INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE INTELIGÊNCIA ARTIFICIAL




Ajuste Dinâmico de Dificuldade em Videojogos Usando Aprendizagem Automática

JORGE EMANUEL COELHO MENDONÇA DE ANCIÃES FELÍCIO
outubro de 2024

P.PORTO

Dynamic Difficulty Adjustment in Video Games Using Machine Learning

Jorge Emanuel Coelho Mendonça de Anciães Felício

Student No.: 1181244

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Artificial Intelligence

Supervisor: Luiz Felipe Rocha de Faria, Coordinating Professor at Instituto Superior de Engenharia do Porto do Instituto Politécnico do Porto

Evaluation Committee:
President:
Isabel Cecília Correia da Silva Praça Gomes Pereira, Coordinating Professor at Instituto Superior de Engenharia do Porto do Instituto Politécnico do Porto

Members:
António Vieira de Castro, Adjunct Professor at Instituto Superior de Engenharia do Porto do Instituto Politécnico do Porto
Luiz Felipe Rocha de Faria, Coordinating Professor at Instituto Superior de Engenharia do Porto do Instituto Politécnico do Porto

Porto, October 14, 2024

Statement of Integrity

I declare that I have conducted this academic work with integrity.

I have not plagiarized or engaged in any form of misuse of information or falsification of results

throughout the process leading to its creation.

Therefore, the work presented in this document is original and authored by me, having not

been used previously for any other purpose.

I also declare that I am fully aware of the P.PORTO Ethical Conduct Code.

ISEP, September 2024

Abstract

In the constantly evolving field of video games, traditional difficulty settings fail to accommodate the wide range of skill levels among players. The resulting mismatch between the player's skill and the game's challenge can make the game boring for skilled players or frustrating for less experienced ones, negatively affecting player engagement. Dynamic Difficulty Adjustment (DDA) seeks to resolve this issue by adapting the game's difficulty in real time in response to the player's performance. While advancements in artificial intelligence (AI), particularly machine learning (ML), have enabled more adaptive DDA systems, the full potential of certain advanced techniques or tools has yet to be explored.

This thesis thus explores possible innovations in the integration of AI in DDA systems for video games. The research begins by reviewing the techniques used for DDA, focusing on methodologies such as player modeling, rule-based systems, and ML. Based on this research, potential areas for innovation were identified and the application of Deep Reinforcement Learning (DRL) in the Unity game development platform through the usage of the ML-Agents toolkit was chosen as a promising approach for this research.

Using this methodology, this research aims to implement a DDA system that adjusts a game's difficulty based on the player's skills, enhancing their engagement and maintaining a consistent challenge. This project has several critical phases of development, including the creation of a game prototype, data collection for model training, development and integration of the DDA system into the game prototype, and conducting an experiment comparing the prototype with DDA integrated with a version of the prototype that used traditional static difficulty scaling.

The experiment conducted was done with 20 participants of varying skill levels and used a combination of collected gameplay metrics and a modified Game Experience Questionnaire (GEQ) survey to evaluate the DDA system's effectiveness.

The results showed that the DDA system demonstrated a statistically significant increase in the player engagement component and appropriately adjusted the difficulty to be harder for participants of higher skill. However, the system sometimes exhibited some issues with drastic adjustments in difficulty between levels, which led to a slightly lower Post-Game positive experience score compared to the static difficulty scaling system. Despite these fluctuations, the proposed system demonstrates the potential of the ML-Agents toolkit in implementing DDA with DRL in games made on the Unity platform.

By identifying underexplored areas in the current literature and applying advanced techniques like DRL, this thesis aims to contribute to both academic research and game development regarding the approach to DDA in video games.

Keywords: Dynamic Difficulty Adjustment, Artificial Intelligence, Machine Learning, Video Game Design, Deep Reinforcement Learning, Unity Platform

Resumo

Na área dos videojogos, as configurações de dificuldade tradicionais não conseguem acomodar a ampla variedade de níveis de habilidade dos jogadores. A irregularidade resultante entre a habilidade do jogador e o desafio do jogo pode tornar o jogo demasiado aborrecido para jogadores mais experientes ou demasiado frustrante para os menos experientes, afetando negativamente o envolvimento dos jogadores. O Ajuste Dinâmico de Dificuldade (DDA) procura resolver este problema, ajustando a dificuldade do jogo em tempo real em resposta ao desempenho do jogador. Embora os avanços na inteligência artificial (IA), particularmente na aprendizagem automática (ML), tenham permitido sistemas de DDA mais adaptáveis, o potencial total de certas técnicas ou ferramentas avançadas ainda não foi totalmente explorado.

Esta tese explora possíveis inovações na integração da Inteligência Artificial (IA) em sistemas de Ajuste Dinâmico de Dificuldade (DDA) para videojogos. O estudo começou com a revisão das técnicas utilizadas para DDA, com foco em metodologias como modelação de jogadores, sistemas baseados em regras e aprendizagem automática. Com base nesta investigação, foram identificadas áreas com potencial para inovação e a aplicação de Deep Reinforcement Learning (DRL) na plataforma de desenvolvimento de jogos Unity, através do uso do kit de ferramentas ML-Agents, foi escolhida como uma abordagem promissora para este trabalho de investigação.

Usando esta metodologia, este estudo procurou implementar um sistema de DDA que ajuste a dificuldade do jogo com base nas habilidades do jogador, aumentando o seu envolvimento e mantendo um desafio consistente. Este projeto inclui várias fases críticas de desenvolvimento, tais como a criação de um protótipo de jogo, a recolha de dados para o treino do modelo, o desenvolvimento e integração do sistema de DDA no protótipo, e a realização de uma experiência que compara o protótipo com DDA integrado com uma versão do protótipo que utiliza um sistema tradicional de escalamento de dificuldade estática.

A experiência foi realizada com 20 participantes de diferentes níveis de habilidade e usou uma combinação de métricas de jogo recolhidas e um questionário modificado da Experiência de Jogo (GEQ) para avaliar a eficácia do sistema de DDA.

Os resultados mostraram que o sistema de DDA demonstrou um aumento estatisticamente significativo no envolvimento dos jogadores e ajustou adequadamente a dificuldade para ser mais difícil para os participantes de maior habilidade. No entanto, o sistema apresentou, por vezes, problemas com ajustes drásticos de dificuldade entre níveis, o que resultou numa pontuação ligeiramente inferior na experiência positiva pós-jogo em comparação com o sistema de escalamento de dificuldade estática. Apesar destas flutuações, o sistema proposto demonstra o potencial da ferramenta ML-Agents para a implementação de DDA com DRL em jogos desenvolvidos na plataforma Unity.

Ao identificar áreas pouco exploradas na literatura atual e ao aplicar técnicas avançadas como DRL, esta tese procura contribuir tanto para a investigação científica como para o desenvolvimento de jogos no que diz respeito à abordagem de DDA em video jogos.

Palavras-chave: Dynamic Difficulty Adjustment, Artificial Intelligence, Machine Learning, Video Game Design, Deep Reinforcement Learning, Unity Platform

Acknowledgement

First, I would like to sincerely thank my thesis supervisor, Professor Luiz Faria, for his invaluable guidance throughout the thesis process. His constructive feedback and willingness to clarify my questions were essential to helping me stay on track and improve my work.

I'm also deeply grateful to Instituto Superior de Engenharia do Porto (ISEP) for providing such a solid education during my master's, and to the many dedicated and caring professors who supported us throughout these years.

My heartfelt thanks also goes to my family and friends for their unwavering support and constant encouragement. I'd like to specifically mention my parents, my sister, and my grandmother, for their constant support. My close friend, Nuno Resende, also deserves a sincere thanks for always being willing to offer constructive criticism on the thesis.

To those who generously volunteered as research participants for this thesis- your help was invaluable, and I truly appreciate the time and effort you dedicated.

I would also like to give a special thanks to my colleague, Gabriel Ribeiro, for always being a thoughtful and reliable project partner throughout our master's journey.

I am also grateful to my trainer, Paulo Vasconcelos, for encouraging me to stay physically active and fit throughout this whole process.

Lastly, I'd like to honor my dear dog, Kira, who sadly passed away during the first year of my master's. Her companionship was a valued source of comfort, and she will always be remembered.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AI	Artificial Intelligence.
DDA	Dynamic Difficulty Adjustment.
DL	Deep Learning.
DRL	Deep Reinforcement Learning.
FPS	First-Person Shooter.
GDPR	General Data Protection Regulation.
GEQ	Game Experience Questionnaire.
ID	Identification.
IQR	Interquartile Range.
JSON	JavaScript Object Notation.
ML	Machine Learning.
MMORPG	Massively Multiplayer Online Role-Playing Game.
MPRL	Multiple-Periodic Reinforcement Learning.
NN	Neural Networks.
NPC	Non-Player Character.
PC	Personal Computer.
RL	Reinforcement Learning.
UX	User Experience.
VR	Virtual Reality.

Glossary

Artificial Intelligence	The simulation of human-like intelligence in machines, such as being programmed with the ability to learn or process complex data.
Curriculum Learning	A training strategy in machine learning where tasks are presented in a progressively challenging order. The agent starts by learning how to do simple tasks, which then progresses to more complex ones, allowing the agent to build upon previous knowledge.
Deep Reinforcement Learning	An advanced form of reinforcement learning that utilizes deep neural networks.
Dynamic Difficulty Adjustment	A method in video games where the difficulty level is automatically adjusted in real-time based on the player's performance.
Evolutionary Algorithms	A subset of AI based on algorithms that draw inspiration from biological evolution, focused on iteratively evolving towards optimal solutions for a given problem.
First-Person Shooter	A genre of video games focused on weapon-based combat in first-person perspective.
Flow State	A mental state in which a person is fully immersed in an activity, enhancing their focus and enjoyment.
Imitation Learning	A method in machine learning where an agent learns to perform tasks by observing and imitating the behavior of an expert. Instead of learning through trial and error, it is trained on a dataset composed of demonstrations from an expert, which significantly improves the performance of the agent.
Machine Learning	A subset of AI focused on algorithms and models that allows for computers to learn and adapt without explicit instructions.

Massively Multiplayer Online Role-Playing Game	A genre of video games that mixes elements from role-playing games with the vast worlds and large player bases from the massive multiplayer online (MMO) genre.
Multiple-Periodic Reinforcement Learning	A type of machine learning that uses neural networks with multiple layers of processing.
Multiple-Periodic Reinforcement Learning	A specialized form of reinforcement learning that evaluates multiple objectives over different periods.
Neural Networks	Computational models inspired by the human brain's structure and function. Used in machine learning to recognize patterns, process data and make decisions.
Non-Player Character	Characters in a video game that are not controlled by the player, but instead by the game's AI.
Player Behavior Modeling	The process of utilizing computational methods to understand and predict the player's behavior.
Procedural Content Generation	Techniques in game design that automatically create game content, making the experience for each player unique.
Reinforcement Learning	A type of machine learning where an agent learns to make decisions by performing actions in an environment to achieve some reward.
Roguelike	A game genre characterized by traversing through procedurally generated levels and permanent death of the player character, resetting the loop.
Souslike	A game genre characterized by challenging combat and a significant emphasis on the player's skill.
User Experience	The experience of a user using a product.
Virtual Reality	A simulated experience created by computer technology that immerses the user in a virtual environment.
WASD	Standard movement keys (used in games for directional controls).

Chapter 1

Introduction

1.1 Contextualization

Dynamic Difficulty Adjustment (DDA) is an important concept in modern video game design and is a significant evolution from static, traditional gaming experiences. Its importance lies in the ability to dynamically tailor the game's behavior to individual player's skill levels, therefore maintaining an engaging experience and player satisfaction throughout the gameplay.

Central to DDA is the concept of 'flow state', a term coined by psychologist Mihaly Csikszentmihalyi (Csikszentmihalyi 1990). This state occurs when a person's skill level is perfectly matched with the challenge presented, leading to a heightened state of focus and engagement. In the field of video games, achieving this balance is critical for maintaining player engagement and satisfaction. DDA seeks to keep players in this state by dynamically altering the game's difficulty, ensuring that the player is neither bored nor frustrated. Figure 1.1 provides a visual representation of Csikszentmihalyi's flow state model, showing the desired diagonal "flow zone", which lies between the undesired states of anxiety and boredom.

The traditional approach to dealing with difficulty in video game design often involves multiple static difficulty levels (Ex: Easy, Medium, or Hard), which are rigid and usually do not account for the individual player's characteristics, like their skill level, dexterity, or learning rate. In contrast, DDA introduces a more player-centric approach to game difficulty, in which the game adapts dynamically to the player instead of the opposite. This leads to a more inclusive game experience that accommodates a wider range of player's skill levels and preferences, preventing disengagement from the player due to poorly matched difficulty levels.

The evolution of DDA has been significantly influenced by various advancements in Artificial Intelligence (AI). Early DDA systems normally relied on rule-based algorithms, which, while effective, had limited flexibility. Later integration of Machine Learning (ML) and other AI techniques has led to more responsive and adaptive DDA systems that learn from player data to adapt the difficulty of maintaining engagement. Certain modern DDA systems have even gone beyond reactive adjustments and have the capacity to anticipate future challenges to keep players engaged due to incorporating predictive mechanisms. This thesis builds upon these advancements by exploring innovative underutilized AI techniques for DDA, as detailed in Section 2.2, to propose a novel implementation.

Overall, an effective DDA system should be capable of tracking and adapting in response to the player's evolving skill level, continually adjusting without disrupting the cohesiveness of the gaming experience. AI-based DDA systems, particularly with ML integrated, make

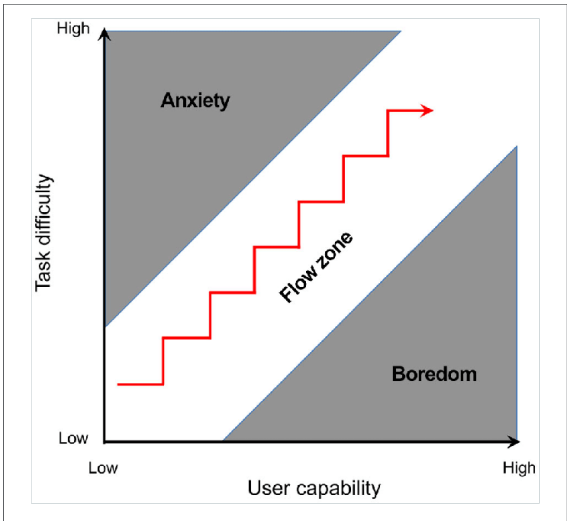


Figure 1.1: Illustration of Flow State (Hiremath et al. 2015).

more nuanced and effective adjustments, leading to a more personalized gaming experience (Zohaib 2018).

The integration and evolution of DDA in video games represents a significant difference in how games are enjoyed or experienced, influencing modern game design philosophy to prioritize player experience and engagement. As AI technology continues to evolve, so too will the effectiveness of DDA systems and their importance in the future of video game development.

1.2 Problem Statement

The field of video game design is constantly evolving, yet the challenge of creating a truly dynamic DDA system persists. As mentioned in section 1.1, a persistent issue in video game design is the static approach to game difficulty, which fails to consider the wide range of skill levels in the global player base. While various advancements in DDA have been made, using AI to create more adaptive gaming experiences, there remains untapped potential in the area. Certain DDA systems, for example, often disturb the game's flow or respond to the player's performance after the fact instead of adjusting preemptively to the player's growing skill.

Additionally, while advancements in AI have revolutionized DDA through ML and Neural Networks (NN), the full potential of certain advanced methodologies, like Deep Reinforcement Learning (DRL), have not been fully explored or are yet to be realized in this domain, especially within less mainstream game genres that may benefit significantly from DDA.

Thus, the purpose of this research is not only to investigate and identify underutilized AI techniques and potential areas for innovation but also to implement and evaluate a novel solution based on these research findings. This system will then be applied to a game genre or game development platform that, to date, appears to have been less explored in the context of the researched studies for DDA, with the objective of filling out this gap in current research.

1.3 Objectives

The primary goal of this thesis is to propose an innovative DDA system that builds upon existing methodologies and techniques used in the area. The effectiveness of this system is then assessed through the implementation of a video game prototype that incorporates the proposed system. Through this, the research seeks not only to contribute to academic knowledge in the field but also to offer practical insight for game developers.

This project is composed of several phases, with each one being dedicated to a specific objective crucial to the overarching goal of the project. Each phase of the project builds upon the previous phase, guaranteeing that the development approach of this system is comprehensive and well-documented.

1. **State of the Art Research**: This phase is dedicated to researching different techniques that were used for DDA, such as player modeling, rule-based systems, and various ML techniques. The purpose of this phase is to identify areas that have potential for innovation, which would then serve as a basis for the choice of video game genre for the prototype and the technique that would be used for the DDA system.
2. **Prototype Creation**: Development of a simple game prototype.
3. **Data collection integration**: Designing and integrating a data collection mechanism in the game prototype to gather player behavior and performance metrics, adhering to data anonymity regulations. The data collected is then used to train the model for the DDA system.
4. **Prototype DDA system Development**: Implementing the DDA system within the game prototype using the chosen ML technique.
5. **Validation Experiment**: The effectiveness of the DDA system is assessed by conducting an experiment with 20 participants of varying game experience and skill levels. Each participant played two versions of the prototype that were tested: one with the DDA system, and the other with static difficulty scaling. A modified Game Experience Questionnaire (GEQ) (IJsselsteijn, de Kort, and Poels 2013) was then used to measure the different game experience components when playing both prototypes.
6. **Results Analysis**: The results of the experiment are analyzed and discussed to evaluate the proposed system.

Chapter 2

State of the Art

2.1 DDA: An Evolutionary Overview

This section is dedicated to outlining the evolution of DDA, from its origin in the nascent stages of video gaming to its current state-of-the-art implementations observed in recent years.

2.1.1 Early Beginnings

The concept of DDA emerged in the early years of the video game industry, where the challenge was to create a game that was accessible to a wide range of players. Classic arcade games were the first to incorporate a rudimentary form of DDA, with the objective of keeping players engaged so that they could insert more coins. One of the first examples of difficulty adjustment was the unintended result of a glitch. In this game, 'Space Invaders (1978)', the enemies became faster as the player progressed. This was caused due to the initial swarm of enemies overloading the game, thus causing lag. As the enemies were then defeated by the player, this freed up resources, causing the remaining enemies to become faster. Due to this feature, the game became largely known for being the foundation for the concept of 'difficulty curves', in which the difficulty increases as time passes (Jagoda 2018).

Despite being largely revolutionary at the time, these initial implementations of DDA were still quite basic, as any changes to the difficulty were based on predetermined triggers, such as number of enemies defeated or time passed, rather than changing based on the analysis of the player's skill or performance.

Eventually, these systems shifted from purely mechanical triggers to more player-centric adjustments with the introduction of rule-based systems.

2.1.2 The Shift in Difficulty Adjustment

Albeit still limited in scope, the introduction of rule-based systems for DDA allowed for games to adjust their behavior based on the player's skills.

