemed relevant to the task of learning production strategies. Let $n \in \mathbb{N}$ be the number of variables chosen to describe the state. We have:

$$\boldsymbol{s} = (s_1, s_2, ..., s_n), \forall i \in \{1, ..., n\} : s_i \in \mathbb{R}$$

Several components of $\boldsymbol{s}$ are variables that can be directly influenced by production orders. Those are the variables that describe the number of buildings of each type available or planned, the cumulative number of units of each type produced or planned and whether each technology is researched or planned. If a technology is researched or planned, the corresponding variable is equal to 1, otherwise, it is equal to 0. Let m be the number of these variables and let s_{p_1}, s_{p_2}, ..., s_{p_m} be the components of $\boldsymbol{s}$ that correspond to these variables.

When in state $\boldsymbol{s}$, a player u can select an action vector $\boldsymbol{a} \in \mathcal{A}$ of size m that gathers the "production orders". The j^{th} component of this vector corresponds to the production variable s_{p_j}. When an action $\boldsymbol{a}$ is taken, the production variables of $\boldsymbol{s}$ are immediately modified according to:

$$\forall j \in \{1, ..., m\} : s_{p_j} \leftarrow s_{p_j} + a_j$$

We define a production strategy for player u as a mapping $P : \mathcal{S} \rightarrow \mathcal{A}$ which selects an action vector $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\boldsymbol{a} = P(\boldsymbol{s})$$

5 Learning Architecture

We assume that a set of recorded games constituted of state vectors $\boldsymbol{s}^u \in \mathcal{S}^u$ of player u is provided. Our objective is to learn the production strategy P^u used by player u. To achieve this, we use supervised learning to learn to predict each production variable s_{p_j} based on the remaining state $\boldsymbol{s}_{-p_j}$ defined below. We then use the predicted s_{p_j} values to deduce a production order $\boldsymbol{a}$. Since there are m production variables, we solve m supervised learning problems. Formally, our approach works as follows.

For any state vector $\boldsymbol{s}$, we define the remaining state for each production variable s_{p_j} as $\boldsymbol{s}_{-p_j}$:

$$\forall j \in \{1, ..., m\} : \boldsymbol{s}_{-p_j} = (s_1, s_2, ..., s_{p_j-1}, s_{p_j+1}, ..., s_n)$$

For each production variable, we define a learning set $\{(\boldsymbol{s}^u_{-p_j}, s^u_{p_j})\}_{\boldsymbol{s}^u \in \mathcal{S}^u}$ from which we learn a function $\hat{P}^u_j$ which maps any remaining state $\boldsymbol{s}_{-p_j}$ to a unique $\hat{P}^u_j(\boldsymbol{s}_{-p_j})$. Knowing each $\hat{P}^u_j$, we can deduce a mapping $\hat{P}^u$ and estimate a production order $\boldsymbol{a}$ for any given state vector $\boldsymbol{s}$:

$$\begin{aligned}\boldsymbol{a} = \hat{P}^u(\boldsymbol{s}) = (&\hat{P}^u_1(\boldsymbol{s}_{-p_1}) - s_{p_1}, \\ &\hat{P}^u_2(\boldsymbol{s}_{-p_2}) - s_{p_2}, ..., \hat{P}^u_m(\boldsymbol{s}_{-p_m}) - s_{p_m})\end{aligned}$$

Using this approach, we learn the production strategy used by player u by learning m $\hat{P}^u_j$ functions to estimate production variables given the remaining state variables. Each $\hat{P}^u_j$ is learned separately using supervised learning. In other words, we learn m models. For each model, the input for the learning algorithm is the state vector $\boldsymbol{s}$ stripped from the component the model must predict, which becomes the output. This process is illustrated in Figure 2.

It is worth stressing that the action vector $\boldsymbol{a}$ computed by the mapping $\hat{P}^u$ learned may not correspond to, due to the constraints imposed by the game, an action that can be taken. For example, $\boldsymbol{a}$ may send among others an order for a new type of unit while the technology it requires is not yet available. In our implementation, every component of $\boldsymbol{a}$ which is inconsistent with the state of the game is simply set to zero before the action vector is applied.

6 Experimental Results

The proposed method was tested in StarCraft II by teaching a Terran agent facing a Protoss opponent production strategies.

A total of $n = 108$ variables were selected to describe a state vector. These state variables are:

- $s_1 \in \mathbb{N}$ is the time elapsed since the beginning of the game in seconds
- $s_2 \in \mathbb{N}$ is the total number of units owned by the agent
- $s_3 \in \mathbb{N}$ is the number of SCVs (Space Construction Vehicles)

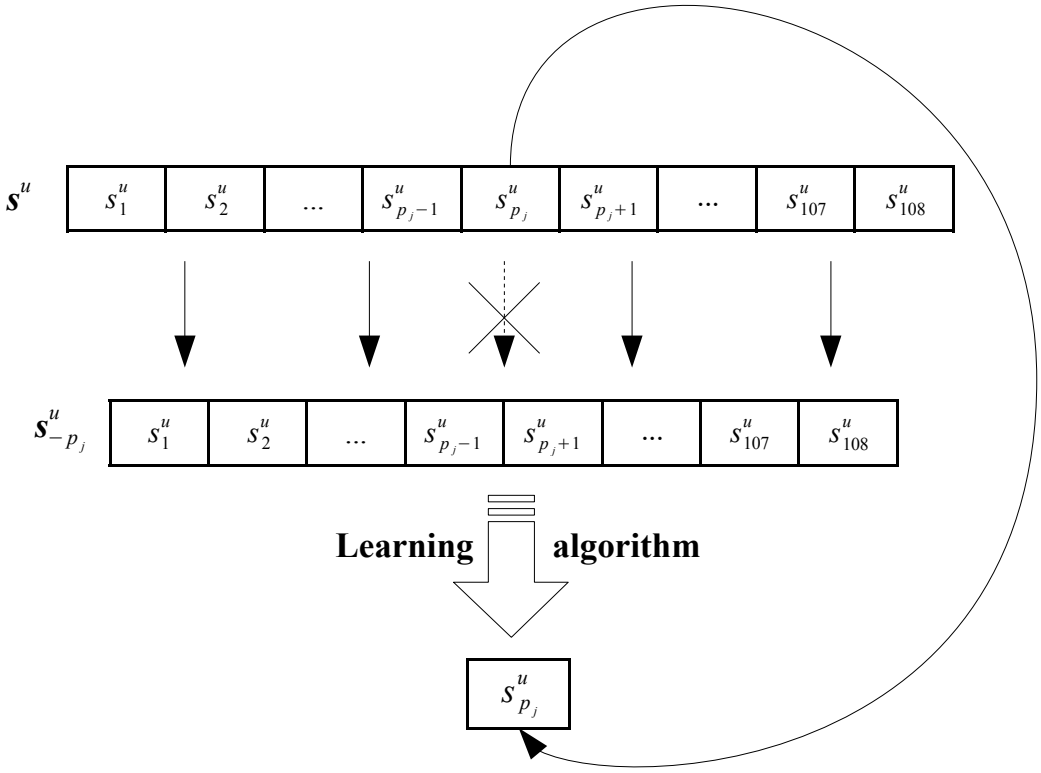


Figure 2: Learning the p_jth model

- $s_4 \in \mathbb{N}$ is the average mineral harvest rate in minerals per minute
- $s_5 \in \mathbb{N}$ is the average gas harvest rate in gas per minute
- $s_u \in \mathbb{N}, u \in \{6, ..., 17\}$ is the cumulative number of units produced of each type
- $s_b \in \mathbb{N}, b \in \{18, ..., 36\}$ is the number of buildings of each type
- $s_t \in \{0, 1\}, t \in \{37, ..., 63\}$ indicates whether each technology has been researched
- $s_e \in \{0, 1\}, e \in \{64, ..., 108\}$ indicates whether an enemy unit type, building type or technology has been encountered

Among these, there are $m = 58$ variables which correspond to direct production orders: 12 s_u unit variables, 19 s_b building variables and 27 s_t technology variables. Therefore, an action vector is composed of 58 variables. These action variables are:

- $a_u \in \mathbb{N}, u \in \{1, ..., 12\}$ corresponds to the number of additional units of each type the agent should produce
- $a_b \in \mathbb{N}, b \in \{13, ..., 31\}$ corresponds to the number of additional buildings of each type the agent should build
- $a_t \in \{0, 1\}, t \in \{32, ..., 58\}$ corresponds to the technologies the agent should research

The Terran agent learned production strategies from a set of 372 game logs generated by letting a Very Hard Terran computer player (u) play against a Hard Protoss computer player on the Metalopolis map. State vectors were dumped every 5 seconds in game time. Each $\hat{P}_j^u$ was learned using a feedforward neural network with a 15-neuron hidden layer and the Levenberg-Marquardt backpropagation algorithm [Lourakis, 2005] to update weights. Inputs and outputs were mapped to the $[-1, 1]$ range. A tan-sigmoid activation function was used for hidden layers.

Because it is not possible to alter production decisions in the Very Hard Terran player without giving up the remaining non production decisions, these 58 neural networks were combined with a simple scripted combat manager which handles when the agent must attack or defend. On the other hand, the low level unit AI is preserved. During a game, the agent

periodically predicts production orders. For any given building type, unit type or technology, if the predicted target value $\hat{P}_j^u(\boldsymbol{s}_{-p_j})$ is greater than the current number s_{p_j}, a production order a_j is passed to reach the target value. This behavior is illustrated in Figure 3.

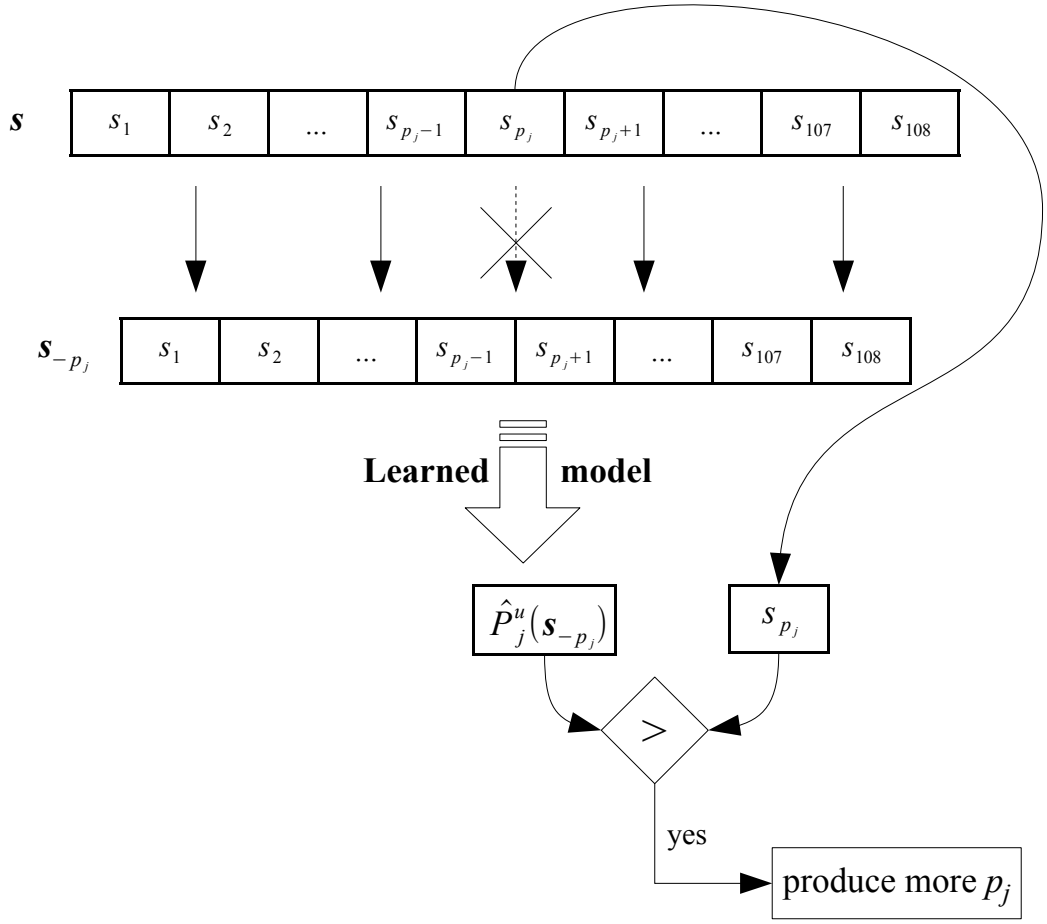


Figure 3: Agent production behavior

The final agent was tested in a total of 50 games using the same settings used to generate the training set. The results are summarized in Table 1. With a less sophisticated combat handler, the imitative learning trained agent (IML agent) managed to beat the Hard Protoss computer player 9 times out of 10 on average while the Hard Terran computer player lost every game. This performance is not far below that of the Very Hard Terran computer player the agent learned from, which achieved an average win rate of 96.5%. In addition to counting victories, we have attempted to verify that the agent indeed replicates to some extent the same production

strategies as those from the training set. Roughly, two different strategies were used by the Very Hard Terran computer player. The first one (A) primarily focuses on infantry while the second one (B) aims at faster technological development. Formally, a game is given the label Strategy A if no factories or starports are built during the first 5 minutes of the game. Otherwise it is labeled Strategy B. Figure 4 shows, for the training set, the average number of barracks, factories and starports built over time for each strategy. Two corresponding strategies were also observed for the learning agent over the 50 test games, as shown in Figure 5. For each strategy, the frequency of appearance is shown in Figure 6.

