UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ADAPTATION AND LEARNING OF INTELLIGENT AGENTS IN INTERACTIVE ENVIRONMENTS

Daniel Álvaro Fonseca Policarpo

MESTRADO EM ENGENHARIA INFORMÁTICA
Interacção e Conhecimento

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ADAPTATION AND LEARNING OF INTELLIGENT AGENTS IN INTERACTIVE ENVIRONMENTS

Daniel Álvaro Fonseca Policarpo

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor Paulo Jorge Cunha Vaz Dias Urbano

e co-orientado por Pedro Jorge da Costa Amado

MESTRADO EM ENGENHARIA INFORMÁTICA

Interacção e Conhecimento

2011

Acknowledgements

First of all, I would like to thank Dr. Paulo Urbano and Tiago Loureiro for their assistance and support during this project.

I would also like to thank my co-workers at *vectrLab* for their friendship and for providing an enjoyable working environment.

My friends, colleagues and ex-colleagues, without mentioning names, who helped me grow professionally and personally, also deserve my appreciation and respect.

Many thanks go to my girlfriend Ana, for her patience, love and support over the years.

I cannot end without thanking my family, on whose constant encouragement and love I have relied throughout my time at the university.

To my parents, brothers and to Ana.

Abstract

Videogame industry evolved from a niche market to a globally recognized opportunity for entrepreneurship and profit. Presently, it is one of the economic sectors that generate more jobs and capital, even competing with the cinematographic industry. As technology evolves, videogames become more appealing both visually and technically, taking full advantage of the top features technology has to offer. Since more appealing videogames provide better profit from sales, this industry is one of the driving forces for technological evolution.

Videogame Artificial Intelligence is growing more complex and realistic to keep up with player requirements. Despite this, most games still fail to provide true adaptability in their behaviors, resulting in situations where an intermediate level player is able to predict the non-player opponents' behavior in a short amount of time, leading to a predictable and boring game experience. Creating a truly adaptive Videogame Artificial Intelligence would greatly benefit a videogame's intrinsic value by providing a more immersive and unpredictable game experience.

This document describes the research and development of an Artificial Intelligence system for the First-Person Shooter videogame genre. After a period of research for related work and state-of-art, we decided to adopt the Dynamic Scripting technique as a basis for the Artificial Intelligence system, so as to create adaptable non-player opponents that provide more unpredictability and challenging characters to commercial videogames.

Dynamic Scripting is a technique for machine learning of behaviors for videogame characters that maintains several "rulebases", one for each type of agent in the videogame, from which rules are extracted to create scripts that control agent behaviors. These rulebases adapt to the players' actions, learning which rules translate into better performance scripts. After a number of validation tests, this technique was implemented on First-Person Shooter scenarios that were previously constructed with the Unity3D videogame engine. I also implemented a number of extensions for the Dynamic Scripting technique, namely, improvements to the original technique and a Goal Oriented approach to behavior selection.

Keywords: Adaptation; Machine-Learning; Videogame Artificial Intelligence; Rule-based behaviors; Goal Oriented behaviors

Resumo

A indústria de videojogos evoluiu de um mercado de nicho para um mercado globalmente reconhecido pelas suas oportunidades de empreendedorismo e lucro. Actualmente é um dos sectores económicos que gera mais emprego e capital, ultrapassando o volume de facturação da indústria cinematográfica e competindo até com a indústria musical. Com a evolução da tecnologia, os videojogos tornam-se cada vez mais apelativos, tanto visualmente como técnicamente. Uma vez que jogos mais apelativos providenciam melhores resultados nas vendas, esta indústria é uma das forças motrizes para a evolução tecnológica das plataformas físicas de videojogos.

A Inteligência Artificial nos videojogos é cada vez mais complexa e realista, de modo a acompanhar as exigências dos jogadores. Apesar disto, a maioria dos jogos ainda não fornece verdadeira adaptabilidade nos comportamentos dos seus personagens, resultando em situações em que um jogador de nível intermédio é capaz de prever o comportamento dos adversários num curto espaço de tempo, levando a uma experiência de jogo previsível e aborrecida. Criar uma Inteligência Artificial verdadeiramente adaptável beneficiaria muito o valor intrínseco de um videojogo, fornecendo uma experiência de jogo mais envolvente e imprevisível.

Este documento descreve a pesquisa e desenvolvimento de um sistema de Inteligência Artificial para o género de videojogos First-Person Shooter. Após um período de investigação sobre trabalhos relacionados e o estado-da-arte, decidiu-se adoptar a técnica Dynamic Scripting como base para o sistema, permitindo a criação de adversários com comportamentos adaptáveis, que fornecem mais imprevisibilidade e desafio em videojogos comerciais.

Dynamic Scripting é uma técnica de Inteligência Artificial para aprendizagem de comportamentos para videojogos, que mantém várias bases de dados de regras, uma para cada tipo de agentes no videojogo, a partir das quais são extraídas as regras utilizadas para controlam os comportamentos dos agentes. Estas bases de regras adaptam-se às acções dos jogadores, aprendendo quais as regras que traduzem em melhor desempenho do comportamento do agente. Após uma série de testes de validação, esta técnica foi implementada em cenários típicos de videojogos do género First-Person Shooter, construídos previamente com a utilização do motor de jogo *Unity3D*. Para além disso, implementaram-se uma série de extensões para a técnica Dynamic Scripting, nomeadamente, melhorias na técnica original e uma abordagem de construção de comportamento orientada para objectivos.

Palavras-chave: Adaptação; Aprendizagem; Inteligência Artificial em videojogos; Comportamentos baseados em regras; Comportamentos orientados para objectivos.

Index

Figure List ... ix

Table List ... xi

Chapter 1 Introduction ... 1

1.1 Motivation ... 1

1.2 Objectives ... 2

1.3 Scientific Contributions ... 2

1.4 VectrLab ... 3

1.5 Planning ... 3

1.6 Document's Organization ... 4

Chapter 2 Background and Related Work ... 6

2.1 Videogame Basics ... 6

2.1.1 Definition ... 6

2.1.2 History ... 7

2.1.3 Videogame Genres and Types ... 8

2.2 First-Person Shooter Genre ... 9

2.3 Videogame Artificial Intelligence ... 10

2.3.1 History ... 10

2.3.2 Videogame AI *vs* Academic AI ... 11

2.3.3 Machine Learning in Videogame Environments ... 12

2.3.4 Machine Learning in First-Person Shooter Environments ... 13

Chapter 3 Dynamic Scripting ... 15

3.1 Basic Mechanics ... 15

3.1.1 Rules and Rulebases ... 16

3.1.2 Script Selection ... 17

3.1.3 Rule Policy ... 17

3.1.4 Rule Weight Updating ... 18

3.2 Central Characteristics ... 20

3.3 Extensions to Dynamic Scripting 21
3.3.1 Automatic Rule Ordering 21
3.3.2 Goal-Directed Dynamic Scripting 22
Chapter 4 Implementation of Dynamic Scripting in First-Person Shooter Scenarios 25
4.1 Methodology 25
4.1.1 Unity3D 25
4.2 Implementation of the Basic Components 26
4.2.1 Rules and Rulebases 26
4.2.2 Script Selection 26
4.2.3 Rule Policy 27
4.2.4 Rule Weight Updating 27
4.3 Experimental Evaluation 29
4.3.1 Registering Results 30
4.3.2 First Scenario 30
4.3.3 First Scenario Results 34
4.3.4 First Scenario Results Discussion 39
4.3.5 Second Scenario 40
4.3.6 Second Scenario Results 42
4.3.7 Second Scenario Results Discussion 48
Chapter 5 Conclusions 51
5.1 Future Work 52
Bibliography 53
Appendix A 56

Figure List

2.1 Atari's *Pong* Gameplay 7

2.2 Typical view of a FPS, *Wolfenstein 3D* 10

3.1 General representation of the Dynamic Scripting technique 16

4.1 Representation of Scenario 1, with a label describing each element 31

4.2 Fitness average of Simple DS for Scenario 1 34

4.3 Weight average of SidestepRocketAttack rule on Simple DS for Scenario 1 35

4.4 Fitness average of DS with Automatic Rule Ordering extension for Scenario 1 36

4.5 Fitness average of DS with the Goal-Directed extension for Scenario 1 37

4.6 Fitness average of Extended DS for Scenario 1 38

4.7 Weight average of SidestepRocketAttack rule in Extended DS for Scenario 1 39

4.8 Representation of Scenario 2, with a label describing each element 41

4.9 Fitness average of Simple DS for Scenario 2 43

4.10 Weight average of AdvanceRocketAttack rule in Simple DS for Scenario 2 44

4.11 Fitness average of DS with Automatic Rule Ordering extension for Scenario 2 45

4.12 Fitness average of DS with Goal-Directed extension for Scenario 2 46

4.13 Fitness average of Extended DS for Scenario 2 47

4.14 Weight average of HuntPathA rule in Extended DS for Scenario 2 48

Table List

3.1 Representation of a relation-weight table 21

4.1 List of rules used in Scenario 1 34

4.2 List of additional rule used in Scenario 2 42

Chapter 1

Introduction

This chapter is organized as follows: the next Section (Section 1.1) describes the motivation behind the work developed; in Section 1.2, I present the objectives of this work; scientific contributions are described in Section 1.3; Section 1.4 introduces the external institution where this work was carried out, and describes the integration with this institution; Section 1.5 describes the original plan for the completion of the work and the adjustments made to it; finally, Section 1.6 describes this document's organization.

1.1 Motivation

Ever since the Atari company developed in 1972 a simple ping-pong simulation called *Pong*, triggering the huge boom of the videogame industry, videogames evolved tremendously. Nowadays, the videogame industry moves millions of workers and currency, generating profit that competes with the top economic sectors. Modern videogames require specific hardware to cope with the huge amount of calculations necessary for special effects and detailed environments, therefore working as a driving force for technological advancement.

As better technology is applied in videogame production, players' expectations of game environments rich in detail and credibility raise. Graphic wise, the industry is answering to said expectations, delivering videogames that are visually and technically appealing. However, the application of artificial intelligence (AI) in videogames is currently under a constant need to evolve in an attempt to meet the growing demand of players for realism, immersion and credibility in the behavior of characters in the game environment. One common aspect of current videogame AI is the lack of learning processes: the behavior of game agents is usually static and thus unable to adapt to the player. A major disadvantage of non-adaptive videogame AI is that once a weakness is discovered, nothing stops the human player from exploiting it, leading to a predictable and boring game experience.

In general, research in AI has been mainly made by the academic sector, with characteristics and objectives very different from the business sector in which videogames are located. From the variety of AI techniques currently available, only some are suitable for use in videogames, often due to resource constraints imposed by market requirements. The dynamic and interactive environments of videogames can become good platforms for the research and testing of new AI techniques by the academic community [1,2], even if some of the techniques are not commercially implemented by developers. Creating a truly adaptive videogame AI that learns through the interaction with the player, without forgetting the constraints of limited resources, is a challenge already accepted by a number of academic researchers. The application of such techniques in videogames, and therefore its success, is only dependent on the interest shown by the most influential companies in the market.

1.2 Objectives

In simulated synthetic environments such as videogames, creating intelligent agents equipped with the ability to learn from the reactions of human players becomes increasingly necessary.

Therefore, this dissertation's goal is the application of the Dynamic Scripting (DS) technique [3] in the First-Person Shooter (FPS) videogame genre [4]. This technique, created by Pieter Spronck, is already applied and validated for three different videogame genres, namely Computer Role-Playing [3], Real-Time Strategy [5, 6] and Turn-Based Strategy [7]. As far as we know, there is no application yet to the FPS genre. As such, it is necessary to construct virtual scenarios that validate and test this technique in that particular videogame genre. These scenarios depict an adaptive character controlled by the DS technique against a static character controlled by a Finite-State Machine. In addition to the basic DS technique, two extensions are applied to better accommodate the technique for the FPS videogame genre, namely the Automatic Rule Ordering [8] extension and the Goal-Directed [5] extension. The resulting AI system is meant to be used in future First-Person Shooter commercial videogames produced by *vectrLab*, the company that sponsored this work.

1.3 Scientific Contributions

A contribution for the scientific community was written in the form of a paper. This paper was submitted and accepted in the second Workshop on Intelligent Systems and Applications (WISA), a workshop inserted in the fifth Conferência Ibérica de

Sistemas e Tecnologias de Informação (CISTI). The paper [9], with the title *Dynamic Scripting Applied in a First-Person Shooter*, was presented on 19th June 2010, at Santiago de Compostela, Spain, and is available for consultation in Appendix A.

1.4 VectrLab

The work described in this document took place in *vectrLab*, a Portuguese company located in *Instituto de Ciência Aplicada e Tecnologia* (ICAT), on the *Faculdade de Ciências da Universidade de Lisboa* Campus.

In 2007, Tiago Loureiro and Pedro Amado, the company's founding members, shaped the idea of *vectrLab*, a company that would market the areas of entertainment and interactivity. The main objective was to be one of the top companies in those areas and exceed the expectations of its customers through its services and innovative and high quality products.

By winning the IAPMEI / ANJE "Ideia" award for the best idea for an innovative company, the project gained the attention of *InovCapital*, a Portuguese venture capital company where innovation and high technology are the main areas of investment.

The company was established in February 2008 with the mission of continuous quality improvement of its services and products through planning and investment in R & D efficiency, with the premise to never forget the satisfaction of their customers.

The newly created company has a partnership with the *Laboratório de Modelação de Agentes* (LabMag), a research unit located at FCUL. Within four months after its creation, the company received funding from *InovCapital* because of its recognition as an emerging company with innovative potential.

Currently, *vectrLab* is developing interactive content for touchscreen systems on behalf of the *Comissão Nacional para as Comemorações do Centenário da República* (CNCCR), having as theme the centenary of the Portuguese Republic, as well as the remake of an original, yet to be released, ZX Spectrum iconic game.

1.5 Planning

The work described in this document started in October, 2009. In the beginning a plan was made and presented in the preliminary report. The initial plan for this thesis was the following:

1. October 2009 - November 2009: Problem analysis and state of art of the application of adaptation and learning in videogame characters. Familiarization with the prototyping tools. Writing of the preliminary report;
2. December 2009 - January 2010: Definition of the experimental test-bed. Research for the application of machine learning techniques in videogames;
3. February 2010: Construction of the experimental test-bed;
4. March 2010: Testing and debugging of the experimental test-bed;
5. April 2010-May 2010: Acquisition of results from the test-bed. Application of the results obtained from the test-bed to a FPS videogame developed by *vectrLab*;
6. June 2010: Writing of this report;

As the work progressed, modifications to the initial plan were necessary. Some aspects took longer than expected, such as the familiarization with the prototyping tool, the construction and testing of the experimental scenarios; other aspects were added, such as the writing and submission of a paper for an international conference, the attendance to this conference; and, finally, some aspects were revised.

After the research for the state of art in machine learning applied to videogames, I proposed to *vectrLab* the application of the Dynamic Scripting technique as my main goal for the thesis. This technique has a number of features both scientifically interesting and commercially practical, as is explained in chapter 4.

Since the work took longer than expected, and the FPS videogame that *vectrLab* is developing is not ready for publication, the application of the Dynamic Scripting technique in a commercial videogame was left to future work.

1.6 Document's Organization

This document is organized as follows:

Chapter 2 - Background is given on certain concepts of videogames and their history and terminology, as well as on videogame AI.

Chapter 3 - Related work by other authors is presented and described.

Chapter 4 - Dynamic Scripting technique is described in detail.

Chapter 5 - The implementation of Dynamic Scripting is described along with the methodology adopted throughout the duration of the work, and a description of the tools used for its construction.

Chapter 6 - The experimental results obtained are presented and discussed.

Chapter 7 - Conclusions and possible future work are presented.

Chapter 8 - Referenced bibliography is presented in this chapter.

Chapter 2

Background and Related Work

The goal of this chapter is to provide relevant background information for this thesis, as well as provide a brief description of some of the major works that focus on machine learning and adaptive AI techniques applied to videogames in general, and to FPS in particular.

The next Section briefly introduces videogame concepts, genres and history. In Section 2.2, the First-Person Shooter genre is described in more detail, and Section 2.3 has a brief introduction to Videogame AI history, as well as descriptions of the major works in machine learning in Videogames.

2.1 Videogame Basics

Videogame industry is one of the economic sectors that move the most capital and job opportunities. From a *niche* market to a globally recognized one, this industry continues to drive forth technological advancements with its profit and entrepreneurship opportunities.

2.1.1 Definition

Videogames are variously referred to as "computer games", "electronic games", and even "digital entertainments" [4]. These terms cannot be taken to be strictly synonymous: "computer game" refers to game on a personal computer; "electronic games" might also refer to toys; while "digital entertainments" is sometimes used to refer exclusively to console games such as those on the X-box 360 or Playstation 3. This report will adopt "videogames" as the general term because it is the term that dominates current usage and the one that the industry has adopted.

Therefore, a videogame is an electronic game that involves interaction between a person and an electronic system, which typically generates visual feedback on a video device. The electronic systems used to play videogames are known as platforms, with personal computers and videogame consoles being examples of the former. These platforms range from large mainframe computers to small handheld devices.