One of the early examples of the implementation of a rule-based DDA system was in the 2001 video game 'Max Payne'. Through the usage of predefined rules, the game was able to adjust certain parameters of its difficulty based on certain performance metrics of the player, such as reducing the enemy's health and accuracy if the player was frequently dying in one level (Hullett and Whitehead 2010).

These rule-based systems were a significant milestone since games now could have the capacity to respond and adapt to the player's performance, albeit within a fixed framework.

Building upon this foundation, improvements were made by the industry to the basic rule-based systems, such as the integration of more complex player performance metrics. While still operating under predefined rules, these systems could now adjust the difficulty in a more nuanced manner by considering a broader range of player-related data, such as hit/miss ratio and level completion times (Pedersen, Togelius, and Yannakakis 2009).

Another advancement in DDA was the introduction of scripted adaptive difficulty, which incorporates a level of adaptation to the traditional rule-based systems. An example of such a technique is explored in the article 'Online Adaptation of Computer Opponent AI' (Spronck, Sprinkhuizen-Kuyper, and Postma 2003), in which the script of the predefined rules of the computer opponent adapts to the performance of the player by changing certain weights of the predefined parameters. By doing this, these systems could start to understand and adapt to player behavior less rigidly, and more sensitively to context.

These advancements in DDA eventually led to the next evolutionary step of integrating ML and AI into these systems. These emerging technologies led to a shift in these systems, changing static rules into dynamic and adaptive systems capable of adapting to the players in real-time.

2.1.3 Harnessing AI into DDA

During the early 2000s, new AI-based approaches began to be integrated into DDA systems. Initial efforts focused on introducing basic ML algorithms, which laid the groundwork for more recent innovations of AI algorithms in DDA. An example from this early era is the study by Charles et al. (Charles and Black 2004), which explores the use of NN to adapt the game's difficulty in response to the player's performance.

As AI and ML progressed, DDA systems began to utilize AI to predict and analyze the player's behavior instead of depending on purely rule-based systems. The study by Hunicke (Hunicke 2005) during this early period discusses the use of ML for DDA and describes the growing interest in games becoming more intelligent and adaptable for each player.

With the usage of ML, researchers were also able to create predictive models capable of reacting to the player's action and predicting their future needs, which allowed DDA systems to take a more proactive approach to difficulty adjustment. One such research by Pedersen et al. (Missura and Gärtner 2009) focused on modeling the player experience in the video game "Super Mario Bros" by linking different characteristics such as the player's emotions, behavior and the level design's characteristics. Such models show potential in being used for procedural game level generation that adapts to the player.

As the potential of AI in DDA becomes more widely explored, DDA systems start to learn from player actions and are constantly evolving and adapting in real time in response to player interaction, guaranteeing a challenge uniquely appropriate to that player. A more recent example of this advanced integration of AI is the application of DRL in a recent work by Or et al. (Or, Kolomenkin, and Shabat 2021), which showcases a system that optimizes the user's experience through a User Experience (UX)loss function.

This new approach to DDA not only reflects the technological advancements of AI but also a shift in how developers approach challenge and player satisfaction in games. With the context into how DDA systems have progressed, the next section shall describe the state-of-the-art techniques used for such systems.

2.2 Techniques in DDA

This section will explore in detail the various techniques and methodologies used for DDA, along with describing various studies in which they were implemented for DDA. By researching these different techniques, a better understanding will be gained of what methodologies have been driving the field of DDA, and what techniques have shown promise for future innovation.

2.2.1 Player Modeling

Player modeling is an essential concept of DDA and refers to the creation of a computational representation of the player's skills, behavior, and preferences, which is then used for implementation of the DDA system.

These models are not static, and constantly change in response to the player's actions so that they accurately reflect the player's abilities. To do this, these models need to track certain metrics, such as reaction times, accuracy, and progression rates.

One example of this is the study by Missura et al. (Missura and Gärtner 2009) in which they used Player Modeling in combination with clustering techniques in their approach for DDA. They first gathered the player data by logging various gameplay elements, like score, character's health, and difficulty level, along with their timestamp. After gathering this data, they used clustering to identify the patterns from the different player types. With the clustering done, their algorithm averaged the data within each cluster to obtain a model for each player type, which was then used to predict which cluster new players likely belong to, adjusting the game difficulty accordingly.

In their master thesis, Rodriguez (Rodriguez 2020) follows a similar approach as the above study for using player models to classify players. They collected the data both from gameplay metrics, but also from surveys made to the players, analyzing how the player's self-perceived skill correlated with their actual performance. After gathering this data, they used clustering and classification models to predict the player's skill levels and then applied this model to predict the skill of new participants.

In a different study, Pfau et al. (Pfau, Smeddinck, and Malaka 2019) used an advanced form of player modeling called Deep Player behavior Models. Unlike traditional models, these models use NN to continuously learn and process gameplay data. In this study, the objective of this model was to continually adapt an enemy that responds to the player's actions. To implement this, they used a multilayer perceptron NN that constantly processes the data gathered, which includes various player actions and contextual game elements like movement patterns, damage taken, and frequency of specific actions.

To evaluate their approach, they designed a platform fighting game called "Korona: Nemesis" and tested different enemy types, including basic, random, optimal, and player-based ones. After comparing the different approaches, they concluded that the player-based one indicated a significant improvement in the player's experience.

In a more recent study by the same authors (Pfau, Smeddinck, and Malaka 2020), Deep Player behavior Models were implemented in a Massively Multiplayer Online Role-Playing Game (MMORPG) they created to study the effects of the models on long-term engagement, comparing it with traditional parameter tuning methods. They found that the models increased the player's intrinsic motivation, as indicated by higher scores in effort-importance,

and players preferred game experiences featuring the models in comparison to those featuring traditional methods.

As can be observed from these studies, Player Modeling has been an important component for DDA and there are various approaches in how one can implement this technique, such as with the clustering demonstrated by Missura et al (Missura and Gärtner 2009).

Notably, the advancement of AI has led to the creation of more advanced models, such as Deep Player behavior Models, which utilize NN. As was observed in both studies by Pfau et al. (Pfau, Smeddinck, and Malaka 2019) (Pfau, Smeddinck, and Malaka 2020), the models were able to be used to create enemies that continuously adapt to the player's behavior, leading to an increase in player satisfaction and engagement.

2.2.2 The Traditional Approach to Adaptation: Rule-based systems

Rule-based systems were one of the first AI techniques that were used for DDA systems. While less adaptive compared to their modern counterparts, such methods served as the basis for modern DDA systems, which still incorporate the basic principles albeit in a more nuanced manner.

Rule-based systems can be characterized by their reliance on a set of predefined rules that determine how the game's difficulty adapts in response to specific actions from the player, or other values of the game. To decide which predetermined rule is triggered to adjust the difficulty, such systems monitor various game and player-related metrics. Here are some common types of metrics used:

- **Player Performance** – These are values directly related to the player's skill, such as his score, level completion time, and accuracy.
- **Health and Damage** – Includes the player's health and damage taken. By tracking these values, the game could adjust the damage of the enemy and the availability of healing items.
- **Consumable Resources** – These are limited resources that the player can utilize, such as healing items or ammunition. By tracking the amount of resources, the game can adjust the availability of these resources.
- **Enemy Encounter Outcomes** – Involves tracking the values related to the player's encounter with enemies, such as the number of enemies defeated or which enemy type causes the most deaths. By tracking this data, the system can adjust the number of enemies faced at once or the frequency of a certain type of enemies.

In response to these values reaching a certain predetermined threshold, the system triggers generally straightforward rules that affect the difficulty of the game.

Figure 2.1 is a flowchart of a simple yet effective rule-based system for a hypothetical Virtual Reality (VR) game based on whack-a-mole. The system observes whether the mole is hit within the defined TOH (Time Out of Hole), and adjusts the TOH depending on the player's performance.

In more advanced implementations, a rule-based system can utilize mathematical formulas for a more accurate DDA system. For example, in "The case for dynamic difficulty adjustment in games" (Hunicke 2005), various formulas are proposed for a more nuanced and

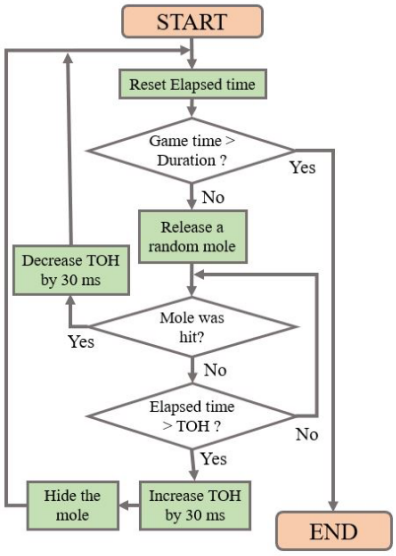


Figure 2.1: Rule-based System Flowchart Example (Caldas et al. 2023).

precise difficulty adjustment. One of the proposed formulas, shown in Figure 2.1, estimates the probability of player death based on Gaussian distribution.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.1}$$

Where:

- $p(x)$ is the probability density function of the Gaussian distribution.
- x represents the damage values.
- μ is the mean damage.
- σ is the standard deviation of the damage.

By using these formulas to calculate the probability of receiving a certain amount of damage or of dying at any given time, the game can have a more dynamic assessment of the player's situation in real-time.

Overall, rule-based systems, with their straightforward implementation and potential for improving their efficacy through the implementation of probabilistic formulas, still show some effectiveness in DDA. However, their rigid structure and simplicity can lead to less personalized game experiences not truly capable of suiting their difficulty for the individual player's skill (Zohaib 2018).

2.2.3 Evolutionary Algorithms

Evolutionary Algorithms are a subset of AI that draw inspiration from natural biology such as natural selection or genetics (Miettinen and Neittaanmaki 1999). In the context of DDA, such algorithms are generally used to find the most optimal solution for each player (Andrade et al. 2016), and usually iteratively follow these steps:

- **Generation** – A diverse population of game difficulty parameters is generated.
- **Evaluation** – Each set of parameters is evaluated against player performance metrics, evaluating how well each difficulty setting aligns with the player's gameplay style.
- **Selection** – From the population, the sets of parameters that best match the desired player experience (highest fitness) are selected.
- **Adaptation** – The selected parameters undergo mutation to generate the next population, creating new difficulty configurations that might better fit the player's evolving skills.

A study by Andrade et al. (Andrade et al. 2016) explores the use of Evolutionary Algorithms for DDA in the context of rehabilitation robotics. Their approach utilizes the algorithm to modify two key game parameters based on the user's performance to achieve the optimal challenge for rehabilitation. These key parameters, target speed, and distance were constrained within the defined boundaries of the game, so that, for example, the target distance would not exceed 50 percent of the game's screen.

The used fitness function was a coefficient representing the game distance, game speed, the player's position relative to the target, and the player's use of time. This function was designed in such a way that the fitness value changed in response to the round's difficulty and player experience. The fitness value increased proportionally with the difficulty of the round to ensure the challenge of the game, but also decreased for rounds that were too frustrating for the player, maintaining the balance between challenge and frustration. Results of the study showed that the adjustment successfully led to a difficulty level that more closely matched the player's skill.

A different study by Weber et al. (Weber and Notargiacomo 2020) focused on the application of Genetic Algorithms for DDA in the "Empire Game", which is a mobile game designed for the iPad. In this game, players and AI-controlled players compete for resources on a planet, to construct buildings to score points. The integration of the algorithm involved giving each player a unique population for individual difficulty adjustment, with the implementation of metrics such as continuous fitness, which had the goal of reducing score differences between rounds, and final fitness, which had the goal of reducing score differences across all games. Unlike traditional genetic algorithms, their implementation didn't utilize a stop condition, since their goal was continuous adjustment of the difficulty throughout the gameplay.

Using different configurations for the algorithm, a total of 54,000 tests were made for this study, and found that the game duration averaged 19 rounds when played with AI agents, and 23 rounds when played without. In the study, the difference in rounds is attributed to the effectiveness of the DDA system in making the game more efficient and balanced.

As can be observed from both of these studies, the iterative process of these algorithms offers a structured, but also flexible framework for continuous improvement of the game experience.

2.2.4 Machine Learning Techniques

Unlike conventional methods, ML-based DDA systems don't rely on predefined rules or scripts. They are capable of learning and improving without it being explicitly programmed. In the context of DDA, they are essential for guaranteeing that the difficulty adjustments remain effective over time.

ML algorithms are capable of identifying patterns, predicting outcomes, and making decisions based on their dataset. By constantly analyzing the player's behavior, preferences, and skill, the algorithms can maintain an optimal state of game difficulty, a state often referred to as the 'flow' state (Csikszentmihalyi 1990).

In this subsection we explore ML techniques often used in DDA, describing how these techniques work, and how they were applied to a DDA System in a related work.

2.2.4.1 Neural Networks

NN are a fundamental technique of ML inspired by the structure of the human brain. In the context of DDA, NN are particularly suited for it due to their ability to understand non-linear relationships in the data, which is a useful function for understanding player behavior (Or, Kolomenkin, and Shabat 2021).

NN consists of layers of interconnected nodes, each capable of simple computations. They process input data to recognize patterns and then make decisions based on it.

In a study by Li et al. (Li et al. 2010), they utilize a 3-layered Feed-Forward Artificial Network, using the classic game 'Pac-man' to test their approach. They trained the NN with data generated from the UCT (Upper Confidence bounds applied to Trees) method, which involved simulating multiple game scenarios to collect performance data. With this approach, Li et al. were capable of using the NN to change the behavior of the game opponents (Ghosts in Pac-Man) based on the skill level of the player.

A different study by Wang et al. (Wang and Tseng 2013) presents an interesting approach for DDA by combining NN with fuzzy logic rules in a First-Person Shooter (FPS). They trained artificial NN using datasets containing data of players of different skill levels and then used the NN to control the behavior of the Non-Player Character (NPC) in the game. By integrating fuzzy logic, they were able to dynamically balance the game by adjusting the enemy's aggressiveness based on the difference in the health between the player and the opponent.

Both studies showcased that NN trained on player data can control enemies with human-like behavior and tactics, offering a greater challenge over time compared to traditional game AI methods.

2.2.4.2 Deep Learning

Deep Learning (DL) is essentially an advanced form of NN, with the key difference being the "depth" of the model, which refers to the number of layers in the model. DL Models are known for having many layers, sometimes hundreds or thousands, which enables the models to learn complex patterns in the input data (Rayhan 2023).

Due to their greater capacity in learning patterns, these techniques are particularly well suited for analyzing player patterns and behavior for DDA (Or, Kolomenkin, and Shabat 2021).

In their study, Or et al. (Or, Kolomenkin, and Shabat 2021) present a system for DDA using DL. This system optimizes the user experience by not only considering individual player styles but also broader gameplay constraints, which are parameters that determine how challenging a game feature is. The system uses an innovative UXLoss function in the NN, which utilizes a completion rate constraint, corresponding to the percentage of players that completed a certain task or challenge.

This study mentions difficulty in incorporating this constraint, due to it being a constant function with zero gradients in most areas, meaning that simple adjustments in difficulty might not affect the completion rate until a certain threshold is reached, causing the difficulty to become too hard for most players. To resolve this issue, the study used a variation of the projected gradient descent algorithm, which projects the parameters of the NN into a subspace that adheres to the completion rate constraints, thus guaranteeing that the difficulty adjustments remain within acceptable limits.

This approach was tested during an online game and was shown to better adapt to the skills of a wide player base, outperforming traditional manual difficulty settings.

A recent study by Mendez et al. (Romero-Mendez et al. 2023) also explores the use of DL for DDA, but this time more directly based on the player's skill. In this study, a Feedforward NN was used to classify the players into different skill levels (beginner, intermediate, and expert), adjusting the difficulty accordingly. To test this system, they created a custom game based on 'Space Invaders', and collected the gameplay data from 56 players, including metrics like accuracy and score. The model showed a very high precision of above 99 percent in classifying player skill levels, and players noted that the dynamic adjustment improved their experience. However, the study notes the need for more computationally efficient models and for testing the model's adaptability for different genres.

In these studies, the potential of DL for DDA is shown, especially its ability to learn complex patterns, which is effectively used to adapt the game to different player skill levels. Both of these studies also show a promising avenue for further research, with more computationally efficient models, and extending them to different game genres.

2.2.4.3 Reinforcement Learning

Reinforcement Learning (RL) is a ML technique based on rewarding desired behaviors and punishing undesired behaviors. The agent learns to make decisions by performing actions in a simulated environment to achieve the maximum possible reward (Carew 2023).

DDA systems that utilize RL continuously assess player data, which is then used by the RL agent to adjust the game's difficulty to maximize the player's engagement.

A study by Massoudi et al. (Massoudi and Fassihi 2013) propose an integration of RL with fuzzy logic for DDA, in which the player's performance is monitored and assessed using fuzzy logic, and RL is used to train the game agent. This approach was applied to an action tower defense game and results showed that players of all skill levels were appropriately challenged, as shown by the similar kill percentages across all players.

In a different study, Sekhavat (Sekhavat 2017) presents an innovative approach to applying a specialized form of RL for DDA in rehabilitation games. Their proposed approach managed to increase patient satisfaction and recovery speed by regulating the game's difficulty in real-time. The method that they used, the Multiple-Periodic Reinforcement Learning (MPRL) technique, is a specialized form of RL that has the ability to evaluate multiple objectives in separate periods, offering a more effective DDA system for rehabilitation. While traditional RL focuses on maximizing a single reward function, MPRL is capable of addressing multiple goals with different evaluation periods.

More recently, a study by Rahimi et al. (Rahimi et al. 2023) proposes a continuous RL-based DDA system for a visual working memory game, using a Proximal Policy Optimization algorithm, which is a type of policy gradient method for RL. With this implementation, the

game, composed of a series of trials, adapted its difficulty based on the player's performance and game difficulty from the previous trial. From feedback gathered from 52 subjects, they found that this method maintained the player's engagement over a series of trials, enhanced the player's experience regarding tension and challenge, and also resulted in higher win rates compared to traditional rule-based methods.

As can be observed from these studies, RL can be used for DDA due to its ability to learn and adjust based on player interactions, and can offer a continuously challenging experience to players of differing skill levels. The integration of this technique with other methodologies like fuzzy logic, or the utilization of specialized forms of RL like MPRL, shows the flexibility of the technique and the potential for innovation in this area.

2.2.4.4 Deep Reinforcement Learning

In recent years, more complex approaches were used for DDA, such as DRL, which combines the decision-making capabilities of RL with the powerful data processing capabilities of Deep NN.

Such systems generally utilize a DL component, such as a Deep NN, to process and interpret complex input data, and a RL component, which would involve an agent trained on the data that adjusts his behavior over time for optimal difficulty adjustments.

A study by Noblega et al. (Noblega, Paes, and Clua 2019) proposes a DRL approach for game balancing, in which DL is used to process and interpret input data from the game environment, including player behavior and game states. This was then used to train the NPC agents to adapt their strategies in real-time by using a Proximal Policy Optimization algorithm. They also utilize a unique reward function based on a balancing constant, which is defined by the skill difference between the player and the NPC, to guide the NPCs in maintaining the game's balance.