Table 1: Terran performance against Hard Protoss

	Terran win rate	**Total games**
Very Hard Terran	96.5%	372
Hard Terran	0%	50
IML agent	**90%**	**50**

The frequency at which each strategy is used was not faithfully reproduced on the test set. This can be partly explained by the more limited combat handler, which may fail to acquire the same information on the enemy than was available in the training set. Moreover, Strategy B seems to be less accurately replicated than Strategy A. This may be caused by the lower frequency of appearance in the training set. Nevertheless, the results obtained indicate that the agent learned both production strategies from the Very Hard Terran computer player. Subsequently, we may rightly attribute the agent's high performance to the fact that it managed to imitate the efficient production strategies used by the Very Hard Terran computer player.

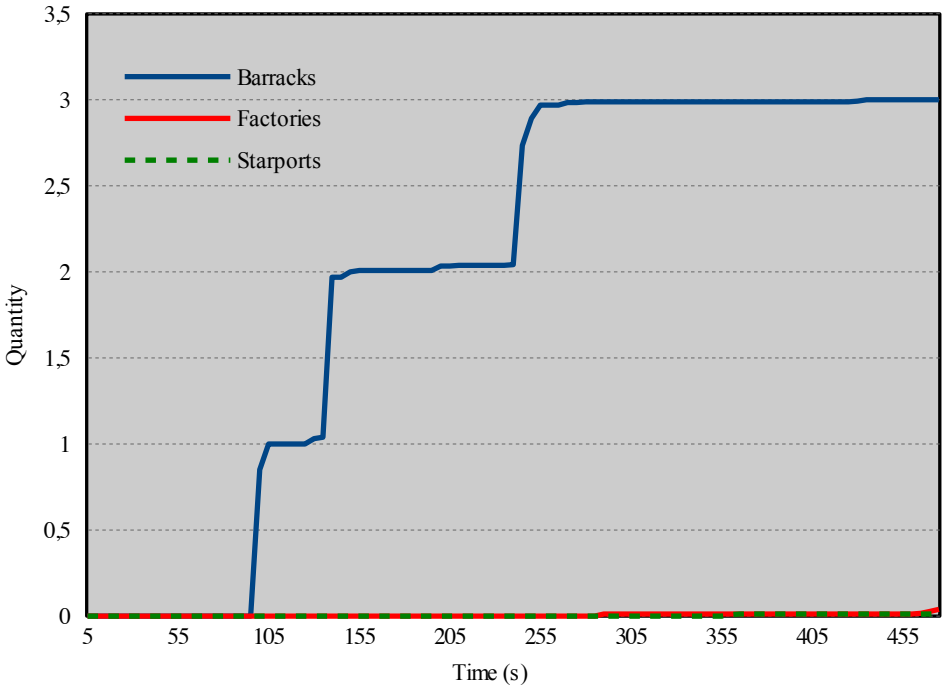
7 Conclusion and Future Work

In this paper, we have presented a method for integrating imitative learning in real-time strategy agents. The proposed solution allowed the creation of an agent for StarCraft II capable of learning production strategies from recorded game data and applying them in full one-on-one games. However, since the training data was artificially generated, the agent is restricted to a specific matchup type. A larger and more diverse dataset would be required to significantly impact the performance of agents against human

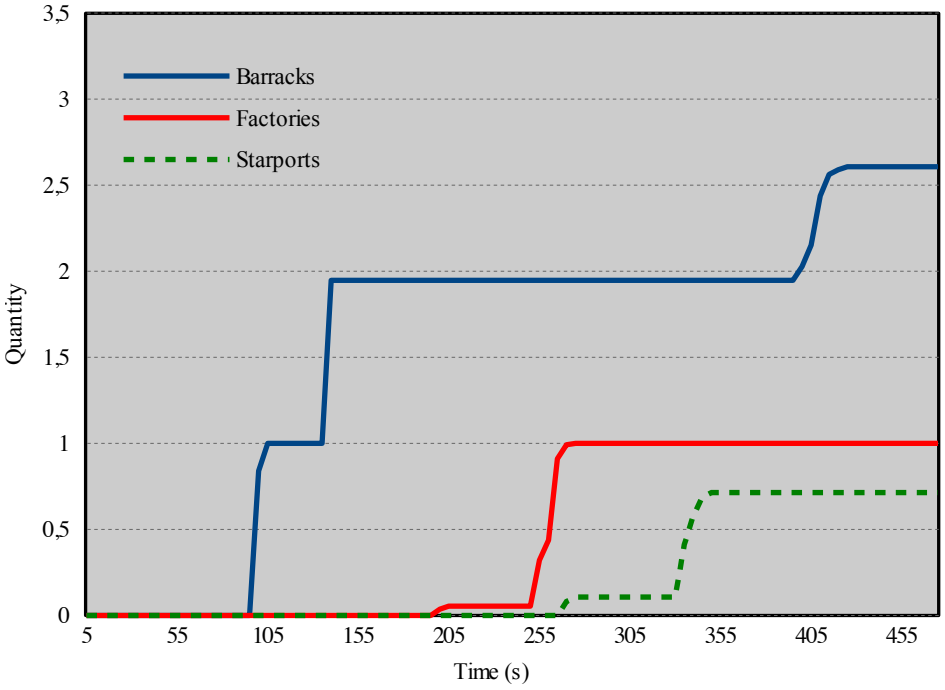
players. We therefore plan on extending this work to larger datasets.

In order to efficiently learn from richer sets, potentially collected from various sources, we suspect clustering will be required to organize records and maintain manageable datasets. Furthermore, the manually generated training data only contained desirable production strategies. When training data is automatically collected from various sources, selection techniques will be required to filter out undesirable production strategies. We believe that with a large enough set, the learned production strategy models should be robust enough to be used against human players.

Besides production-related improvements, there are other areas worth investing in to increase agent performance such as information management or combat management. Enhanced information management can allow an agent to better estimate the state of its opponents and for example predict the location of unit groups that could be killed before they can retreat or be joined by backup forces. As for combat management, it may lead to much more efficient unit handling in battle and for example maximize unit life spans.

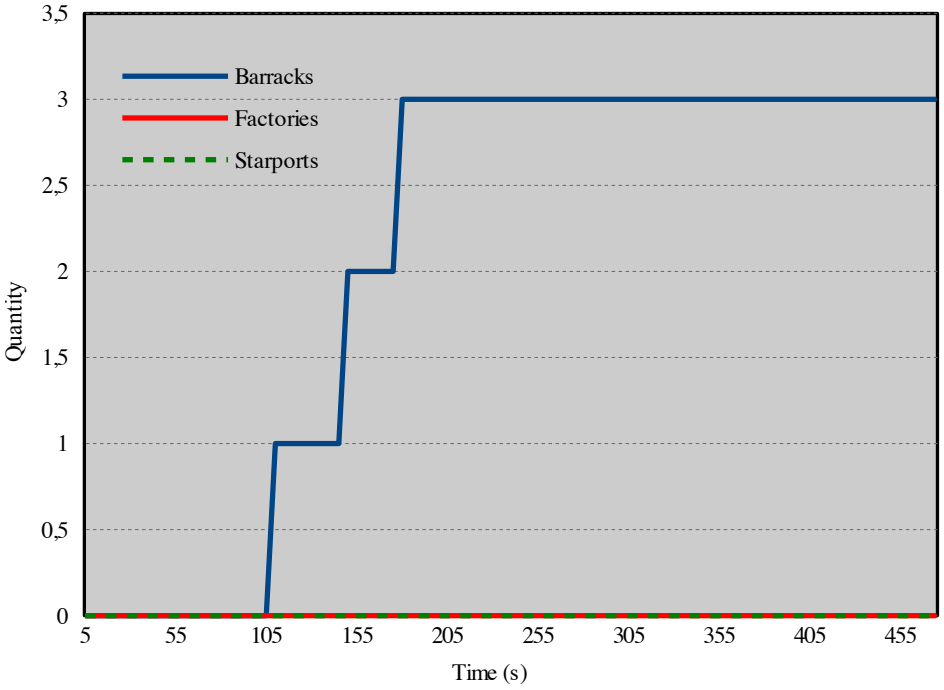


(a) Strategy A

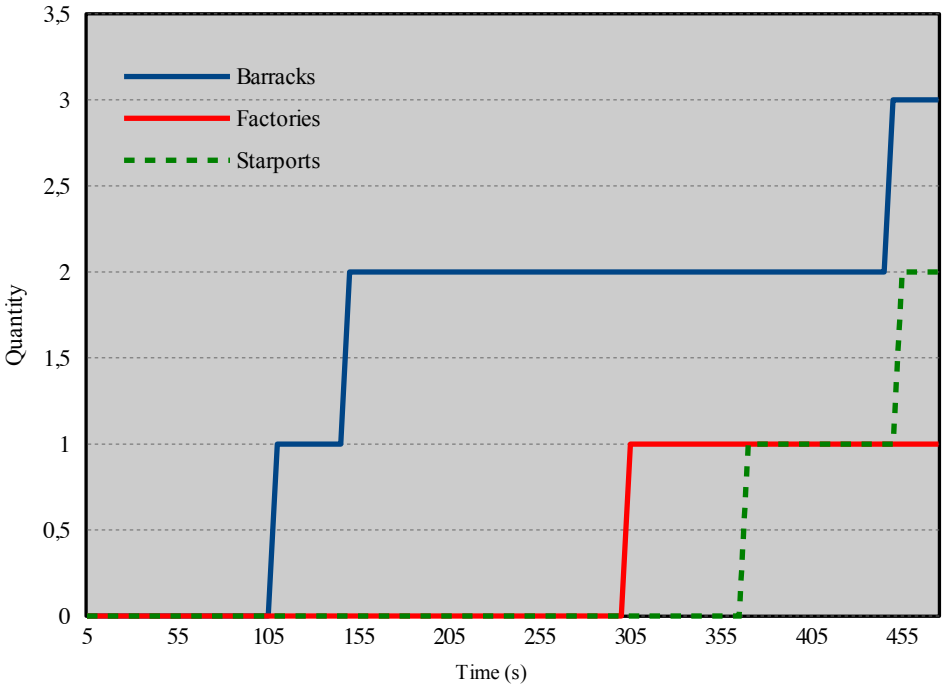


(b) Strategy B

Figure 4: Training set strategies



(a) Strategy A



(b) Strategy B

Figure 5: Test set strategies

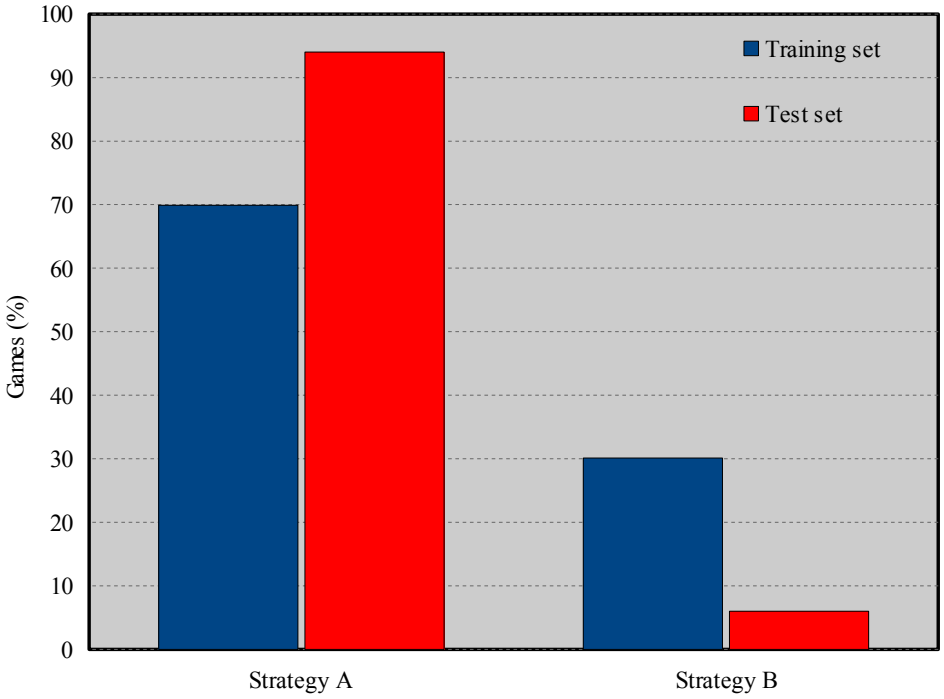


Figure 6: Strategy frequencies

Parallel Cascade Correlation: A GPU Implementation

Firas Safadi, Raphaël Fonteneau and Damien Ernst

Abstract

This paper presents a new implementation of the cascade correlation architecture which leverages the parallel computing capabilities of GPUs. It shows that by combining the inherent parallelization potential of evolutionary algorithms with the caching properties of the cascade correlation architecture, the algorithm can be simplified to a few easily parallelizable operations. It also comes with an open-source CUDA C++ implementation of the resulting genetic cascade correlation algorithm.