2.1.2 History

The first videogame that was commercially successful, although not the first videogame ever created, was *Pong*, by the Atari company (Figure 2.1). *Pong* was created by Allan Alcorn as a training exercise assigned to him by Atari founder Nolan Bushnell. Soon after its release, several companies began producing games that copied *Pong*'s gameplay, and in time released new types of games. As a result, Atari encouraged its staff to produce more innovative games.

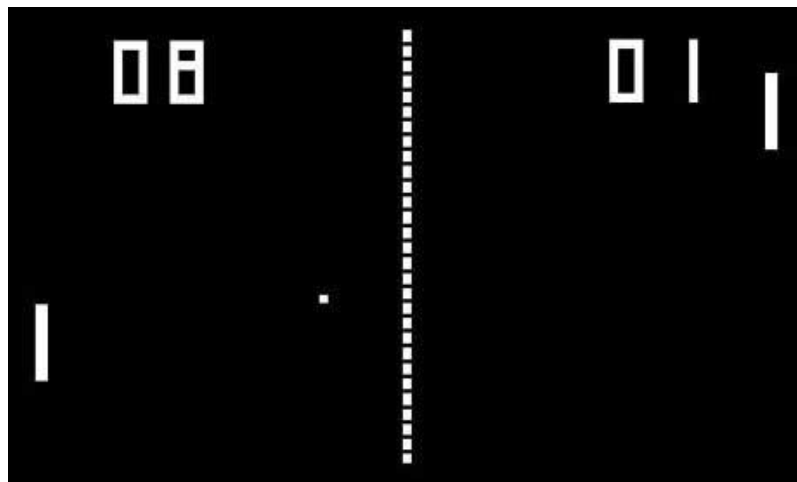

Figure 2.1: Atari's *Pong* gameplay.

While originally game developers needed to support a wide variety of computers, in the mid-1980s the IBM-PC became the industry standard for home computing and thus for home gaming. Videogame consoles also became popular, starting with the Nintendo Entertainment System in 1986 [10].

With the continuous advances in technology, processing power and the increasing capabilities of home computing, videogames became more and more complex. Development teams grew accordingly, from teams with five people in the 1980s to teams of hundreds in the 1990s, and with them grew also the cost of producing a single videogame. In the late 1990s specialized 3D video cards became affordable and

widespread, freeing up processing power for other gameplay features, such as artificial intelligence [11].

Nowadays the videogame industry has grown to surpass the Hollywood movie industry in revenues [12, 13] and the market for videogames is primarily made up of large development companies, supported by wealthy publishers.

2.1.3 Videogame Genres and Types

Videogames can be categorized into genres based on many factors such as methods of gameplay, types of goals, art styles and more [14]. Because genres are dependent on content for definition, they have changed and evolved as newer styles of videogames have come into existence. Nevertheless, genres are classified independently of the game setting of world content, since an action game is still an action game, regardless of whether it takes place in a fantasy world or outer space [15]. A general description of a number of different genres is presented below:

- **Action** - An action game requires players to use quick reflexes, accuracy, and timing to overcome obstacles. It is perhaps the most simple of gaming genres, and certainly one of the broadest. Action games tend to have gameplay with emphasis on combat.

- **Adventure** - Unlike adventure films, adventure games are not defined by story or content. Rather, adventure describes a manner of gameplay without reflex challenges or action. They normally require the player to solve various puzzles by interacting with people or the environment, most often in a non-confrontational way.

- **Role-Playing** - In role-playing games, the player typically assumes the role of a game character and is sent on quests, which mainly involve exploration and combat. These games are usually story-driven, with player decisions affecting the gameplay and storyline.

- **Simulation** - Simulation games require players to observe and interact with a simulation, and have two main types: vehicle simulations and management simulations.

- **Strategy** - Strategy videogames focus on gameplay requiring careful and skillful thinking and planning in order to finish each level or achieve a specific goal. In most strategy videogames the player is given a godlike view of the game world, indirectly controlling the units under his command.

Videogames can also be classified into types based on a number of factors: Hardcore games (or simply core games) are generally defined by their intensity, depth of play or scale of production involved in their creation; Casual games are defined by their ease of accessibility, simple-to-understand gameplay and quick to grasp rules; Serious games are designed for a primary purpose other than pure entertainment, like conveying information or a learning experience of some sort to the player.

2.2 First-Person Shooter Genre

First-Person Shooters (FPS) are videogames that feature a first-person point of view (hence the name) in which the player sees the action through the eyes of the player's character [15], allowing the player to focus on aiming. This genre can be classified as a subgenre of action games, as the primary design element is combat, mainly involving firearms. There are different environments where the action takes place, called arenas or maps. During the game, a player can pick up different items to improve the abilities of the character. Typically, the goal of a FPS is to stay alive as long as possible and to kill as many opponents as possible.

Most FPS's are very fast-paced and require quick reflexes, where the game environment is typically represented in 3D, with great emphasis in realism. Components such as gravity, light and collisions give the player a great sense of immersion, which contributes to a better gaming experience.

Since the first successively commercial FPS videogame released in 1992, named *Wolfenstein 3D* (Figure 2.2), this genre is also used to demonstrate the state-of-art of technical advances in gaming platforms, particularly their graphical component.

In the 21st century, the FPS is one of the biggest and fastest growing videogame genres, receiving awards for their artistry, narrative, innovation and graphic quality.

Figure 2.2: Typical view of a FPS, *Wolfenstein 3D*.

2.3 Videogame Artificial Intelligence

Artificial Intelligence (AI) has been a standard feature in videogames since the birth of the videogame industry. It encompasses many subject areas such as interaction, path finding, machine learning, flocking, formations, difficulty scaling and decision making [2, 11, 12].

The area of Artificial Intelligence is extremely vast and has inspired a great amount of research since its inception sixty years ago. Since this thesis is in relation to AI techniques applied specifically to commercial videogames, I focused my research on more recent studies.

2.3.1 History

The first videogames developed in the 1970s were implemented on discreet logic and based on competitions for two players, without involving AI. The earliest real artificial intelligence in gaming was the computer opponent in *Pong* or variations thereof (of which there were many) [11]. The computer paddle would do its best to block the ball from scoring by hitting it back at the user. Determining where to move the paddle was accomplished by a simple equation that would calculate at exactly what height the ball would cross the goal line and move the paddle to that spot as fast as allowed. Depending on the difficulty setting, the computer might not move fast enough to get to the spot or may just move to the wrong spot with some probability.

At the beginning of the 1990s, games gradually moved away from basic and heavily scripted character behavior. With more powerful microprocessors being

developed at the time, programmers started using Finite State Machine techniques in their in-game AI [11].

Since the second half of the 1990s non-deterministic AI methods were used. The game *Goldeneye 007*, released in 1997, was one of the FPS to use characters that would react to players' movements and actions. They could also take cover and dodge fire. Although the AI was unfair, because the characters knew where the player was at all times, the game was far more realistic than other games in the same genre [11]. In 1999, two popular FPS were released. First, on November 30, *Unreal Tournament* was released by Epic Games. The game really focused on multiplayer action. The AI was scripted, which means that scenarios are written out for the characters. This results in predictable behavior and actions of the characters [12]. Ten days later Id Software released *Quake III Arena*. The AI for this game used fuzzy logic to make decisions, thereby making sure that the character doesn't exhibit strictly predictable behavior [12].

Later games have used bottom-up AI methods, like the emergent behavior and evaluation of player actions in games like *Creatures* or *Black & White* [2].

2.3.2 Videogame AI *vs* Academic AI

There is an important distinction to be made between the AI studied by the academia and that used in videogames. While typical academic research in AI focuses in solving problems optimally, with less emphasis on hardware or time limitations, videogame AI programmers have to work with limited resources, making compromises and, more often than not, design AI to be suboptimal, with entertainment of the player as primary goal. Most players will quickly become frustrated if they face an opponent that is controlled by an optimal AI that always wins, therefore, in order to create an enjoyable gameplay experience to the average player, the videogame AI must be challenging, even though losing more often than triumph [16, 17, 18].

Although videogame AI is of interest to academia and game developers, there is little communication between these groups [19]. Game developers complain that academics fail to help them with the practical implementation of videogame AI [11,20] and academics claim to be held back in game development because of industry secrets, tight schedules and lack of funding [19,16].

2.3.3 Machine Learning in Videogame Environments

Reinforcement learning is an approach to artificial intelligence that emphasizes learning by the agent from its interaction with its environment, receiving positive or negative reinforcements. It has provided an inspiration to the development of a number of approaches for the generation of policies for both game agent control [21,22,23,24] and game agent strategies [25,3], however, the algorithms used have been based on the concept of reinforcement learning, rather than the explicit use of reinforcement learning algorithms. In the work of Szita and Lõrincz [23, 24] game agent control policies have been successfully generated for the digital games *Tetris* [23] and *Ms. Pac-Man* [24]. In [23] value functions, represented by linear combinations of basis function, were iteratively learned by applying a modified version of the Cross-Entropy Method (CEM) optimization algorithm [24] in order to determine the weights of the basic functions. Similarly, in [24] CEM has been used to determine a control policy, comprising a prioritized rulebase learned from sets of predefined rules associated with observations and corresponding actions for both the game agent and the game environment.

Reinforcement learning has also inspired the development of Dynamic Scripting by Spronck et al. [25,3] for the online creation and adaptation of rule based policies, using a self-play mechanism, within the role-playing game *Neverwinter Nights*. This technique is the basis of this thesis' work, and is explained further in the next chapter (Chapter 3). Dahlbom and Niklasson [5] proposed a goal-directed hierarchical extension to the dynamic scripting approach for incorporating learning into real-time strategy games, where goals are used as domain knowledge for selecting rules (a rule is seen as a strategy for achieving a goal). This particular work got our attention, and a goal-oriented component for our FPS application was later developed (explained in more detail in Chapter 4).

The concept of reinforcement learning through explicit player guided supervision, rather than self-play, was used in the development of the combat training game NERO in research conducted by Stanley et al. [21,22], though evolutionary neural networks were used to implement the real-time reinforcement learning mechanism. In this game a team of agents can be trained for combat by human players. The real-time version of the NEAT algorithm (rtNEAT) was used in this game. Developers show that this technique is flexible and robust enough to support real-time interactive learning of decision tasks.

In [26], the authors proposed to create non-player characters that can evolve and adapt through motivated reinforcement learning agents in the commercial videogame *Second Life*. In their paper, the authors explain that motivated learning agents are meta-learners which use a motivation function to provide a standard reinforcement learning algorithm with an intrinsic reward signal that directs learning. This function uses

domain independent rules based on the concept of interest in order to calculate an intrinsic motivation signal. Motivated reinforcement learning agents explore their environment and learn new behaviors in response to interesting experiences, allowing them to display progressively evolving behavioral patterns.

2.3.4 Machine Learning in First-Person Shooter Environments

FPS videogames have received attention as a machine learning test-bed due to their popularity and applicability as a model for real-life situations. After the release of the source code of *Quake 3 Arena* videogame and the test bed *GameBots* [27] for *Unreal Tournament*, a lot of research has been done on AI in FPS videogames. Laird used the cognitive architecture SOAR to implement anticipation in a *Quake 3 Arena* character [28]. The SOAR character simulated the perception of human players, like visual and auditory information. The character was able to predict the behavior of the enemy. A disadvantage of this technique is that the character shows predictable behavior, if a similar situation occurs again. Zanetti et al. [29] implemented a character for *Quake 3 Arena* that used three neural networks: one for movement of the character in a combat situation, one for the aiming and shooting skills, and one for path planning. These networks were trained using recorded actions from expert players.

In [30], Bakkes et al. introduce an adaptability mechanism based on genetic algorithms for a team in *Quake 3 Arena*. The key idea is that NPC do not evolve by themselves, but as a team with a centralized agent control mechanism. The team-oriented behavior is learned by cooperation between multiple instances of an evolutionary algorithm. Each instance learns a relatively uncomplicated behavior. Although they were able to generate adaptive behavior to non-static opponent behavior, the time needed to learn this behavior was extremely long.

Doherty et al. explore the effects of communication on the evolution of team behaviors for teams of characters in FPS's [31]. They show that using communication in difficult environments increases the effectiveness of the team. Westra uses evolutionary neural networks to train characters in *Quake 3 Arena* [32]. He uses weapon and item selection as learning tasks. The character that evolved performed better than the original *Quake 3 Arena* character.

In [33], the author studied the performance of different supervised learning techniques in modeling player behavior in *Soldier of Fortune 2* FPS. He showed that neural networks with a large dataset generally outperformed other supervised learning techniques (decision trees, k-nearest neighbor and Bayesian classification). In [34], the

authors conclude that it is possible to observe realistic behaviors in AI controlled agents using hierarchical learning techniques. A behavior controller selects which subsystem takes control of the agent at a certain time and that subsystem learns through neural networks trained with genetic algorithms. This technique requires a great number of training iterations though, limiting the adaptability of the AI.

Reinforcement learning techniques applied in commercial games are quite rare, because in general it is not trivial to decide on a game state vector and the agents adapt too slowly for online games. In [35] the authors conclude that by using Sarsa(λ) algorithm, an agent can learn how to navigate an environment (avoiding obstacles, attacking enemies and fleeing if losing) through reinforcement learning and environment interaction. RETALIATE [36] is a reinforcement learning algorithm that learns to choose tactics for teams of agents playing a *Domination* style of FPS. This algorithm can rapidly adapt in case of environmental changes by switching team tactics.

Chapter 3

Dynamic Scripting

This chapter describes the Dynamic Scripting technique. In Section 4.1, the basic mechanics are explained, with special emphasis to the four basic mechanisms: rule construction, script selection, rule policy and rule weight updating. In Section 4.2, I list the central characteristics of Dynamic Scripting that helped us choose this technique as the base of the thesis' work. In Section 4.3, extensions to the Dynamic Scripting technique are presented.

3.1 Basic Mechanics

Dynamic Scripting is an unsupervised learning algorithm with a simple yet efficient mechanism for dynamically constructing proper behavior composed by a set of rules from a given rulebase. The aim for Dynamic Scripting is to learn behaviors for non-player characters of videogames. Each character is controlled by a script, which is composed of rules. A rule is a representation of an action of a character, and typically is composed by a condition clause and an effect clause. If the condition is valid, the effect is executed. As an example, consider a rule that translates into the action "shoot the opponent". The condition clause could be "If the opponent is seen and a weapon is equipped" and the effect clause could be "Aim the weapon for the opponent and shoot". The learning process does not change or create rules, but chooses which rules to use in the current context. Rules have to be designed manually.

Each character type is represented by a knowledge base (rulebase) that contains a list of rules that may be inserted in a game script. A game script is a set of rules that represent the behavior of a character. The rules that compose the script are sequentially verified each time an action from a character is required, so that the first rule whose condition is valid corresponds to the action that is going to be executed. The number os rules present in the script may vary. As an example, consider an arbitrary script composed by three rules: A, B and C. Each time an action is required from this script's

character, the rules' conditions are verified. If the condition of rule A is not valid for the current state, rule B's condition is verified instead, and so on for rule C. If none of the rules conditions are valid, a default rule is executed. This default rule does not have a condition, so its effect is always executed if required. This is to prevent the character to become static in the game, always waiting for a rule to be executed.

Every time a new character is placed in the game, the rules that comprise the script controlling the behavior are extracted from the corresponding rulebase. Each rule in the rulebase has an attribute called rule weight. The probability for a rule to be selected for a script is proportional to the associated rule weight.

After an encounter (typically a combat) between the human player and an opponent, which is called a learning episode, the opponent's rulebase adapts by changing the rule weight values in accordance with the success or failure of the rules that were activated during the encounter. This enables the dynamic generation (hence the name) of high quality scripts for basically any given scenario. Scripts (and therefore tactics) are no longer static but rather flexible and able to adapt to even unforeseen game strategies. In figure 4.1, a representation of Dynamic Scripting is presented.

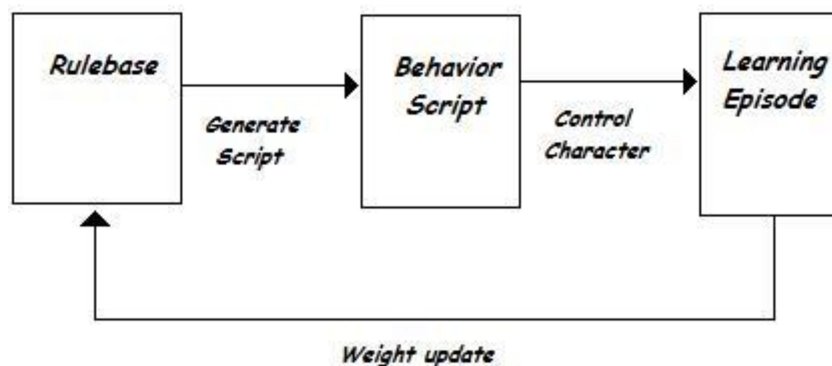

Figure 3.1: General representation of the Dynamic Scripting technique.

There are four main components in the Dynamic Scripting technique: a set of rules, script selection, rule policy, and rule weight updating.