This approach was tested in a 3D fighting game simulation, emulating different play styles of players (aggressive, defensive, etc...). The results showed that the NPCs were able to adjust their strategies based on the player's fighting style, while still maintaining the game's balance.

Another notable advancement in the application of DRLL in gaming has been the integration of the ML-Agents toolkit for Unity, a widely recognized game development platform (Juliani et al. 2020). This toolkit provides a robust framework for game developers to implement and refine intelligent game agents within game environments through the implementation of DRL to train the agent's models.

The toolkit serves as the bridge between the Unity game environment, which uses C#, and the DRL algorithms, which are implemented in Python. To develop and train an intelligent game agent through this toolkit, developers generally have to follow these steps:

- **Environment Setup** – Design a simulated environment in Unity mimicking a typical game scenario that the agent would interact with.
- **Agent Configuration** – The agents are configured with their sensors (What information they perceive from the environment) and actuators (How they are controlled and interact with the world). By defining these, the agent can perceive its surroundings and take action accordingly.

- **Reward System** – Define the positive and negative rewards of the agent based on its actions and outcomes.
- **NN architecture configuration** – Customize the structure of the neural network that the toolkit will use for the agent.
- **Training Process** – The agent is simulated in the game environment and continually adjusts the neural network based on the rewards received. Each iteration resets the environment until the end of training, which produces a model that can then be integrated into the agent.

In their Thesis, Xiong (Xiong 2019) explores the possibility of implementing a DDA system through the usage of the ML-Agents toolkit in Unity. The prototype that he implemented to test this approach was a Shoot'em Up game where the player controls a ship to attack waves of enemies and dodge their projectiles, collecting points whenever destroying them. The DDA system was implemented in the Gamemaster AI, an intelligent agent responsible for controlling the game's flow. By collecting data related to the player's performance and status, such as their score and health, and data related to the enemy waves, such as the number of waves and the average damage dealt by each wave, the gamemaster decides which pre-defined wave to send next, aiming to keep the player's health at an "expected" amount, a value which depends on the target time of each gameplay session.

To examine the effectiveness of this approach, 25 participants were asked to play two different versions of the game, one with a traditional deliberately programmed AI and the other with the developed machine learned AI. This experimentation was followed by the players responding to a survey and the collection of the game logs. Results found that the machine-learned AI outperformed its counterpart in terms of score differentiation and health reduction, and participants found that version more enjoyable. While the thesis acknowledges the need to better refine the training method used and to adapt the game to more traditional progression methods, these initial findings show promise for further research in implementing DDA in Unity through this toolkit.

While relatively unexplored compared to other approaches, such a hybrid approach shows promising potential for innovation for DDA.

2.3 Conclusion

In the preceding sections of this chapter, the evolution of DDA was described, from its early roots in simplistic mechanisms in classic arcade games to the modern, AI-driven systems that are used today. After describing this evolution, various techniques and relevant studies were described, showing the various advancements in the field. This concluding section summarizes the key findings of this chapter, as well as identifies potential areas for future innovation in the field.

The initial subsection of this section summarizes the research presented throughout the chapter, focusing on critical milestones and technological advancements in DDA.

Subsequently, the following subsection then examines the implications of the findings, identifying the unexplored potential within the field. This subsection will be instrumental in guiding the focus of the DDA prototype that will be developed later in this thesis, ensuring that the selected methodologies and techniques for the prototype contribute to this field.

2.3.1 Summary

Initially, DDA mechanisms in video games were rudimentary and primarily used in classic arcade games. Its concept originated from the classic game 'Space Invaders (1978)', in which a design error accidentally led to a form of DDA where the game became more difficult over time. These early systems, however, were more focused on maintaining player engagement through mechanical triggers based on pre-existing conditions, instead of truly adapting to the individual player.

As the video game industry matured throughout the years, so did the complexity of DDA systems. The introduction of rule-based systems represented a significant leap at the time, enabling DDA systems to adjust difficulty based on player performance metrics, such as player health or frequency of deaths, albeit within the confines of fixed rules.

Afterward, more advancements were developed for rule-based systems, such as the usage of more nuanced player performance metrics, as seen in the study by Pedersen et al. (Pedersen, Togelius, and Yannakakis 2009), or the integration of probabilistic formulas capable of predicting a player's chance of failure, as demonstrated in the study by Hunicke (Hunicke 2005). One significant advancement for rule-based systems was scripted adaptive systems, which allowed for a more context-sensitive adaptation to player behavior. Studies like 'Online Adaptation of Computer Opponent AI' (Spronck, Sprinkhuizen-Kuyper, and Postma 2003) highlighted how these systems became more complex and began to adapt the difficulty in response to a broader range of player actions.

The integration of more advanced AI techniques and ML marked the next significant step of DDA. Early implementations of ML as described in studies like the one by Charles et al. (Charles and Black 2004) laid the groundwork for more sophisticated systems, moving beyond using static rules to using algorithms capable of learning and adapting to player patterns. The use of NN and predictive models, as seen for example in the study by Li et al. (Li et al. 2010), further enhanced the adaptability of these systems, leading to a more dynamic and proactive adjustment of the difficulty.

More recently, advancements in DDA have been significantly influenced by the utilization of DL and RL techniques. DL models, capable of processing large datasets and learning complex player patterns, have shown their effectiveness in DDA, as demonstrated in the study by Or et al. (Or, Kolomenkin, and Shabat 2021). RL, on the other hand, can adequately continuously balance the game in response to the player, as shown in the studies by Sekhavat (Sekhavat 2017) and Rahimi et al. (Rahimi et al. 2023). DRL, the combination of these two techniques, has also been recently researched in the study by Noblega et al. (Noblega, Paes, and Clua 2019).

The integration of these technologies into modern DDA systems has led to more personalized and engaging experiences for the players. However, with these advancements come new challenges and opportunities for innovation. The next subsection will focus on analyzing these findings and identifying potential future approaches for innovation.

2.3.2 Research Insights and Discussion

In many of the analyzed studies, researchers demonstrate the effectiveness of their proposed DDA system integrated into a recreation of a classic game, in an existing game, or a game prototype made to showcase their system. However, many of these demonstrated games were usually of genres that have been extensively explored historically with DDA, such

as FPS, fighting games, and platformers, leaving various other genres unexplored. These other genres could greatly benefit from DDA, especially those with more complex gameplay mechanics like strategy or simulation, or skill-focused genres like roguelike or soulslike.

DRL is also an interesting avenue for future research and, while studies like the one by Noblega et al. (Noblega, Paes, and Clua 2019) have begun to explore its potential, there is ample room for deeper investigation. With its ability to combine the decision-making capability of RL with the pattern-recognition power of deep NN, it could lead to more sophisticated and responsive DDA systems that learn from the player in real-time. One challenge that would need to be resolved for such implementations would be the issue of such technology taking up a lot of processing power from the game, which could force developers to take undesirable measures, such as downgrading certain graphical aspects of the game, or to increase the game's system prerequisites. Unity's ML-Agents toolkit, which utilizes DRL to help game developers train intelligent agents, could also be a novel method for implementing DDA, since current existing research related to the topic is scarce, except for the study done by Xiong (Xiong 2019) or by Noblega et al. (Noblega, Paes, and Clua 2019).

A limitation of most existing DDA systems is that they are tailored for only a specific game, which limits their applicability. Creating a more flexible DDA framework could lead to systems that can be easily integrated into different games or even genres with only a few necessary changes. Such a framework could recognize certain generalized player performance metrics that are usually used across most genres, such as player health and score, while certain metrics that are unique to the game would have to be manually adjusted. Ideally, this adjustment would be minimal due to the hypothetical framework's flexibility.

Another promising area for future innovation is the integration of DDA with procedural content generation. While this has been explored in the past for certain platformer games, such as in the study by Pedersen et al. (Pedersen, Togelius, and Yannakakis 2009), the concept could be explored for other genres with procedural content generation, such as roguelikes or adventure games focused on exploration. Ideally, these alterations would focus on parts of the environment related to difficulty, such as the adjustment of the number of traps found, the placement of enemies, or traversable paths that the player can use.

Overall, while the field of DDA has progressed significantly, especially with the advancements in AI, it's an area that still holds considerable potential. Considering the results of this initial research, this study will aim to contribute to this field by implementing DDA in Unity using the ML-Agents toolkit, building upon the promising results of existing research (Xiong 2019).

Chapter 3

Implementation

This chapter of the thesis is dedicated to describing the methodology employed in the different phases of this research, from conceptualization to the implementation and evaluation of the proposed DDA system within a game prototype.

The methodology is divided into the following sections, with each dedicated to a different phase of the implementation of the solution:

- **Tools and Methods Selection** – Initial phase dedicated to selecting the tools and methods that were used in the subsequent phases.
- **Data Protection, Security and Ethical Standards** – Before the actual development of the prototype and data collection, this phase is dedicated to addressing ethical and data protection concerns, including the addition of a consent form for research participants.
- **Game Prototype Creation** – This phase is dedicated to describing the design and development of the game prototype, describing the implementation of certain aspects, such as player controls, enemies, hazards, and procedural level generation.
- **Enemy Agent Experimentation** – Describes the first round of practical application of the chosen method for implementing the DDA system, focused initially on using the method for training enemy agents to control the behavior of the different enemy types.
- **Data Gathering** – Dedicated to implementing a data collection mechanism within the prototype to analyze gameplay metrics and train the necessary player models for implementing the DDA system.
- **DDA Implementation** – This phase of development is dedicated to describing the development and integration of DDA within the game prototype, detailing the training process of each iteration of the model.

As can be observed, the game prototype was created before the enemy agent experimentation and data collection process. This would normally not be necessary for most studies that utilize machine learning, due to the availability of usable online datasets. However, due to the lack of suitable public datasets for training the machine learning model and how the chosen tools work, it was deemed necessary to first create the game prototype. This prototype would not only be the testing ground of the DDA system but it would also be used as the mechanism to gather the necessary gameplay data for implementing the system. Consequently, the game prototype is a prerequisite for the data collection phase.

3.1 Tools and Methods Selection

For the development of the prototype, Unity was selected as the game development platform. Its robust support for scripting, animation, and physics made it ideal for the design of the game prototype, which emphasizes simple, but robust gameplay mechanics. Its intuitive user interface, extensive community support, and various first and third-party tools also facilitate the rapid development of the prototype and the subsequent testing and refinement of the DDA system.

While Unity covers most development aspects of the research, other tools were chosen to assist in other necessary tasks. Unity's ML-Agents Toolkit was used to configure and train the intelligent agents for our DDA system in the Unity development environment (Juliani et al. 2020). Consequently, the ML method used was DRL, which is supported by the toolkit. Unity Profiler was also important in monitoring the game's performance to ensure that the implementation of the DDA system did not significantly impact the game's performance.

Aside from Unity-related tools, other supplementary tools were used to assist in the analysis and development process. Git, a version control system, was used to efficiently manage and document the development process, and to ease troubleshooting of any emerging issues. For analyzing the gameplay data, Python was used along with relevant libraries, like Pandas, NumPy, and Matplotlib for processing and analyzing the large datasets of gameplay data. For creating the surveys for the players to fill out, a simple survey from Google Forms was used to do so.

3.2 Data Protection, Security and Ethical Standards

In this research, the primary data that was collected were gameplay metrics and timestamps from different participants. To ensure that no sensitive data is tied to the individual participants, the player Identification (ID) codes are anonymous, and the timestamps focus on the elapsed time of the gameplay instead of actual dates. This data was gathered during real-time gameplay and, given its non-sensitive nature, encryption was not deemed necessary.

Aside from this gameplay data, some additional data was collected from the participants through surveys, including their age and gaming experience. Like the gameplay data, this data is purely anonymous and cannot be traced back to the participant. Thus, the process of our data collection aligns well with the General Data Protection Regulation (GDPR) standards, especially due to the non-sensitive nature of our data and the anonymization of demographic data. Further details of the data collection are described in 3.5.

While the data collected was not sensitive, basic security protocols were nonetheless implemented to prevent any unauthorized access to the data, such as by using secure data storage for data related to the research. Standard security protocols were also used for the transmission of data from the game to the storage to prevent any interception or corruption.

Regarding other potential ethical concerns, the research provided a consent form for the game prototype and surveys, detailing what data was collected and how it would be used in the research. This form specified the voluntary nature of the participation in this research, with the option to withdraw at any time without any repercussions. It was integrated into the game prototype, where the player will be able to click on agreeing or denying participation. The full form is included in Appendix A.

The game also avoided flashing lights at a certain frequency to keep away the possibility of triggering any reaction from participants with photosensitive epilepsy. Lastly, the research only involved adult participants (ages 18 and above), removing the need for parental consent for minors.

These measures guarantee that the anonymity and rights of all involved participants were appropriately protected. While the participants won't receive individual feedback on the results, the overall findings of this research are published in the thesis to be freely accessed by the participants and other interested parties.

3.3 Game Prototype Creation

This section describes the development of the game prototype upon which the DDA system was integrated and tested. The development of certain features and game entities is also detailed, such as player mechanics, enemies, hazards, and the procedural generation of levels.

The prototype that served as the primary testing ground of the DDA system was a simple top-down 2D shooter, featuring certain skill-based gameplay mechanics. In this prototype, the player-controlled character can shoot projectiles at enemies, deflect enemy projectiles back at the enemy with the correct timing, and dash in the direction it's moving to dodge projectiles.

Due to time constraints, the design of the prototype was simplistic, so the development was focused on the gameplay mechanics. Following this style, any art elements used in this game were created from free assets or shapes provided by Unity, and no sound elements were added.

Both the player character and the enemies were represented as distinct colored shapes. Different enemy types also exist in the game, with varying attack patterns, projectile speeds, and other attributes.

For the level design, the prototype used procedural level generation to vary the placement of enemies and hazards each time a new level was generated. This approach was then integrated with DDA in section 3.6 to analyze the feasibility of procedural level generation that adjusts the layout according to the player's skill level, such as favoring the placement of more difficult enemies to increase the challenge or reducing the number of enemies placed to lessen the difficulty.

Lastly, the prototype was developed for Personal Computer (PC), more specifically the Windows operating system, so that it was accessible to a wider range of players participating in this research.

The following subsection describes the initial implementation of the game prototype, which served as a basis for the development of the rest of the prototype.

3.3.1 Initial Prototype

The initial prototype was developed considering only the essential and core features of the prototype for initial testing of the Unity toolkit. As such, the development was focused on the player-controlled movement, projectile and deflection mechanics, an initially rudimentary enemy AI, and a basic damage and health system, all set within a simplistic arena. Figure 3.1 shows an example of this game environment and its main components.

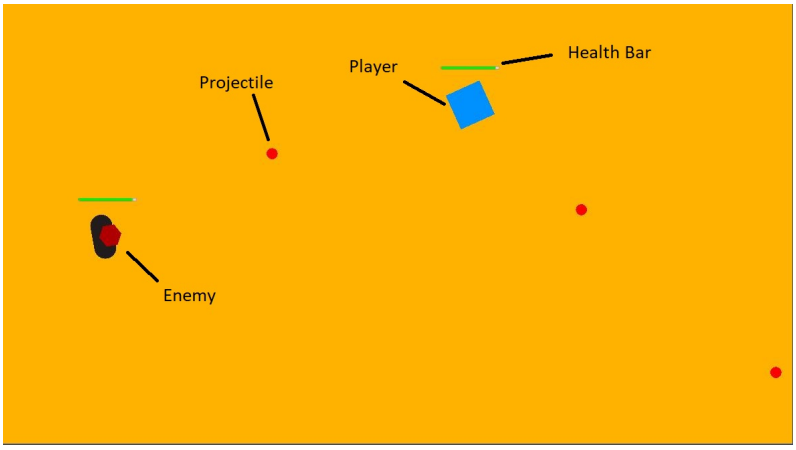


Figure 3.1: Initial Prototype Game Environment

The main game components of this prototype are:

- **Player** – The player-controlled character is controlled by a blue square capable of moving in any direction through directional inputs, shooting, and deflecting projectiles with the right timing.
- **Enemy** – The basic enemy developed takes the form of a black capsule, with a distinct red shape indicating its orientation and shooting direction. It moves around the arena and rotates to aim and shoot projectiles against the player. The ML-Agents toolkit was used to train the model that controls the AI of the enemy. This process is further detailed in 3.4.
- **Projectile** – A basic red sphere is used to represent the projectiles launched by both the enemy and the player. Whenever colliding against a target (enemy or player), it deals a specific amount of damage to them. Projectiles were programmed so that the player is immune to their own projectiles and the enemy immune to their own projectiles launched by other enemy instances.
- **Health Bar** – The player and each enemy are instantiated with a health bar that moves with them. Whenever taking damage, the health bar is reduced by the corresponding amount, turning yellow and red when reaching a certain threshold. When an entity reaches zero health, the entity and its health bar are removed from the environment.

The developed environment for this initial prototype was structured as a closed rectangle, delineated by a wall that served as a boundary preventing any of the game entities from accidentally moving away from the observed environment.

This initial prototype provided valuable information regarding the features of the Unity platform and confirmed the feasibility of this project. With the core basis of the prototype established, the next step involved refining and adding more depth to the player mechanics, which the following subsection describes.

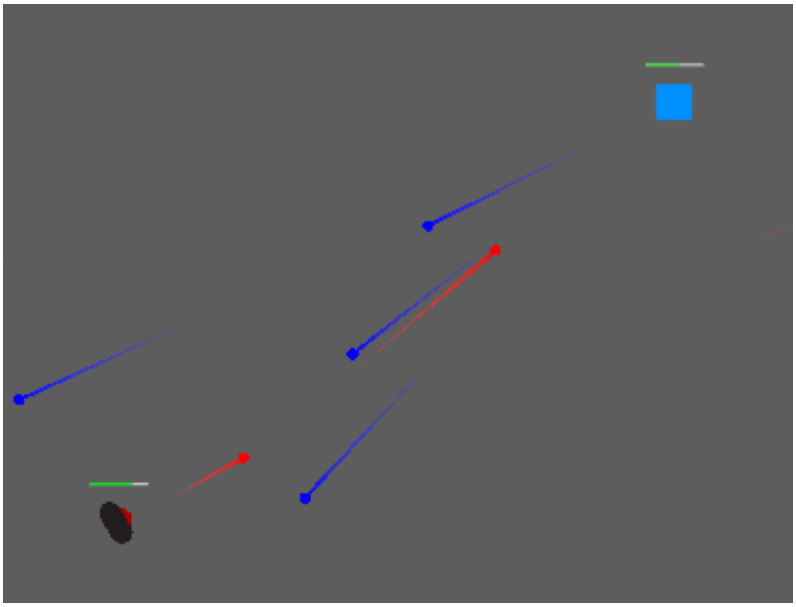


Figure 3.2: Prototype Shooting Gameplay Example

3.3.2 Implementing Player Mechanics

The mechanics of the player-controlled character form the core of the gameplay experience in this prototype and, as such, were the focus of the initial development. The player character was represented by a blue square and was equipped with various abilities to deal with challenges defensively or offensively.