1 Introduction

Today, parallel computing is highly accessible and widespread thanks to cheap GPUs with thousands of cores and well-documented APIs. New algorithms are therefore being designed with parallelization in mind. Existing ones however need to be rethought and adapted to parallel architectures. In this article, the genetic cascade correlation algorithm is reexamined and implemented for NVIDIA's parallel computing architecture, CUDA.

Cascade correlation is a supervised learning algorithm which adaptively builds a neural network during the training phase and presents several advantages compared to other algorithms, such as learning very quickly [Fahlman and Lebiere, 1990]. It does however suffer from overfitting issues [Hansen and Pedersen, 1994, Tetko and Villa, 1997].

In the genetic cascade correlation algorithm, the standard Quickprop optimization algorithm used in the original cascade correlation algorithm is replaced with a genetic algorithm, which presents several benefits such as

reducing the possibility of convergence to local optimums instead of global ones and support for applications in which the gradient of the optimization function cannot be computed [Potter, 1992].

This work focuses on the parallelization advantages of using genetic algorithms in the cascade correlation learning architecture. It shows how the resulting algorithm can be simplified to a few basic operations which are easy to parallelize or for which optimized parallel solutions already exist. The CUDA C++ implementation of the algorithm can be downloaded online [Safadi, 2014a].

2 Cascade Correlation

The cascade correlation architecture is a type of neural network characterized by a number of features. In this section, it is described in the context of regression where the training dataset is composed of $n \in \mathbb{N}^*$ input-output pairs, or samples, where the input is a scalar vector of size $a \in \mathbb{N}^*$ and the output is a scalar:

$$\mathcal{D} = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\},\ \forall i \in \{1, ..., n\}, \boldsymbol{x}_i \in \mathbb{R}^a, y_i \in \mathbb{R}$$

2.1 Network Topology

The network is automatically built by adding hidden layers during the training phase. Each hidden layer contains one neuron which outputs a signal computed using the inputs of the hidden layer and which serves as an additional input for all subsequent layers. In addition, network inputs are directly connected to all layers. The number of inputs of a hidden layer is thus equal to that of the previous layer plus one. Any layer in the network can be described using a vector of scalar weights. The network itself can be described as a vector of layers. Initially, the network starts with only an output layer connected to the network inputs as well as a bias signal. The network starts as:

$$\boldsymbol{N}_0 = (\boldsymbol{o}_0),\ \boldsymbol{o}_0 \in \mathbb{R}^{a+1}$$

After l hidden layers have been added, the network becomes:

$$\boldsymbol{N}_l = (\boldsymbol{h}_1, ..., \boldsymbol{h}_l, \boldsymbol{o}_l),\ \boldsymbol{h}_1 \in \mathbb{R}^{a+1}, ..., \boldsymbol{h}_l \in \mathbb{R}^{a+l}, \boldsymbol{o}_l \in \mathbb{R}^{a+l+1}$$

Thus we have that:

$$\boldsymbol{h}_l \in \mathbb{R}^{a+l} \; \forall l \in \mathbb{N}^*$$
$$\boldsymbol{o}_l \in \mathbb{R}^{a+l+1} \; \forall l \in \mathbb{N}$$
$$\boldsymbol{N}_l \in \begin{cases} \mathbb{R}^{a+1} & \text{if } l = 0 \\ \mathbb{R}^{a+1} \times ... \times \mathbb{R}^{a+l+1} & \text{if } l \geq 1 \end{cases}$$

The number of weights in the output layer increases as the network grows. The final size of the network can be determined using different criteria, such as reaching a residual error goal or a maximum number of hidden layers.

2.2 Network Construction

The algorithm for adding a hidden layer to the network works as follows. First, the weights of the output layer are trained to minimize $E_l,\ l \in \mathbb{N}$, the total error of the network with l hidden layers over the training dataset. Then, as long as some stopping criteria is not met, the following operations are repeated. The output layer is temporarily disconnected from the network and replaced with a new hidden layer. The weights of the new hidden layer are then trained to maximize the absolute value of the covariance between its output signal and the residual error in the network over the training dataset. More precisely, the maximized value $S_l,\ l \in \mathbb{N}^*$ is defined for a hidden layer as

$$S_l = \left| \sum_{i=1}^{n} (v_{i_l} - \bar{v}_l)(e_{i_l} - \bar{e}_l) \right|$$

where v_{i_l} is the output value of the lth hidden layer for the ith sample in the training dataset, $\bar{v}_l$ is the mean output value of the lth hidden layer over all samples, e_{i_l} is the observed network error for the ith sample before adding the lth hidden layer and $\bar{e}_l$ is the mean observed network error over all samples before adding the lth hidden layer.

Once trained, the hidden layer is permanently added to the network and its weights are frozen. The output layer is reconnected with an additional weight for the new connection and is retrained. Because of the correlation between the output signal of the new hidden layer and the residual error in the network, the latter should be reduced further when the weights of the output layer are retrained. A pseudocode for this algorithm is presented below.

Algorithm 1 Network construction algorithm

$l \leftarrow 0$
$\boldsymbol{H} \leftarrow ()$
$\boldsymbol{o} \leftarrow rand(a+1)$
$train(\boldsymbol{o})$
while not $stop$ **do**
  $l \leftarrow l+1$
  $\boldsymbol{h}_l = rand(a+l)$
  $trainh(\boldsymbol{h}_l)$
  $\boldsymbol{H} \leftarrow \boldsymbol{H} \parallel \boldsymbol{h}_l$
  $\boldsymbol{o} \leftarrow \boldsymbol{o} \parallel rand()$
  $train(\boldsymbol{o})$
end while
$\boldsymbol{N} = \boldsymbol{H} \parallel \boldsymbol{o}$

In this code, the $rand$ function returns a random vector of specified dimension or a random scalar if no dimension is specified. The $train$ procedure tunes the weights of the output layer $\boldsymbol{o}_l$ so as to minimize E_l, whereas the $trainh$ procedure tunes the weights of the new hidden layer $\boldsymbol{h}_l$ to maximize S_l. Note that because the weights of the hidden layers are frozen, training the network is reduced to training the output layer.

3 Genetic Algorithms

Genetic algorithms are an optimization technique inspired by biological evolution. A genetic algorithm involves maintaining a population of individuals each representing a solution to the optimization problem. The performance or quality of individuals can be evaluated using a fitness function and is improved by repeatedly evolving them using genetic operators such as selection, combination and mutation. In this section, the technique is described in the context of optimizing weights in a neural network. Individuals are therefore represented by scalar vectors.

3.1 Weight optimization

Optimizing a weight vector of dimension b using genetic algorithms works as follows. First, a population $\boldsymbol{P}$ is created by randomly generating many individuals. Let p be the number of individuals in the population. $\boldsymbol{P}$ is a $b \times p$ matrix where each column is an individual in the population:

$$\boldsymbol{P} = (\boldsymbol{d}_1, ..., \boldsymbol{d}_p),\ \boldsymbol{d}_1, ..., \boldsymbol{d}_p \in \mathbb{R}^b$$

The population is evaluated using a fitness function $f : \mathbb{R}^{b \times p} \rightarrow \mathbb{R}^p$ specific to the optimization problem which measures the quality q of the solution represented by an individual by assigning to it a positive scalar called fitness which grows larger as the quality improves:

$$f(\boldsymbol{P}) = \boldsymbol{q} = (q_1, ..., q_p),\ q_j \geq 0\ \forall j \in \{1, ..., p\}$$

The population is then evolved using genetic operators g times, where g is the number of epochs in an evolution cycle. The fitness vector $\boldsymbol{q}$ is used to eliminate weak individuals and filter out weak genetic material from one generation to the next, keeping strong genetic material and using it to create stronger solutions. It is computed at the end of each epoch. After g epochs, the evolution cycle is complete and the individual with the highest fitness $\boldsymbol{d}_s \mid q_s \geq q_j\ \forall j \in \{1, ..., p\}$ is used as the optimal solution. A pseudocode for this algorithm is shown below.

Algorithm 2 Weight vector optimization

$\boldsymbol{P} \leftarrow rand(b, p)$
$\boldsymbol{q} \leftarrow f(\boldsymbol{P})$
for $i \leftarrow 1$ **to** g **do**
  $\boldsymbol{P} \leftarrow evolve(\boldsymbol{P}, \boldsymbol{q})$
  $\boldsymbol{q} \leftarrow f(\boldsymbol{P})$
end for
$s \leftarrow imax(\boldsymbol{q})$

Here, the $rand$ function returns a random matrix of the specified dimensions. Given a population and fitness vector, the $evolve$ function generates a new population using genetic operators. The $imax$ function finds the index in a vector of the component with the highest value.

4 Cascade Correlation using Genetic Algorithms

The merits of using genetic algorithms in the cascade correlation architecture are discussed in [Potter, 1992]. In addition to the theoretical advantages, the genetic cascade correlation algorithm is very well suited for parallelization. Indeed, the inherent parallelization potential of genetic algorithms due to the use of many independent solutions can be combined

with the caching properties of cascade correlation due to the freezing of hidden layer weights to greatly simplify the simulation of the network over the entire training dataset for an entire population.

Since the weights of a hidden layer are frozen when it is added to the network, it is possible to cache the output values of the hidden layer for all training samples. Because the output of a hidden layer is connected to all subsequent layers, this is equivalent to adding a new input to be used in any subsequent training every time a hidden layer is added. Therefore, with l hidden layers, it is possible to directly compute the output of the network $\hat{y}_{i_l}$ for a training sample $(\boldsymbol{x}_i, y_i)$ using a vector $\boldsymbol{c}_{i_l},\ i \in \{1, ..., n\}$ defined as

$$\boldsymbol{c}_{i_l} = (bias, x_{i_1}, ..., x_{i_a}, v_{i_1}, ..., v_{i_l})$$

where v_{i_l} is the output value of the lth hidden layer for the ith training sample. Let m be defined as $m = a + l + 1$. With $\boldsymbol{o}_l = (w_1, ..., w_m)$, we have:

$$\hat{y}_{i_l} = \boldsymbol{c}_{i_l} \cdot \boldsymbol{o}_l$$

The output of the network for all training samples can then be computed using

$$\hat{\boldsymbol{y}}_l = \boldsymbol{C}_l \boldsymbol{o}_l$$

where $\boldsymbol{C}_l$ is a $n \times m$ matrix where each row is a vector $\boldsymbol{c}_{i_l}$ corresponding to the ith training sample. This can then be extended to compute the output of the network for all training samples using multiple different output layers:

$$\hat{\boldsymbol{Y}}_l = \boldsymbol{C}_l \boldsymbol{P}$$

In that case, $\boldsymbol{P}$ is a $m \times p$ matrix representing a population of p output layers and $\hat{\boldsymbol{Y}}_l$ is a $n \times p$ matrix containing the output values of the network for all n training samples and all p individuals. This is interesting because several high-performance parallel implementations for matrix multiplication operations already exist and these matrix multiplications are at the heart of the network construction algorithm.