3.1.1 Rules and Rulebases

A rule translates into an action for the character. As mentioned before, Dynamic Scripting creates behaviors by putting together a set of rules. These rules need to be implemented, which is totally dependent on the videogame. All implemented rules are

registered in a rulebase, which is an index of all rules and their corresponding weights. Please note that there may be several rulebases, one for every agent class in the videogame. An agent class is the generic name for all computer controlled agents in a videogame sharing the same behavior. Each rule may optionally contain a condition clause that limits its applicability based on the current game state. In the case of Dynamic Scripting, it is assumed that rules are created manually, though previous work has focused on the automatic creation of rules [37]. Each individual rule in the set of rules has a single weight value associated with it. In the beginning of the learning process, each rule starts with the same weight value, previously defined. Rules in the rulebase never change; instead their weight values are updated to translate their usability in the game. This is one of the most important components of the algorithm, as the performance of the behavior script can only be as good as the rules that it contains.

3.1.2 Script Selection

A learning episode is defined as the component of the videogame where character learning takes place. Before each learning episode the agent creates a subset of the available rules to use in the episode - this is known as a script. The script generation works as follows: First an empty script is generated, containing only a default rule (which specifies an action that can always be performed or an action that is equivalent to "do nothing"). This is just to ensure that characters always have a valid behavior, as this default rule remains unchanged. Then new rules are added to the script by copying them from the rulebase. The order of the rules in the script is important, as the rule policy component orderly processes it and performs the first rule that is applicable to the current game state, as is explained in the next sub-section. A free parameter n determines the size of the script. This parameter is defined by the developers, and usually (but not strictly) depends on the complexity of each rule and on the total number of rules. The script selection component uses a form of fitness proportionate selection to select n rules (without replacement) from the complete set of rules in the rulebase based on their assigned weight value. Rules with a higher weight have a higher probability of getting selected. Rules with a weight of 0 are inactive and cannot be inserted in a script. However, their weight can increase again over time.

3.1.3 Rule Policy

Rule policy determines how rules are selected within a learning episode. This component orderly processes the script component and performs the first rule that is

applicable to the current game state. For example, a rule may require that a character's health be below 50%. If this is not the case then the rule does not apply.

Rules are ordered by their priority. Even though priorities are generally assigned by the behavior developer, there is still some research being done on learning rule priorities in dynamic scripting [38]. In the event of a priority tie, rules are selected based on the highest weight value. This is the secondary use of rule weight values in the Dynamic Scripting technique.

3.1.4 Rule Weight Updating

Besides implementing the rules, the behavior developer creates a fitness function that provides feedback on the utility of the script as a whole. This fitness function produces a number between 0 and 1 indicating how good the script performed during the episode. High fitness indicates strong performance and low fitness indicate low performance. At the end of the learning episode, this fitness function is used to scale the rewards and penalties to the rule weights. The fitness function depends on the videogame, as it evaluates the performance of the script on that particular videogame. Therefore, the same fitness function is not guaranteed to correctly evaluate a character on different videogame types.

Rules that are selected within a learning episode are called "active" rules. These are the rules for which a weight adjustment is calculated. Every other rule has their weight values adjusted depending on the active rules' adjustment.

A standard value called "break-even point" represents a neutral point. If the fitness function returns a value equal to the break-even point then the performance of the agent was neither very good nor really bad. As a consequence all rule weights stay the same, there are no adjustments. Again, the concrete value of the break-even point is dependent on the fitness function and therefore on the game. Basically the point lies between the fitness values of the worst winning agent and the best losing agent. The difference between the fitness value and the break-even point determines the magnitude of the weight modulations.

The Dynamic Scripting technique specifies some parameters affecting the weight adjustments and therefore the whole learning process. In addition to the break-even point there are five parameters limiting the rewards and penalties:

- R_{MAX} : Maximum possible weight increase per trial;

- P_{MAX} : Maximum possible penalty per trial;
- W_{MAX} : Maximum weight for a single rule;
- W_{MIN} : Minimum weight for a single rule;
- W_{INIT} : Initial weight for each rule;

Comparing the return value of the fitness function to the break-even point determines whether an increase (reward), a decrease (penalty) or nothing should happen to the activated rules' weights. The formula to determine the weight adjustment for the active rules (those that were selected to activate during the learning episode) is defined as follows:

$$w = \begin{cases} -P_{MAX} * \dfrac{b-F}{b}, (F < b) \\ R_{Max} * \dfrac{F-b}{1-b}, (F \geq b) \end{cases}$$

Equation 3.1: Formula for determining the weight adjustment value

In this equation w is the weight adjustment for each active rule; F is the calculated fitness value and b is the break-even point. The function evaluates to a negative value if $F < b$ (resulting in a weight decrease), a positive value if $F \geq b$ (weight increase) and 0 if $F = b$ (weights remain unchanged). This is the reward value that is added to each active rule's weight. We can define three different types of rules at the end of an episode: an activated rule, which was present in the script and was activated during the learning episode, receiving the full reward value; a non-activated rule, which was present in the script but was not activated during the episode, receiving half of the reward value; and a non-selected rule, which was not present in the script, receiving the compensation value. Therefore, a half reward is given to each rule in the script that was not selected to activate, which can happen because the rule was never applicable or because the rule had a relatively low priority. Compensation is applied to all rules that are not part of the script (rules in the rulebase), so that the sum of all rule weights always stays constant.

Through the compensation mechanism, the rule weight updating component is responsible for distributing the rule weight “value points" among the available rules. As an example, if there are 10 rules with an initial weight value of 100, there are 1000 value points that can be distributed across all rules. A rule can have higher weight value

than others because it was successfully activated in many winning scripts or because it was not selected to participate in losing scripts and the character lost many matches.

3.2 Central Characteristics

As already mentioned, the research goal of this project was to find AI techniques that could be successfully applied to commercial videogames for controlling the behavior of non-player characters. I tried to find techniques that had certain required characteristics, as a commercial project is very different from a purely academic one.

Dynamic Scripting was selected as a base of our work for the following characteristics presented by the original author:

- Computational effectiveness: Dynamic Scripting is effective, as all rules in the rulebase are based on domain knowledge. Every action which an agent executes through a script that contains these rules is an action that is at least reasonably effective;

- Clarity: Dynamic Scripting generates scripts, which can be easily understood by videogame developers;

- Computational speed: Dynamic Scripting is computationally fast, as it only requires the extraction of rules from a rulebase and the updating of the rules' weights once per episode;

- Variety: Dynamic Scripting generates a new script in every episode, thus providing a high variety in behavior;

- Computational robustness: Dynamic Scripting is robust, as rules are not removed immediately when punished, they instead get selected less often. Their selection rate will automatically increase again, either when they are included in a script that achieves good results or when they aren't included in a script and that script has low performance.

3.3 Extensions to Dynamic Scripting

Previous works from other authors focused on extensions to the basic Dynamic Scripting technique. Two such extensions were applied in our implementation of Dynamic Scripting: Automatic Rule Ordering [8] and Goal-Directed Dynamic Scripting [5]. In the next Sub-Sections, I will explain each extension.

3.3.1 Automatic Rule Ordering

In the basic ordering mechanism, rules appeared in the script ordered by weight values. The rule with the highest weight is always checked first. This approach, while intuitive, has some flaws, as the rule with the highest weight is not necessarily the best rule to be the first one activated. For example, a more specific rule like "picking up an item", should be checked first than a more general rule, like "patrol a corridor".

By learning the rule priorities in parallel with the rule weights, scripts can be obtained that take into account the priority relation between each rule. In this mechanism a relation-weights table stores two values for each combination of two rules of a rulebase, one value for each of the two possible orderings of the two rules. The values are an indication of the effect that the rules have on the performance of the script, when they occur in the specified order.

The relation-weights table is represented in Table 4.1. R_i represents rule number i and R_j represents rule number j. The rulebase contains a total of N rules. Usually relation weights w_{ij} and w_{ji} are not equal. Many rules have a "natural" location in the script, for example, rules that deal with very specific circumstances should occur earlier in the script than more general rules. Still, it is possible that both relation weights are positive, when both have a beneficial effect regardless of where they occur in the script. Also, it is possible that both are negative, when they always have a detrimental effect.

Rule	R1	R2	Rn
R1	-	w_{12}	w_{1n}
R2	w_{21}	-	w_{2n}
Rn	w_{n1}	w_{n2}	-

Table 3.1: Representation of a relation-weight table

The table is updated every time there is a rule weight update. The value of each rule weight update is added to the corresponding value in the table. A single entry w_{ij} is an integer and has the following meaning: The rule i occurred before rule j in a script and gained the weight w_{ij}.

If a new script is to be generated, the rule ordering is determined by adding up the relative priorities for each rule per row and sorting the rule by descending priorities. The rule with the highest priority is inserted at the first position, followed by the one with the second-highest priority and so on.

3.3.2 Goal-Directed Dynamic Scripting

Similarly to Dynamic Scripting, Goal-Directed Dynamic Scripting maintains several rulebases, one for each basic character type in a game. Each rule in a rulebase has a purpose to fill and several rules can have the same purpose, e.g. to attack an enemy but in different ways. Goal-Directed Dynamic Scripting extends the amount of domain knowledge by grouping rules with the same purpose, and defining a common goal for these rules. Hence, goals are introduced and a rule is seen as a strategy for achieving a goal, which can be seen as domain knowledge used to direct the behavior. The learning mechanism operates on the probability that a specific rule is selected as strategy for achieving a specific goal. In order to allow for reusability of rules, so that many goals can share individual rules, weights are detached from rules and instead attached to the relationships between goals and rules. By assigning weights to each goal-rule relationship, adaptation can occur in a separate learning space for each goal. This can allow for higher flexibility and reuse. With the Goal-Directed Dynamic Scripting, the learning process requires fewer trials than the simple Dynamic Scripting to arrive at proper behavior, as instead of selecting from a large pool of rules, the mechanism only selects the rules that matter for the current goal.

As an example, consider a rulebase composed of 10 rules, where 4 of them are offensive-oriented rules (different ways to attack an opponent), 4 are defensive-oriented rules (different ways to defend an area) and 2 are both offensive and defensive-oriented rules. We define two different goals, Attack and Defend, and associate the offensive rules to the Attack goal and the defensive rules to the Defend goal. If we were to use the simple Dynamic Scripting technique, the script selection component would have to choose rules from the entire rulebase, and the learning process would take longer to arrive at proper behavior scripts than if we were to use the Goal-Directed Dynamic Scripting, since the script selection component would only choose rules that have the

same goal as the character, and each goal is associated to only 6 rules (4 offensive or defensive-oriented rules + 2 offensive and defensive-oriented rules).

Chapter 4

Implementation of Dynamic Scripting in First-Person Shooter Scenarios

This chapter is organized as follows: the next Section (Section 4.1) describes the methodology behind the implementation, with a brief description of the tools used; Section 4.2 describes the Dynamic Scripting implementation details; Section 4.3 describes the experimental evaluations made with both the basic and the extended Dynamic Scripting techniques in the two FPS scenarios constructed.

4.1 Methodology

This implementation of Dynamic Scripting will be used for creating characters with adaptive behaviors in future *vectrLab* videogames, and as such, the methodology was dictated by the resources the company provided, as well as the company's own methodology.

The development tool used was *Unity3D* [39], a videogame engine. This engine allows the use of three different programming languages: *C#*, *Javascript* and *Boo*. For the implementation of the Dynamic Scripting technique, *C#* language was used. The next subsection briefly describes this videogame engine.

4.1.1 Unity3D

Unity3D is an integrated authoring tool for creating 3D videogames or other interactive content such as architectural visualizations or real-time 3D animations. The editor runs on *Windows* and *Mac OS X* and can produce games for *Windows*, *Mac*, *Wii*, *iPad*, *iPhone, iPod Touch, Android, PS3 and Xbox 360* platforms. It can also produce browser games that use the Unity web player plugin, supported on *Mac* and *Windows*.

Unlike tools such as *XNA* [40], *Unity3D* provides a visual editor, which greatly facilitates the distribution of elements within the game. This editor also enables you to set all objects, initialize values of attributes and edit scripts. The tool has a friendly user interface and offers powerful programming languages.

Unity3D offers components for working with physical particles, audio, video, lighting, networking, animation, terrain, cameras and much more. The game logic can be programmed in *JavaScript*, C#, or *Boo* (*Python*), and scripts written in one language can interact with scripts written in another language without problems, since they are all supported by *Unity3D*.

4.2 Implementation of the Basic Components

For the implementation of Dynamic Scripting, the four basic components of this technique, previously explained in Chapter 4, had to be considered: rules and rulebases; script selection; rule policy; and rule weight updating.

4.2.1 Rules and Rulebases

We decided to implement each rule manually, leaving a possible automatic rule construction system for future work, since this was our first attempt to implement Dynamic Scripting, and, as such, a greater level of control of the learned behaviors was required. Since the actual implementation of each rule strongly depends on the videogame scenario, the implementation details are explained in each scenario's description, in Section 4.4.

4.2.2 Script Selection

The script selection mechanism is responsible for selecting a number of rules to be added to a character's script. Our implementation of this mechanism selects 4 rules from the rulebase. This number was defined based on the simplicity of the scenarios constructed and on preliminary results obtained. The rules selected are different from each other and the selection itself is based on a form of fitness proportionate selection, where rules with higher weight value have more probability to be selected than rules with lower weight value.

4.2.3 Rule Policy

The rule policy mechanism determines how rules are selected within a learning episode. As a FPS videogame is played in real-time, without pauses to the action, each character is required to execute an action at all times. Therefore, this mechanism is responsible for continuously verifying which rule in the character script should be currently executed. The rule executed must be the highest priority one that has its condition valid in the current state. Hence, the mechanism sequentially and cyclically traverses the script searching for a valid rule to activate.

4.2.4 Rule Weight Updating

This mechanism is responsible for all the learning that takes place in the Dynamic Scripting technique. Two components were of greater importance while implementing the mechanism: the fitness function, which, similarly to the rules, strongly depends on the videogame type and scenario; and the learning parameters, that affect the exploration and exploitation of the learning process.

Fitness Function

The scenarios constructed to test the Dynamic Scripting technique, which are further described in Section 4.4, feature a Dynamic Scripting controlled agent against a static AI controlled agent (a typical non-adaptive finite-state machine agent). The agents fight each other and whoever reaches 0 Health Points (HP) first, loses. Each agent uses firearms to reduce the opponent's HP, and are able to pick up items that restore HP. Therefore, the most successful agent is the one that can keep its HP's number high while reducing the opponents HP. If the Dynamic Scripting agent cannot win a match in ten minutes, it automatically loses that match, and a fitness value of zero is given to it. This is to avoid matches that take too long to finish, as 10 minutes is more than enough to win in the constructed scenarios. The translation of these goals in a fitness function capable of evaluating the Dynamic Scripting agent correctly is presented below:

$$F(a,g) = \frac{4 * H(a) + 4 * D(g) + 2 * T(g)}{10}$$

Equation 4.1: Formula for determining the fitness value

In this equation, the several components have different factors that reflect their respective weight. The *a* parameter refers to the agent and *g* refers to the match. The components are *H(a)*, that represents the remaining health of agent *a*, *D(g)*, representing the total damage done to the opponent in the match *g* (the *o* parameter in the *D(g)* equation below refers to the opponent), and *T(g)*, that represents how fast the agent won or how slow the agent lost match *g* (where t_m refers to the maximum time and t_t refers to the current time). If the agent lost because time reached the 10 minutes limit, the fitness function returns the value zero. This is to force the agent to learn tactics that result in confrontations with the opponent. After analyzing preliminary results and testing different weight values for each component, the best results were obtained with higher values in *H(a)* and *D(g)* than *T(g)*. The equations for the different components are presented below:

$$H(a) = \frac{h_t(a)}{h_0(a)} \qquad D(g) = \frac{(h_0(o) - h_t(o))}{h_0(o)} \qquad T(g) = \begin{cases} \dfrac{t_t}{t_m}, & a \text{ lost;} \\ \dfrac{(t_m - t_t)}{t_m}, & a \text{ won.} \end{cases}$$

Equation 4.2: The three components of the fitness function

Learning Parameters

Dynamic Scripting learning parameters affect the rate of exploration/exploitation of the learning process by limiting rewards and punishments. The following is a description of each parameter, as well as the values used in the registration of the results described in the Section 5.3:

- **Winit** - Initial weight for each rule. The sum of all initial weights specifies the available weight points that can be redistributed. The initial weight implemented is 100.

- **Wmin** – Minimum weight for a single rule. By using a minimum weight of 0, the probability for selection of rules that have that value is null. The minimum weight implemented is 0.

- **Wmax** – Maximum weight for a single rule. Increasing this parameter will increase the exploitation of successful tactics. The maximum weight implemented is 1000.

- **Rmax** – Maximum possible reward. Higher values lead to faster learning but local optimums. The maximum reward implemented is 50.

- **Pmax** – Maximum possible penalty. The maximum penalty implemented is 50.

4.3 Experimental Evaluation

To measure the strength of the characters controlled by Dynamic Scripting, I constructed two different First-Person Shooter scenarios, where these agents compete with agents controlled by a static Finite-State Machine that represents a typical FPS character's behavior. The first scenario implemented was supposed to be simple and quick, with the validation of the Dynamic Scripting technique in the First-Person Shooter genre as goal. The second scenario is more complex than the first.