The player's primary offensive ability is to shoot projectiles toward any location on the screen by left-clicking with the mouse. This was implemented by calculating the direction vector from the player's current position to the target position (where the mouse was clicked) and then instantiating a projectile at the player's position with the calculated direction and a predefined speed. A short cooldown was also implemented for this system to prevent the player from being able to shoot too many projectiles at once by repeatedly clicking the mouse button. Player and enemy projectiles were instantiated with different colors so that the player could easily distinguish between them, and bullet trails were also added so that it was easier for the player to see the direction of the fired bullets. Figure 3.2 shows an example of how these visual indicators look during gameplay.

To navigate the game's environment, the standard WASD or directional keys were used for the player character's movement and the left shift key was used to dash. The movement was designed to be fluid, allowing the player to quickly change direction from one moment to the next, with the square representing the player also rotating to always face the direction of the movement. The dash mechanic allowed the player character to dash forward in the direction that it was moving, quickly rapidly moving a predefined distance. To balance this powerful ability, a short cooldown period was added to prevent the player from repeatedly using it.

Lastly, by right-clicking the mouse, the player could activate a shield that defends against enemy projectiles, deflecting them in the opposite direction and possibly hitting enemies with their own projectiles. This shield was indicated by a visual effect and lasted half a

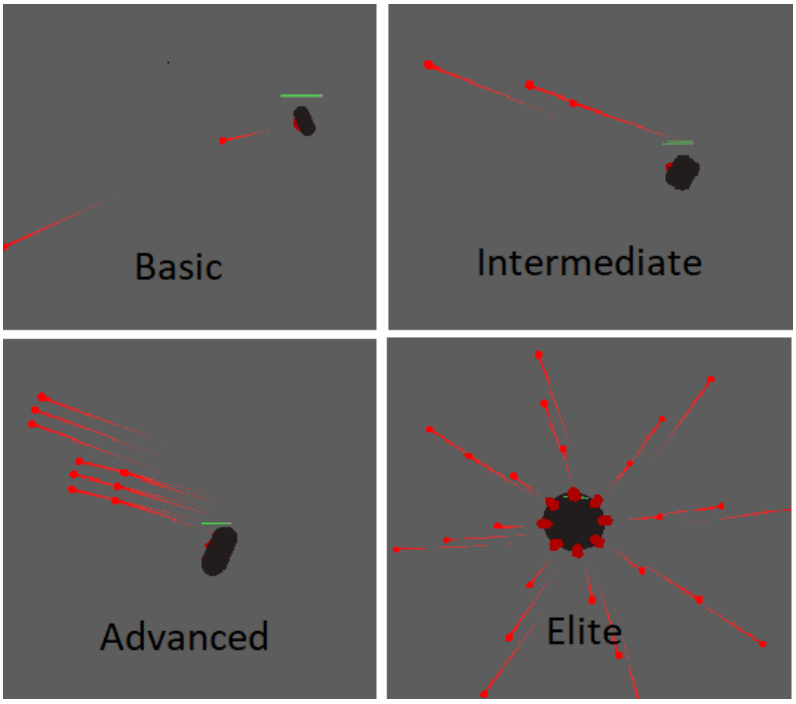


Figure 3.3: Prototype Enemy Types

second, needing a balance of timing and positioning from the player when using this ability to maximize its effectiveness.

These abilities were designed and implemented to be straightforward and easily understood by less experienced players, while still offering enough depth for more experienced players by allowing them to combine the abilities in more strategic ways, such as by dashing behind enemies for an optimal shooting angle or by perfecting the deflection timing and positioning to hit the enemy with their own projectiles.

This design approach for the player mechanics was chosen taking into account the eventual integration of the DDA system. By making it so that the player mechanics were easy to learn but hard to master, the DDA system could more effectively be tested on a broader range of player skill levels.

With these core player mechanics implemented, the next stage in the development process involved creating different enemy types, each with its own behavior and attributes, to make the gameplay experience more dynamic and challenging.

3.3.3 Implementing Enemies

Four distinct enemy types were implemented for this prototype, with each being more difficult than the previous one due to the increasing health and complexity of the projectile patterns. To compensate for more complex projectile patterns, each enemy type had a longer cooldown than the previous one. Figure 3.3 shows the visuals of the different enemy types and their projectile patterns.

The **Basic Enemy**, which was based on the enemy created for the initial prototype, was the simplest enemy to defeat. It had the least health and shot a single projectile at short

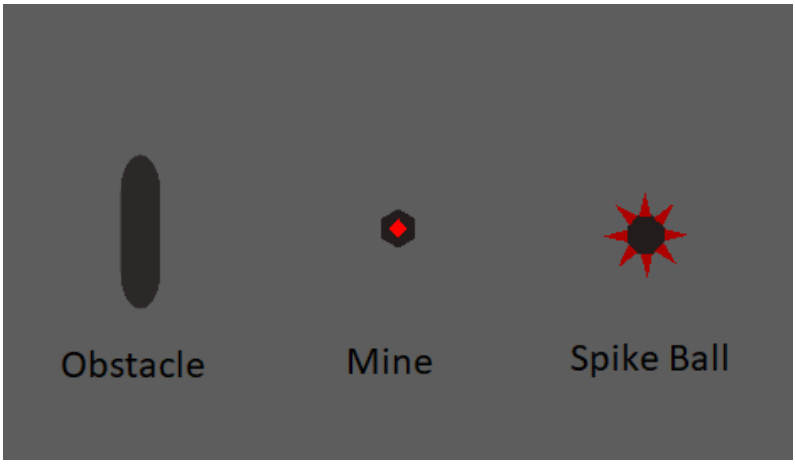


Figure 3.4: Prototype Hazard Types

intervals. This enemy's function was to introduce players to the core mechanics of dodging and shooting, without needing any strategy to defeat it.

The next implemented enemy type was the **Intermediate Enemy**, which was the fastest of the enemy types, forcing the player to use the dash mechanic if they wanted to move away from it. Additionally, this enemy shot three projectiles in quick succession, being hard to dodge if the enemy managed to get close to the player.

The third enemy type was the **Advanced Enemy**, which was potentially the most dangerous for the player due to its projectile pattern. It fired a spread of three projectiles simultaneously and repeated this three times in a row. This projectile pattern could hit the player multiple times at once if they didn't dodge it, but strategic players could take advantage of the deflection mechanic to deflect most of the projectiles back to this enemy.

The last enemy type that was developed was the **Elite Enemy**, which had the largest health, but was the slowest and easiest to hit to compensate. It shot three projectiles in quick succession from multiple points around its body, spreading the projectiles around the environment. Due to it having the longest cooldown and size, it was relatively straightforward for a player to defeat it, but when mixed with other enemy types, the projectiles of the elite enemy created a hazardous environment for the player.

Each enemy was controlled by a DRL model, trained using Unity's ML-Agents toolkit. The training process and the resulting behavior of the different enemy types are detailed in section 3.4.

With varied enemy types created for the prototype, the development then focused on creating various hazards to increase the complexity and variability of each level.

3.3.4 Implementing Hazards

To increase the complexity of each level, three different hazards were created: obstacles, mines, and spike balls. Each hazard was not considered an enemy that needed to be defeated by the player, but instead, elements of the environment that could disrupt the player or be strategically used by them for their benefit. Figure 3.3 shows the visual representation of these hazards within the game.

The obstacles were designed as moving barriers that could obstruct the path of the player or enemies, and block any projectiles that collided against them. When instantiated, these obstacles moved between two points on the map (calculated based on the obstacle's starting position) and quickly switched the movement direction when colliding against the outer walls of the environment. Players could also use these obstacles to their benefit by using them as a cover against enemy projectiles and could quickly dash to move in front or behind these obstacles.

Mines were a more dangerous element that was introduced as a potential danger for both the player and enemies. These static objects were instantiated with collision detection that caused them to detonate when they came into contact with a player, enemy, or a certain amount of projectiles. The explosion damaged and pushed all entities around it in a large radius, potentially causing a chain reaction of other nearby mine hazards. A player needed to be aware of their positioning to prevent any collision against the mines, but could also strategically shoot the mines from a distance to cause them to detonate and damage nearby enemies.

The last hazard that was implemented, the spike ball, was the type that posed the most danger to the player. These objects constantly moved around the map, changing direction at regular intervals to ensure that their movement pattern was unpredictable. Upon colliding with the player, the player was damaged and pushed back a certain distance away to allow the player to dodge and prevent another collision. While the spike balls didn't damage enemies when colliding, they could still push them, potentially disrupting their attack pattern.

The addition of these elements created a more interactive and challenging environment for the players, potentially disrupting their strategy, but also offering more experienced players ways to use these elements to their advantage.

With the implementation of the player mechanics and the necessary game elements for the environment, such as the enemies and hazards, the next step was to explore the use of procedural level generation to dynamically place these entities every time a new level began.

3.3.5 Procedural Level Generation

This section details the implementation of procedural level generation in the prototype, to ensure that every level was different from the previous by dynamically placing different types and quantities of enemies and hazards. The game was structured into ten levels so that a more comprehensive comparison could be made between the traditional difficulty scaling detailed in this section and the implementation based on DRL for DDA in section 3.6. The algorithm was implemented in a class responsible for managing the game environment, named Level Manager.

For a player to beat a level, he needed to defeat all enemy entities present and, upon doing so, the level was cleared of any other entities, such as hazards, before the next level was generated. Since this version of the prototype used traditional static difficulty scaling, meaning each level increased the difficulty by the same amount, each enemy and hazard type was assigned a difficulty value. For example, while the basic enemy had a difficulty value of only two points, more complex enemy types had a higher value, such as the elite enemy with five points. Thus, the first level began with a difficulty value of two, and this value increased by two with each subsequent level.

To prevent issues with performance or too many entities on the screen, the limit for the number of enemies and hazards spawned was four each. The algorithm started by randomly choosing primary and secondary enemy and hazard types, making sure that whenever there was a higher number of enemies or hazards, there would be some variety in the types that were present.

Then, until the difficulty value for that level was reached, the algorithm started placing enemies and hazards on the level with a random starting position and rotation, making sure that they did not overlap with existing objects or each other in a certain radius. The algorithm also made sure that at least one enemy was placed for the player to defeat.

After generating the level, the Level Manager class was notified when all enemies were defeated, giving the player a few seconds to prepare before clearing the level and generating the next one. In the case of beating the tenth level, the player was returned to the main menu with a congratulatory message. Figure 3.5 shows a snippet of the game prototype after all the game elements have been implemented.

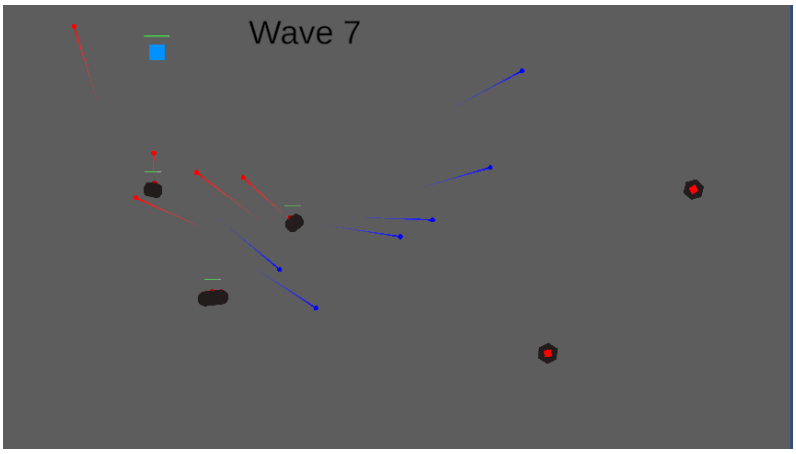


Figure 3.5: Prototype Gameplay Showcase

With the prototype fully working with a stable game, the next step in the process was to experiment with the ML-Agents toolkit to train the different enemy agents responsible for controlling the behavior of the implemented enemy types.

3.4 Enemy Agent Experimentation

The primary objective of this phase was to test and examine the feasibility of using the ML-Agents toolkit, which uses DRL, in this project. To do this, the toolkit was used to turn the different enemy types implemented in the developed prototype (detailed in section 3.3) into agents with intelligent behavior, capable of navigating the environment and competently aim towards the player and survive.

The process of training the different enemy types involved creating and simulating an appropriate environment where they could interact and learn through episodic training, where the NN maximized its accumulated rewards across multiple episodes. As usual in reinforcement learning, positive and negative rewards were used to guide the agent toward desired behaviors and away from undesired ones. The details of the environment and the training process of the different enemy types are shown in the following subsections.

3.4.1 Setup and Environment

To create and train this AI, a learning environment was first created in Unity to simulate the agent's interactions during the training process. This training process was based on episodic learning, where the NN attempts to maximize its accumulated rewards across a series of episodes. This iterative process allows the NN to refine its strategy based on the feedback from each episode, leading to more desirable behaviors for the AI.

Through the toolkit, each episode was programmed to end when the player or enemy entity died, or after a defined maximum amount of training steps. At the beginning of each episode, the 'OnEpisodeBegin' function was used to reset the environment, reinitializing the training conditions to guarantee consistency in the training process. This function was implemented for both the player and enemy agents to reset certain key attributes, such as resetting the health to the maximum value and randomizing their starting position and orientation to introduce variability in training, preventing overfitting and encouraging generalization of the enemy's behavior. Code 3.1 shows the implementation of this function for the player and enemy agents. The logic of this function is used consistently for training every enemy type so that the training environment is standardized.

```
public override void OnEpisodeBegin()
{
    //Reset physics and training variables
    this.GetComponent<PolygonCollider2D>().enabled = true;
    this.GetComponent<Health>().enabled = true;

    //Below lines are only utilized in the function for the enemy
    this.KilledPlayer = false;
    this.Died = false;

    //Reset health
    this.healthComponent.currentHealth = this.healthComponent.maxHealth;

    //Randomize starting position and rotation
    this.transform.localPosition = GetRandomStartPosition();
    this.transform.rotation = Quaternion.Euler(0, 0,
    GetRandomStartRotation());

    //Set the parent position to the local training environment
    transform.parent = LevelParent;
}
```

Listing 3.1: Initialize Episode Code

Ray perception sensors were added as components to each enemy type so that the enemy entities could observe and learn from their environment. Except for the elite enemy type, the sensors were placed at the front of each enemy type to provide them a form of cone-like vision, and one was placed at their back to prevent it from colliding with something behind it. In the case of the elite enemy, the sensors were placed all around it to provide it with 360-degree vision, since its shooting pattern fired projectiles all around it. The placement of the sensors in the different enemy types is shown in figure 3.6.

This type of sensor, provided by the ML-Agents toolkit, emitted invisible rays from the entity in predetermined directions, functioning similarly to a radar. When a ray intersects with an object, it returns data about the distance to the object and the type of object encountered, such as hazard, player, or enemy. This type of data, combined with the observations of

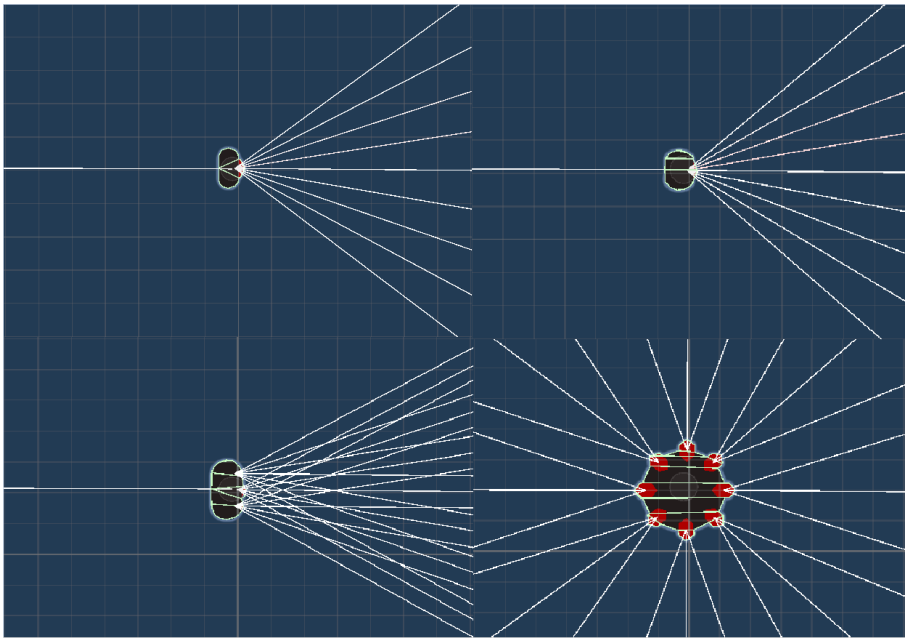


Figure 3.6: Enemy Sensors Placement

the enemy's own state and that of the opposing player, detailed in the following section, provided a comprehensive view of the local environment to the agent.

To enhance the training efficiency, the environment was cloned multiple times to allow for parallel training across various instances. This means that multiple independent episodes could run simultaneously, allowing each environment to learn from the results of others and increasing the speed of the learning process. The number of simultaneous environments used in this phase varied between 9 and 16, depending on the complexity of the enemy type being trained. The need to reduce the number of environments during training occurred due to the lack of computational power of the hardware used for training. Since each additional environment increased the computational load, the hardware used struggled to train more complex enemy types, which instantiated more projectiles and had a higher number of active sensors.

So that the enemies could handle complex scenarios, curriculum learning was used as the approach for training. This involved making small modifications to the environment throughout training, to simulate increasingly complex tasks. The initial tasks were simple, such as shooting a static target, while later ones were complex, such as shooting a moving target while dodging its projectiles. Eventually, the models were also trained in a full simulation of a level to teach the enemy to navigate generated hazards and cooperate with other enemies. This led agents to build upon previously learned behaviors, allowing them to handle more complex scenarios.

The following section details the general training process for each enemy type and the results of that process.

3.4.2 Training the Enemy Agents

As part of training the different agents, a reward system was also defined for the environment to steer the agents towards the desired behavior, keeping track of whenever certain events happened during each episode and applying each reward accordingly. Positive reinforcement was applied whenever the agent hit or managed to defeat the player, and negative reinforcement was applied whenever the agent was hit, died, or collided against an obstacle.

To encourage the enemy to aim toward the player more efficiently, a small negative reinforcement was consistently applied per time step during an episode. Additionally, to encourage a more aggressive behavior and to prevent the enemies from adopting overtly evasive tactics, the negative reinforcement from being damaged or defeated was designed to be less significant than the positive reward from successfully damaging and defeating the player. The exception to this was the basic enemy type, which was encouraged to adopt a more evasive behavior to compensate for its low health.

This reward logic was consistently applied across all enemy types, with the specific reward values being adjusted to account for their capabilities. For example, the rewards for damaging and defeating the player were reduced for more advanced enemy types. This was necessary to compensate for the larger spread of projectiles instantiated by the enemy, which would naturally achieve positive rewards more frequently.