5 CUDA Implementation

This work is accompanied by an open-source CUDA C++ implementation of the genetic cascade correlation algorithm called *parallel-cc*. The code

comes in the form of a CUDA 6.5 project for Visual Studio 2013 and can be downloaded online [Safadi, 2014a]. This section describes some of the choices made for this implementation. In the CUDA context, *host* refers to the system board and *device* refers to the graphics board.

5.1 Data Representation

The primary data structure used throughout the project is a duplex matrix stored in row-major order. The data can be stored on host or on device, or both and can be synchronized in either direction. On device, the *cudaMallocPitch* function is used to ensure that row addresses are correctly aligned for coalescing. Data matrices are also shaped according to work parallelization so as to avoid uncoalesced access to global memory on device as much as possible.

Scalars are represented using a *real* type which can be defined as a single-precision floating point type or a double-precision floating point type. Switching between 32- and 64-bit precision is easily done by toggling a macro which affects type definitions and functions across the project.

5.2 Genetic Algorithm

A multi-population genetic algorithm is used for weight optimization. Contrary to a standard genetic algorithm, this genetic algorithm maintains a species rather than a population. A species is composed of multiple independent populations each with its own genetic parameters. The genetic parameters of each population can be set manually or generated randomly using species parameters. This presents a number of advantages. For example, rather than manually looking for interesting genetic parameters for a particular problem, specifying broad species parameters and working with more populations and lesser individuals per population can make the search more automatic. Another advantage is that it is faster to sort multiple small fitness vectors instead of a single large one. Improving spacial locality in memory reference while evolving populations is yet another example. With small populations, memory access patterns during evolution are delimited, leading to better L1/L2 cache exploitation.

The *cuRAND* API is used for random number generation. Each individual in a population has its own generator. The generator is not only used to generate random scalars but also for any stochastic operation.

The genetic operators used are roulette-wheel selection, elitism, one-point and uniform crossover, mutation and random generation. For each population, a proportion can be specified for the use of each operator. The roulette-wheel selection operator is implemented using the roulette-wheel selection via stochastic acceptance algorithm, which provides $O(1)$ performance instead of the standard logarithmic complexity obtained with a dichotomic search and a sorted fitness vector [Lipowski and Lipowska, 2012].

After a species is evaluated, the populations are sorted according to fitness using the bitonic sort algorithm. The bitonic sort algorithm is very easy to parallelize and works very well for small to medium-sized arrays on a parallel architecture, a case for which it was shown to be much more efficient than other popular parallel sorting algorithms such as the radix sort found in the *Thrust* library [Safadi, 2014b].

Alternating containers are used for storing populations. This allows tracking changes from one generation to the next and makes the evolution code simpler by avoiding the use of any temporary storage.

Typically, the workload is parallelized across individuals in all major functions, meaning that individuals are handled separately, each by a different thread.

5.3 Cascade Correlation

The cascade correlation algorithm is used in conjunction with the genetic algorithm for regression problems. Data sets can be loaded from files and divided into a training set and a test set by specifying a desired test sample proportion. Test samples can either be chosen from the end of the file or randomly. The test set is used to detect overfitting, a common issue with the cascade correlation algorithm.

The training phase can be controlled in different ways. An error goal may be specified to stop the training once it is reached. A maximum number of hidden layers in the network can also be set to limit the training phase. An overfitting detection trigger may be used as well. The training stops after a specified number of hidden layers have been added without reducing the error on the test set. Alternatively, the user can manually stop the training any time. When the training is stopped, the network with the

best performance on the test set is returned.

The error on a data set is measured using the sum of absolute sample errors $E = \sum |\hat{y} - y|$. The fitness function used for the output layer weight optimization is $f_o(\boldsymbol{o}) = \frac{1}{1+E}$. For hidden layer weight optimization, the fitness function is $f_h(\boldsymbol{h}) = S$. A sigmoid function, more specifically $f(x) = \frac{x}{1+|x|}$, is used as the activation function for hidden layers.

The *cuBLAS* API is used to perform matrix multiplications. Since the API works with matrices that are stored in column-major order, the provided functions cannot be used in a standard way with the matrices in this project which are stored in row-major order. A matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ is computed by requesting that $\boldsymbol{C}^T = \boldsymbol{B}^T\boldsymbol{A}^T$ be computed instead and inverting the dimensions of each matrix. Providing a matrix $\boldsymbol{A}$ stored in row-major order as is but specifying inverted height and width causes it to be read as $\boldsymbol{A}^T$ in column-major order. This avoids unnecessary format conversion.

The amount of data transfers between host and device is minimal and the majority of the workload is executed on device, making this implementation largely dependent on device performance.

6 Experimental Results

This section presents results obtained using a GeForce GTX 460 1GB with 336 CUDA cores. It includes 2 primary experiments. The implementation is tested on the *Housing* data set [UCI] with varying combinations of population count and size in order to gain some insight on how using multiple independent populations affects learning performance both in terms of training time and model accuracy. The first experiment is done using fixed training and test set partitioning. In the second experiment, the test set is randomly selected every run. In both experiments, the genetic parameters are fixed and the test set accounts for 10% of the samples in the data set, or 50 samples, leaving 456 samples in the training set. The results of the first and second experiments are reported in Tables 1 and 2 and Tables 3 and 4, respectively.

For different total individual counts (i.e., population count times population size), different combinations of population size and count are tested. For each combination, the average training time, the average number of

hidden layers in the final network, the MSE on the training set and the test set and the best and worst MSE achieved on the test set over 10 runs are reported. The average training time corresponds to the time it takes to build and train the network from 0 to 50 hidden layers, in seconds. The average number of hidden layers corresponds to the average number of hidden layers in the optimal network found (i.e., the network with the lowest test error). The remaining columns are self-explanatory.

In the first experiment (Tables 1 and 2), the first thing to notice is the disparity between training and test errors as well as the relatively small network size, which suggest that the seemingly subtle similarities between the training set and the test set are not captured fast enough by the algorithm. Decreasing the error on the training set increases the error on the test set in most cases, causing networks to be small. The worst case error seems to decrease as the number of populations grows for this particular test, but the large variance in the results makes it difficult to draw any conclusion. The impact of using multiple populations on worst case performance is investigated further using auxiliary experiments described at the end of this section. The results show that there is no significant impact.

In terms of training time, the results confirm that working with multiple small populations is faster than working with a single large one when the training phase is controlled by the maximum number of hidden layers. This is best illustrated in the table with 32,768 total individuals, where the difference between the highest and lowest average training time is over 10 seconds. It is also worth noticing that while the training time decreases as the population is broken into smaller and smaller populations for the reasons mentioned in the previous section, it starts increasing after a certain point, typically when population size drops below 128. This is not surprising and is due to the block size used for the population evolution kernel. Populations are evolved in parallel using blocks of 128 threads (one thread per individual). When population size becomes lower than the block size, the workload for each block becomes less uniform, leading to lower instructions per warp (IPW) and more stalled warps. This can be easily verified by modifying the block size. Using 512 threads per block causes peak performance to shift towards a population size of 512 instead of 128.

The impact of evolving multiple populations with different configurations in a single block can be assessed using the following experiment. First, the time required to evolve 1,000 times a species made of 1024 populations of 32 individuals (32,768 total individuals) with a random generation rate

of 1.0 (i.e., an entirely new population is generated each epoch, discarding the previous one) and using blocks of 128 threads (4 populations per block) is measured. In this case, the workload in a block is completely uniform and consists in generating 128 random individuals. A time of 336 ms is recorded. Next, the test is repeated with a mutation rate set to 1.0 (i.e., each generation is a mutation of the previous one). Mutation is a more costly operator than random generation. A time of 383 ms and an IPW of 3,865 (2 or more eligible warps 82% of the time in warp issue efficiency) are recorded. Then, the species mutation rate range is set to $[0.0, 1.0]$, resulting in populations with a mutation rate of different values in that range. This means that an evolution consists in a certain number of mutations and random individual generations. This results in 8 interleaved sub-blocks of mutation and random generation in each block. This time, the test completes in 523 ms with an IPW of 2,311 (2 or more eligible warps 72% of the time), thus proving that block performance can severely suffer from non-uniform workloads.

In the second experiment (Tables 3 and 4), the disparity between training error and test error is significantly reduced compared to the first experiment. This shows that the fixed training and test set partitioning used in the first experiment is particularly difficult to learn with. Randomly selected test samples lead to more consistent results and useful hidden layers. Worst-case performance however becomes largely dependent on the training and test set partitioning and the benefits of using multiple small populations become insignificant.

Tables 5 and 6 present results similar to those obtained in the first experiment, only with a number of evolution epochs set to 50 down from 100. The impact on accuracy is small, but the training time is cut by a factor two. This is interesting because on a parallel architecture, compensating for some sequential optimization by drastically increasing the number of individuals can lead to better resource exploitation and result in lower execution time.

In order to further investigate the results obtained in the first experiment, the following auxiliary experiments are conducted. The algorithm is tested using different configurations of 1024 total individuals and 4096 total individuals. In these experiments, the test set contains 20% of the dataset samples (101 samples), leaving 405 training samples. The maximum number of hidden layers is increased to 114 and an overfitting detection trigger set to 50 hidden layers is used, meaning that training may be

interrupted before reaching the maximum allowed number of hidden layers if the error on the test set cannot be reduced after adding 50 hidden layers. For the experiment with 1024 total individuals, the test set is randomly selected at each run and the algorithm is run 100 times for each different configuration. For the experiment with 4096 total individuals, 20 tests are generated by randomly partitioning the dataset into training and test sets 20 times. For each test, the training and test set are fixed and the algorithm is run 100 times for each configuration. The results of the experiments are presented in Figures 1 and 2 and Figures 3 and 4 in the form of box plots showing the minimum, first quartile, median, third quartile and maximum training error, test error, hidden layers and training time. Each box plot in the experiment with 1024 total individuals represents 100 runs each with a different training and test set partitioning, whereas in the experiment with 4096 individuals each one represents 2,000 runs figuring 20 training and test sets shared across configurations.

A few conclusions can be drawn from the plots. First, using multiple small populations instead of a single large one has little impact on accuracy. It may prevent extremely bad performance as hinted by the high maximum test error measured when using a single population compared to using multiple populations (66.08 for 1x1024 vs. 26.69 for 4x256 and 62.98 for 1x4096 vs. 34.41 for 32x128), though more tests are necessary to assert this. Another clear observation is the increasing network size as populations become smaller. Smaller populations lead to more hidden layers, in turn leading to longer training times when the network size is unbounded. Thus, although optimizing multiple small populations is faster than optimizing a single large one, they ultimately result in a longer training when the training phase is controlled by the overfitting detection trigger and not the maximum number of hidden layers because they tend to converge more slowly to an optimum.

7 Conclusion

This article explores some interesting characteristics of the genetic cascade correlation algorithm when it is implemented on a parallel architecture. By combining the caching properties of the cascade correlation algorithm with the inherent parallelization potential of genetic algorithms, the algorithm is simplified to basic operations such as matrix multiplications for which efficient solutions already exist for parallel architectures, making it easier for an efficient implementation of the algorithm to be produced.

The work includes an open-source CUDA C++ implementation of the algorithm which supports multi-population genetic algorithms to further take advantage of the target architecture. The implementation is tested on a GeForce GTX 460 1GB with 336 CUDA cores and some interesting conclusions are reached during the discussion of the results. Using multiple small populations is shown to be faster than using a single large one on a parallel architecture such as CUDA, but also to converge more slowly with each hidden layer and ultimately lead to a longer training phase when the latter is not controlled by the size of the network. In addition, using a large enough total individual count makes it possible to depend less on sequential optimization through evolution and better exploit resources in a parallel architecture.

One possible improvement to the implementation could be an alternate workload distribution for the evolution of populations. Currently, populations are divided into potentially heterogeneous blocks where different genetic operators may be used by different threads, leading to potentially lower workload uniformity within blocks and therefore potentially lower performance, especially with a very small population size. A better solution may be to group all similar operations from the different populations into the same blocks, creating only uniform blocks (i.e., mutation blocks, random generation blocks, and so on).