As mentioned in Chapter 3, the performance of the agents (and therefore the AI controlling the agent) in the Dynamic Scripting technique, is measured with the fitness function. However, the outcome of a single match has no real significance. That is because, as any First-Person Shooter videogame, winning by chance is possible even with an inferior tactic. To make a statement about the general performance of an AI, one should look at the average results over a large number of matches.

Two scenarios were constructed to test the FPS application. The first one was meant to experiment the basic technique in a FPS videogame scenario. The second one was constructed to test and compare the extensions made to the technique (described in Chapter 4, Section 4.3) with the original Dynamic Scripting. After considering a number of preliminary trials and registering the script sizes that provided the best results, we decided to use scripts composed of four rules for our experimentations.

4.3.1 Registering Results

Each of the following scenarios looped through 5 groups of 100 matches each, resulting in 500 matches per scenario. The rule weights are reset after each group, which means that all learned tactics are discarded and the learning restarts at zero. It should be noticed that 100 matches are a very low value for machine learning methods. This number of trials were chosen because the results obtained in preliminary setups from the 100th match onwards stabilize in general. Normally 100 matches would not be sufficient for most standard techniques, but Dynamic Scripting was designed by Spronck to learn from a small number of encounters.

I decided to average the results in a similar way as Pieter Spronck did in his tests [3]. First, the fitness values for each match are averaged across the groups, i.e., the 5 fitness values of the first match of the 5 groups are averaged, and so on. This results in 100 average values, one for each match. The fitness average graphics for each scenario shows the match number on the x-axis. A point on the y-axis is the average fitness value for that match across all groups, calculated as mentioned above. The rule weight values graphics represent the average weight of that particular rule across all groups, calculated similarly to the fitness average.

4.3.2 First Scenario

The goal for the first scenario was to validate the Dynamic Scripting technique, so it was simple and direct, where faster results could be produced and analyzed to conclude if Dynamic Scripting could be used in FPS scenarios. Later, when the extensions to the technique were finalized, the results obtained from the first experiment were compared to the results obtained using the Extended Dynamic Scripting, that is, with both the Automatic Rule Ordering and the Goal-Directed extensions to the basic Dynamic Scripting technique applied. The results obtained from each extension in separate are also registered, so that each extension's individual results can be analyzed and compared.

Scenario Description

In the first scenario implemented, the Dynamic Scripting controlled agent and the agent using the static Finite-State Machine (FSM) are placed in an arena-type environment, with a few items distributed about.

The character controlled by Dynamic Scripting is able to learn different tactics, while the static character always uses the following tactic: if the opponent is not in

range, the character patrols a predetermined area; if the opponent is in range, the character approaches the opponent; if the opponent is at half of the maximum shooting distance or more, the character uses the rocket launcher while approaching the opponent; if the opponent is at less than half the maximum shooting distance, the character uses the machine gun while staying put.

There are three types of items: one is represented by a heart and increases the health of the character that picks it up (to a maximum of the initial health that each starts with), another, represented by a barrel, explodes if it is damaged, hitting anything that is near the blast and therefore damaging it, and the other is represented by a green box-like item and increases the character's ammo count (to a maximum of the initial ammo that each character starts with). Figure 6.1 is a representation of the scenario previously described.

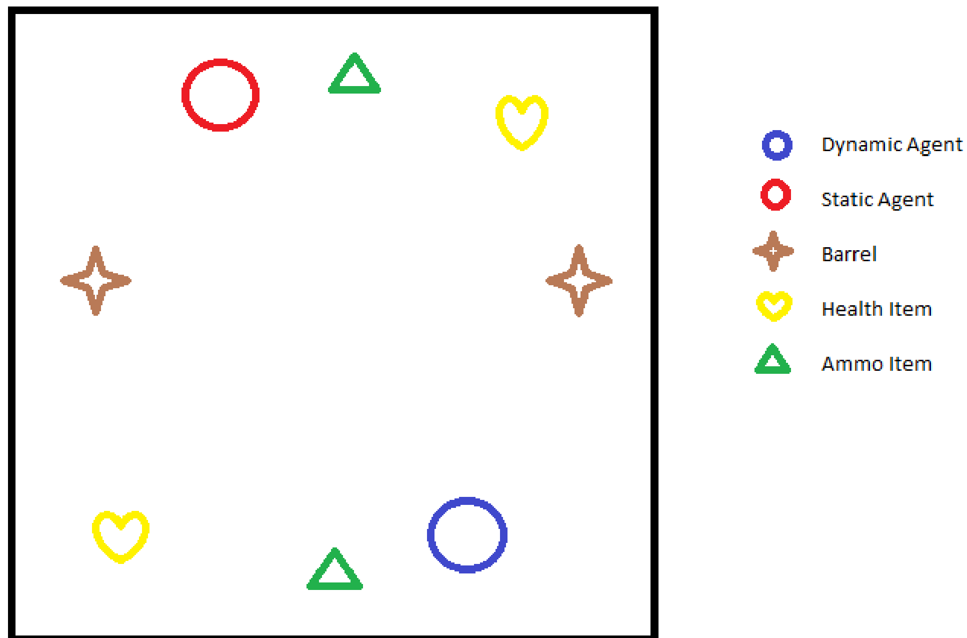

Figure 4.1: Representation of scenario 1, with a label describing each element.

Each character can wield two different weapons that are used to decrease the health of the opponent: a rocket launcher and a machine gun, that shoots faster but making less damage than the first. Rockets from the rocket launcher explode when they collide, causing up to 100 Health Points (HP) of damage (values change depending on the distance from the point of impact). Bullets from the machine gun cause 5 HP of damage each. Each character starts with 200 HP and both have the same weapons and parameters, so that the only difference between them is the behavior.

Rulebase

The implementation of each rule depends upon the programming language used and the game engine, as each must be manually designed (future development will include creating a middleware engine that is able to be easily integrated with the most common game engines). Since those dependencies were already chosen, I had to work with the available functions of the game engine to control characters. For example, if I want to implement a rule that makes the character move, I have to assign animations to the character in order to get more realistic movements, as well as assign velocity and destination parameters. Because of this, the implementation of each rule tends to be too long to list the source code in this document. Therefore, I will simply describe the condition and effect of each rule. A total number of 14 rules were implemented for this scenario. These 14 were used in the simple Dynamic Scripting tests. In tests involving the Goal-Directed extension, an offensive-oriented goal was determined for the agent. From the original 14 rules, 10 were selected to be offensive-oriented rules. The remaining 4 were left out from these particular tests, because they don't belong to the offensive-oriented goal.

The following table lists the rules used in the first scenario:

Rule Name	Condition	Effect
AdvanceGunAttack (Offensive)	Character has machine gun ammo and can see an opponent	Advances towards the opponent and shoots with the machine gun if opponent is in range
AdvanceRocketAttack (Offensive)	Character has rocket launcher ammo and can see an opponent	Advances towards the opponent and shoots with the rocket launcher if opponent is in range
StationaryGunAttack (Offensive)	Character has machine gun ammo and can see an opponent and opponent is in shooting range	Remains in the same place and shoots the opponent with the machine gun

StationaryRocketAttack (Offensive)	Character has rocket launcher ammo and can see an opponent and opponent is in shooting range	Remains in the same place and shoots the opponent with the rocket launcher
SidestepGunAttack (Offensive)	Character has machine gun ammo and can see an opponent and opponent is in shooting range	Moves sideways while shooting the opponent with the machine gun
SidestepRocketAttack (Offensive)	Character has rocket launcher ammo and can see an opponent and opponent is in shooting range	Moves sideways while shooting the opponent with the rocket launcher
BarrelGunAttack (Offensive)	Character has machine gun ammo and there is a barrel close to an opponent that is in shooting range	Shoots the barrel with the machine gun
BarrelRocketAttack (Offensive)	Character has rocket ammo and there is a barrel close to an opponent that is in shooting range	Shoots the barrel with the rocket launcher
TakeAmmo	Character does not have ammo in at least one his weapons and he can see an ammo item	Advance towards the ammo item
TakeHealth	Character has less than 25% of health and can see a health item	Advance towards the health item
Escape	Character has less than 25% of health and can see the opponent	Advances in the opposite direction of the opponent
Idle	Character cannot see the opponent	Remains stationary
Patrol (Offensive)	Character cannot see the opponent	Advances to predetermined locations sequentially

Search (Offensive)	Character cannot see the opponent and has its last known position	Advances to the last known position of the opponent

Table 4.1: List of rules used in Scenario 1.

4.3.3 First Scenario Results

Results are divided in two Sections: Section A presents the results obtained with the simple Dynamic Scripting technique, i.e., without the extensions; Section B presents the results obtained with the Extended Dynamic Scripting, as well as each extension's individual results.

A - Simple Dynamic Scripting

The following figure represents the evolution of the fitness average obtained:

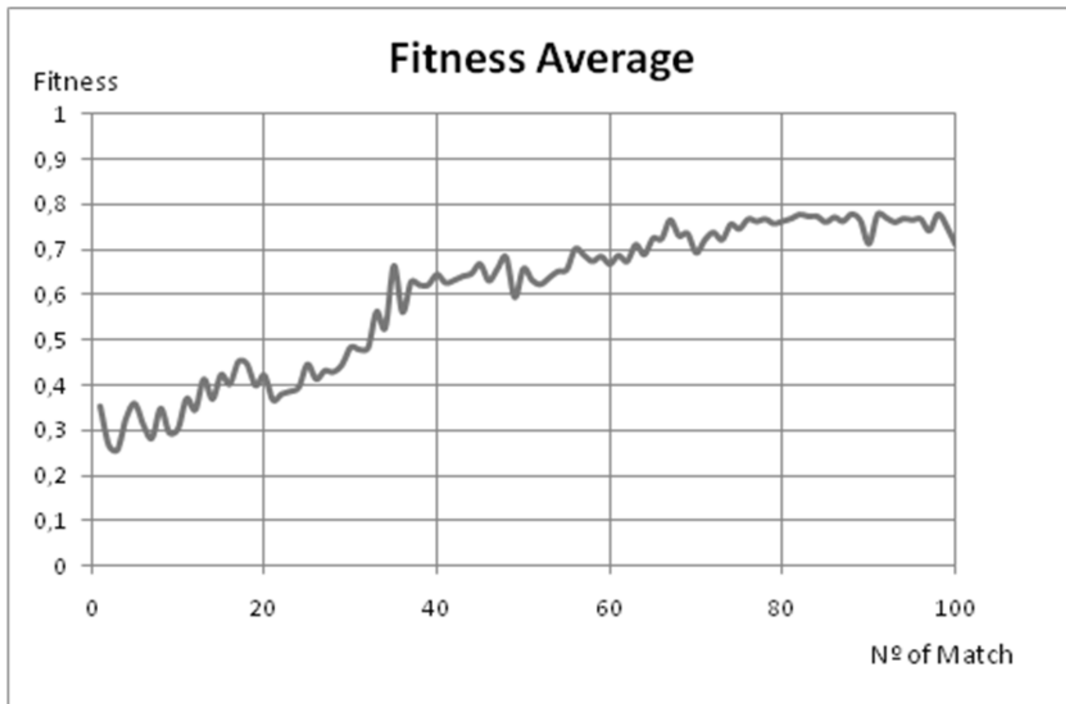

Figure 4.2: Fitness average of Simple DS for Scenario 1

The following is a list of the three rules with the highest weight:

1. SideStepRocketAttack

2. TakeHealth
3. SideStepGunAttack

The following is a list of the three rules with the lowest weight:

1. Idle
2. TakeAmmo
3. Search

The following graphic represents the averaged weight value of the SideStepRocketAttack rule:

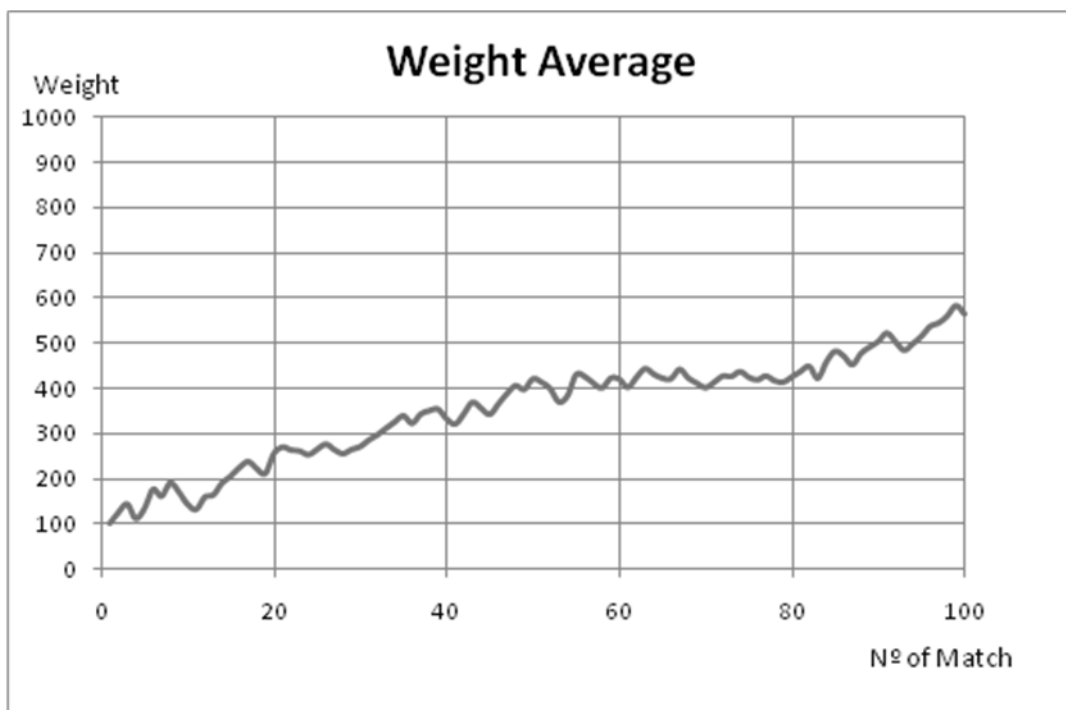

Figure 4.3: Weight average of SidestepRocketAttack rule on Simple DS for Scenario 1.

B - Extended Dynamic Scripting

The following graphic represents the evolution of the fitness average obtained by applying the Automatic Rule Ordering extension to the basic Dynamic Scripting technique:

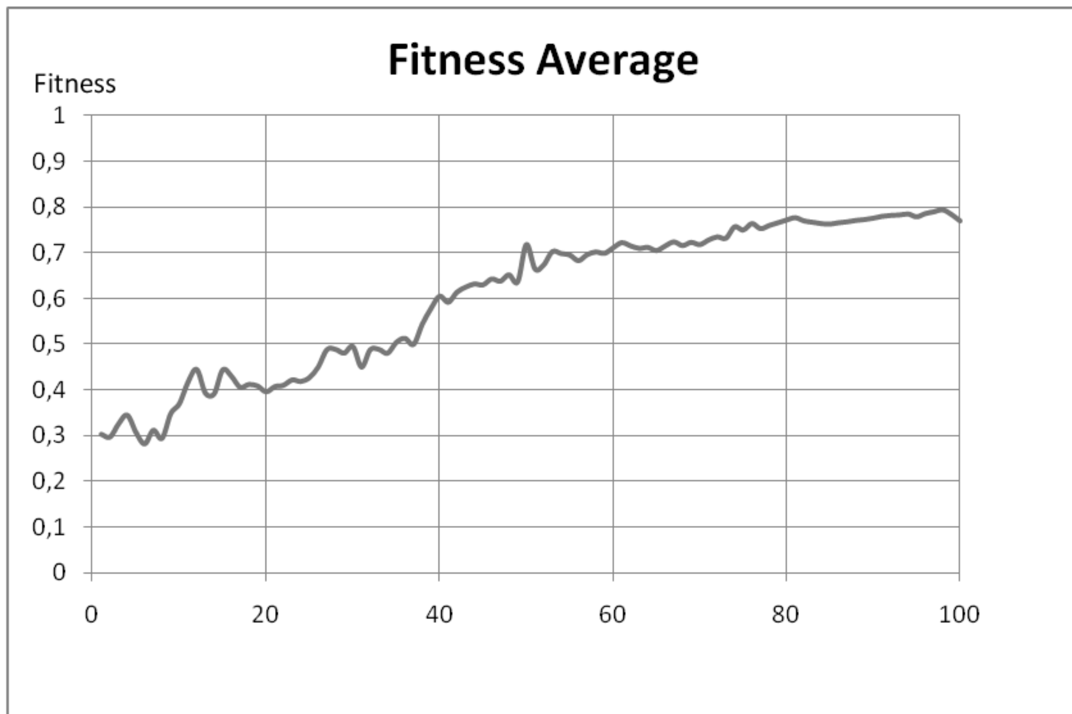

Figure 4.4: Fitness average of DS with Automatic Rule Ordering extension for Scenario 1.