Aside from the data returned by the ray perception sensors, the enemy agent also constantly collected other observations from itself and the player that were important for its behavior. These observations were the same across the different enemy types and were all normalized to ensure that the neural network could effectively process these inputs. The following list details the different observations added:

- **Local Position** – The enemy's current position relative to the game environment, counting as three different observations (x, y, and z-axis).
- **Relative Position to Player** – The direction and distance to the player, to help the enemy orient itself towards the player.
- **Player's Movement Direction** – The direction of the player's movement. This was a late addition to help the enemy predict the player's future position, consequently improving its aim and accuracy.
- **Player's Current State** – The player's current health, to encourage the agent to be more aggressive when the player is more vulnerable.
- **Enemy's Own State** – The enemy's current health to help the agent decide between acting more evasively and continuing to attack.
- **Shooting Cooldown** – The shoot cooldown timer, so that the agent can learn when it can shoot.
- **Rotation** – The enemy's current rotation to help in aiming towards the player.
- **Enemy's Rotation Relative to the Player** – The relative rotation angle between the enemy's forward shooting direction and the direction towards the player. This was a late addition to fine-tune the agent's aiming for better accuracy.

As the agent received these observations, the NN architecture was programmed to output four values between 0 and 1 which were received as parameters by the 'OnActionReceived'

method, which is responsible for converting these values into commands that direct the behavior of the agent. The first three values are continuous variables that govern the movement of the agent in the x and y axis and its change in rotation. The fourth value, a discrete variable, is a binary decision of whether the agent shoots or not.

During the training process, multiple models were iteratively developed and refined. Each trained model was built upon the learning of its predecessor until the final one was created. This final model was then integrated into the enemy agent, serving as its "brain", constantly receiving inputs from its environment and observations and outputting its chosen commands based on the patterns that it had learned.

Between each iteration, certain adjustments were made to the reward parameters and certain parameters of the NN based on the performance of previous models. Below is a brief description of the main parameters of the NN that were adjusted during this process:

- **Batch Size** – The number of experiences collected before updating the NN.
- **Buffer Size** – The total number of experiences stored. Once the buffer is filled, older experiences are replaced by newer ones.
- **Learning Rate** – How drastically the NN's weights are updated during the training.
- **Num Epoch** – The number of times that the algorithm will pass through the entire training dataset.
- **Hidden Units** – The number of units in each hidden layer of the NN.
- **Num Layers** – The number of hidden layers in the NN, determining its depth.
- **Max Steps** – The total number of steps taken in the environment during the training process.
- **Time Horizon** – How many steps are considered for calculating future rewards.

The results of the main iterations for the different enemy types are detailed in the following subsections.

Basic Enemy

The most extensive experimentation was made during the training of the basic enemy, due to it being the first enemy type that was trained using the ML-Agents toolkit. During this process, eight successful models were trained, with each improving upon the previous. Other models were attempted between each of these main iterations but were stopped in the middle when it was clear that the change in the parameters led to a significant reduction in the model's performance.

Between each iteration, adjustments were made to certain parameters of the NN or the rewards, and the tasks of each iteration were adjusted to be increasingly complex, as part of the curriculum learning approach. The results and configuration of each model are detailed in Table 3.1, while Figure 3.7 shows the training performance graph generated by TensorBoard, highlighting the rewards accumulated by each model during training. In certain cases, other variables of the game were adjusted, such as the player's health, to balance the game.

By the final iteration, the basic enemy demonstrated the desired capacity to accurately locate and aim to shoot at the player while moving evasively to reduce the chance of being hit.

Table 3.1: Initial Experimentation Results for Basic Enemy

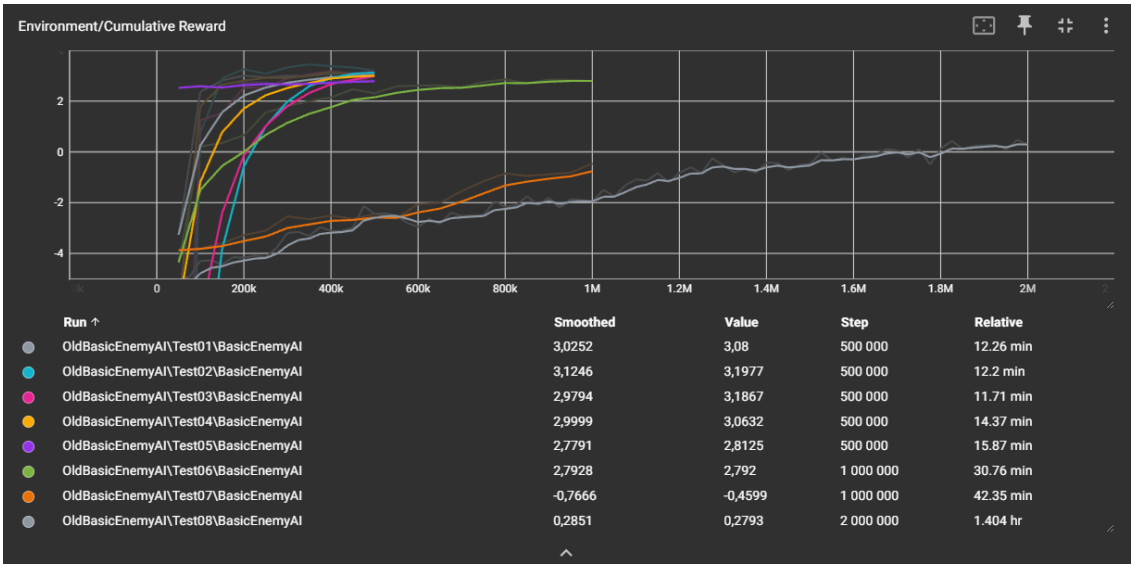
Model	Parameters	Rewards	Task	Results
1	**batch size** = 1024 **buffer size** = 10240 **learning rate** = 0.0003 **num epoch** = 3 **hidden units** = 128 **num layers** = 2 **max steps** = 500000 **time horizon** = 64	**Defeat player** = 1.0 **Die** = -0.5 **Hit player** = 0.25 **Got Hit** = -0.25 **Collision** = -0.25	**Task:** Shoot a static player at a fixed position in the environment.	**Reward** = 2.92 Agent learned to rotate to shoot toward the player.
2	Same as model 1.	Same as model 1 with the below changes: **Time Penalty**= -0.01 **Hit player**= 0.125	**Task**: Shoot a static player at a fixed position in the environment.	**Reward** = 3.22 Agent now rotates and shoots towards a player faster due to pressure imposed by the time penalty reward, which applies this negative reward per time step of the episode.
3	Same as model 2.	Same as model 2 with the below changes: **Time Penalty**= -0.02 **Note:** Also fixed some collision bugs.	**Task**: Shoot a static player at a random starting position in the environment.	**Reward** = 3.06 Agent is now more capable of shooting at a static player in different positions of the environment thanks to randomizing the player's start position. The overall reward is lesser than model 2 due to increasing the time penalty negative reward, but the enemy demonstrates overall improvement in its accuracy.

Table 3.2: Initial Experimentation Results for Basic Enemy (Continued)

Model	Parameters	Rewards	Task	Results
4	Same as model 3 with the below changes: **num epoch** = 4 **hidden units** = 256 **num layers** = 3 **time horizon** = 128	Same as model 3 with the below changes: **Time Penalty**= -0.05 **Note:** Added player's direction and own orientation towards the player as observations. Due to this, the training process was reset from this model onwards.	**Task:** Shoot a static player at a random starting position in the environment.	**Reward** = 3.39 Agent now reacts faster when detecting a player. Reward increased even despite the increase in the time penalty and the reset of training.
5	Same as model 4.	Same as model 4.	**Task:** Shoot at a moving player in the environment.	**Reward** = 2.98 Agent was now capable of rotating to follow the player's movement to keep shooting.
6	Same as model 5 with the below changes: **hidden units** = 512 **num layers** = 4 **max steps** = 1000000	Same as model 5 with the below changes: **Time Penalty**= -0.02 **Note:** Some game values were adjusted for balancing, including the player's health and enemy damage.	**Task:** Shoot at a moving player in the environment.	**Reward** = 2.90 Agent maintains the same behavior despite adjustments.
7	Same as model 6.	Same as model 6 but added a small negative reward for each shot fired to encourage greater accuracy. Also, removed the time penalty since the task is more focused on surviving player projectiles.	**Task:** Shoot at a moving player in the environment that shoots projectiles at the agent.	**Reward** = -0.20 Agent begins to adopt more evasive maneuvers but struggles to avoid all projectiles while trying to hit the player.
8	Same as model 7 with the below changes: **num layers** = 5 **max steps** = 2000000	Same as model 7, but removed the negative reward for shooting.	**Task:** Shoot at a moving player in the environment that shoots projectiles at the agent.	**Reward** = 0.16 Agent is still hit but adopts the desired behavior of moving in more evasive patterns.

The lessons learned and the improvements made during the training of the basic enemy greatly benefited the training of subsequent enemy types, especially with the addition of observations like the player's direction or orientation towards the player. Additionally, resolving various physics-related bugs improved the stability and accuracy of future training simulations.

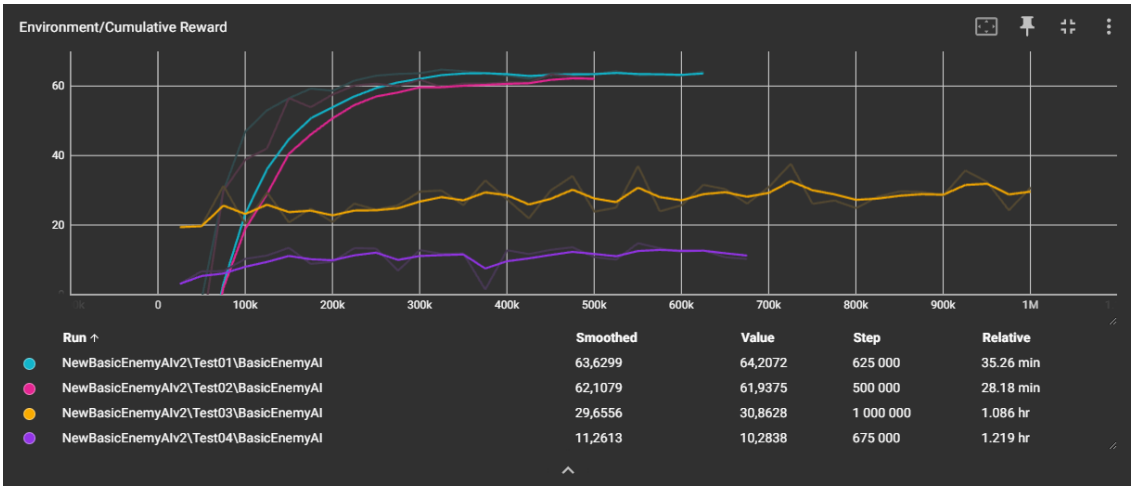
During the training of the other enemy types, the training process was streamlined to only



Run	Smoothed	Value	Step	Relative
OldBasicEnemyAI\Test01\BasicEnemyAI	3,0252	3,08	500 000	12.26 min
OldBasicEnemyAI\Test02\BasicEnemyAI	3,1246	3,1977	500 000	12.2 min
OldBasicEnemyAI\Test03\BasicEnemyAI	2,9794	3,1867	500 000	11.71 min
OldBasicEnemyAI\Test04\BasicEnemyAI	2,9999	3,0632	500 000	14.37 min
OldBasicEnemyAI\Test05\BasicEnemyAI	2,7791	2,8125	500 000	15.87 min
OldBasicEnemyAI\Test06\BasicEnemyAI	2,7928	2,792	1 000 000	30.76 min
OldBasicEnemyAI\Test07\BasicEnemyAI	-0,7666	-0,4599	1 000 000	42.35 min
OldBasicEnemyAI\Test08\BasicEnemyAI	0,2851	0,2793	2 000 000	1.404 hr

Figure 3.7: TensorBoard Graph for Models of Basic Enemy

take four models each, with minimal adjustments, thanks to optimization of the parameters and rewards. The training environment was also adjusted to simulate real levels to train the enemy to react to hazards and to cooperate with other enemy entities. Considering that the basic enemy model from initial experimentation was more inefficient compared to the models for other enemy types, the basic enemy was retrained using the streamlined process. Table 3.3 describes the results from the models, and Figure 3.8 shows the generated graph.



Run	Smoothed	Value	Step	Relative
NewBasicEnemyAIv2\Test01\BasicEnemyAI	63,6299	64,2072	625 000	35.26 min
NewBasicEnemyAIv2\Test02\BasicEnemyAI	62,1079	61,9375	500 000	28.18 min
NewBasicEnemyAIv2\Test03\BasicEnemyAI	29,6556	30,8628	1 000 000	1.086 hr
NewBasicEnemyAIv2\Test04\BasicEnemyAI	11,2613	10,2838	675 000	1.219 hr

Figure 3.8: TensorBoard Graph for Optimized Models of Basic Enemy

Table 3.3: Optimized Model Results for Basic Enemy

Model	Parameters	Rewards	Task	Results
1	**batch size** = 1024 **buffer size** = 20480 **learning rate** = 0.0003 **num epoch** = 5 **hidden units** = 512 **num layers** = 5 **max steps** = 1000000 **time horizon** = 32	**Defeat player** = 4.0 **Die** = -2.0 **Hit player** = 1.0 **Got Hit** = -0.5 **Collision** = -0.5 **Time Penalty** = -0.02	**Task:** Shoot a static player in the environment. Both the player and the enemy have a random starting position and rotation.	**Reward** = 64.21 Agent efficiently locates and aims toward the target. **Note:** Training stopped at step 625000 due to the rewards hitting a plateau.
2	Same as model 2.	Same as model 2.	**Task:** Shoot at a moving player in the environment.	**Reward** = 61.94 Agent now capable of tracking and shooting a moving player. **Note:** Training stopped at step 500000 due to the rewards hitting a plateau.
3	Same as model 2.	Same as model 2.	**Task:** Shoot at a moving player while reacting to generated hazards of the level.	**Reward** = 30.86 Agent hits the player fewer times but successfully learns to avoid hazards, such as mines. In some cases, it uses certain hazards to its advantage, such as using them as a barrier to defend from the player's projectiles.
4	Same as model 3.	Same as model 3.	**Task:** Shoot at a moving player while reacting to generated hazards of the level and cooperating with other enemies	**Reward** = 11.26 Agent struggles to hit the player as many times due to competition with other enemy entities but successfully learns to avoid disrupting the movement and projectiles of other enemies.

Intermediate Enemy

As mentioned in the previous subsection, after some brief experimentation with parameters, a more optimal approach for training the enemies was developed, with each enemy type only needing four model iterations to be fully trained. The chosen parameters for the NN and tasks for each model were the same as the ones detailed in table 3.3, with the main difference being that the positive rewards for hitting and defeating the player and being damaged were proportionally adjusted to take into consideration the increased capacity of this enemy type to deal and receive damage, due to its higher health and fire rate. Figure 3.9 shows the accumulated rewards from the training of each iteration. As can be observed in this figure, the agent had the most difficulty in the third model iteration, in which hazards were spawned in the environment. This was primarily due to its higher speed, which made learning to avoid collisions particularly difficult during the initial training.

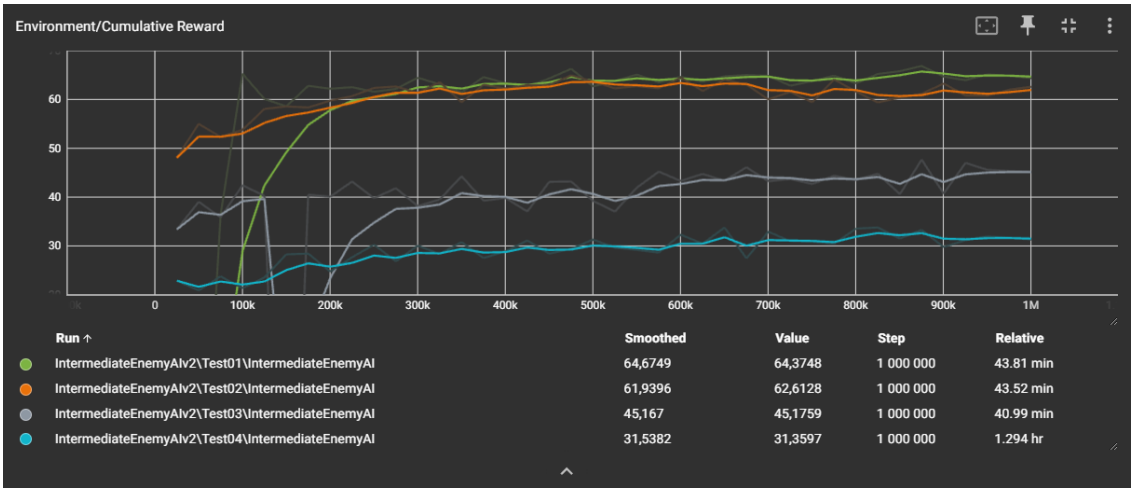


Figure 3.9: TensorBoard Graph for Optimized Models of Intermediate Enemy

After training this enemy type, the resulting agent has learned to take advantage of its higher speed to pursue the players closely to have a higher chance of hitting them.

Advanced Enemy

Just like the previous enemy type, four model iterations, with the same parameters and tasks described in table 3.3, were created during the training of this agent, with the only difference being a proportional adjustment of the rewards to take the agent's higher health and higher projectile spread into consideration. Figure 3.10 shows the accumulated rewards from the training of each iteration. As can be observed, the reduction in rewards is especially visible in the fourth iteration, which struggled to significantly increase from the initial rewards. This ended up being a consequence of the wider shape of the enemy type, which led to more collisions with other entities when rotating to fire toward the player and its higher cooldown for shooting. Despite these issues, the resulting agent learned to perform competently, taking advantage of its wider projectile spread to affect a larger area around the player. Additionally, it learned to cover a wider area of the environment by cooperating with other enemy entities.

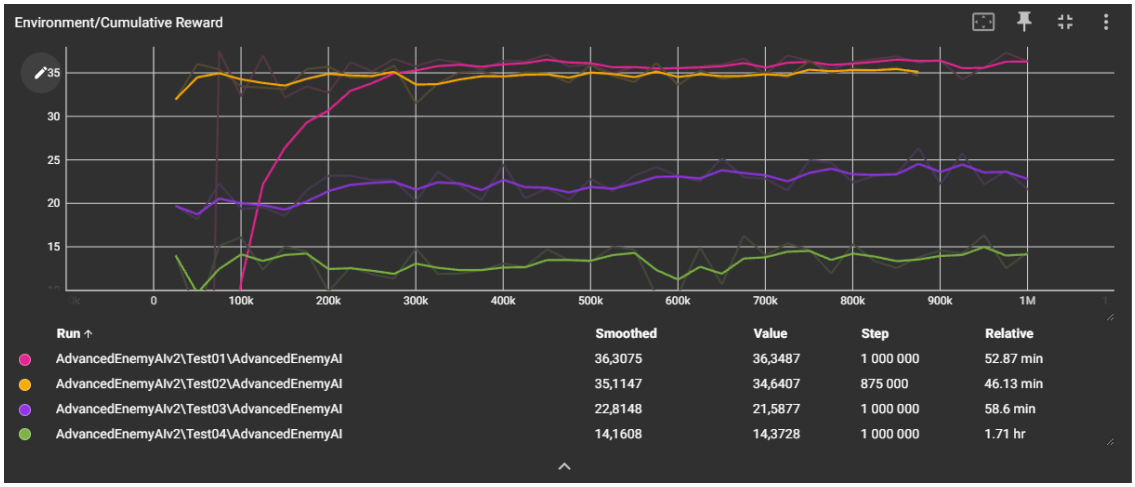


Figure 3.10: TensorBoard Graph for Optimized Models of Advanced Enemy

Elite Enemy

This last enemy type was trained by following the same approach described in the previous subsections. Figure 3.11 shows the accumulated rewards from the training of each iteration. This agent struggled the most during the training of the third and fourth model iterations due to it being the largest enemy type. Consequently, it had difficulty avoiding collisions with other entities and struggled to hit the player when he was behind certain obstacles that the enemy could not pass through due to its size. To compensate for this issue, the agent ended up learning to stay at a certain distance from other entities and try to stay close to the center of the environment to maximize its spread of projectiles.