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	5.7444	7.6	25.1532	15.0253	9.9296	25.3749
2	512	5.7507	12.7	23.6276	15.1427	8.87243	24.5904
4	256	5.7097	6.3	28.502	14.5934	11.0936	23.9173
8	128	5.68	19.1	23.0815	12.7926	7.72544	18.2764
16	64	5.7195	12.1	30.1349	12.6693	9.86672	19.0923
32	32	5.8223	7.1	29.7822	11.9925	8.57852	14.8908
Total		5.7378	10.8	26.7136	13.7026	9.34439	21.0237

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	6.815	14.7	22.3864	16.1352	10.8022	22.8103
2	1024	6.802	6.2	22.5142	13.3874	11.7459	16.3759
4	512	6.7237	1	32.8006	13.7729	10.3181	18.2186
8	256	6.7126	3.3	30.4429	13.3755	10.0057	20.4157
16	128	6.6247	3.5	27.3918	12.7235	8.40769	18.0643
32	64	6.701	6	27.7456	13.1441	10.7849	16.6599
64	32	6.8261	5.1	34.3868	12.5976	10.3125	16.0724
Total		6.7436	5.7	28.2383	13.5909	10.3396	18.3739

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	8.9534	10.7	23.1412	16.215	11.299	24.8604
2	2048	8.881	11.7	19.0772	16.1186	13.0309	23.689
4	1024	8.793	11.1	21.5032	13.2302	9.73053	16.6072
8	512	8.5829	9.4	21.5898	13.5255	9.69317	17.542
16	256	8.4892	11.7	20.3152	11.8112	9.90674	14.9905
32	128	8.432	13.1	21.9509	13.7596	10.0077	17.7071
64	64	8.5082	7.5	30.659	13.3795	11.0917	17.9532
128	32	8.7011	3.3	33.9822	13.3303	10.2261	15.5214
Total		8.6676	9.8	24.0273	13.9212	10.6232	18.6089

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	17.6427	5.1	19.9755	14.9292	9.63952	18.7254
2	4096	17.3068	5.9	21.6052	14.1965	9.78466	18.2335
4	2048	17.0237	6.2	22.9352	12.5862	9.56828	16.0705
8	1024	16.6	5.3	27.856	13.3651	9.16044	18.3874
16	512	16.1657	12.5	23.5779	12.6474	9.56242	16.1686
32	256	15.9795	6	24.9239	11.8402	8.59518	16.1702
64	128	15.9188	7.6	22.2146	12.8953	9.18679	17.5616
128	64	16.1825	0.9	35.0585	13.4192	9.97808	15.6321
256	32	16.6815	9.4	25.1965	13.042	9.42702	16.5128
Total		16.6112	6.5	24.8159	13.2135	9.43360	17.0513

Table 1: Results obtained with fixed training and test sets (1a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	33.0061	3.7	19.4949	13.8875	11.9197	16.4154
2	8192	32.0269	2.1	23.1108	14.3605	11.8449	19.632
4	4096	31.1067	4	23.1619	13.2571	10.2325	14.9378
8	2048	30.3363	7.4	20.9396	13.796	10.5962	18.701
16	1024	29.5315	5.1	22.1481	13.1562	10.5932	17.2023
32	512	28.7155	1.6	26.0216	14.1073	11.5454	17.1557
64	256	28.3895	11.8	20.7006	14.533	8.26138	20.742
128	128	28.4287	1.1	28.7517	12.7224	7.37068	18.1893
256	64	28.9285	4.5	26.0835	11.8389	9.51295	16.0983
512	32	29.6812	6.2	28.3104	12.7063	9.57904	16.5264
Total		30.0151	4.8	23.8723	13.4365	10.1456	17.5600

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	65.7259	13	15.1833	13.4346	10.6843	19.8109
2	16384	63.6212	6.2	21.4143	13.337	7.26945	17.5696
4	8192	61.6337	6.9	22.4735	13.9666	11.9585	18.3517
8	4096	59.4162	3.5	20.8499	13.502	11.4353	18.0738
16	2048	57.895	12.5	16.1652	13.3681	10.7183	15.6923
32	1024	56.0563	21.9	13.3399	11.5464	9.78405	14.0714
64	512	54.4305	8.9	21.3979	13.6064	10.3957	19.9792
128	256	53.8164	7.9	21.97	13.5777	12.029	16.6165
256	128	53.921	6.6	24.2377	12.535	10.9093	14.3517
512	64	54.852	1.9	28.563	11.3465	8.51013	13.8668
1024	32	56.5238	6.5	26.2171	12.7646	9.13264	15.7992
Total		57.9902	8.7	21.0738	12.9986	10.2570	16.7439

Table 2: Results obtained with fixed training and test sets (1b)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	5.7866	30.7	8.79973	16.2081	8.68358	27.9637
2	512	5.7828	21.9	11.1445	18.3999	9.82574	37.3507
4	256	5.7254	26.5	10.201	15.1386	8.24654	43.3569
8	128	5.6837	36.8	8.08679	16.6551	8.91677	27.4543
16	64	5.7254	33	11.7211	16.2391	7.25801	25.5675
32	32	5.828	36.7	12.7327	21.2692	8.8733	30.1395
Total		5.7553	30.9	10.4476	17.3183	8.63399	31.9721

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	6.8076	20.6	10.4119	12.5716	7.24871	16.4769
2	1024	6.8415	17.2	12.2465	13.626	6.70158	19.0794
4	512	6.7526	28.9	7.16465	13.8171	7.7502	22.039
8	256	6.7058	33.7	8.61757	14.218	7.87063	21.7652
16	128	6.6317	37.3	7.8377	14.3334	5.83973	21.2703
32	64	6.7047	28.6	14.9117	11.175	8.02507	16.5312
64	32	6.834	40.4	11.3318	17.3591	5.79048	49.7445
Total		6.7540	29.5	10.3603	13.8715	7.03234	23.8438

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	8.9666	23.7	7.90538	14.1123	7.90678	23.6956
2	2048	8.8635	17.6	12.5183	15.6521	9.55615	28.2149
4	1024	8.7665	16.4	11.3872	16.5327	8.8409	32.594
8	512	8.5904	28	9.21454	12.4991	5.73309	41.0194
16	256	8.5073	21.4	13.1608	12.281	6.05751	27.9876
32	128	8.4413	32.3	9.04951	12.8213	4.81874	25.843
64	64	8.5229	40.7	8.68574	14.544	9.95593	22.6624
128	32	8.7021	31.7	13.859	17.9074	10.9365	29.5512
Total		8.6701	26.5	10.7226	14.5437	7.9757	28.9460

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	17.6891	19.1	9.97156	14.9819	6.33594	24.8311
2	4096	17.3195	29.1	6.09559	14.319	9.38157	22.1386
4	2048	17.0635	22.8	10.4267	15.3268	7.57837	31.6053
8	1024	16.5957	24.7	7.84281	14.2335	7.81084	29.1284
16	512	16.1536	18.3	11.8615	13.2321	9.09361	19.4294
32	256	15.9389	24.6	11.5548	14.3233	9.23231	20.5262
64	128	15.9407	34.5	8.09448	16.4088	7.15903	29.7323
128	64	16.2059	28.5	11.5789	16.6488	4.93472	41.7021
256	32	16.6839	38.6	11.3399	13.1785	6.87062	21.7681
Total		16.6212	26.7	9.86292	14.7392	7.59967	26.7624

Table 3: Results obtained with random training and test sets (a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	33.1539	20.8	8.28689	11.4533	6.21434	19.3758
2	8192	32.2125	20.5	7.07369	14.8537	8.1422	25.9423
4	4096	31.2328	26.1	6.22508	17.0958	9.86969	30.6308
8	2048	30.3883	26	9.30055	16.056	7.53701	31.4495
16	1024	29.5724	23.6	9.06147	15.0136	9.94383	24.0633
32	512	28.7664	20.3	8.83704	15.2959	7.6053	29.8551
64	256	28.4327	31.7	9.80965	15.5661	6.74968	39.2089
128	128	28.4607	16.3	13.8391	13.1809	7.2251	25.9738
256	64	28.9291	33.5	10.1962	14.4106	7.06882	24.2269
512	32	29.71	36.9	12.1729	15.4333	9.55274	22.3314
Total		30.0859	25.6	9.48026	14.8359	7.99087	27.3058

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	65.0992	20.2	7.68169	11.9521	5.41838	29.6517
2	16384	63.4841	19.8	8.86801	11.314	6.68024	20.0277
4	8192	61.5051	27.7	7.16731	16.9692	8.25641	26.3433
8	4096	59.3211	19.7	7.00603	12.082	6.01189	22.2692
16	2048	57.8055	29.8	6.81576	13.3609	7.93704	27.4768
32	1024	56.0592	18.8	9.34539	16.4611	10.1129	23.6008
64	512	54.4021	34.5	5.74977	17.5853	7.00735	42.7785
128	256	53.8582	21.5	10.8576	19.832	10.0531	36.1215
256	128	53.8308	28.7	11.6495	18.6566	10.6928	31.2416
512	64	54.8088	32.9	10.2964	14.5821	7.13799	20.7592
1024	32	56.4425	39.4	10.3247	19.6797	9.96527	40.9079
Total		57.8742	26.6	8.70565	15.6795	8.11576	29.1980

Table 4: Results obtained with random training and test sets (b)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	1024	2.9376	10.4	28.6406	17.7922	10.6933	31.748
2	512	2.945	10.8	24.1731	16.11	10.2608	22.4682
4	256	2.9302	4.3	35.6086	14.1943	6.91624	22.6771
8	128	2.914	8.7	38.0709	13.0964	9.23085	16.6226
16	64	2.9323	11	27.5158	14.3576	11.1315	16.1959
32	32	2.9851	8	38.0299	14.1412	6.82335	18.9557
Total		2.9407	8.9	32.0065	14.9486	9.17601	21.4446

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	2048	3.5058	7.3	25.397	16.4004	11.0134	32.0249
2	1024	3.4844	11.8	20.8501	16.4594	10.2645	25.913
4	512	3.4567	6.4	29.4426	14.7056	10.1829	20.1048
8	256	3.436	6.1	32.4958	14.3556	11.441	17.428
16	128	3.3905	7.3	30.717	13.4743	7.68237	24.1688
32	64	3.4323	9.9	26.9228	13.0089	10.4453	17.975
64	32	3.4926	15.7	32.8819	12.3857	8.16403	19.1199
Total		3.4569	9.2	28.3867	14.3986	9.88479	22.3906

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	4096	4.6091	15.6	21.885	15.0998	10.697	28.4413
2	2048	4.5177	9.7	25.5822	13.2116	11.4431	16.712
4	1024	4.4892	5	28.1588	15.1242	10.7369	18.7719
8	512	4.3733	6.2	28.0812	13.4197	7.87407	16.3264
16	256	4.3272	4.3	26.8258	14.0732	9.19711	21.4063
32	128	4.2987	17.7	23.4574	13.2223	10.6287	15.7383
64	64	4.3389	9.2	24.103	13.0291	10.0032	16.2899
128	32	4.4258	15.1	29.1083	12.8179	9.01312	16.7183
Total		4.4225	10.4	25.9002	13.7497	9.94915	18.8006

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	8192	8.9584	15.6	19.9823	15.854	12.5602	24.8782
2	4096	8.7448	14.8	19.3946	13.254	10.8309	16.7977
4	2048	8.6227	10.4	23.4762	14.2682	9.15028	21.0271
8	1024	8.4192	7.8	23.9407	11.17	9.45436	12.9457
16	512	8.2094	10.9	21.0673	12.1152	10.0815	15.6551
32	256	8.0856	15.7	25.2934	12.2781	10.2268	15.0444
64	128	8.0576	3	32.8097	12.0329	9.82625	16.1738
128	64	8.1955	14.8	24.2395	12.9621	9.34046	18.4505
256	32	8.4579	6.8	30.0937	12.4577	8.5789	15.4924
Total		8.4168	11.1	24.4775	12.9325	10.0055	17.3850

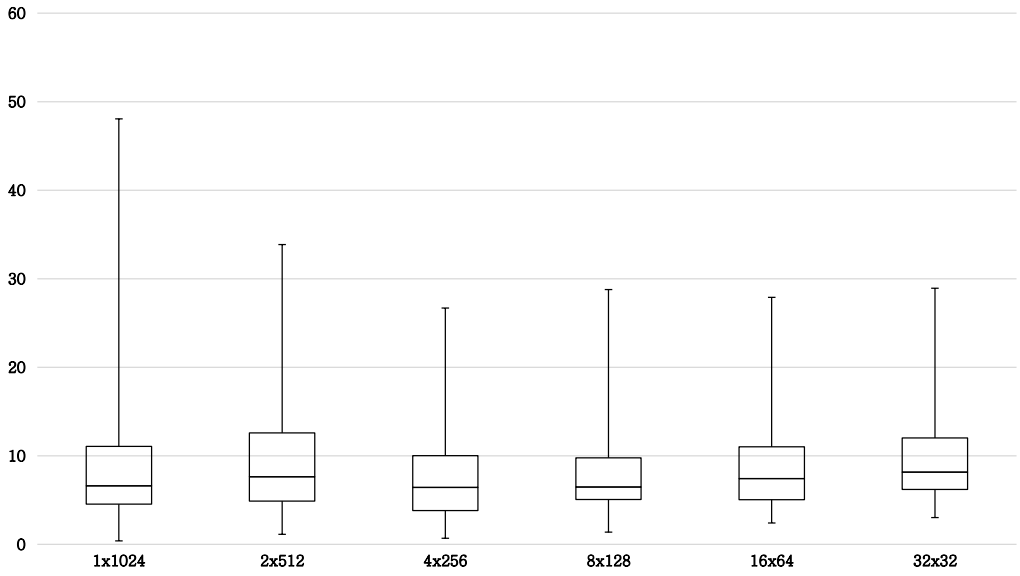
Table 5: Results obtained with fixed training and test sets (2a)