The following is a list of the three rules with the highest weight:

1. TakeHealth
2. SideStepGunAttack
3. SideStepRocketAttack

The following is a list of the three rules with the lowest weight:

1. Idle
2. TakeAmmo
3. Search

The following is an ordered list of the three rules with highest priority:

1. TakeHealth
2. SideStepRocketAttack
3. SideStepGunAttack

The following graphic represents the evolution of the fitness average obtained by applying the Goal-Directed extension to the basic Dynamic Scripting technique:

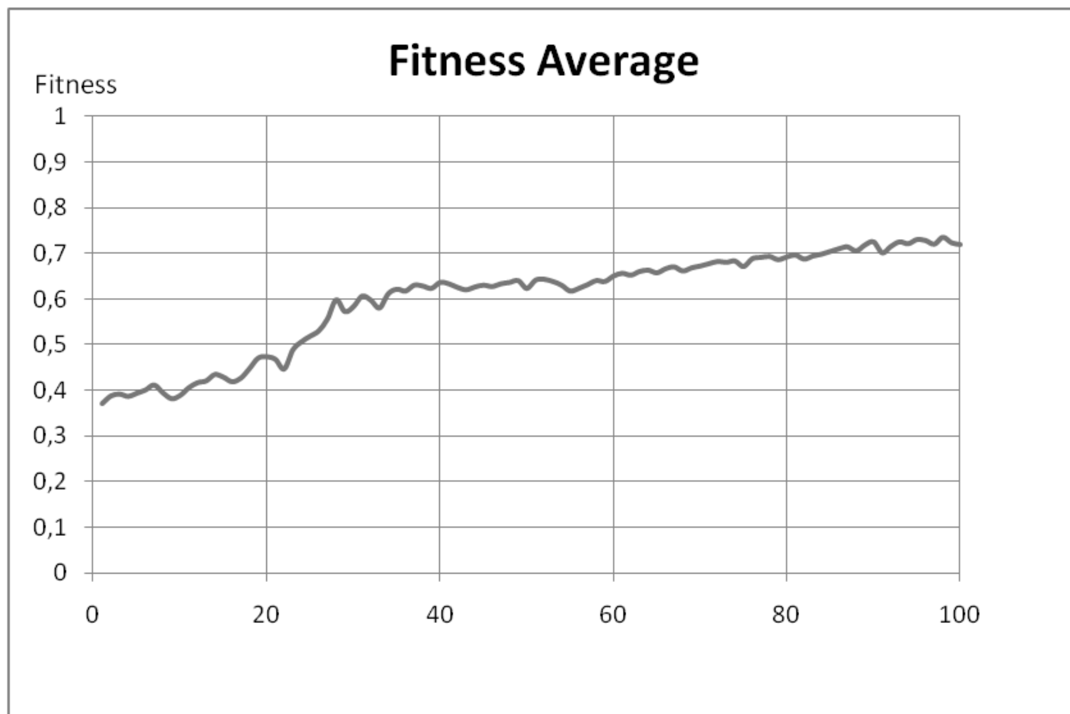

Figure 4.5: Fitness average of DS with the Goal-Directed extension for Scenario 1.

The following is a list of the three rules with the highest weight:

1. SideStepGunAttack
2. AdvanceGunAttack
3. SideStepRocketAttack

The following is a list of the three rules with the lowest weight:

1. Search
2. StationaryGunAttack
3. Patrol

The following graphic represents the evolution of the fitness average obtained by applying both extensions to the basic Dynamic Scripting technique:

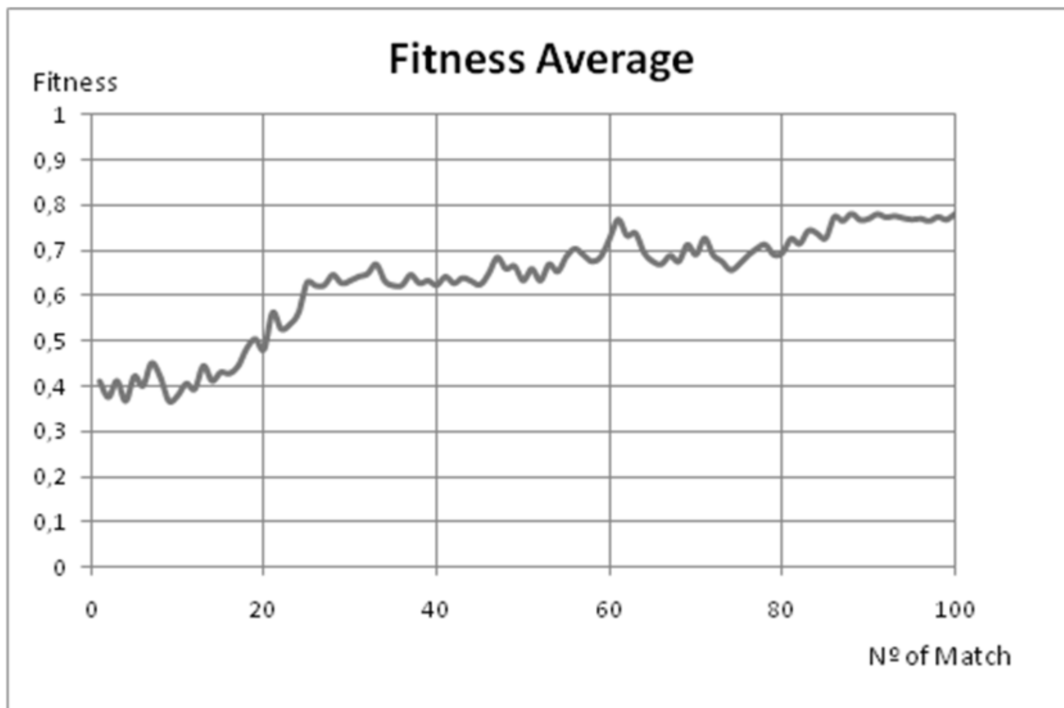

Figure 4.6: Fitness average of Extended DS for Scenario 1

The following is a list of the three rules with the highest weight:

1. SideStepRocketAttack
2. SideStepGunAttack
3. AdvanceGunAttack

The following is a list of the three rules with the lowest weight:

1. StationaryGunAttack
2. Search
3. StationaryRocketAttack

The following graphic represents the averaged weight value of the SideStepRocketAttack rule:

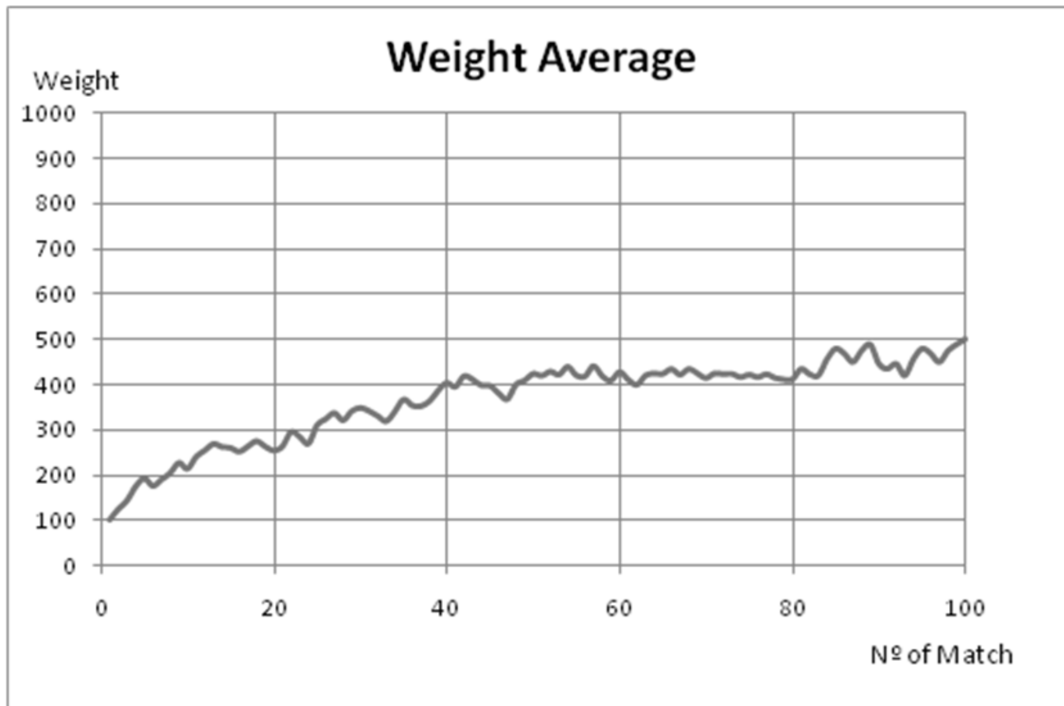

Figure 4.7: Weight average of SidestepRocketAttack rule in Extended DS for Scenario 1.

4.3.4 First Scenario Results Discussion

As explained above, this scenario is very limited and small, resulting in fast matches as both characters detect each other almost immediately. The experimental objective for this scenario was to discover if a Dynamic Scripting controlled agent could learn tactics that successfully exploit weaknesses in an agent controlled by a static AI in a FPS videogame. That objective was completed successfully, as the character controlled by the simple Dynamic Scripting technique (without the extensions later developed) emerged victorious from the 30th match onwards.

We can observe that in the first 30 matches the average fitness values are below 0.5 and fluctuating. This means that in all batches the dynamic character lost more times than it won, although there were peaks of high and low fitness. This probably resulted from rules being tested out and their values changing, when there was more exploration of rules than exploitation. From the 40th match onwards, the fitness steadily raised to above 0.7. In these matches, the best rules were probably chosen already, or in the process of being discovered.

The rule with the highest weight was SidestepRocketAttack, followed by TakeHealth. Since rockets do more damage than machine gun bullets, it was predictable that rules involving rockets were going to have better values. Also, moving to the sides while shooting is a good strategy to evade damage. The TakeHealth rule allows a character to gain health when in the imminence of being defeated. This probably saves many matches and allows better fitness values.

Using the Automatic Rule Ordering extension, in theory scripts are better organized, and as such the agent arrives at better behavior. This isn't mush noticeable in the results obtained with only this extension.

The Goal-Directed extension provides the agent with greater domain knowledge by selecting the appropriate rules for the appropriate goal. Therefore, the agent should arrive at better scripts faster. The results obtained using only this extension show that the agent arrives at the fitness value of 0.5 faster than using the Simple Dynamic Scripting and using the Automatic Rule Ordering extension.

When comparing the results obtained from the Simple Dynamic Scripting *versus* the Extended Dynamic Scripting, the differences were almost unnoticeable. Since Extended Dynamic Scripting has greater domain knowledge and better organized scripts, in theory it arrives at better results faster than using the Simple Dynamic Scripting. This scenario is inherently simple, so even without the Goal-Directed extension and the Automatic Rule Ordering component, the Dynamic Scripting technique is capable of discovering better tactics than the opponent in as little as 30 matches. With the Extended Dynamic Scripting, the number of matches necessary to learn a winning tactic dropped to 20. In a complex environment like the second scenario, the differences are much more noticeable.

4.3.5 Second Scenario

The goal for the second scenario was to test the Dynamic Scripting technique in a more complex environment, and compare the results obtained with the Extended Dynamic Scripting against those from the Simple Dynamic Scripting.

Scenario Description

The main differences between the first scenario and the second scenario are in the number of obstacles that are included in the environment and the scale of that environment. It is a much bigger area, and unlike the first scenario, both characters can't detect each other right away. Figure 5.7 is a representation of the second scenario. There

are 2 paths that character can take to find each other, where path *A* is shorter than path *B*.

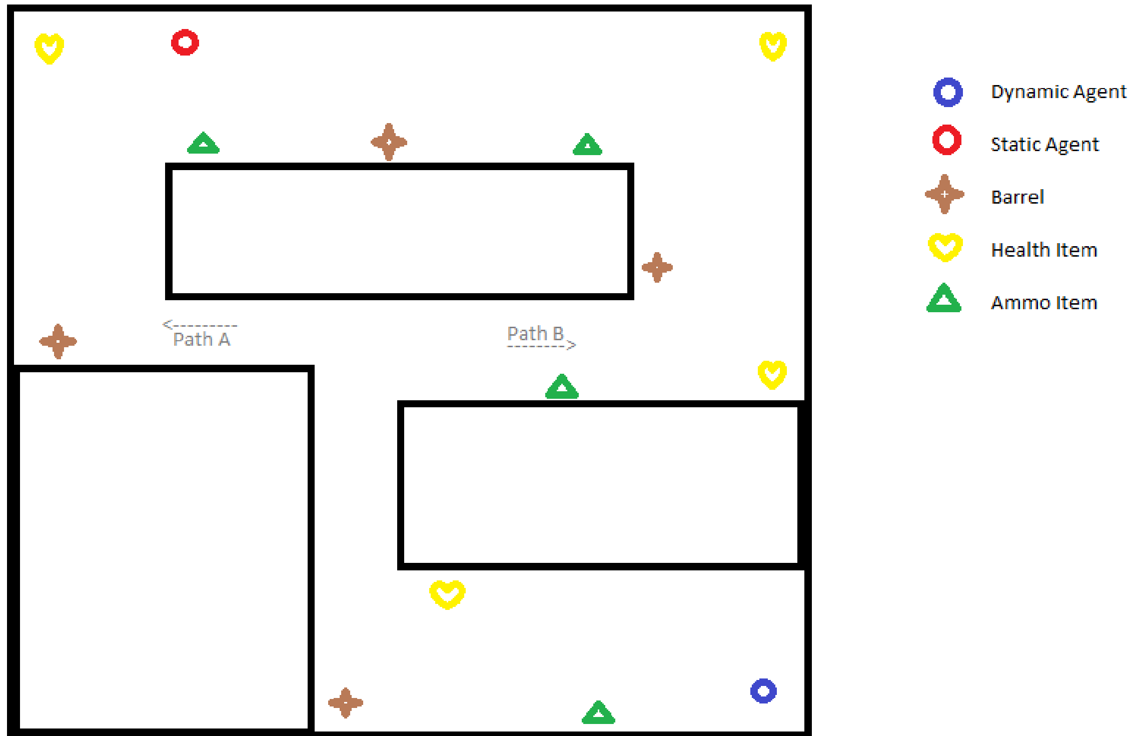

Figure 4.8: Representation of scenario 2, with a label describing each element.

The equipment, parameters and items available to both characters is the same as in the first scenario.

The character controlled by the FSM has the same behavior as in the first scenario, except that instead of patrolling an area, it remains in the same place until it detects its opponent. After detection, the character shoots its weapons while backing away from its opponent. Therefore, the behavior for the FSM controlled character is the following: if the opponent is not in range, the character stands in the same place; if the opponent is in range, the character backs away from the opponent; if the opponent is at half of the maximum shooting distance or more, the character uses the rocket launcher while staying put; if the opponent is at less than half the maximum shooting distance, the character uses the machine gun while backing away from the opponent.

Rulebase

As was already explained, the implementation of each rule is too long and complex to list the source code in this document, so I'll only list each rule's condition and effect. The same rules that were present in the first scenario were used in this one, with an added number of four rules, bringing the total number of rules used to 18. From these rules, 14 were selected to be offensive-oriented rules, used in the Extended Dynamic Scripting tests. For the tests without the extensions, all rules were used.

The following table lists the additional rules used in the second scenario:

Rule Name	Condition	Effect
AdvanceSidestepGunAttack (Offensive)	Character has machine gun ammo and can see an opponent	Advances towards the opponent while sidestepping and shooting with the machine gun if opponent is in range
AdvanceSidestepRocketAttack (Offensive)	Character has rocket launcher ammo and can see an opponent	Advances towards the opponent while sidestepping and shooting with the rocket launcher if opponent is in range
HuntPathA (Offensive)	Character cannot see the opponent	Searches for the opponent through predetermined path A
HuntPathB (Offensive)	Character cannot see the opponent	Searches for the opponent through predetermined path B

Table 4.2: List of additional rules used in Scenario 2.

4.3.6 Second Scenario Results

As before, results are divided in two Sections: Section A presents the results obtained with the simple Dynamic Scripting technique; Section B presents the results obtained with the Extended Dynamic Scripting.

A - Simple Dynamic Scripting

The following graphic represents the evolution of the fitness average obtained:

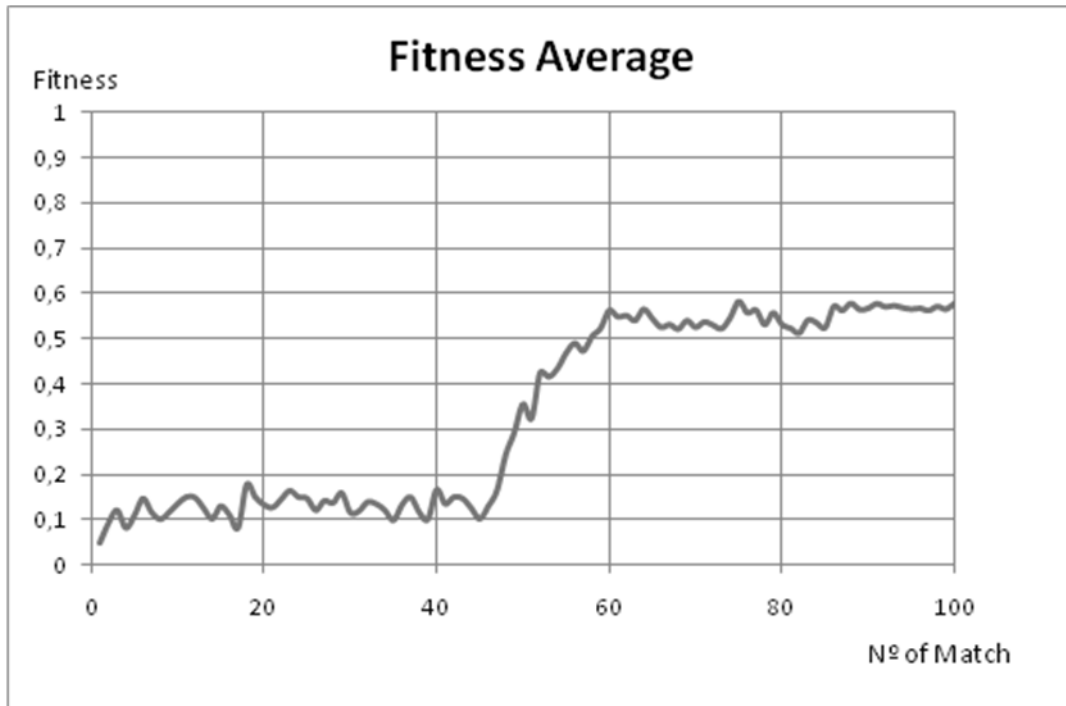

Figure 4.9: Fitness average of Simple DS for Scenario 2.