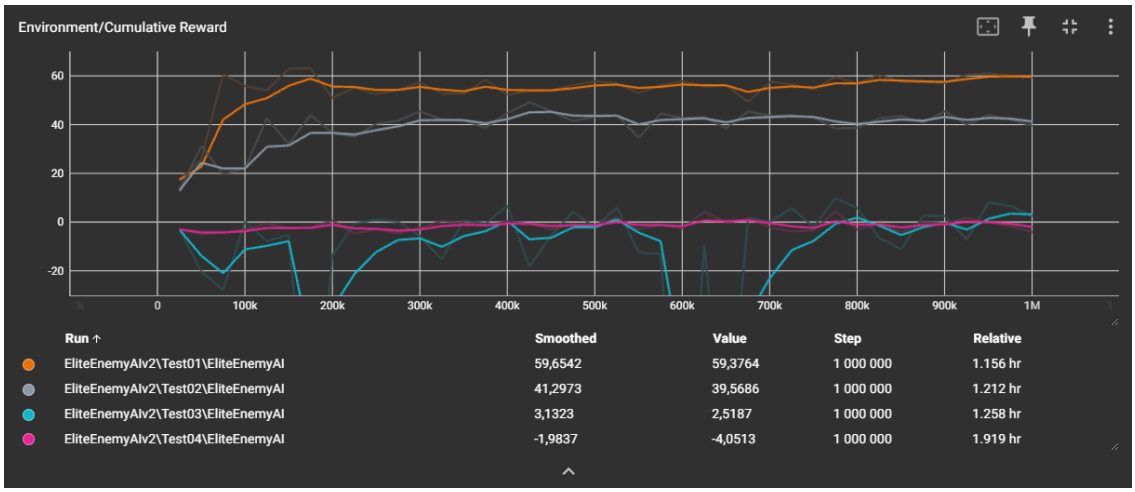


Figure 3.11: TensorBoard Graph for Optimized Models of Elite Enemy

After experimenting with the ML-Agents toolkit and verifying the viability of using this tool, the next phase of the research focused on the implementation of a data collection mechanism into the game prototype and the subsequent collection of player data that would be utilized for implementing the DDA system.

3.5 Data Gathering

To effectively train the DDA system using Unity's ML-Agents toolkit, the training scenarios took into account the behavior of players of different skill levels. Unfortunately, this training process could not occur in real-time because Unity didn't support training and updating the models during the game's execution. However, the toolkit did support collecting gameplay recordings through the usage of demonstration recorders, which collected observations and actions taken by an agent. These demonstrations could then be used to train the player models through behavior cloning. This allowed for training agents that observed and learned to mimic the actions taken by the human players, allowing for the creation of more realistic training scenarios for the DDA system.

Additionally, a secondary data collection mechanism was implemented to collect relevant player performance metrics from each gameplay session for data analysis and more precise classification of the gameplay recordings into different skill levels.

The participants in this data gathering phase were volunteers who were selected based on their availability and gaming experience to guarantee a wider range of skill levels.

This section will describe the implementation of the two data collection mechanisms and analyze the results of the data collected during this phase.

3.5.1 Collection of Gameplay Data

To collect the desired gameplay recordings using the ML-Agents toolkit, it was necessary to transform the player entity into an agent capable of collecting relevant information from its environment and performing actions based on those observations. This is because a demonstration recorder could only be integrated as a component into an Agent entity.

To do this, the player-related code was rewritten to function as an agent while still keeping the existing implemented controls and mechanics. Relevant methods from Unity's agent class were implemented, allowing for the collection of observations, handling actions, and handling the player's input. Additionally, various ray perception sensors were added around the player entity, configured to identify enemies, hazards, and projectiles and their distance from the player. These visual observations were then added to the observations programmed in the player agent as part of the 'CollectObservations' method, which includes the following:

- **Local Position** – The player's current position relative to the game environment, counting as three different observations (x, y, and z-axis).
- **Current Health** – The player's current health, normalized between 0 and 1.
- **Current Accuracy** – The player's current accuracy, is calculated as the proportion of successful hits compared to shots fired.
- **Current Deflection Success Rate** – The success rate of deflections, calculated as the proportion of successful deflections to failed ones (meaning no projectile was deflected during that deflection).
- **Current Level Reached** – The current level that the player has reached.
- **Current Score** – The current value of points' worth of enemies defeated.
- **Average Health Lost Per Level** – The average amount of health that the player has lost per level.

- **Average Time Spent Per Level** – The average duration that the player takes to beat a level.
- **Velocity** – The player's current velocity.
- **Direction** – The player's current direction.
- **Closest Distance to Enemy** – The calculated distance to the nearest enemy.
- **Enemy Density** – The number of enemies within a certain radius around the player.

Some of the above observations are values calculated from a class named 'Performance-MetricsLogger' that is responsible for logging and updating relevant performance indicators from the player. The implementation of this class is detailed in subsection 3.5.2.

Afterward, the 'OnActionReceived' method was implemented, which serves to interpret actions provided by the neural network and translates them into actions taken by the agent. In this case, since the player agent didn't have any model trained as a brain and was supposed to be controlled by the player during gameplay, the 'Heuristics' method was used to convert mouse and keyboard inputs into discrete and continuous values that were then translated into actions by the 'OnActionReceived' method.

For example, movement inputs were continuous values that were translated into directional movement using the existing functionality for movement, while deflecting and dashing were executed based on discrete action inputs (0 to do nothing, 1 to execute that action). The shooting was a bit more complex, with the shooting direction being calculated from two continuous values and the decision of whether to shoot is based on a discrete action input.

Once the player entity was successfully converted into an agent, and the controls were tested, a 'DemonstrationRecorder' component was added. As mentioned previously, this component can record the observations and actions of the agent during gameplay, generating demonstration files that can be used for behavior cloning.

The existing logic for the level manager was adjusted so that the component started recording whenever a new game started and the player entity was instantiated, stopping only when the player lost or won. After stopping, the recorder saved the demonstration file in a local folder with a unique identifier and the timestamp as the file name.

With the demonstration recorder integrated, the prototype could now generate the required files for training the player agents based on real gameplay data. However, to accurately train agents representing players of different skill levels, it was crucial to classify the demonstration files accordingly. This is because using gameplay data from highly skilled players to train an agent representing low-skilled players would confuse the model during training, leading to erratic results.

Therefore, a secondary data collection mechanism was implemented to log the relevant player performance metrics, to allow for correct classification of demonstration files according to the represented skill level. The following subsection describes the defined player performance metrics and the implementation of the collection mechanism.

3.5.2 Collection of Player Performance Metrics

To collect the player performance metrics during gameplay, a class named 'Performance-MetricsLogger' was created, responsible for logging and updating various metrics throughout the gameplay.

For this, several performance indicators were designed, some of which were used as observations of the player agent, as mentioned in the previous subsection:

- **Average Health Lost Per Level** – The average amount of health that the player has lost per level.
- **Average Time Spent Per Level** – The average duration that the player takes to beat a level.
- **Max Level Reached** – The highest level that the player reached during the game.
- **Average Accuracy** – The player's average shooting accuracy throughout the game.
- **Average Deflection Success Rate** – The player's average deflection success rate throughout the game.
- **Max Difficulty** – The maximum difficulty level faced by the player, calculated from the total points worth of enemies and hazards present in a level.
- **Score** – The total points worth of enemies defeated

The 'PerfomanceMetricsLogger' class uses listeners and subscribers to track and update these metrics during gameplay. For example, to track and update the average accuracy, whenever the player shoots a projectile, the logger logs the shot fired and attaches a listener to the projectile's 'OnHitEnemyEvent'. This example can be observed in lines 15 and 16 of code excerpt 3.2.

```
1     private void ShootTowards(Vector2 targetPosition)
2     {
3         Vector2 direction = (targetPosition - (Vector2)transform.
      position).normalized;
4
5         GameObject projectile = Instantiate(projectilePrefab, transform.
      position, Quaternion.identity);
6         Rigidbody2D rb = projectile.GetComponent<Rigidbody2D>();
7         rb.velocity = direction * projectileSpeed;
8
9         Projectile projectileScript = projectile.GetComponent<Projectile
      >();
10        projectileScript.targetTag = "Enemy";
11
12        SpriteRenderer projectileSprite = projectile.GetComponent<
      SpriteRenderer>();
13        if (metricsLogger != null)
14        {
15            metricsLogger.LogShotFired();
16            projectile.GetComponent<Projectile>().OnHitEnemy +=
      metricsLogger.LogShotHit;
17        }
18        if (projectileSprite != null)
19        {
20            projectileSprite.color = Color.blue;
21        }
22
23        Destroy(projectile, 12.0f);
24    }
```

Listing 3.2: Logger Listener Example

The logger thus increments the shot counter every time a projectile is fired and, by subscribing to the 'OnHitEnemy' event, the logger increments the shots hit whenever that event is triggered. With both of these values logged, the logger always knows the average accuracy of the player.

To track metrics related to level completions like max level reached or average health lost per level, the 'LevelManager' class calls the 'LevelCompleted' method in the logger, which updates these metrics accordingly.

Like the demonstration recorder described in the previous subsection, the logger was programmed to start logging the metrics whenever a new game started and to stop and save the metrics when the player lost or won. These values were saved to a local JavaScript Object Notation (JSON) file named with a unique randomly generated ID code and the timestamp.

Another benefit of how the logger was implemented is that, due to the logger constantly tracking and updating these metrics, other classes could receive updates on these values from the logger during runtime. This benefit was later used for training and implementing the DDA system, allowing for the trained model to observe the player's performance during runtime.

After implementing this mechanism within the prototype, it was shared with a limited number of volunteers who played the game multiple times to generate the necessary data. This process and results are detailed in the following subsection.

3.5.3 Results of Data Gathering

A total of seven volunteers participated in this data gathering phase, with each being given the prototype to be played between ten to thirty times, generating a demonstration and metrics file for each game played. A short tutorial was implemented within the prototype to help the participants understand the game mechanics and controls. This tutorial is described in further detail in Appendix B.

The collected player performance metrics files were then analyzed to classify the associated demonstration files into four different skill levels: low, medium, high, and very high. The main classification criteria were in which level the player lost, or if they won the game:

- **Low Skill** – The player lost between levels 1 and 3.
- **Medium Skill** – The player lost between levels 4 and 6.
- **High Skill** – The player lost between levels 7 and 8.
- **Very High** – The player lost between levels 9 and 10, or successfully won the game.

During the analysis of these metrics, some outliers were identified and removed to maintain the integrity of the dataset, such as instances where the player lost or won the game due to an unforeseen bug, or due to external factors, such as having to lose the game on purpose due to having to deal with real-life interruptions.

Additionally, the average values of the performance indicators were calculated for each participant to provide insight into their overall skill level, helping with the classification of the demonstration files. In some participants' cases, a participant could also have their associated demonstration files belonging to different skill levels, due to their skill level improving as they were learning the game between gameplay sessions. Table 3.4 shows the calculated average values for each participant. For data protection, each participant was anonymized

and represented as Participant 1, Participant 2, etc... Regarding the average health loss per level, it's important to note that the player entity had a total of 250 health.

Table 3.4: Average Performance Indicators per Participant

Participant	Avg. Health Lost/ Level	Max Level Reached	Avg. Level Duration (s)	Avg. Accuracy	Avg. Deflect. Rate	Score	Max Difficulty
1	89.78	3.45	15.40	31.88%	22.58%	16.68	8.89
2	154.67	1.36	11.59	39%	20%	5.45	4.73
3	61.74	6.48	11.42	42%	31%	44.81	14.26
4	82.81	2.8	12.55	38.14%	10%	11.9	7.6
5	88.73	2.85	11.18	34.74%	18.17%	12.25	7.7
6	36.51	6.33	10.59	54%	43%	39.58	14.5
7	33.62	7.0	11.79	49.23%	0%	44.71	15.71

Table 3.5 shows the distribution of the demonstration files across the different skill categories after classification.

Table 3.5: Distribution of Demonstration Files By Skill Level

Skill Level	Number of Demonstrations
Low	46
Medium	45
High	20
Very High	21

To balance the number of samples per category, data augmentation was used to increase the number of "ideal" samples from the high and very high skill categories, This involved selecting the most representative samples from these categories and duplicating them to balance the dataset.

With the data collection phase completed, the research progressed to the core implementation phase of this study: the implementation of the DDA system.

3.6 DDA Implementation

This section is dedicated to describing the implementation of DDA in the game prototype. To do this, the existing logic for the procedural generation of levels was adjusted so that this process was controlled by an agent trained to generate the levels based on the player's current and previous performance.

To effectively train this agent to consider scenarios with players of various skill levels, different player agents representing different skill levels were trained so that they could be used during the training of the DDA agent. The data that was collected to train the player agents to imitate real behaviors from players is detailed in section 3.5.

The following subsection describes how the different player agents were trained and the results of that process.

3.6.1 Training the Player Agents

As detailed in section 3.5, the player entity was previously converted into an agent capable of collecting observations from the environment and transforming input into actions. Initially, however, this agent could only be controlled through mouse and keyboard, with no trained model that could output actions based on the inputted observations. This subsection details the process of training four distinct player models, each representing a player of low, medium, high, and very high skill levels.

This training process used a form of imitation learning, which is a type of machine learning where the model learns to perform tasks by mimicking the behavior demonstrated by other humans or agents. In the context of the ML-Agents toolkit, this process was facilitated through behavior cloning, which involved using demonstration files to train the model. As described in subsection 3.5.1, these files record the actions taken by the human player along with the corresponding observations.

To train each model, the below additional parameters from the ML-Agents configuration file were altered:

- **demo path** – The path to the demonstration files. In this case, a folder named low, medium, high, or `Very_High` contained the corresponding files.
- **steps** – The total number of steps in which behavior cloning is used.
- **strength** – Determines the influence of the demonstration files during training. This value slowly decreases to 0 until the number of steps of training reaches the value defined in steps.

The training environment was a simulation of the game using the implemented procedural level generation as if the model was a player participating normally in the game. Due to certain performance issues, perhaps due to the complexity of the player entity, only four parallel environments were used for training. Consequently, the training process took longer than usual.

The rewards defined for the training of each model were made to guarantee that the resulting player agent learned to stay within specific ranges of performance indicators, such as accuracy and deflection rate. The values for the lower and upper boundaries of performance metrics were different for each model and were derived from the approximate average values observed in the collected metrics of the players from the corresponding skill level (see subsection 3.5.2). The model received positive rewards if the values stayed within the target range, and negative rewards if the values fell below the lower bound or exceeded the upper bound. This reward structure was designed to encourage the ideal performance for that skill level and punish deviations from it.

Section C.1 from Appendix C contains the flowcharts that describe the logic of the reward system. The rewards were calculated based on these performance metrics:

- **Accuracy and Deflection Rate** – The model receives constant small positive and negative rewards depending on whether these values are within the desired range.
- **Health Lost and Completion Time** – At the end of each level, rewards are applied based on the health lost and the time that it took to beat the level. For levels beyond the first one, the average health loss per level and average completion time are also checked to apply additional positive or negative rewards.

- **Score and Level Completion** – When the player is defeated, rewards are calculated based on the final score and the level at which the player was defeated. Positive rewards are applied if the score is within a certain range or if the player was defeated within the target level ranges, Otherwise, negative rewards are applied instead.
- **Player Victory** – Lastly, if the player wins the game, a large negative or positive reward is applied depending on whether winning was expected for that skill level.

To calculate these rewards, except for player victory, score, and level completion, a helper function was used to determine the optimal reward based on the present value, lower bound, upper bound, and optimal value. Following these values, the function then adjusted the base reward based on how close the present value was to the optimal value. In the case of when the present value was outside the defined bounds, the reward applied instead was a negative value equal to the base reward.

In the case of calculating the reward for the score and level completion, the positive reward for the score was calculated as the total score multiplied by the base reward, and the reward for the level reached was calculated as the level reached multiplied by the base reward.

Except for the low skill model, other small rewards were also applied through event listeners to help direct the player's behavior, such as applying positive rewards for successfully deflecting, hitting, and defeating targets, and negative rewards when damaged.

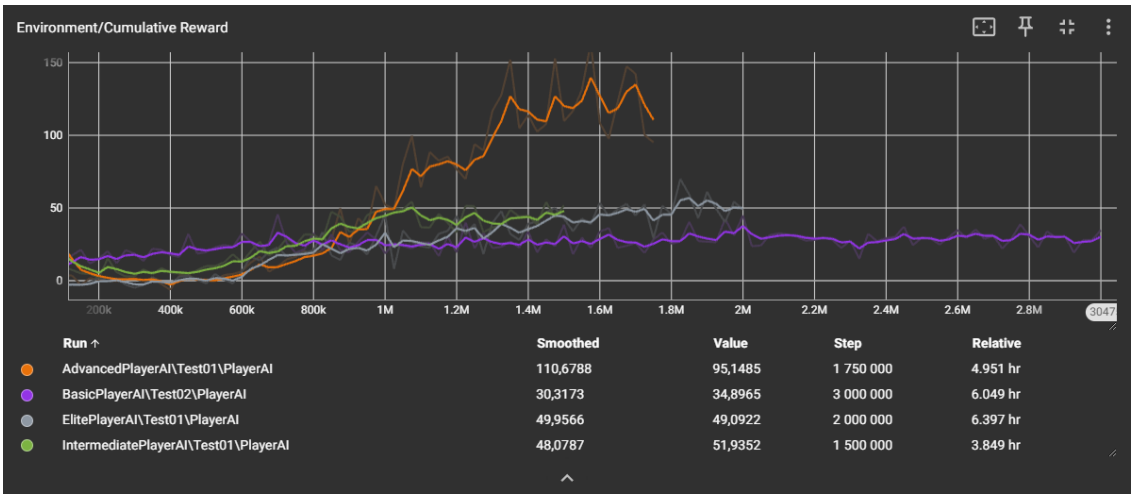


Figure 3.12: TensorBoard Graph for Player Models

The specific boundary, and optimal values defined for each skill level are presented in Table 3.6. These values are presented in the following order: lower bound, optimal value, and upper bound. In the case of score and level reached, these metrics do not have an optimal value since they do not utilize the helper function for calculating the reward.