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	16384	16.7614	3.5	26.8023	14.283	10.7267	19.9507
2	8192	16.1715	16.1	15.1361	12.996	10.7763	15.6309
4	4096	15.7204	8	19.978	13.894	7.93584	15.8702
8	2048	15.3757	4.3	26.4008	14.3963	11.1337	18.9091
16	1024	14.9467	6.5	24.0266	13.4192	12.2344	15.4028
32	512	14.5467	11.9	21.3798	11.9368	8.83372	13.867
64	256	14.409	6.1	25.3503	14.2053	10.7836	19.6191
128	128	14.3923	14.5	21.8821	14.18	9.35829	18.7834
256	64	14.6231	4.7	28.4069	12.5931	7.78428	20.2294
512	32	15.0278	10.3	29.741	12.0001	9.44003	14.3281
Total		15.1975	8.6	23.9104	13.3904	9.90069	17.2591

Population		Average				Best	Worst
Count	Size	Time	Hidden Layers	Training Error	Test Error	Test Error	Test Error
1	32768	33.027	11	20.2192	14.0776	10.1778	19.2708
2	16384	31.9724	7.9	23.0951	13.9906	10.5227	18.3219
4	8192	31.0466	6.9	23.2507	13.7711	10.6745	17.4305
8	4096	29.9824	9.7	16.0172	11.8862	8.48604	14.352
16	2048	29.1772	10.6	20.5569	13.6273	10.8444	18.0473
32	1024	28.2283	6.7	21.7664	13.9419	12.801	16.1334
64	512	27.3834	10.5	21.7159	13.6081	9.17518	17.8265
128	256	27.1135	7	24.2265	13.2249	9.70003	16.5823
256	128	27.1462	4	28.3921	12.5484	8.66757	15.8086
512	64	27.6345	10.7	22.9369	13.4483	8.23694	18.0201
1024	32	28.4831	3.6	31.5642	12.4989	9.59946	17.2554
Total		29.1995	8.1	23.0674	13.3294	9.89869	17.1863

Table 6: Results obtained with fixed training and test sets (2b)

(a) MSE on the training set



(b) MSE on the test set

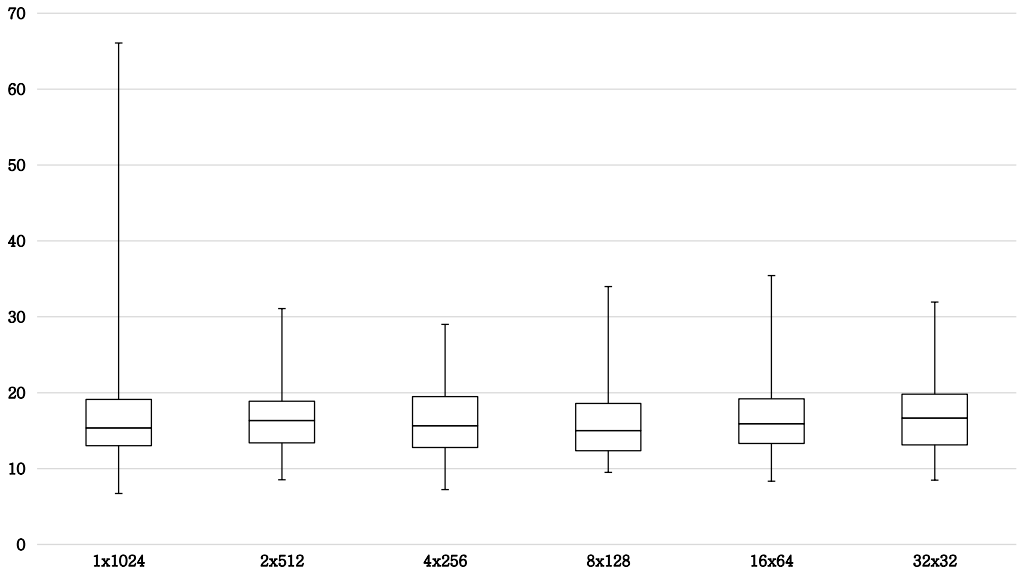
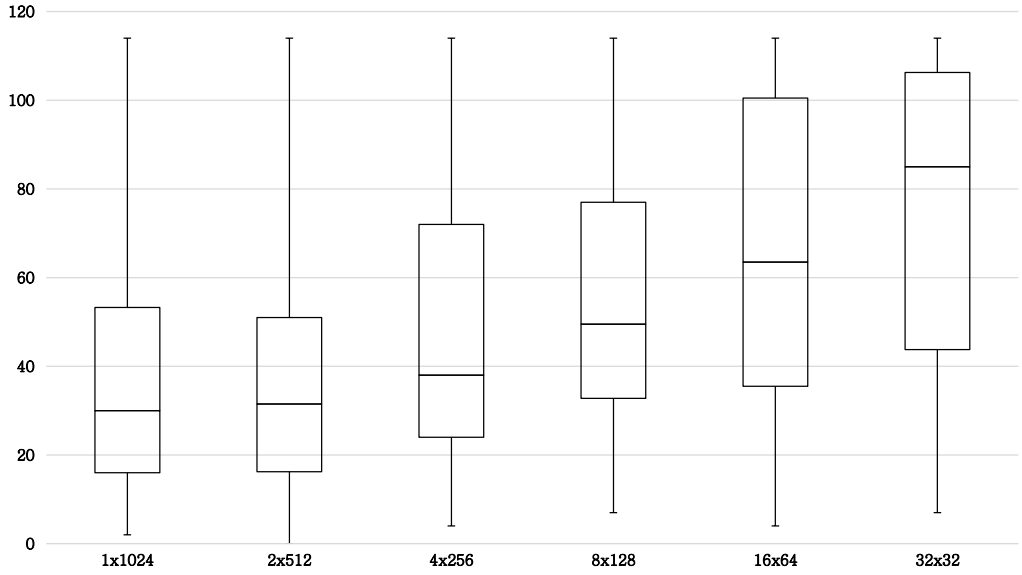


Figure 1: Training and test error versus population configuration (1a)

(a) Hidden layers



(b) Training time in seconds

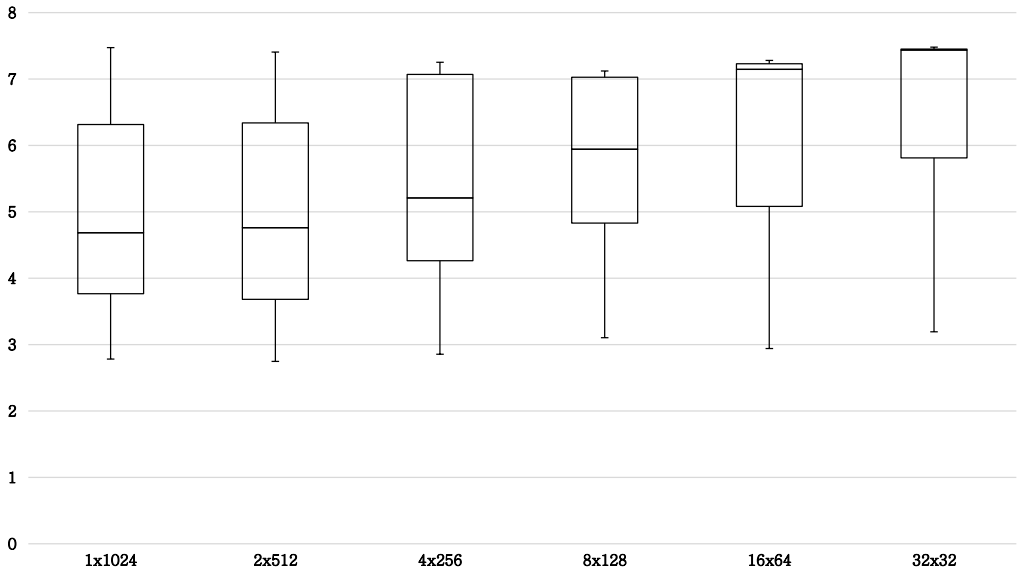
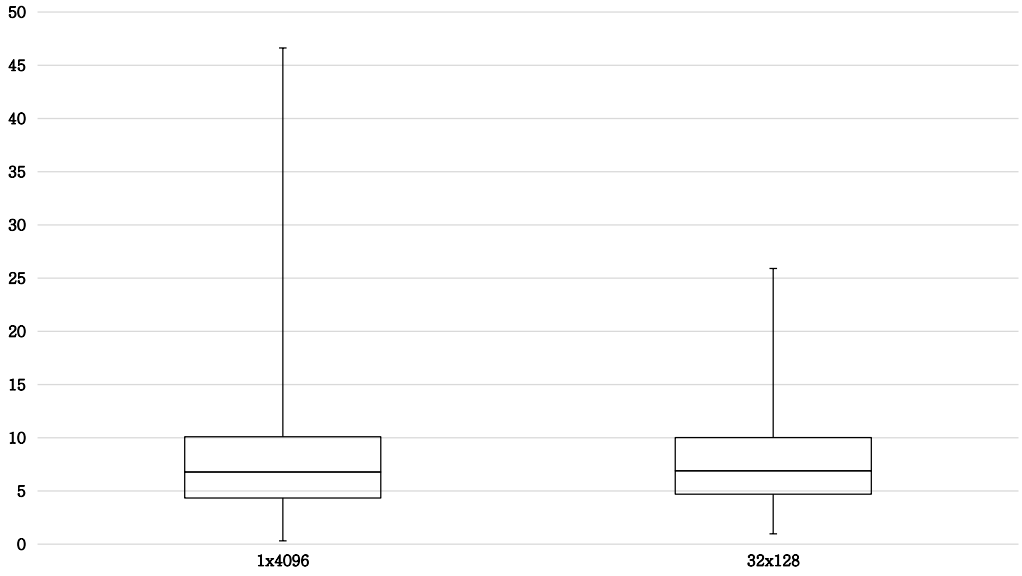


Figure 2: Training time and hidden layers versus population configuration (1b)