The following is a list of the three rules with the highest weight:

1. AdvanceRocketAttack
2. HuntPathB
3. TakeHealth

The following is a list of the three rules with the lowest weight:

1. Idle
2. Escape
3. StationaryGunAttack

The following graphic represents the averaged weight value of the AdvanceRocketAttack rule:

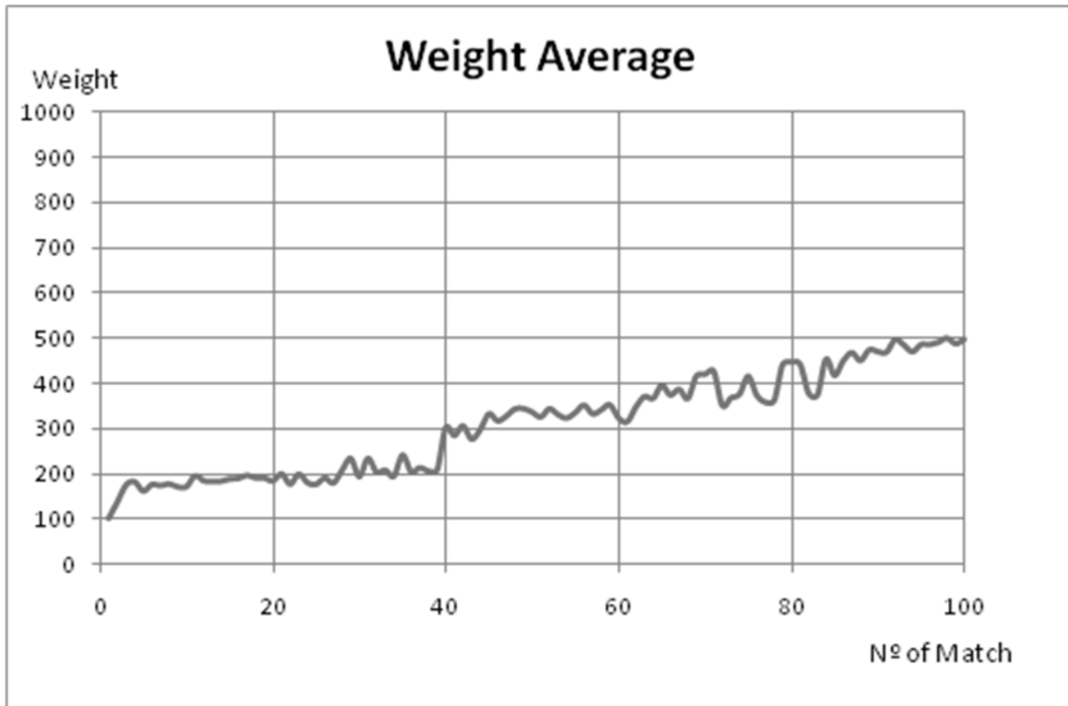

Figure 4.10: Weight average of AdvanceRocketAttack rule in Simple DS for Scenario 2.

B - Extended Dynamic Scripting

The following graphic represents the evolution of the fitness average obtained using the Automatic Rule Ordering extension:

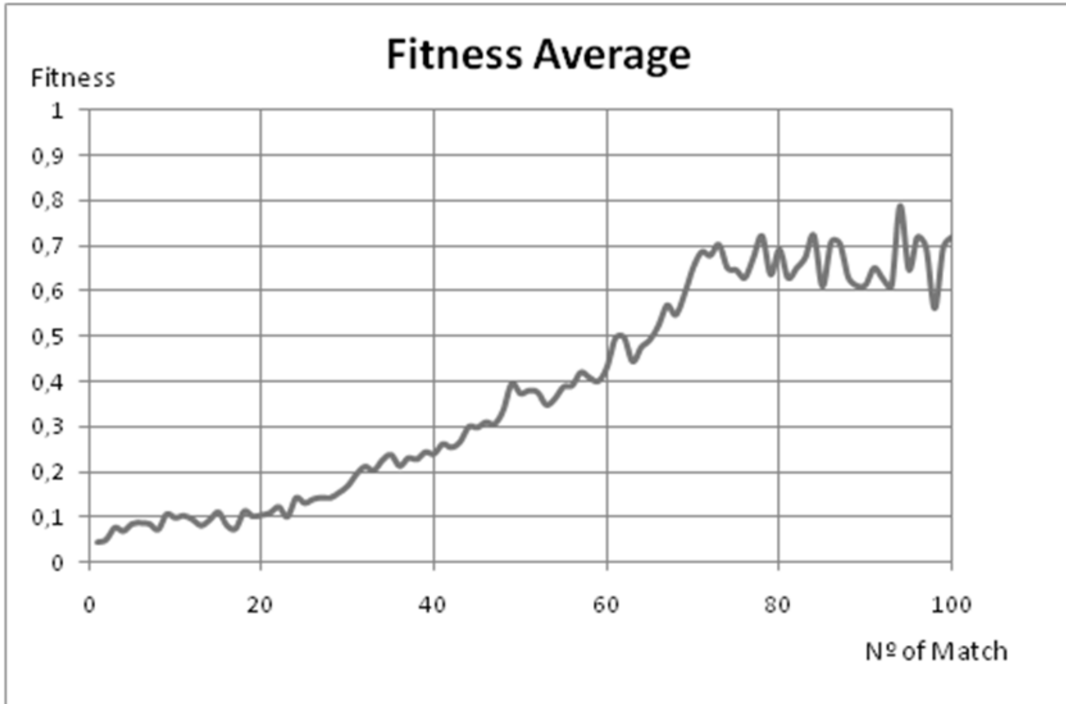

Figure 4.11: Fitness average of DS with Automatic Rule Ordering extension for Scenario 2.

The following is a list of the three rules with the highest weight:

1. AdvanceSidestepRocketAttack
2. TakeHealth
3. HuntPathA

The following is a list of the three rules with the lowest weight:

1. Idle
2. TakeAmmo
3. Escape

The following is an ordered list of the three rules with highest priority:

1. TakeHealth
2. HuntPathA
3. AdvanceSidestepRocketAttack

The following graphic represents the evolution of the fitness average obtained using the Goal-Directed extension:

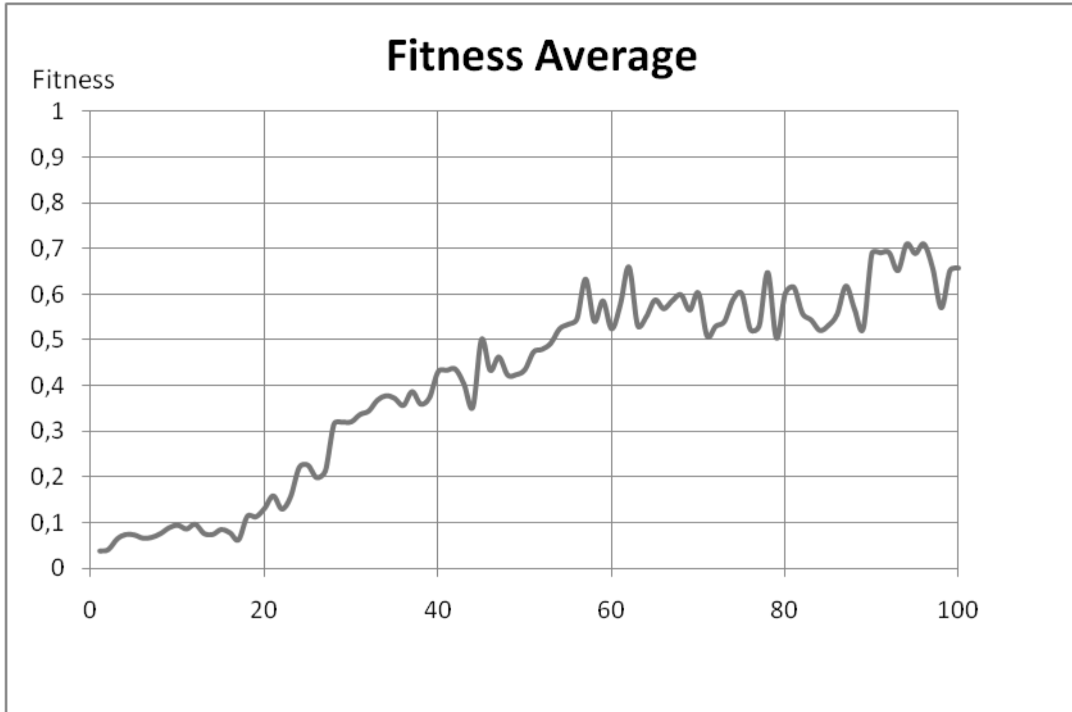

Figure 4.12: Fitness average of DS with Goal-Directed extension for Scenario 2.

The following is a list of the three rules with the highest weight:

1. HuntPathA
2. AdvanceSidestepGunAttack
3. AdvanceSidestepRocketAttack

The following is a list of the three rules with the lowest weight:

1. StationaryGunAttack
2. StationaryRocketAttack
3. BarrelGunAttack

The following graphic represents the evolution of the fitness average obtained by applying both extensions to the basic Dynamic Scripting technique:

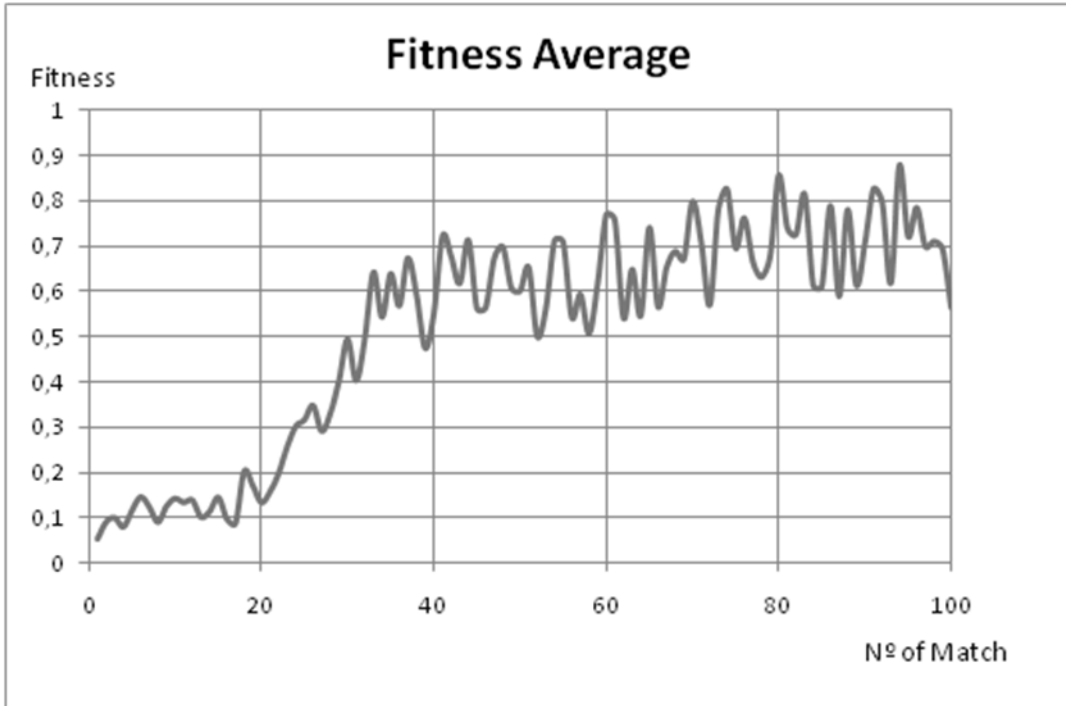

Figure 4.13: Fitness average of Extended DS for Scenario 2.

The following is a list of the three rules with the highest weight:

1. HuntPathA
2. AdvanceSidestepRocketAttack
3. AdvanceGunAttack

The following is a list of the three rules with the lowest weight:

1. StationaryGunAttack
2. BarrelGunAttack
3. StationaryRocketAttack

The following graphic represents the averaged weight value of the HuntPathA rule:

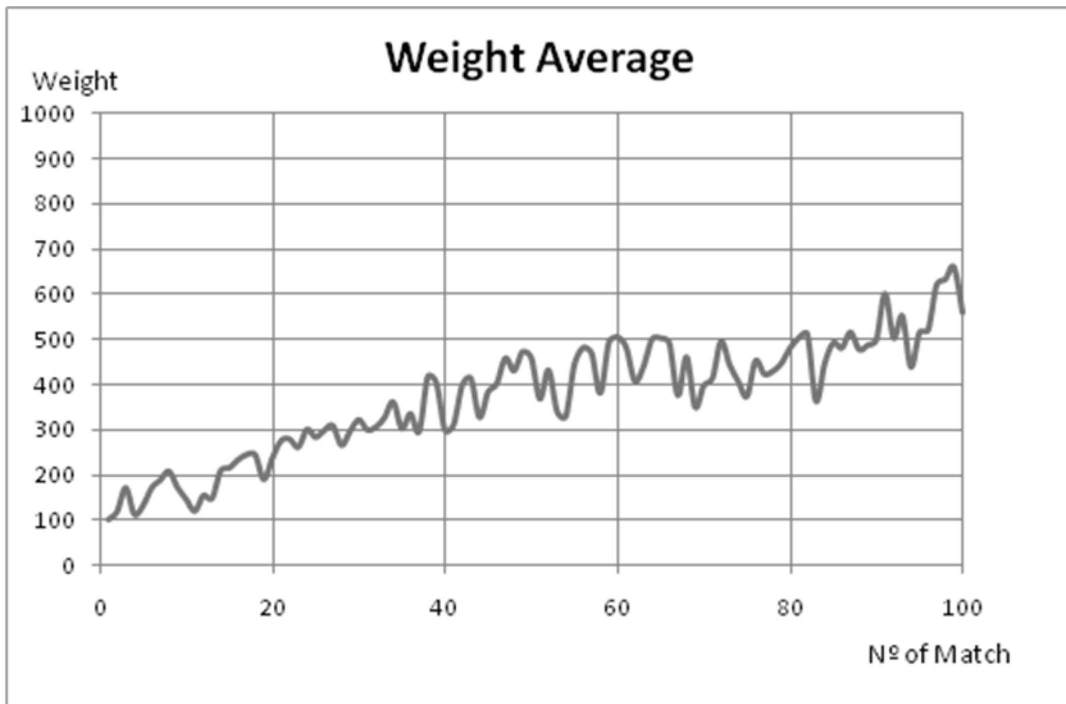

Figure 4.14: Weight average of HuntPathA rule in Extended DS for Scenario 2.

4.3.7 Second Scenario Results Discussion

The experimental objective for this scenario was to discover if the Goal-Oriented extension for Dynamic Scripting can produce better results. Since this scenario was much more complex than the previous one, more domain knowledge makes a big difference, as results from the Goal-Oriented Dynamic Scripting are better when comparing with those from the Simple Dynamic Scripting.

The big difference between this scenario and the previous one is that only a specific combination of rules would lead the character to victory. This combination is as follows: one rule to reach the opponent (as it will stay put waiting for the character), and if the time runs out, the character loses; and one rule to attack while advancing (since when the opponent detects the character's movements, it will try to run away from it).

The rules that make the character reach the opponent are HuntPathA and HuntPathB. If the behavior script does not contain one of these rules, the character loses from time-out. HuntPathA reaches the opponent faster than HuntPathB, so choosing the first over the second is a better tactic.

The rules that attack the opponent while advancing are AdvanceRocketAttack, AdvanceGunAttack, AdvanceSidestepGunAttack, AdvanceSidestepRocketAttack. If the

behavior script doesn't contain one of these rules, the character doesn't have time to defeat the opponent, as it will always run away from the character.

Since the Extended Dynamic Scripting (with both extensions applied to the basic technique) has fewer rules to explore, as only certain rules had the same goal as the character, the solution for the combination of rules is reached earlier than with the Simple Dynamic Scripting. With fewer rules to explore, the best rule combination (HuntPathA with AdvanceSidestepRocketAttack) has a greater probability of being discovered, hence the better fitness value of the character.