Table 3.6: Performance Metrics Boundaries By Skill Level

Skill	Accuracy	Deflection Rate	Health Lost	Completion Time	Score	Level Reached
Low	0.10 0.25 0.40	0.00 0.05 0.10	75 115 150	8 19 30	0 / 45	1 / 3
Medium	0.30 0.50 0.70	0.20 0.40 0.60	0 25 50	6 13 20	12 / 90	3 / 6
High	0.40 0.60 0.80	0.30 0.50 0.70	0 20 40	6 13 20	24 / 120	6 / 8
Very High	0.40 0.70 1.00	0.30 0.65 1.00	0 13 25	4 12 20	32 / 150	8 / 10

Using imitation learning, the player model for low skill was successfully trained. However, training higher level skill models proved to be increasingly challenging, with the model struggling to imitate them. This occurred likely due to the more diverse and complex strategies used by skilled players or an insufficient amount of demonstration files.

To resolve this issue, several configurable methods were implemented to mechanically assist the agents by adjusting their aim, deflection timing, and dodge timing and direction, with varying parameters according to the skill level of the player agent.

For aim adjustment, the method adjusted the shooting direction of projectiles toward the nearest enemy. This degree of adjustment varied with the skill level defined on the agent, with higher skill levels adjusting the aim more precisely.

To improve the deflection timing, the player agents were configured to execute parries based on the proximity of the projectiles, with higher skill level agents only triggering the parry when the projectile was in a more effective range.

Lastly, improving dodge timing was done by calculating the safest direction to dash based on incoming projectiles. In this case, to help differentiate between the skill levels, lower skill levels had a chance of failing to trigger a dash representing the fact that less experienced players tended to ignore that mechanic.

By implementing these methods, the four models were able to be successfully trained, with each demonstrating a considerable difference in skill level. Other attempts with different parameters were made when training the models, but they were stopped mid-way when it was clear that the rewards would not reach a satisfactory value for the model. The described four models were the ones that demonstrated the desired results and thus were trained until the end. Table 3.7 shows the parameters and results of these models. The rewards listed on the table for the performance metrics, except score and level reached, are the base values that are adjusted depending on whether the specified performance metric is within the target range and how close it is to the optimal value. In the case of score and level reached it is the positive reward that is multiplied by the score and level reached in case the metric is within the target range.

Figure 3.13 shows the accumulated rewards over time for the four models. As can be observed, the models show a trend of increasing rewards from basic to intermediate to advanced, due to rewards from reaching the target levels increasing. The advanced player model shows the biggest improvement, peaking at around 1.6 million steps before plateauing. However, the elite player model, despite significant initial progress, started to struggle with reaching the desired level range, leading to a significant slowdown in cumulative rewards.

Additionally, the training time for these player agents was significantly longer compared to the enemy agents. This is possibly due to the increased complexity of the player entity, which led to the necessity of increasing the number of training steps to reach a learning plateau.

Despite the challenges faced in training the different player models, including the longer training time and the need to create helper methods when the demonstration files weren't enough, the overall training process was successful. Despite the elite player model underperforming compared to the other models, each model managed to reach a performance plateau that accurately represented its respective skill level and the corresponding player behavior, which was essential for the next phase of implementing DDA into the game prototype.

In the following subsection, the process of training the model for DDA is detailed. Using the player models developed in this phase, player agents of different skill levels were simulated to train the DDA agent with realistic game scenarios.

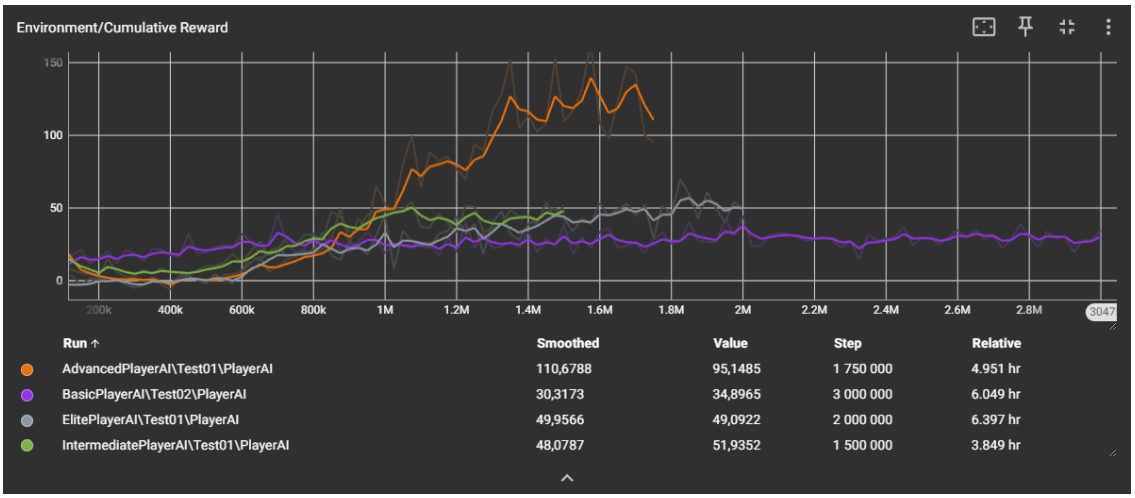


Figure 3.13: TensorBoard Graph for Player Models

Table 3.7: Player Model Results

Model	Parameters	Rewards	Results
Low Skill	**batch size** = 1024 **buffer size** = 20480 **learning rate** = 0.0005 **num epoch** = 5 **hidden units** = 512 **num layers** = 5 **max steps** = 3000000 **time horizon** = 64 **strength** = 0.4	**Accuracy** = 0.05 **Deflection Rate** = 0.05 **Health Lost** = 0.5 **Completion Time** = 0.5 **Level Reached** = 2.0 **Score** = 0.2	**Reward** = 34.9 Agent adopts behavior patterns of low skill players, such as shooting from corners, avoiding dashing, and using deflection without regard for timing.
Medium Skill	Same as the above model, with these changes: **buffer size** = 40960 **max steps** = 1500000 **strength** = 0.5	Same as the above model, with these additions: **took damage** = -0.1 **hit target** = 0.05 **deflection** = 0.05 **defeated enemy** = 0.02* enemy's difficulty value	**Reward** = 51.94 Agent is more competent, regularly reaching level 4 or 5, thanks to more strategic positioning and the assistance of the implemented helper methods.
High Skill	Same as the above model, with these changes: **max steps** = 2000000 **strength** = 0.4	Same as the above model, with this addition: **win game** = 5.0	**Reward** = 95.15 Agent regularly reaches level 7 or 8, frequently dodging and deflecting projectiles. **Note**: Training stopped at step 1750000 due to reward hitting a plateau.
Very High Skill	Same as the above model.	Same as the above model.	**Reward** = 49.09 Agent does not show as significant of an improvement compared to previous agents, possibly due to reaching a learning plateau. However, this agent regularly reaches level 8 or 9, despite rarely winning the game.

3.6.2 Training the model for DDA

As mentioned previously, DDA was implemented in the game prototype by adjusting the procedural level generation to be controlled by a model that adjusted the difficulty when generating each level to maintain the player's engagement. To do this using the ML-Agents toolkit, the Level Manager responsible for the procedural level generation had to first be converted into an agent with its own observations and actions.

Various observations, all normalized, were added to the Level Manager, mostly focused on observing various metrics related to the player's performance and the game's state. The Performance Metrics Logger implemented in 3.5 was used to gather the relevant metrics during gameplay. In the case of the first level of a game being generated, most observations had a default value because there were no previous data points to draw from, such as the health lost or time spent last level. However, when generating the first level of a game after another game had already been played during the same session, the agent retained the results from the last level played in the most recent game. This way, the agent could observe the results of the previous game and adjust the first level of the next one accordingly. The observations added were the following:

- **Player's Current Health** – The current health of the player, to indicate how well the player is surviving against the game's challenges.
- **Health Lost Last Level** – The health lost by the player on the previous level. This observation helps to understand the impact of recent difficulty adjustments.
- **Current Level Reached** – Records the current level number,
- **Current Score** – The player's score, to reflect their offensive capability.
- **Time Spent Last Level** – How much time the player spent to beat the most recent level. This helps the agent measure the pacing of the game.
- **Deflection Success Rate** – Shows how often the player succeeds in deflecting projectiles.
- **Accuracy** – The player's precision in hitting targets.
- **Previous Difficulty Level** – This corresponds to the difficulty score of the previous level completed, which is calculated as the total amount of points that each hazard and enemy instantiated in the level is worth.
- **Previous Enemy and Hazard Counts** – These observations are the number of hazards and enemies instantiated in the previous level.
- **Previous Enemy and Hazard Type Focus** – These observations record what enemy and hazard types were chosen to be instantiated in the previous level.
- **Level Initialization Flag** – Indicator for whether a new level has already started or not. This is used to help the agent distinguish whether a level is ongoing or whether it has been completed and awaits the generation of a new level.

The algorithm for procedural level generation was also altered to receive its parameters based on the outputs of the NN. Instead of the traditional static difficulty scaling approach of generating each level by instantiating enemies and hazards until the predetermined difficulty value for that level was reached, as described in 3.3.5, the number and types of enemies and hazards instantiated for a level were decided by the agent's neural network.

This was decided by discrete actions outputted by the neural network that specified the number of enemies and hazards to be instantiated, as well as the main and secondary types for both enemies and hazards. The type focus is a parameter used to specify which types of enemies and hazards are predominantly featured in a level. For example, the neural network could decide that the main enemy type of a level was a larger and slower enemy type, while the secondary enemy type was a faster enemy type that helped to compensate for the weakness of the main enemy type.

When each level is generated, the algorithm also saves the chosen parameters of that level as observations, so that the model can more effectively analyze the result of its decision on the player's state.

The reward mechanisms for training the model were designed to teach the agent to adjust the game's difficulty from level to level. During training, whenever a level ended, the model received positive or negative reinforcement from various factors, mostly focused on the impact on the player. Section C.2 from Appendix C contains the flowcharts that describe the logic of the reward system. The rewards were calculated as follows:

- **Health Lost** – One of the most important metrics to evaluate the model's performance is how much health the player has lost. In the game, 25 points of health is equivalent to being damaged by a single projectile, and all damage values are multiples of 25. This way the number of hits the player had taken was also calculated.
  - For the first half of a game, levels 1 to 5, the ideal health loss was set at 25, equivalent to being hit once on a level. If the health lost exceeded this value, the model would receive a negative reward of 0.02 for each additional health point lost beyond 25, meaning 0.5 (0.02*25) per additional hit taken. If the health loss was less than 25, the agent was also punished accordingly.
  - For the latter half of the game, from levels 6 to 10, the optimal health loss increased to 50 points, equivalent to two hits, following the same proportional reward and penalty structure.

  This approach was used to maintain a base level of challenge that was easy enough for the player to succeed in the first half of the game while making it hard enough so that winning the entire game was significantly harder.
- **Level Completion Time** – The level completion time was another important metric to keep the player's engagement to prevent levels from being too short or long. The model was trained to keep the level completion time within 6 to 45 seconds, with an ideal time set at 25 seconds. The reward for this metric was calculated using the optimal utility function described in 3.6.1, which adjusted a base reward, in this case, 0.25, based on how close the completion time was to the optimal value of 25.
- **Player Defeat** – When the player is defeated during training, the reward applied is based on which level the player is defeated.
  - If the player dies in the first half of the game, levels 1 to 5, it indicates that the game was too challenging, and so the model is penalized with a value based on how close it died to the beginning of the game. For example, dying in level 3 would apply a negative reward equal to 2.0 (1.0*(5-3)).

  - For the latter half of the game, from levels 6 to 10, the reward is calculated through the optimal utility function based on how close the level is to 8, which was defined as the optimal level for the player to be defeated in.

- **Player Victory** – If the player wins the game, the model receives a reward based on the player's remaining health at the end of the game, according to the formula in 3.3.

```
winReward = 0.5 * (-10 + 2 * healthBasedDifficulty);
```

Listing 3.3: Win Formula

  In this formula, the attribute 'healthBasedDifficulty' represents how many hits the player has taken throughout the entire game. In this case, the player would have to at least have received 5 hits, equivalent to half of his health, throughout the game to prevent a negative reward. In the case of having received more hits, the model receives a proportionally positive reward.

- **Skill-based Adjustments** – Another metric that was considered to check if the difficulty was appropriate to the player's skill level was by calculating the average of their accuracy and deflection rate and checking the total number of hits that he had taken. A player was considered skilled if the average between his accuracy and deflection rate was at least 40%, and unskilled if it was below 20%. If a skilled player took fewer hits than expected, the model received a negative reward of 1.0 for not having increased the challenge appropriately. If an unskilled player took more hits than expected, the same negative reward was applied for not having decreased the difficulty.

- **Level Generation Adjustments** – Some additional rewards unrelated to DDA were also implemented to resolve certain issues found when training earlier iterations of the model.

  - To prevent repetitive gameplay, the model received a negative reward of 0.5 if the main enemy or hazard type chosen was the same as in the previous level.

  - To ensure that the second half of the game was not overtly simple in its difficulty, the model also received a negative reward of 1.0 if it chose to generate only a single enemy or hazard from level 6 onwards.

During the process of training the DDA model, five different models were trained, each with distinct parameters and alterations to evaluate which configuration demonstrated the best results. Table 3.8 shows the parameter configuration and results for each model. The details and values for the rewards mentioned in the table are described in the above list. Each training episode involved a full simulation of a game, starting from the initial instantiation of the player agent until he won or lost. So that the model learned to respond to players of different skill levels, the player agent was instantiated with a randomly selected player model, chosen from the ones trained in the previous subsection, representing either low, medium, high, or very high player skill level.

Several challenges were encountered during the above process of training the DDA model, mostly due to computational limitations. Due to these limitations, only a single training environment could be simulated at a time, resulting in a significantly longer training duration due to no parallel environments being used. Even after optimizing the game's physics and resolving existing memory leaks, this issue persisted, with the training simulation crashing if more than one training environment was being used. This issue, which was not as significant

when previously training the player and enemy agents, could be due to the increased computational load caused by multiple active agents in the training environment, which included the player agent, multiple enemy agents, and the DDA agent itself.

Table 3.8: DDA Model Results

Model	Parameters	Rewards and Other Changes	Results
1	**batch size** = 1024 **buffer size** = 30720 **learning rate** = 0.005 **beta** = 0.004 **num epoch** = 5 **hidden units** = 512 **num layers** = 5 **max steps** = 5000000 **time horizon** = 64	Implemented Rewards for: **Player Victory** **Player Defeat** **Completion Time** **Health Lost**	**Reward** = 0.97 Agent-generated levels that were too easy to maximize the rewards from the player winning or being defeated in the later levels. It avoided generating more than one enemy and hazard per level, resulting in simplistic and unengaging gameplay.
2	Same as the above model, with these changes: **buffer size** = 20480 **time horizon** = 128	Same as the above model, with these additions/changes: -Adjusted the reward logic for **Player Victory** -Added negative reward for generating only one enemy and hazard in the second half of the game.	**Reward** = 3.10 Agent found a strategy that generally produced better rewards, but was not desirable for a real gameplay scenario since it always chose the same enemy and hazard type, only varying in quantity. This lack of variability led to repetitive gameplay.
3	Same as the above model, with these changes: **learning rate** = 0.003 **beta** = 0.005 **num epoch** = 3	Same as the above model, with these additions/changes: -Added rewards for **Skill-Based Adjustments** -Added negative reward for selecting the same enemy or hazard type twice in a row.	**Reward** = -0.36 Agent began to exhibit the desired behavior by starting to adjust the difficulty in response to the player's performance, while still varying each level. **Note**: Training stopped at step 1750000 due to reward hitting a plateau.

Table 3.9: DDA Model Results (continued)

Model	Parameters	Rewards and Other Changes	Results
4	Same as the above model, with these changes: **buffer size** = 30720 **learning rate** = 0.003 **beta** = 0.02 **hidden units** = 256 **num layers** = 4	Same as the above model.	**Reward** = -2.03 Similar behavior as model 3, with the model still being capable of adjusting the difficulty. However, it struggled more to decrease the difficulty for players of lower skill levels.
5	Same as the above model.	Same as the above model, with these additions/changes: -Reward values for **Health Lost**, **Player Victory** and **Player Defeat** were adjusted to observe if the model's behavior steered towards the desired behavior.	**Reward** = -2.07 Agent produced very similar results to model 4. It might have further improved if it was fully trained, but training had to be stopped prematurely at 450,000 steps due to time constraints and the lack of observed improvements over the previous model.

Due to the complexity of the DDA model, each model required a significantly larger number of training steps to demonstrate significant results, in comparison to the training duration of the player and enemy agents. Some of the models took almost a week to train fully, which was a substantial time investment. Figure 3.14 shows the accumulated rewards over time for the five models.

The training of each model had to be interrupted during the night and resumed the following morning since the computer could not be left on continuously throughout the entire training duration. This process was facilitated by the ML-Agents toolkit, which supports the saving and loading of training checkpoints. When training was paused at night, the current state of training was saved to a checkpoint file, containing the weights of the neural network and other relevant training parameters. The following morning, the checkpoint was then loaded to resume training from that point.

Due to the time investment required to train each model and the necessity to advance to the experiment phase of the research, the third trained model was selected for implementation, since it was the one that demonstrated results closest to the desired behavior.

Figure 3.14: TensorBoard Graph for DDA Models

This urgency to have a DDA model ready was critical because the experiment phase, which involved gathering various research participants and carrying out validation experiments, could not proceed without it and needed to be accomplished in June. This timing was essential to avoid conflicts with the participants' exam schedules or vacations, which would impact their availability. Coordinating the schedules of multiple participants required careful planning and significant time investment, and further delays caused by training more model iterations would have disrupted this process. Therefore, a model had to be chosen to meet the research timeline and guarantee the participants' availability.

Despite the possible improvements, when doing some initial testing with the chosen model, it demonstrated the capacity to adjust the difficulty from level to level. With a functional model trained, the next step was to validate the effectiveness of this solution through testing. The next chapter of this thesis describes the experiment conducted to validate whether the solution adjusted the difficulty appropriately in response to the player's performance and whether it was beneficial for the player's engagement.

Chapter 4

Experiment

The main objective of this chapter is to evaluate the effectiveness of the DDA model integrated into the game prototype developed in the previous chapter. The experiment was designed to assess whether the model managed to successfully adjust the game's difficulty in response to a player's skill level, and if it managed to consequently improve the player's engagement.

Given the nature of DDA, where the effectiveness of the difficulty adjustment is mainly subjective, and varies between each player, traditional metrics to evaluate success such as accuracy are not sufficient. Unlike typical ML models, where clear metrics can determine whether the model was successful, the success of the DDA model is mainly based on player feedback and engagement levels. As a result, an experiment needed to be conducted with real participants to gather qualitative data. Thus, the experiment compares the results of participants playing the prototype with the DDA model, with the results of the version of the prototype with static difficulty scaling.