(a) MSE on the training set



(b) MSE on the test set

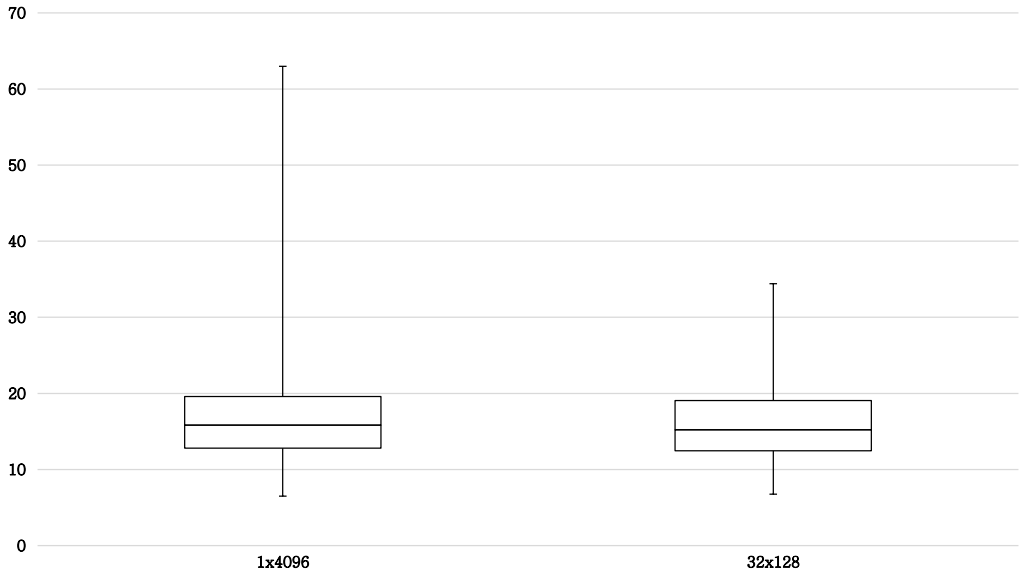
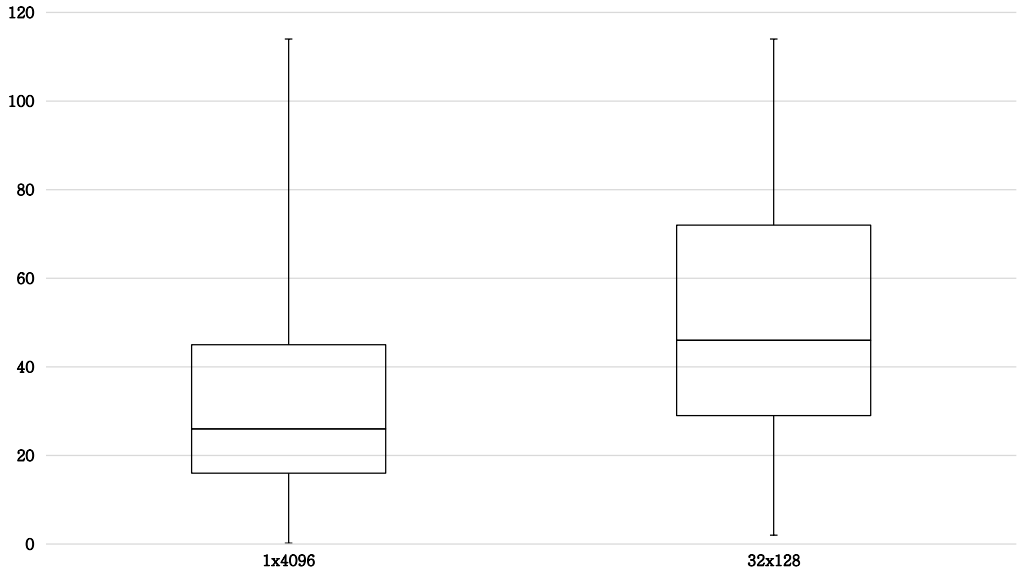


Figure 3: Training and test error versus population configuration (2a)

(a) Hidden layers



(b) Training time in seconds

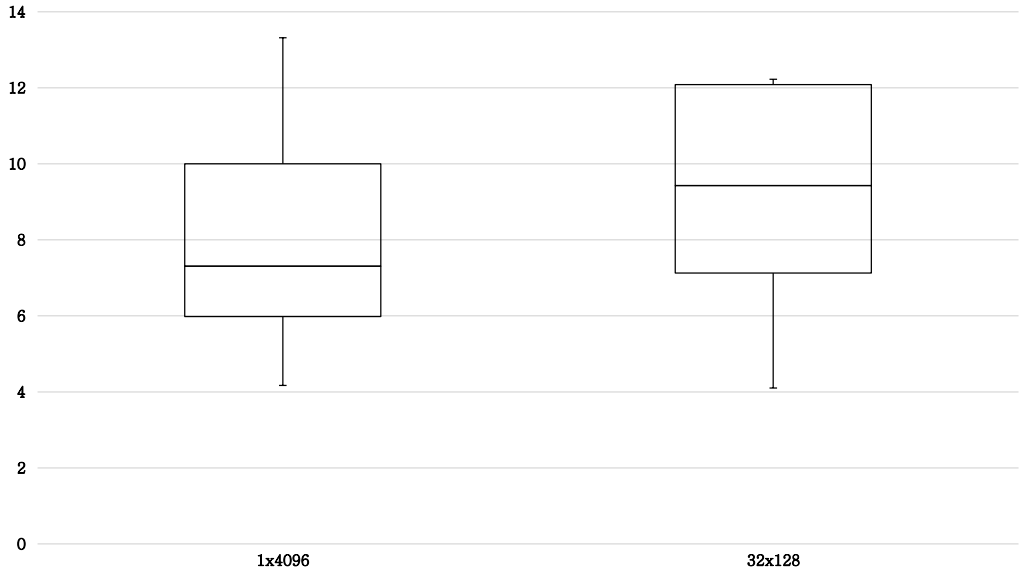


Figure 4: Training time and hidden layers versus population configuration (2b)

Legal Notice

Bibliography

[1] Computer-Aided Information Systems for Gaming.
http://www.dtic.mil/dtic/tr/fulltext/u2/623091.pdf.

[2] AI Center - Alexander Nareyek - Activities.
http://www.ai-center.com/home/alex/activities.html.

[3] AiGameDev.com | Your Online Hub for Game/AI.
http://aigamedev.com/.

[4] CRYENGINE | The complete solution for next generation game development by Crytek.
http://cryengine.com/.

[5] Programming Guide :: CUDA Toolkit Documentation.
http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] Dota 2 - Overview - The International.
http://www.dota2.com/international/overview/.

[7] Electronic Games Magazine (March 1982).
https://archive.org/details/electronic-games-magazine-1982-03.

[8] Soccer: World Cup money pot increased to $576m | Reuters.
http://www.reuters.com/article/2013/12/05/
us-soccer-world-prizemoney-idUSBRE9B40QG20131205.

[9] GameDev.net Game Development Community.
http://www.gamedev.net/page/index.html, .

[10] US government recognizes League of Legends players as pro athletes - GameSpot.
http://www.gamespot.com/articles/
us-government-recognizes-league-of-legends-players-as-pro-athletes/
1100-6411377/, .

[11] Season 3 World Championship - Leaguepedia - Competitive League of Legends Wiki.
http://lol.gamepedia.com/Season_3_World_Championship, .

[12] Havok.
http://www.havok.com/.

[13] Video Games and Artificial Intelligence - Microsoft Research.
http://research.microsoft.com/en-us/projects/ijcaiigames/, .

[14] TrueSkill™Ranking System: Details - Microsoft Research.
http://research.microsoft.com/en-us/projects/trueskill/details.aspx, .

[15] A Long Time Ago, in a Lab Far Away . . . - New York Times.
http://www.nytimes.com/2002/02/28/technology/a-long-time-ago-in-a-lab-far-away.html.

[16] 1939 New York World's Fair Home Page.
http://www.1939nyworldsfair.com/.

[17] NVIDIA Nsight Visual Studio Edition User Guide.
http://docs.nvidia.com/nsight-visual-studio-edition/4.1/Nsight_Visual_Studio_Edition_User_Guide.htm.

[18] Video Game History Timeline | The Strong.
http://www.museumofplay.org/icheg-game-history/timeline/.

[19] UCI Machine Learning Repository: Housing Data Set.
https://archive.ics.uci.edu/ml/datasets/Housing.

[20] Patent US2215544 - Machine to play game of nim - Google Patents.
http://www.google.com/patents/US2215544, .

[21] Patent US2455992 - Cathode-ray tube amusement device - Google Patents.
http://www.google.com/patents/US2455992, .

[22] Unity - Game Engine.
http://unity3d.com/.

[23] Unreal Engine Technology | Home.
https://www.unrealengine.com/.

[24] Wikipedia, the free encyclopedia. `http://en.wikipedia.org/`.

[25] A survey of game portability,.

[26] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

[27] D. W. Aha and M. Molineaux. Integrating learning in interactive gaming simulators. In *Challenges of Game AI: Proceedings of the AAAI04 Workshop*, 2004.

[28] D. W. Aha, M. Molineaux, and M. J. V. Ponsen. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR'05)*, pages 5–20, 2005.

[29] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1983.

[30] E. F. Anderson. Playing smart – artificial intelligence in computer games. 2003.

[31] E. F. Anderson. A NPC behaviour definition system for use by programmers and designers. 2004.

[32] E. F. Anderson. Scripting Behaviour – Towards a New Language for Making NPCs Act Intelligently. In *Proceedings of zfxCON05 2nd Conference on Game Development*, 2005.

[33] E. F. Anderson. SEAL – A Simple Entity Annotation Language. In *Proccedings of zfxCON05 2nd Conference on Game Development*, pages 70–73, 2005.

[34] E. F. Anderson. Scripted smarts in an intelligent virtual environment. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 185–188. ACM, 2008.

[35] E. F. Anderson. *On the Definition of Non-Player Character Behaviour for Real-Time Simulated Virtual Environments*. PhD thesis, Bournemouth University, 2008.

[36] E. F. Anderson. A Classification of Scripting Systems for Entertainment and Serious Computer Games. In *2011 Third International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 47–54, 2011.

[37] E. F. Anderson and C. E. Peters. No more reinventing the virtual wheel: Middleware for use in computer games and interactive computer graphics education. In *31st Annual Conference of the European Association for Computer Graphics*, 2010.

[38] E. F. Anderson, S. Engel, P. Comninos, and L. McLoughlin. The case for research in game engine architecture. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 228–231. ACM, 2008.

[39] W. W. Armstrong, M. Green, and R. Lake. Near-real-time control of human figure models. *Computer Graphics and Applications, IEEE*, 7 (6):52–61, 1987.

[40] N. I. Badler, B. A. Barsky, and D. Zeltzer. *Making them move: mechanics, control, and animation of articulated figures*. Morgan Kaufmann Publishers Inc., 1991.

[41] C. Baekkelund. Academic AI research and relations with the games industry. *AI Game Programming Wisdom*, 3:77–88, 2006.

[42] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the 1968 AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[43] J. Bates et al. The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125, 1994.

[44] C. Bauckhage, C. Thurau, and G. Sagerer. Learning human-like opponent behavior for interactive computer games. *Pattern Recognition*, pages 148–155, 2003.

[45] M. Bauer, S. Biundo, D. Dengler, H. Feibel, J. Koehler, and G. Paul. Reasoning about plans – a progress report. In *ECAI'96 Workshop on Cross-Fertilization in Planning*, 1996.

[46] R. Baumgarten, S. Colton, and M. Morris. Combining AI methods for learning bots in a Real-Time Strategy Game. *International Journal of Computer Games Technology*, 2009, 2008.

[47] C. Berndt, I. Watson, and H. Guesgen. OASIS: an open AI standard interface specification to support reasoning, representation and learning in computer games. In *IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 19–24, 2005.

[48] S. Björk and J. Holopainen. Describing Games – An Interaction-Centric Structural Framework. In *Level Up – Proceedings of Digital Games Research Conference*, 2003.

[49] S. Björk, L. Sus, and H. Jussi. Game Design Patterns. In *Level Up – Proceedings of Digital Games Research Conference*, 2003.

[50] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 47–54. ACM, 1995.

[51] R. A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural computation*, 1(2):253–262, 1989.

[52] M. Buckland. *Programming Game AI by Example*. Jones & Bartlett Learning, 2004. ISBN 1556220782.

[53] M. Buckland and M. Collins. *AI techniques for game programming*. Premier press, 2002.

[54] A. Canossa, S. Björk, and M. J. Nelson. X-COM: UFO Defense vs. XCOM: Enemy Unknown-Using Gameplay Design Patterns to Understand Game Remakes. *Foundations of Digital Games 2014*, 2014.

[55] D. C. Cheng and R. Thawonmas. Case-Based Plan Recognition for Real-Time Strategy Games. In *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE'04)*, pages 36–40, 2004.

[56] M. Chung, M. Buro, and J. Schaeffer. Monte-Carlo planning in real-time strategy games. In *Proceedings of the 1st IEEE Symposium on Computational Intelligence and Games (CIG-05)*, 2005.

[57] I. L. Davis. Strategies for strategy game AI. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 24–27, 1999.

[58] M. DeLoura. The engine survey: Technology results. *Gamasutra Expert Blogs*, 2009.

[59] R. B. D. I. M. Downie and Y. I. B. Blumberg. Creature smarts: The art and architecture of a virtual brain. 2001.

[60] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. *Dagstuhl Follow-Ups*, 6, 2013.

[61] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufmann, 1990.

[62] C. Fairclough, M. Fagan, B. Mac Namee, and P. Cunningham. Research directions for AI in computer games. Technical report, Trinity College Dublin, Department of Computer Science, 2001.

[63] M. Fernando and T. W.-T. Lee. Self organizing neural networks for the identification problem. In *Advances in Neural Information Processing Systems 1*, pages 57–64. Morgan Kaufmann, 1989.

[64] M. W. Floyd and B. Esfandiari. Toward a domain-independent case-based reasoning approach for imitation: Three case studies in gaming. In *Workshop on Case-Based Reasoning for Computer Games at the 18th International Conference on Case-Based Reasoning (ICCBR)*, pages 55–64, 2010.