Chapter 5

Conclusions

This thesis centered in the idea of improving and adapting the behaviors of videogame characters, resulting in better gaming experiences for the human players. By examining the experimental results described in Chapter 5, we can conclude that it is possible for a videogame AI with Dynamic Scripting to learn tactics that successfully exploit weaknesses in the behavior of other opponents.

As part of this thesis, I implemented an AI technique that allows users to add adaptive behavior to their characters, through the *Unity3D* game engine. This AI is based on the Dynamic Scripting technique, where rules encode actions for the character and are selected to participate in behavior scripts. This selection depends on the weight value of each rule, which is updated through a reinforcement learning inspired mechanism, which depends on the fitness function provided by the user.

The biggest worry regarding machine learning is the unpredictability of what is learned. While this is a valid concern which is inherent to learning procedures of all kinds, Dynamic Scripting has many characteristics that minimize this issue. A careful implementation of all rules in a rulebase is the first step to ensure that only meaningful behavior is generated. Another big advantage of Dynamic Scripting is how the learned information is stored. The rule weights which are used to generate the scripts can easily be interpreted by users, as they tend to have a high-level language abstraction; rules with a low weight value are considered bad and rules with a high weight value are probably important for the success of an agent.

The creator of Dynamic Scripting designed it as a technology for deployment in real commercial videogames. This is not academic research just for the sake of it, but a real applicable technique that could greatly improve the fun factor of a upcoming commercial videogame. Assuming a careful implementation, there are little risks and many rewards.

The time required to implement rules for the Dynamic Scripting technique is not greater than applying and testing a non-adaptive AI, since behaviors developed for a

static AI can be integrated in rules for use in Dynamic Scripting. Therefore, future FPS videogames developed by *vectrLab* can use Dynamic Scripting AI without having to waste much more time than the required to develop a static AI, with the added bonus of having characters that adapt their behaviors to the game environment.

5.1 Future Work

A number of improvements can be integrated with our work, namely: a graphical user interface (GUI) for debugging and testing different learning parameters, so that the AI code is generated procedurally; more and different scenarios available to the user; team-based behavior learning; automatic learning of rules and domain knowledge; merging system for rules that work best when activated simultaneously. These improvements are not currently implemented because of time constraints, and also because some of the improvements, like automatic learning of rules, are not interesting enough from the commercial point of view, as some degree of control over what behaviors are learned is a necessity for the videogame industry.

All of the experimental scenarios discussed in this document use static AI opponents for the Dynamic Scripting character. The reason for this is that there was not sufficient time to organize testing sessions with actual human players. An empirical study should investigate the effectiveness and entertainment value of our work in games played against actual human players. Also, testing Dynamic Scripting with different genres of videogames could provide valuable data for future improvements to our work.

Bibliography

[1] Tozour, P., The perils of AI scripting, in AI Game Programming Wisdom, *Charles River Media, Inc., Hingham, MA., USA, pp. 541–547*, 2002.

[2] Millington, I., Artificial Intelligence for Games. *Morgan Kaufmann Publishers Inc., ch. Decision Making, pp. 301–471*, 2006.

[3] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., Postma, E., Adaptive game AI with dynamic scripting, *Machine Learning, v.63 n.3, p.217-248*, June 2006.

[4] Tavinor, G. The Definition of Videogames. *Contemporary Aesthetics, Volume 6*, 2008.

[5] Dahlbom, A., Niklasson, L., Goal-Directed Hierarchical Dynamic Scripting for RTS Games. *Paper presented at the Second AIIDE, Marina del Rey, California*, 2006.

[6] Ponsen, M., Muñoz-Avila, H., Spronck, P., Aha, D., Automatically generating game tactics with evolutionary learning. *Available in: AI Magazine 27(3): pp. 75*, 2006.

[7] Ladebeck, M., Applying Dynamic Scripting to "Jagged Alliance 2", *TU Darmstadt,* 2008.

[8] Timuri, T., Spronck, P., van den Herik, J., Automatic rule ordering for dynamic scripting. *Paper presented at the Artificial Intelligence in Interactive Digital Entertainment, Stanford, CA*, 2007.

[9] Policarpo, D., Urbano, P., Loureiro, T., Dynamic Scripting Applied to a First-Person Shooter, *Proceedings of the 5º CISTI, pp. 570*, 2010.

[10] Baratz, A., The Stage of the Game. *Ars Technica*. 2001.

[11] Tozour, P., The Evolution of Game AI. *AI Game Programming Wisdom (ed. S. Rabin)*, pp. 3-15, Charles River Media, Inc., Hingham, MA. 2002.

[12] Fairclough, C., Fagan, M., macNamee, B., and Cunningham, P., Research Directions for AI in Computer Games. *12th Irish Conference on Artificial Intelligenct & Cognitive Science* (AICS 2001), pp. 333-344. 2001.

[13] Snider, M., Where Movies End, Games Begin. *USA Today*, May 23,2002.

[14] Apperley, T. H., Genre and game studies: Towards a critical approach to videogame genres. *Simulation & Gaming* 37(1), 6-23. 2006.

[15] Adams, E & Rollings, A , Fundamentals of game design, Prentice Hall, NJ. 2007.

[16] Buckland, M., Programming Game AI by Example, *Wordware Publishing*, 2005.

[17] Tozour, P., The Perils of AI Scripting. *AI Game Programming Wisdom*, *(ed. S. Rabin)*, pp. 541-547, Charles River Media, Inc., Hingham, MA, 2002.

[18] Schwab, B., AI Game Engine Programming. *Hingham: Charles River Media*. 2004.

[19] Sawyer, B., Serious Games: Improving Public Policy through Game-based Learning and Simulation. *Foresight & Governance Project, Woodrow Wilson International Center for Scolars*, Washington DC. 2002.

[20] Laird, J.E., Bridging the Gap Between Developers & Researchers. *Game Developers Magazine*, Vol.8, 2000.

[21] Stanley, K., Bryant, B. and Miikkulaine, R., Evolving Neural Network Agents in the NERO Video Game, *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games,* 2005.

[22] Stanley K., Bryant B., Miikkulainen R. Real-time neuroevolution in the NERO video game. *IEEE Trans Evol Comput 9:653-668*,2005.

[23] Szita, I., Lorincz, A., Learning tetris using the noisy cross-entropy method, N*eural Computation, v.18 n.12, p.2936-2941*, December 2006.

[24] Szita I, Lorincz A., Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. *J Artif Intell Res 30:659-684*, 2007.

[25] Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E., Online adaptation of game opponent AI in simulation and in practice. *Proceedings of the 4th international conference on intelligent games and simulation. Eurosis, pp 93-100*. 2003.

[26] Merrick, K., Maher, M.L., Motivated Reinforcement Learning for Non-Player Characters in Persistent Computer Game Worlds, in *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, ACE*, 2006.

[27] Adobbati, R., Marshall, A., Scholer, A., Tejada, S., Gamebots: A 3d virtual world test-bed for multi-agent research. *In Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[28] Laird, J., It knows what you're going to do: adding anticipation to a quakebot. *In Proceedings of the Fifth International Conference on Autonomous Agents, pages 385-392*, Montreal, Canada, 2001.

[29] Zanetti, S., Rhalibi, A., Machine learning techniques for fps in q3. *In ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology, pages 239-244*, 2004.

[30] Bakkes, S., Spronck, P., Postma, E.O., Team: The team-oriented evolutionary adaptability mechanism. *In ICEC '04: Proceedings of the Third International Conference on Entertainment Computing, pages 273-282*. Springer, 2004.

[31] Doherty, D., O'Riordan, C., Effects of communication on the evolution of squad behaviours. *In (AIIDE '08) Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference. AAAI*, 2008.

[32] Westra, J., Evolutionary neural networks applied in First person shooters, MSc Thesis, 2007.

[33] Benjamin, G., An Empirical Study of Machine Learning Algorithms Applied to Modelling Player Behaviour in a First Person Shooter Video Game, *University of Wisconsin - Madison, USA*, 2002.

[34] Hoorn, N., Togelius, J., Schmidhuber, J., Hierarchical Controller Learning in a First-Person Shooter. *In IEEE Symposium on Computational Intelligence and Games*, 2009.

[35] McPartland, M., Gallagher, M., Learning to be a Bot: Reinforcement Learning in Shooter Games, *Proceedings of the Fourth Artificial Intelligence and interactive Digital Entertainment Conference, Stanford, California*, 2008.

[36] M. Vasta, S. Lee-Urban, and H. Muñoz-Avila, RETALIATE: Learning Winning Policies in First-Person Shooter Games, *Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference*, 2007.

[37] Buro, M., ORTS. A Hack-Free RTS Game Environment. *Computers and Games: Third International Conference, Cg,* 2002.

[38] DeBoer, P.-T., Kroese, P., Mannor, S., Rubinstein, Y., A tutorial on the cross-entropy method. *Ann Oper Res 134(1):19-67*, 2004.

[39] Unity3D website: http://unity3d.com.

[40] Microsoft XNA website: http://www.xna.com.

Appendix A

The following pages are a transcription of the paper submitted and accepted in the second Workshop on Intelligent Systems and Applications (WISA), a workshop inserted in the fifth Conferência Ibérica de Sistemas e Tecnologias de Informação (CISTI).

Dynamic Scripting Applied to a First-Person Shooter

Abstract—Videogame Artificial Intelligence (AI) is growing more complex and realistic to keep up with player requirements. Despite this, most games still fail to provide true adaptability in their AI, resulting in situations where an intermediate level player is able to predict the AI's behavior in a short amount of time, leading to a predictable and boring game experience. Creating a truly adaptive AI would greatly benefit a videogame's intrinsic value by providing a more immersive and unpredictable game experience. This paper describes the development of an AI system for the First-Person Shooter (FPS) videogame genre that avoids this problem through the creation of adaptable rule-based behaviors, enabling AI characters to learn the best strategy for any given situation.

Keywords – Videogame AI; Adaptive AI; Rule-based Behaviors; Game Experience

I. INTRODUCTION

Artificial Intelligence (AI) applied to videogames is a topic widely supported by academic researchers and AI R&D teams in the videogame industry. The dynamic and interactive environments of videogames present good test-beds for new and improved AI techniques, even if some of the techniques are not commercially implemented by developers.

In the past two decades, we can clearly observe a co-evolution of commercial videogames, computer graphics and networking. However, and when it comes to AI, for the time being this synergy between the game industry and academic research seems rather an exception than the rule. Although the potential co-evolution is obvious, behavior programming for game characters and AI research are seldom seen together when it comes to evolutionary jumps. The probable reason for this is that videogames are governed by different laws than academic AI Research & Development. In a nutshell, a videogame is a commercial product, and commercial products tend to be based upon industry-proven methods whenever possible. Hardly any well-known videogames publisher will fund the development of a videogame featuring a newly created and market-untested AI process or method. Therefore, most of the commercial products are bulked up with a lot of things that have been successful in the past, leaving a small space for innovation. This is a very different point of view from academic AI

research, where the main goal is to achieve better results with new and innovative methods.

Commercial videogame AI is typically based on non-adaptive techniques [1], [2]. A major disadvantage of non-adaptive game AI is that once a weakness is discovered, nothing stops the human player from exploiting it to the extreme. This disadvantage can be resolved by endowing game AI with adaptive behavior, i.e., the ability to learn and adapt. In practice, adaptive videogame AI is seldom implemented because current used techniques such as neural networks require numerous trials to learn an effective and efficient behavior. In addition, game developers are concerned that applying adaptive game AI to non-playable characters may result in uncontrollable and unpredictable behavior.

This paper describes the work in progress in the development and prototyping of an adaptive videogame AI for commercial FPS videogames, which makes use of Dynamic Scripting [3] to create adaptable rule-based behaviors in Non-Player Characters (NPC). Dynamic scripting is an approach for adaptive game AI that learns, by means of reinforcement learning, which tactics (i.e., action sequences) an opponent should select to play effectively against the human player. The results of the work this paper describes will be used to develop an AI for commercial FPS videogames produced by one of the sponsors of this project. As such, there are some restrictions in the development of the AI, for example, the programming language and game engine used were already defined by the sponsor.

This paper is organized as follows: in the next section, we will start with an overview of the FPS genre of videogames and give a background on AI applied to this type of games. In Section III, we will explain the theory behind the Dynamic Scripting approach, and describe how we applied it in our prototype in Section IV. In Section V, we present the experimental results and finally we conclude discussing future work in Section VI.

II. BACKGROUND

This section will first explain the defining characteristics of FPS and of AI in FPS (Subsection II-A), then discuss related work in the application of AI in FPS videogames (Subsection II-B).

A. FPS Genre and AI

FPS is a videogame genre characterized by the player's viewpoint of the virtual environment as with the character's eyes, as if the player is actually inside the game. This genre of videogames usually has a large focus on realism, with gravity, light, sound, object collision and other components emulating their real-life counterparts. This creates a feeling of immersion in the player, heightening the game experience and fun. Since the creation of the first FPS videogame, this genre is also used to show new technical advances and state of the art components of the various gaming platforms.

Typically, FPS AI is organized in four distinct entities or components [4]: behavior, movement, animation and combat. The behavior component is the highest-level component and determines the objectives, state and immediate destination of the character, communicating with the other components to coordinate the required movement. The movement component determines how the character moves in the game and is responsible for navigation in the environment. The animation component is responsible for the control of the skeleton of the character and his appearance to the player. The combat component is responsible for selecting the tactics and actions of the character when in combat, for example, aiming and firing. This component is the more noticeable to the player, for combat is typically the most common aspect of FPS.

B. Machine Learning in FPS Videogames

FPS games have received attention as a machine learning test-bed due to their popularity and applicability as a model for real-life situations. In [5], the author studied the performance of different supervised learning techniques in modeling player behavior in Soldier of FortuneTM FPS. He showed that neural networks with a large dataset generally outperformed other supervised learning techniques (decision trees, k-nearest neighbor and Bayesian classification).

In [6], the authors conclude that it is possible to observe realistic behaviors in AI controlled agents using hierarchical learning techniques. A behavior controller selects which subsystem takes control of the agent at a certain time and that subsystem learns through neural networks trained with genetic algorithms. This technique requires a great number of training iterations though, limiting the adaptability of the AI.

Reinforcement learning techniques applied in commercial games are quite rare, because in general it is not trivial to decide on a game state vector and the agents adapt too slowly for online games [7]. In [8] the authors conclude that by using Sarsa(λ) algorithm, an agent can learn how to navigate an environment (avoiding obstacles, attacking enemies and fleeing if losing) through reinforcement learning and environment interaction. RETALIATE [9] is a reinforcement learning algorithm that learns to choose tactics for teams of agents playing a Domination style of FPS. This algorithm can rapidly adapt in case of environmental changes by switching team tactics.

III. DYNAMIC SCRIPTING

Dynamic Scripting is an unsupervised learning algorithm with a simple yet efficient mechanism for dynamically constructing proper behavior composed by a set of rules from a given rulebase. The default implementation of Dynamic Scripting is aimed at learning behaviors for NPC opponents. The implementation is as follows: each opponent type is represented by a knowledge base (rulebase) that contains a list of rules that may be inserted in a game script. A game script is a set of rules that represent the behavior of a character. Every time a new opponent is placed in the game, the rules that comprise the script controlling the behavior are extracted from the corresponding rulebase. Each rule in the rulebase has an attribute called rule weight. The probability for a rule to be selected for a script is proportional to the associated rule weight. After an encounter (typically a combat) between the human player and an opponent, the opponent's rulebase adapts by changing the rule weight values in accordance with the success or failure of the rules that were activated during the encounter. This enables the dynamic generation (hence the name) of high quality scripts for basically any given scenario. Scripts (and therefore tactics) are no longer static but rather flexible and able to adapt to even unforeseen game strategies.

There are four main components in the dynamic scripting algorithm: a set of rules, script selection, rule policy, and rule value updating.

The first component is a set of rules that the algorithm can choose from. Each rule may optionally contain a condition clause that limits its applicability based on the current game state. In the case of dynamic scripting, it is assumed that the person developing the game behavior is responsible for creating the set of rules, though previous work has focused on the automatic creation of rules [10]. Each individual rule in the set of rules has a single weight value associated with it. This is one of the most important components of the algorithm, as the performance of the AI script can only be as good as the rules that it contains.

The second component of the algorithm is script selection. A learning episode is defined as a set of actions that occur sequentially: the performance of the AI scripts is measured, rules in the rulebases are updated and new scripts are distributed to the AI characters. Before each learning episode the agent creates a subset of the available rules to use in the episode - this is known as a script. A free parameter n determines the size of the script. The script selection component uses a form of fitness proportionate

selection to select n rules (without replacement) from the complete set of rules based on their assigned weight value.

The third component is the rule policy, which determines how rules are selected within a learning episode. This component orderly processes the script component and performs the first rule that is applicable to the current game state. For example, a rule may require that a character's health be below 50%. If this is not the case then the rule does not apply. Rules are ordered by their priority. Even though priorities are generally assigned by the behavior developer, there is still some research being done on learning rule priorities in dynamic scripting [11]. In the event of a priority tie, rules are selected based on the highest rule value. This is the secondary use of rule values in the dynamic scripting algorithm.

Rule weight updating is the fourth component of dynamic scripting. The behavior developer creates a reward function that provides feedback on the utility of the script as a whole. High rewards indicate strong performance and low rewards indicate low performance. At the end of the learning episode, this reward function is used to create a single numeric reward for the agent's behavior. The full reward is given to each rule in the script that was successfully performed during the encounter. A half reward is given to each rule in the script that was not selected, which can happen because the rule was never applicable or because the rule had a relatively low priority. Compensation is applied to all rules that are not part of the script. Through the compensation mechanism, the rule weight updating component is responsible for distributing the rule weight “value points" among the available rules. As an example, if there are 10 rules with an initial weight value of 100, there are 1000 value points that can be distributed across all rules. A rule can have higher weight value than others because it was successfully activated in many winning scripts or because it was not selected to participate in losing scripts and the character lost many matches.

Dynamic Scripting is a technique that is considered by Spronck to be relatively faster than other learning techniques [3], such as evolutionary learning and neural networks, because of the lower number of training sessions (typically in the hundreds values instead of the thousands). One of the drawbacks of this technique is that the quality of the rules directly influences the quality of the learned behavior.

IV. APPLYING DYNAMIC SCRIPTING TO FPS

This section will explain how Dynamic Scripting was applied in our FPS prototype. We start by describing what customizations were made (Subsection IV-A), next we

show and explain the fitness function used (Subsection IV-B), then we describe the rules implemented (Subsection IV-C) and finally we list the learning parameters values of Dynamic Scripting used in our prototype (Subsection IV-D).

A. Customizations in a FPS

In Dynamic Scripting, learning is achieved within each episode. Choosing what each episode represents in the game is very important to achieve effective and reliable behaviors. This choice depends much on the genre of the videogame that we are applying Dynamic Scripting to. For instance, in a FPS videogame, maps (i.e., environments) are quite often very different from each other and a human player tends to play the same maps over and over again to improve their movement and learn effective strategies. Learning a behavior for every map is probably the best solution for this genre. Therefore, our application of Dynamic Scripting learns AI scripts for entire maps, where a "learning episode" is the entire playtime since the AI character starts a map until it dies or the objectives of that specific map are achieved. This results in one rulebase that adapts to specific situations for each map.

In a FPS videogame, characters act in real-time, without waiting for actions of others. This is very different from Spronck's implementation of Dynamic Scripting [3] in a turn-based game, where each agent chooses one action, i.e., one rule in the script, to perform by going through the entire script in each turn and then waiting for the turn of its opponents. In our implementation of Dynamic Scripting we developed a rule selection mechanism where at all times the AI script is sequentially read to find the first selectable rule. When that rule is found, the mechanism returns to the position of the first rule in the script (the one with the highest priority). That way, the rule with the highest priority that is currently selectable is at all times active.

B. Fitness Function

In the end of each "episode" the fitness function evaluates the success of each script. This function generates a value between 0 and 1 indicating how good the script performed during the last "episode". If it was a perfect performance, the agent controlled by this script played really well and the fitness value is 1. If it was a plain loss, the agent achieved basically nothing and the fitness value is 0. Since videogames tend to be quite different, there is no general fitness function which can be used in every one. Instead different functions have to be designed for each game, based on the goals of that particular game.

Typically, in a FPS videogame, the key element is the combat between the player and a number of opponents. To win, the player needs to defeat all opponents in the map by making their health points (HP) reach zero (Hit Points is also another valid

designation for the same metric that is often found in videogames). HP are a common concept in various types of games. Basically they are integer values modelling the physical condition of a character, the lower the more wounded. Whenever a character is hit by a weapon, his current HP are reduced based on the power of the weapon. There are certain objects that increase the current HP of a character to a maximum value defined in the beginning of the game.

In our prototype we defined a scenario to test Dynamic Scripting against a static AI (a typical non-adaptive finite-state machine). The setting is a match between the Dynamic Scripting agent and an opponent agent with the static AI. The agents fight each other and who ever reaches 0 HP first, looses. This scenario is further explained in the following subsection. The translation of this goal in a fitness function capable of evaluating the Dynamic Scripting agent correctly is presented below:

$$F(a,g) = \frac{4 * H(a) + 4 * D(g) + 2 * T(g)}{10}$$

In this equation, the several components have different factors that reflect their respective weight. The a parameter refers to the agent and g refers to the match. The components are H(a), that represents the remaining health of agent a, D(g), representing the total damage done to the opponent in the match g, and T(g), that represents how fast the agent won or how slow the agent lost in match g. We decided the weight values of each component after analysing the prototype scenario and testing different values. The best results were obtained with higher values in H(a) and D(g) than T(g). The equations for the different components are presented below:

$$T(g) = \begin{cases} \frac{t_t}{t_m}, & a \text{ lost;} \\ \frac{(t_m - t_t)}{t_m}, & a \text{ won.} \end{cases} \qquad D(g) = \frac{(h_0(o) - h_t(o))}{h_0(o)} \qquad H(a) = \frac{h_t(a)}{h_0(a)}$$

In these equations, o refers to the opponent of the agent, ht (x) refers to the HP of agent x in time t of the end of the match, h0 (x) refers to the HP of agent x in the

beginning of the match, tt refers to the time in seconds of the end of the match and tf refers to the maximum permitted time of the duration of the match, also in seconds.

When evaluating the agent's performance our fitness function prioritizes the damage dealt and the agent's remaining health over the time taken to complete the objective.

C. Scenario and characters

In the scenario implemented, the character using Dynamic Scripting AI and the character using the static AI are placed in an arena-type environment, with a few items distributed so that both characters can have easy access to them. There are three types of items: one is represented by a heart and increases the health of the character that picks it up (to a maximum of the initial health that each starts with), another, represented by a barrel, explodes if it is damaged, hitting anything that is near the blast and therefore damaging it, and the other is represented by a green box-like item and increases the character's ammo count (to a maximum of the initial ammo that each character starts with).

Each character can wield two different weapons that are used to decrease the health of the opponent: a rocket launcher and a machine gun, that shoots faster but making less damage than the first. Rockets from the rocket launcher explode when they collide, causing up to 100 HP of damage (values change depending on the distance from the point of impact). Bullets from the machine gun cause 5 HP of damage each. Each character starts with 200 HP and both have the same weapons and parameters, so that the only difference between them is the behaviour. The Dynamic Scripting character is able to learn different tactics, while the static character always uses the following tactic: if the opponent is not in range, the character patrols a predetermined area; if the opponent is in range, the character approaches the opponent; if the opponent is at half of the maximum shooting distance or more, the character uses the rocket launcher while approaching the opponent; if the opponent is at less than half the maximum shooting distance, the character uses the machine gun while staying put.

D. List of Rules

Rulebase design is one of the most important components of Dynamic Scripting. Each rule must be carefully designed to translate useful behaviour in the game, because Dynamic Scripting will learn tactics only as good as the rules implemented in it. A rule has essentially two components: a condition for the rule activation, and the action this rule translates in the game. Some rules can have no condition at all, but the majority has a condition dependent of some state of the game.

The implementation of each rule depends upon the programming language used and the game engine, as each must be manually designed. Since those dependencies were already chosen for us, we had to work with the available functions of the game engine to control characters. For example, if we want to implement a rule that makes the character move, we have to assign animations to the character in order to get more realistic movements, as well as assign velocity and destination parameters. Because of this, the implementation of each rule tends to be too confusing and long to list the source code in this paper. Therefore, we will show them in the form of tables that describe the condition and effect of each rule:

Name	AdvanceGunAttack
Condition	Character has machine gun ammo and can see an opponent
Effect	Advances towards the opponent and shoots with the machine gun if opponent is in range

Name	AdvanceRocketAttack
Condition	Character has rocket launcher ammo and can see an opponent
Effect	Advances towards the opponent and shoots with the rocket launcher if opponent is in range

Name	StationaryGunAttack
Condition	Character has machine gun ammo and can see an opponent and opponent is in shooting range
Effect	Remains in the same place and shoots the opponent with the machine gun

Name	StationaryRocketAttack
Condition	Character has rocket launcher ammo and can see an opponent and opponent is in shooting range
Effect	Remains in the same place and shoots the opponent with the rocket launcher

Name	SidestepGunAttack
Condition	Character has machine gun ammo and can see an opponent and opponent is in shooting range
Effect	Moves sideways while shooting the opponent with the machine gun

Name	SidestepRocketAttack
Condition	Character has rocket launcher ammo and can see an opponent and opponent is in shooting range
Effect	Moves sideways while shooting the opponent with the rocket launcher

Name	BarrelGunAttack
Condition	Character has machine gun ammo and there is a barrel close to an opponent that is in shooting range
Effect	Shoots the barrel with the machine gun

Name	BarrelRocketAttack
Condition	Character has rocket launcher ammo and there is a barrel close to an opponent that is in shooting range
Effect	Shoots the barrel with the rocket launcher

Name	TakeAmmo
Condition	Character does not have ammo in at least one his weapons and he can see an ammo item
Effect	Advance towards the ammo item

Name	TakeHealth
Condition	Character has less than 25% of health and can see a health item
Effect	Advance towards the health item

Name	Escape
Condition	Character has less than 25% of health and can see an opponent
Effect	Advances in the opposite direction of the opponent

Name	Idle
Condition	Character cannot see the opponent
Effect	Remains stationary

Name	Patrol
Condition	Character cannot see the opponent
Effect	Advances to predetermined locations sequentially

Name	Search
Condition	Character cannot see the opponent and has its last known position
Effect	Advances to the last known position of the opponent

Besides this list of rules, characters need to have a default rule in their script that can always be activated, to make sure that if no other rule in the script can be activated, at least it is possible to activate the default rule at all times. The reason for this is that characters must always have an action selected, even if that action is just an animation of the character standing still, as this is a requirement of most game engines. In our scenario, the default rule is equal to the Idle rule described above but without the condition, and each script has a total of 4 different rules.

V. EXPERIMENTAL RESULTS

To obtain experimental results in our prototype, some changes were required to allow automatic generation of results, as the game engine used is designed to provide interactive environments to develop videogames, and not test-beds for AI techniques. Batches of tests were time-consuming to process, since there is no feasible way to turn off the graphical representation. Each character's action must be animated, therefore even a fast computer could not speed up the process.

In our experiments, the Dynamic Scripting controlled character (henceforth referred as "dynamic character") is matched against a character controlled by static AI (henceforth referred as "static character"), to measure the comparative strength between each. When one of the characters is defeated, the environment is reset to the initial situation and after a number of matches all rule weights are discarded and the learning starts again. A sequence of matches with no rule weights reset in between is called a batch.

To obtain these results, we registered the dynamic character fitness values in 5 batches of matches, where each batch contains 100 matches. In the beginning of each batch the rule weight values are reset, so new learning can occur. With these 5 batches, we can observe the dynamic character learning in 5 separate experiments, and compare the fitness values obtained. We averaged the resulting values of each match from the 5 batches and present them below in a graphical representation (Figure 1). The most used and less used rules by the dynamic character are also registered and presented below.

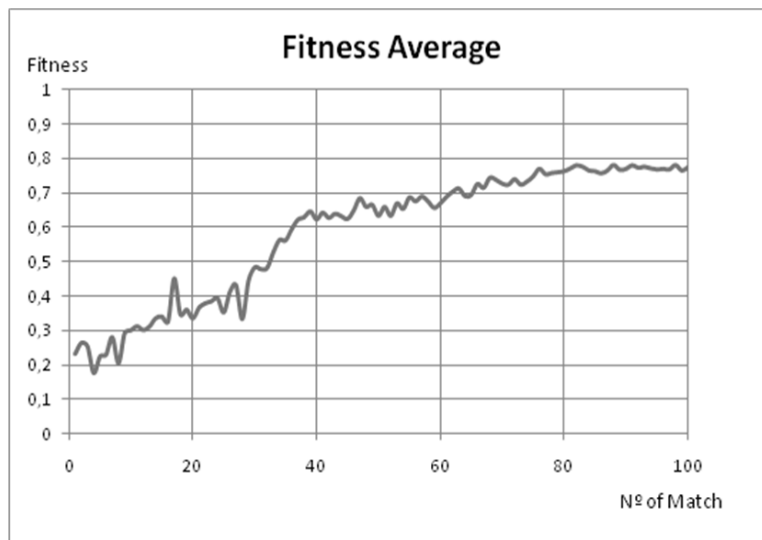

Figure 1 – Represents the fitness average of the 5 batches in each match. The xx coordinate is the number of the match and the yy coordinate is the fitness value.

We can observe that in the first 30 matches the average fitness values are below 0.5 and fluctuating. This means that in all batches the dynamic character lost more times than it won, although there were peaks of high and low fitness. This probably resulted from rules being tested out and their values changing, when there was more exploration of rules than exploitation. From the 40th match onwards, the fitness steadily raised to above 0.7. In these matches, the best rules were probably chosen already, or in the process of being discovered. The best average fitness value registered was 0.779.

The most used rule, i.e., the rule that was chosen for more scripts, was SidestepRocketAttack, followed by TakeHealth. Since rockets do more damage than machine gun bullets, it was predictable that rules involving rockets were going to have better values. Also, moving to the sides while shooting is a good strategy to evade damage. The TakeHealth rule allows a character to gain health when it is almost defeated. This probably saves many matches and allows better fitness values.

The less used rule, i.e., the rule that was chosen for fewer scripts, was Idle, followed by TakeAmmo. This was predicted, as the Idle rule does not translate to any useful behavior, and was inserted in the rulebase for testing purpose only. The TakeAmmo rule was also least selected probably because matches are somewhat fast, and running out of ammo is not that frequent.

VI. CONCLUSIONS AND FUTURE WORK

By examining the experimental results described in the previous section, we can conclude that it is possible for an AI with Dynamic Scripting to learn tactics that successfully exploit weaknesses in an agent controlled by a static AI in a FPS videogame. The time required to apply Dynamic Scripting to a FPS videogame prototype is not greater than applying and testing a static AI, since behaviours developed for a static AI can be integrated in rules for use in Dynamic Scripting. Therefore, future FPS videogames developed by our sponsor can use Dynamic Scripting AI without having to waste much more time than the required to develop a static AI, with the added bonus of having AI characters that adapt their behaviours to the game environment.

For future work, we intend to further expand our prototype by adding different and more complex scenarios and character types that use different rulebases, as time constraints did not allowed for this to be done in the current version. Adding more complex rules that incorporate the character's perception of the state and behaviour of the opponent, as well as adding more opponents and/or teams of characters, are improvements meant to be implemented in the next versions. Also, there is room for improvement of the Dynamic Scripting algorithm, as is described in [10], [11], [12] and [13]. The Goal-Directed Hierarchical approach for Dynamic Scripting described in [13] seems interesting enough to be applied to our FPS prototype, as many videogames of this genre have characters with specific goals and sub-goals.

REFERENCES

[1] P. Tozour, The perils of AI scripting, in AI Game Programming Wisdom, S. Rabin, Ed. Charles River Media, Inc., Hingham, MA., USA, 2002, pp. 541–547, ISBN 1-584-500-778.

[2] I. Millington, Artificial Intelligence for Games. San Francisco, California: Morgan Kaufmann Publishers Inc., 2006, ch. Decision Making, pp. 301–471, ISBN 0-124-977820.

[3] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, Adaptive game AI with dynamic scripting, Machine Learning, vol. 63(3), pp. 217–248, 2006.

[4] Tozour, P., First-Person Shooter AI Architecture. Available in: Game AI Programming Wisdom. Ed. Steve Rabin, Charles River Media, Hingham, MA, 2002.

[5] Benjamin, G., An Empirical Study of Machine Learning Algorithms Applied to Modelling Player Behaviour in a First Person Shooter Video Game, University of Wisconsin - Madison, USA, 2002.

[6] Hoorn, N., Togelius, J., Schmidhuber, J., Hierarchical Controller Learning in a First-Person Shooter in IEEE Symposium on Computational Intelligence and Games, 2009.

[7] Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E., Online Adaptation of Computer Game Opponent AI. Proceedings of the 15th Belgium-Netherlands Conference on AI. pp. 291-298, 2003.

[8] McPartland, M., Gallagher, M., Learning to be a Bot: Reinforcement Learning in Shooter Games, Proceedings of the Fourth Artificial Intelligence and interactive Digital Entertainment Conference, Stanford, California, 2008.

[9] M. Vasta, S. Lee-Urban, and H. Muñoz-Avila, RETALIATE: Learning Winning Policies in First-Person Shooter Games, Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference, 2007.

[10] Ponsen, M.; Muñoz-Avila, H.; Spronck, P.; and Aha, D. 2006. Automatically generating game tactics with evolutionary learning. Available in: AI Magazine 27(3): pp. 75–8.

[11] Timuri, T., Spronck, P., & van den Herik, J. (2007). Automatic rule ordering for dynamic scripting. Paper presented at the Artificial Intelligence in Interactive Digital Entertainment, Stanford, CA.

[12] Ludwig, J., Extending Dynamic Scripting, Department of Computer and Information Science, University of Oregon, Ann Arbor: ProQuest/UMI, 2008.

[13] Dahlbom, A., Niklasson, L., Goal-Directed Hierarchical Dynamic Scripting for RTS Games, School of Humanities and Informatics, University os Skövde, Sweden, 2006.