[65] M. W. Floyd and B. Esfandiari. A case-based reasoning framework for developing agents using learning by observation. In *2011 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 531–538, 2011.

[66] F. Frandsen, M. Hansen, H. Sørensen, P. Sørensen, J. G. Nielsen, and J. S. Knudsen. Predicting player strategies in real time strategy games. Master's thesis, 2010.

[67] J. Funge. Representing knowledge within the situation calculus using interval-valued epistemic fluents. *Reliable computing*, 5(1):35–61, 1999.

[68] J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *Proceedings of the 26th annual conference on Computer graphics and interac-*

tive techniques, pages 29–38. ACM Press/Addison-Wesley Publishing Co., 1999.

[69] J. D. Funge. Making them behave: cognitive models for computer animation, 1998.

[70] J. D. Funge. *AI for games and animation: a cognitive modeling approach*. AK Peters Massachusetts, 1999.

[71] Q. Gemine, F. Safadi, R. Fonteneau, and D. Ernst. Imitative Learning for Real-Time Strategy Games. In *Proceedings of the 8th IEEE Conference on Computational Intelligence and Games (CIG'12)*, 2012.

[72] B. Gorman and M. Humphrys. Imitative learning of combat behaviours in first-person computer games. In *Proceedings of the 10th International Conference on Computer Games: AI, Mobile, Educational and Serious Games*, 2007.

[73] R. M. Gray. Vector quantization. *IEEE ASSP Magazine*, 1:4, 1984.

[74] L. K. Hansen and M. Pedersen. Controlled growth of cascade correlation nets. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 797–800, 1994.

[75] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI. In *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 63–68, 2005.

[76] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[77] R. Huebner. Adding languages to game engines. *Game Developer*, 4 (9), 1997.

[78] Interactive Software Federation of Europe. Video Games in Europe: Consumer Study, September 2012. Newzoo Trend Report.

[79] U. Jaidee, H. Muñoz-Avila, and D. W. Aha. Integrated learning for goal-driven autonomy. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*, pages 2450–2455. AAAI Press, 2011.

[80] F. V. Jensen and T. D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Science+Business Media LLC, 2007.

[81] D. Johnson and J. Wiles. Computer Games With Intelligence. In *FUZZ-IEEE*, pages 1355–1358, 2001.

[82] B. F. F. Karlsson. Issues and approaches in artificial intelligence middleware development for digital games and entertainment products. *CEP*, 50740:540, 2003.

[83] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[84] D. G. Korn. ksh: An extensible high level language. In *Very High Level Languages Symposium (VHLL)*, pages 129–146, 1994.

[85] A. Kovarsky and M. Buro. A first look at build-order optimization in real-time strategy games. In *Proceedings of the 2006 GameOn Conference*, pages 18–22, 2006.

[86] A. H. Kramer and A. Sangiovanni-Vincentelli. Efficient parallel learning algorithms for neural networks. In *Advances in Neural Information Processing Systems 1*, pages 40–48. Morgan Kaufmann, 1989.

[87] B. Kreimeier. The case for game design patterns, 2002.

[88] J. E. Laird and van Michale Lent. Human-level AI's killer application: Interactive computer games. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000.

[89] P. Lankoski and S. Björk. Gameplay design patterns for believable non-player characters. In *Situated Play: Proceedings of the 2007 Digital Games Research Association Conference*, pages 416–423, 2007.

[90] S. Lee-Urban, H. Muñoz-avila, A. Parker, U. Kuter, and D. Nau. Transfer Learning of Hierarchical Task-Network Planning Methods in a Real-Time Strategy Game. In *Proceedings of the 17th International Conference on Automated Planning & Scheduling (ICAPS'07) Workshop on AI Planning and Learning (AIPL)*, 2007.

[91] N. Li, D. J. Stracuzzi, G. Cleveland, T. Könik, D. Shapiro, M. Molineaux, D. Aha, and K. Ali. Constructing Game Agents from Video of Human Behavior. In *Proceedings of the 5th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'09)*, 2009.

[92] L. Lidén. Artificial stupidity: The art of intentional mistakes. *AI Game Programming Wisdom*, 2:41–48, 2003.

[93] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391 (6):2193–2196, 2012.

[94] M. I. A. Lourakis. A brief description of the Levenberg-Marquardt algorithm implemened by levmar. 2005.

[95] A. B. Loyall and J. Bates. Hap: A Reactive, Adaptive Architecture for Agents. Technical Report CMU-CS-97-123, Carnegie Mellon University, School of Computer Science, 1991.

[96] M. Mateas and A. Stern. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[97] M. Mateas and A. Stern. A Behavior Language: Joint Action and Behavioral Idioms. In *Life-Like Characters*, pages 135–162. Springer, 2004.

[98] J. McCarthy. Situations, actions, and causal laws. Technical report, DTIC Document, 1963.

[99] J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University USA, 1968.

[100] J. McCoy and M. Mateas. An integrated agent for playing real-time strategy games. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 1313–1318, 2008.

[101] Microsoft. Windows NT Hardware Abstraction Layer (HAL). `http://support.microsoft.com/kb/99588`.

[102] I. Millington and J. Funge. *Artificial Intelligence for Games*. CRC Press, 2009. ISBN 0123747317.

[103] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, pages 762–767. Morgan Kaufmann, 1989.

[104] J. L. Montana and A. J. Gonzalez. Towards a unified framework for learning from observation. In *IJCAI 2011 Workshop on Agents Learning Interactively from Human Teachers*, 2011.

[105] J. Moody. Fast learning in multi-resolution hierarchies. In *Advances in Neural Information Processing Systems 1*, pages 29–39. Morgan Kaufmann, 1989.

[106] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson. The 2003 Report of the IGDA's Artificial Intelligence Interface Standards Committee. Technical report, International Game Developers Association. `http://www.igda.org/ai/report-2003/report-2003.html`, 2003. Via `http://archive.org/web/`.

[107] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson. The 2004 Report of the IGDA's Artificial Intelligence Interface Standards Committee. Technical report, International Game Developers Association. `http://www.igda.org/ai/report-2004/report-2004.html`, 2004. Via `http://archive.org/web/`.

[108] A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson. The 2005 Report of the IGDA's Artificial Intelligence Interface Standards Committee. Technical report, International Game Developers Association. `http://www.igda.org/ai/report-2005/report-2005.html`, 2005. Via `http://archive.org/web/`.

[109] Newzoo. The Global Games Market, November 2013. European Summary Report.

[110] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Elsevier, 1998.

[111] C. M. Olsson, S. Björk, and S. Dahlskog. The Conceptual Relationship Model: Understanding Patterns and Mechanics in Game Design. In *Proceedings of the 2014 DiGRA International Conference (DiGRA'14)*, 2014.

[112] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram. Case-Based Planning and Execution for Real-Time Strategy Games. In *Proceedings of the 7th International Conference on Case-Based Reasoning (ICCBR'07)*, pages 164–178, 2007.

[113] J. Orkin. Symbolic representation of game world state: Toward real-time planning in games. In *AAAI Workshop on Challenges in Game AI*, 2004.

[114] J. Orkin. Agent Architecture Considerations for Real-Time Planning in Games. In *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 105–110, 2005.

[115] K. Perlin and A. Goldberg. IMPROV: A system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 205–216. ACM, 1996.

[116] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha. Automatically generating game tactics via evolutionary learning. *AI Magazine*, 27 (3):75–84, 2006.

[117] M. A. Potter. A genetic cascade-correlation learning algorithm. In *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 123–133. IEEE Computer Society Press, 1992.

[118] D. Pottinger. Implementing coordinated movement. *Game Developer Magazine*, pages 48–58, 1999.

[119] S. Rabin. The magic of data-driven design. 2000.

[120] S. Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002. ISBN 1584500778.

[121] S. Rabin. *AI Game Programming Wisdom 2*. Cengage Learning, 2003. ISBN 1584502894.

[122] S. Rabin. *AI Game Programming Wisdom 3*. Cengage Learning, 2006. ISBN 1584504579.

[123] S. Rabin. *AI Game Programming Wisdom 4*. Charles River Media, 2008. ISBN 1584505230.

[124] G. Robertson and I. Watson. A Review of Real-Time Strategy Game AI. 2014.

[125] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009. ISBN 0136042597.

[126] F. Safadi. parallel-cc. `http://www.montefiore.ulg.ac.be/~fsafadi/parallel-cc.7z`, 2014.

[127] F. Safadi. Bitonic Sort. `http://www.montefiore.ulg.ac.be/~fsafadi/bitonic.pdf`, 2014.

[128] F. Safadi, R. Fonteneau, and D. Ernst. Artificial Intelligence Design for Real-Time Strategy Games. In *Proceedings of the 25th Conference on Neural Information Processing Systems (NIPS 2011) Workshop on Decision Making with Multiple Imperfect Decision Makers*, 2011.

[129] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[130] T. Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[131] R. B. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *AAAI*, volume 93, pages 689–695, 1993.

[132] B. Schwab. *AI Game Engine Programming*. Cengage Learning, 2008. ISBN 1584505729.

[133] D. Shapiro, T. Könik, and P. O'Rorke. Achieving Far Transfer in an Integrated Cognitive Architecture. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 1325–1330, 2008.

[134] M. Sharma, M. Holmes, J. Santamaria, A. Irani, C. Isbell, and A. Ram. Transfer Learning in Real-time Strategy Games Using Hybrid CBR/RL. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI'07)*, pages 1041–1046, 2007.

[135] R. Straatman and A. Beij. Killzone's AI: dynamic procedural combat tactics. In *Game Developers Conference*, 2005.

[136] P. Sweetser and J. Wiles. Current AI in Games: A review. *Australian Journal of Intelligent Information Processing Systems*, 8(1): 24–42, 2002.

[137] I. V. Tetko and A. E. P. Villa. An enhancement of generalization ability in cascade correlation algorithm by avoidance of overfitting/overtraining problem. *Neural Processing Letters*, 6(1-2):43–50, 1997.

[138] The NetBSD Foundation. Portability and supported hardware platforms.
`http://netbsd.org/about/portability.html`.

[139] C. Thurau, G. Sagerer, and C. Bauckhage. Imitation learning at all levels of game AI. In *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education*, 2004.

[140] J. Togelius, G. N. Yannakakis, S. Karakovskiy, and N. Shaker. Assessing believability. *Believablility in Computer Games*, 2011.

[141] I. Umarov, M. Mozgovoy, and P. C. Rogers. Believable and effective AI agents in virtual worlds: Current state and future perspectives. *International Journal of Gaming and Computer-Mediated Simulations*, 4(2):37–59, 2012.

[142] P. Wavish. Situated action approach to implementing characters in computer games. *Applied Artificial Intelligence*, 10(1):53–74, 1996.

[143] B. L. Webber, C. B. Phillips, and N. I. Badler. Simulating humans: Computer graphics, animation, and control. *Center for Human Modeling and Simulation*, page 68, 1993.

[144] B. Weber and M. Mateas. Conceptual Neighborhoods for Retrieval in Case-Based Reasoning. In *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR'09)*, pages 343–357, 2009.

[145] B. Weber and M. Mateas. Case-Based Reasoning for Build Order in Real-Time Strategy Games. In *Proceedings of the 5th Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE'09)*, 2009.

[146] B. G. Weber and M. Mateas. A data mining approach to strategy prediction. In *Proceedings of the 5th IEEE Symposium on Computational Intelligence and Games (CIG-09)*, pages 140–147. IEEE Press, 2009.

[147] B. G. Weber, M. Mateas, and A. Jhala. Building Human-Level AI for Real-Time Strategy Games. In *AAAI Fall Symposium: Advances in Cognitive Systems*, 2011.

[148] S. Wender and I. Watson. Combining Case-Based Reasoning and Reinforcement Learning for Unit Navigation in Real-Time Strategy Game AI. In *Case-Based Reasoning Research and Development*, pages 511–525. Springer, 2014.

[149] M. West. Domain-Specific Languages. *Game Developer*, 14(7):33–36, 2007.

[150] B. Wilcox. Reflections on Building Three Scripting Languages, 2007.

[151] V. Ye. A Rule-based Approach to Animating Multi-agent Environments. 1996.

[152] B. Yue and P. de Byl. The state of the art in game AI standardisation. In *Proceedings of the 2006 international conference on Game research and development*, pages 41–46. Murdoch University, 2006.

[153] S. Zilberstein and S. J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, volume 93, pages 1402–1407, 1993.