Before proceeding with conducting the experiment, various unit tests were conducted through the Unity test framework to verify the integrity of both prototypes and whether all game-related functions, such as player movement, enemy behavior, and other gameplay mechanics, were correctly working. Additionally, Unity's Physics Debugger was used to identify and correct issues related to game object collisions and other physics-related interactions.

This chapter begins by describing the setup of the experiment, how the experiment was conducted, and the criteria for evaluation. After this, the results of the experiment are presented, and the collected data are analyzed regarding the effectiveness of the DDA model in comparison to the traditional static difficulty scaling.

4.1 Experiment Methodology

Twenty participants with varying levels of experience with video games were selected to guarantee a wide range of skill levels. Each participant played two versions of the developed game: one with the DDA model integrated and the other with static difficulty scaling.

To evaluate each participant's experience with both prototypes, a modified version of the Game Experience Questionnaire (GEQ) was used. It is a widely used tool in the field of game research and is designed to measure various aspects of a gaming experience, such as perceived difficulty, emotional responses, and so on (IJsselsteijn, de Kort, and Poels 2013).

Both prototypes also had the developed performance metrics logger (See section 3.5 activated to record relevant gameplay metrics from each session, so that the collected data could be used to objectively analyze each participant's performance.

The following subsections describe the participants' demographics and experience with video games and how the experiments were conducted, including the measures implemented to prevent bias and how the survey was modified for the experiment.

4.1.1 Participants' Background

The twenty participants were selected for the experiment based on their availability, their varying degrees of experience with video games, as well as different levels of familiarity with the genre of the prototype developed for the experiment. Table 4.1 summarizes basic information and video game experience of each participant.

Table 4.1: Experiment Participants' Information

Participant	Age Range	Gaming Experience	Experience with Prototype Genre	Weekly Gaming Hours in Past Year
1	35-44	More than 10 years	A little experience	Less than 1 hour
2	18-24	More than 10 years	Moderate experience	6-10 hours
3	55-64	Less than 1 year	No experience	None
4	55-64	More than 10 years	A little experience	None
5	18-24	6-10 years	No experience	6-10 hours
6	18-24	More than 10 years	A little experience	11-20 hours
7	18-24	More than 10 years	A little experience	More than 20 hours
8	18-24	More than 10 years	Moderate experience	1-5 hours
9	18-24	More than 10 years	A little experience	More than 20 hours
10	18-24	More than 10 years	Moderate experience	More than 20 hours
11	18-24	More than 10 years	A little experience	6-10 hours
12	18-24	More than 10 years	A little experience	11-20 hours
13	65+	Less than 1 year	No experience	None
14	25-34	More than 10 years	Moderate experience	Less than 1 hour
15	18-24	3-5 years	No experience	More than 20 hours
16	18-24	More than 10 years	A little experience	6-10 hours
17	18-24	More than 10 years	Moderate experience	1-5 hours
18	18-24	More than 10 years	A little experience	11-20 hours
19	25-34	More than 10 years	A little experience	11-20 hours
20	25-34	More than 10 years	A little experience	1-5 hours

The majority of the participants had significant experience with video games, with more than half having more than 10 years of experience. While the research attempted to balance this bias through the diversity in the participants' familiarity with the prototype's genre and varying weekly gaming hours over the past year, the results of the experiment described in section 4.2.1 show that there were not enough samples of lower-skilled players to evaluate the effectiveness of the DDA system in adjusting the difficulty to be easier for them.

The following subsection describes how the experiment was conducted with each participant, how the participants interacted with the prototype, and the measures taken to guarantee that the results were unbiased.

4.1.2 Experiment's Procedure

The experiment was designed to evaluate player engagement when playing the two versions of the game prototype: Prototype 1, which had the DDA model integrated for level generation, and Prototype 2, which used static difficulty scaling. The prototypes were named "Prototype 1" and "Prototype 2" without any indication of their differences to prevent any bias. Participants were told that the research had the objective of measuring player engagement, with the details of the experiment and the real differences between both prototypes being explained only when the experiment concluded.

The order in which the participants played both prototypes was randomized to prevent any bias from starting with the same prototype. Exactly half of the participants started with Prototype 1, and the other half with Prototype 2.

Each participant played each prototype five times, winning or losing each gameplay session. After completing the sessions with one prototype, participants immediately had to fill out a modified version of the GEQ for the prototype that they played and rate the subjective difficulty of that prototype, ranging from 1 (very easy) to 5 (very hard). This process was then repeated for the remaining prototype. The details of the modified GEQ survey are described in subsection 4.1.3.

Before starting the experiment, each participant also had to complete the integrated tutorial to familiarize themselves with the basic controls. This was done so that every participant had a basic understanding of how to play the game, independent of their previous experience with the game's genre.

To prevent "contamination" of the responses, the participants were also asked to not compare their answers for one prototype with those of the other. This was also enforced for the question about rating the subjective difficulty of both prototypes.

This experiment was done both in-person and online. The participants who participated in person did the gameplay sessions on a computer that hosted both prototypes, filling out the questionnaire on Google Forms. For the participants who participated online, which were the majority of the participants (16 compared to the 4 that participated in-person), they received the prototypes through a Google Drive link and played them on their computers, with the research author overseeing the process remotely to provide instructions and to clarify any questions without revealing any information that could cause bias. After completing the experiment, online participants also had to send the generated player performance metrics to the research author for analysis.

In cases where an external issue interrupted a gameplay session, the session was restarted after a short break when the issue was resolved, with the previous unfinished session being ignored.

The next subsection describes how the GEQ works, how it was modified for this experiment, and which values were chosen to measure the player's engagement.

4.1.3 Survey Modification

As mentioned previously, a modified version of the GEQ was used for evaluating player engagement and difficulty perception across both prototypes. Two modules were chosen to be used from the questionnaire, one to evaluate the player's feelings during the gameplay, and the other to evaluate how they felt after playing the game. The full questionnaire sent to the participants can be seen in D. Across all of its modules, the GEQ utilizes a 5-point Likert scale for responses, ranging from "Not at All (0)" to "Extremely (4)".

The core module of GEQ was used to evaluate the in-game experience of the players. This module is structured to evaluate seven different components: Immersion, Flow, Competence, Positive Affect, Negative Affect, Tension, and Challenge. The values for each component are calculated from the average of the values of about five questions of the module each. During the experiment, the participants had to fill out this part of the questionnaire for the corresponding prototype immediately after finishing the five gameplay sessions, so that the experience was fresh in their minds.

While all items of the core module were used in the survey, it's important to mention that some of the questions related to the plot and story of the game, used to evaluate narrative immersion, naturally received low scores since the prototypes had no narrative elements. When analyzing the results, despite all components being considered, emphasis was placed more on the components directly relevant to the gameplay experience, such as Challenge and Competence, and to their engagement, primarily the Flow component.

In addition to the core module, the post-game module from the GEQ was added to the survey to evaluate how the participants felt after playing the game. This module analyzes four components: Positive Experience, Negative Experience, Tiredness, and Returning to Reality. Although the results from this module were not as important for analysis as the components from the core module, this module was chosen to be added to evaluate whether the DDA model led to any significant difference in how the participants felt after playing. Participants had to fill out this module for the relevant prototype shortly after filling out the core module.

The third module of the GEQ, the social presence module, was not chosen due to it being structured to evaluate the psychological and behavioral involvement with other players or in-game characters, which was not an aspect relevant to the prototype.

With the experiments conducted with the described survey, the next step in the process was to gather the results from the experiment and analyze them.

4.2 Results and Analysis

In this section, the results from the experiment will be analyzed to validate the developed solution by verifying if the prototype with the integrated DDA model shows an increase in player engagement, in comparison to the prototype with traditional static difficulty scaling.

This analysis will focus on the calculated scores of the components of the GEQ modules used in the survey and the player performance metrics collected at the end of each gameplay session. Certain components of the modules will be more valued for evaluating player engagement, such as the component "flow", which is a crucial measure for evaluating player engagement compared to components like "Sensory and Imaginative Immersion" or "Tension".

The results are presented through various graphs, which will focus on demonstrating the difference in results between the two prototypes, including their GEQ module scores, and verifying whether the DDA model seems to be adjusting the difficulty appropriately.

The first subsection of this section visually presents the gathered results in the form of graphs, describes them in detail, and statistical significance tests are applied. The second subsection discusses these results, interpreting them and exploring their implications for this research.

4.2.1 Results

This subsection presents the results of the experiment, comparing the results of prototype 1 and prototype 2. The data used to present the graphs was obtained from the results of the questionnaire (See Appendix B) and from the performance metrics files.

To confirm that the findings were reliable and that the observed differences between the two prototypes were statistically significant, paired t-tests with a p-value of 0.05 were applied. The paired t-test was chosen due to it being appropriate for comparing two related samples-in this case, the same participant's responses for the two different prototypes. Python's 'scipy.stats' library, which provides various tools for statistical analysis, was chosen to perform the tests. The library was also used for doing other statistical tests, such as correlation tests to quantify the relationship between certain values.

Figure 4.1 compares the distribution of calculated scores across the GEQ core module components for Prototypes 1 and 2. For each component, the graph shows the median (the central line within the box), the Interquartile Range (IQR), represented as the height of the box, and the spread of the data (represented by the graph whiskers extending from the box) within 1.5 times the IQR. Any outliers are marked as individual points on the graph.

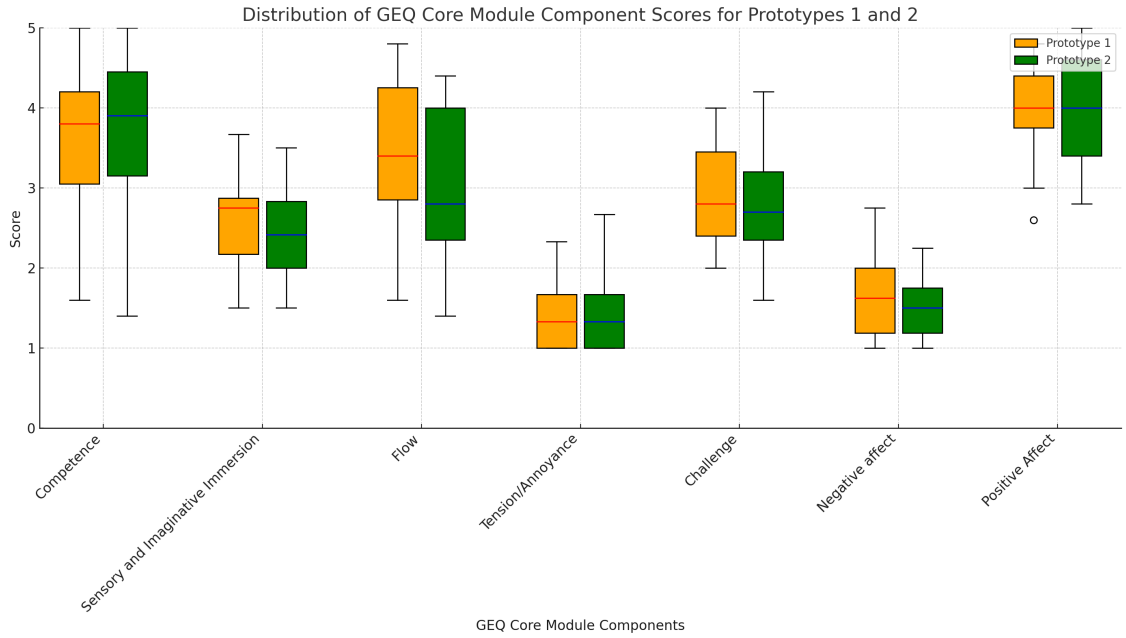


Figure 4.1: GEQ Core Module Component Scores Comparison

The results of the paired t-tests were the following:

- **Competence**: p = 0.681
- **Sensory and Imaginative Immersion**: p = 0.179
- **Flow**: p = 0.006
- **Tension/Annoyance**: p = 0.900
- **Challenge**: p = 0.338
- **Negative Affect**: p = 0.320
- **Positive Affect**: p = 0.531

Both prototypes scored highly on the Competence, Flow, and Positive Affect components, with the Tension/Annoyance and Negative Affect components scoring the lowest. With the significance level set at 0.05, the difference in scores for the Flow component was statistically significant ($0.006 < 0.05$), with the graph showing a higher median and IQR for Prototype 1. The calculated p-values for the other components were above 0.05, meaning that despite there being observable differences in certain aspects of the box, such as the medians or the IQR range, these differences were not statistically significant.

Using the same logic and design of Figure 4.1, Figure 4.2 compares the distributions of calculated scores across the GEQ Post-Game module components for Prototype 1 and 2.

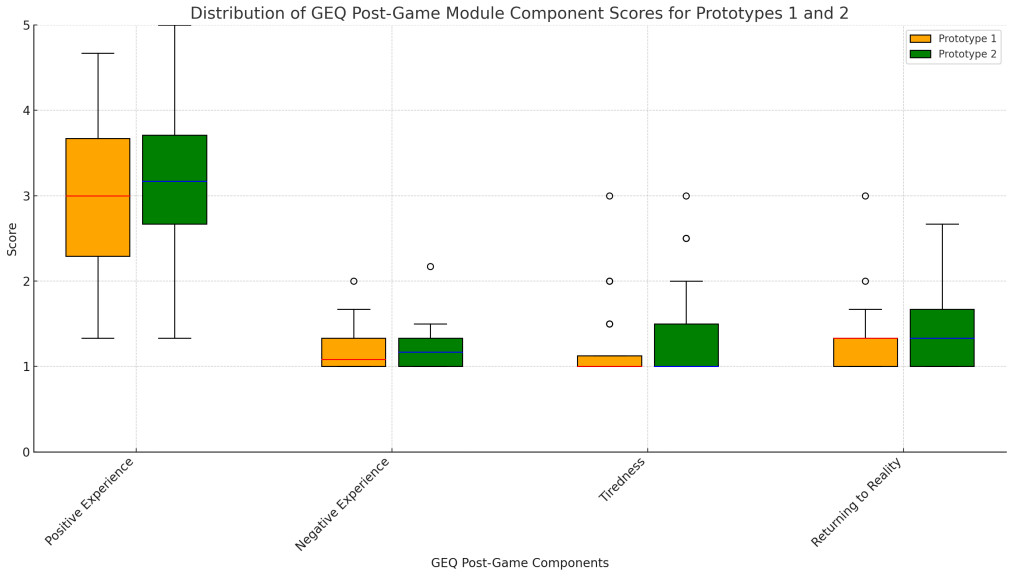


Figure 4.2: GEQ Post-Game Module Component Scores Comparison

The results of the paired t-tests were the following:

- **Positive Experience**: p = 0.036
- **Negative Experience**: p = 0.830
- **Tiredness**: p = 0.330
- **Returning to Reality**: p = 0.546

These results revealed a statistically significant difference in the Positive Experience component ($0.036 < 0.05$), with the graph showing a higher median score and less variability in the IQR for prototype 2. The calculated p-values for the other components were above 0.05, meaning the observed differences are not statistically significant. Both prototypes had a median below 2 for the remaining components, and prototype 1 generally showed a smaller IQR compared to prototype 2, despite more outliers being visible in prototype 1.

The next graph is represented in Figure 4.3. It shows the average difficulty rating provided by the research participants for each prototype, with error bars showing the standard deviation. The y-axis represents the difficulty rating, going from 1 (very easy) to 5 (very hard).

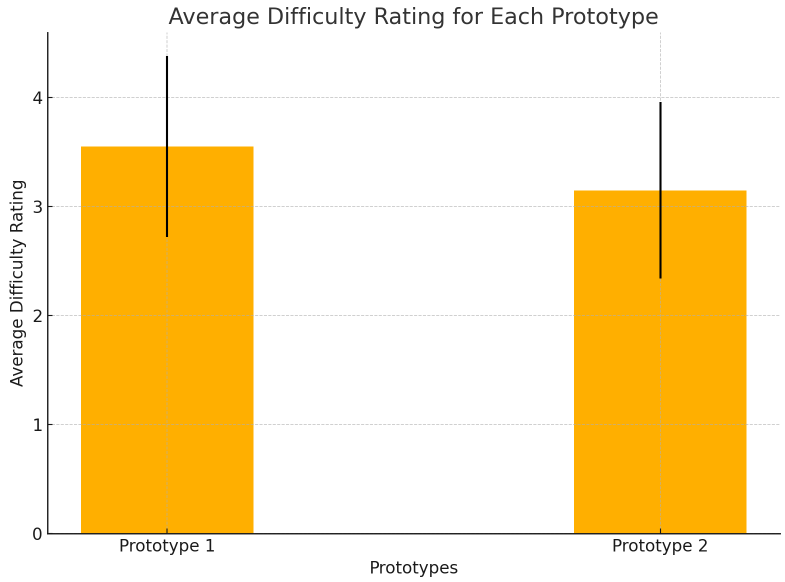


Figure 4.3: Average Subjective Difficulty Comparison

Both prototypes had an average difficulty close to 3 (intermediate), with prototype 1 seemingly having an average difficulty closer to 4 (hard). The result of the paired t-test was a p-value of 0.104, which is above the defined significance threshold of 0.05, meaning that the observed difference is not statistically significant.

Figure 4.4 shows the relationship between the difficulty rating given out by the participants and their actual performance in both prototypes, represented as the average level that they reached. Each participant's data points for both prototypes are connected by dashed lines so that the variations in perceived difficulty and actual performance for both prototypes can be observed.

As can be observed in figure 4.4, the participants who perceived prototype 2 as easier reached a higher average level than prototype 2, while the few data points for prototype 1 varied more significantly in their level reached. The intermediate difficulty showed a wider spread for both prototypes, but the data points for prototype 2 tended to cluster at the higher values of the average level reached. In contrast, the connected data points of prototype 1 were often placed in the hard category, generally with a decrease in the average level reached in comparison. Participants who rated Prototype 1 as hard generally demonstrated a higher level average, while the data points for Prototype 2 showed more variability. For the very hard category, no participants rated prototype 2 as that difficulty.

Figure 4.4: Participant's Subjective Difficulty Versus Average Level Reached Comparison

To further examine the relationships between the perceived difficulty and the average level reached, the Pearson correlation coefficient was calculated to quantify that relationship. For prototype 1, the correlation coefficient was approximately 0.257, showing a weak positive correlation. This means that as the participants rated the difficulty of prototype 1 as higher, they tended to reach higher levels. However, the relationship calculated between these two values was not strong. On the other hand, the correlation coefficient for Prototype 2 was -0.574, meaning a moderate negative correlation. This indicated that higher difficulty perceived for prototype 2 was moderately associated with a decrease in the levels reached.

Additionally, the mean levels reached were calculated for both prototypes to test if the difference was statistically significant. The mean values calculated were approximately 6.411 for prototype 1 and 7.599 for prototype 2, resulting in a p-value of 0.00007 when performing the paired t-test. This demonstrates a statistically significant difference in the participant's performance between both prototypes ($0.00007 < 0.05$). However, there was no statistical significance present in the perceived difficulty between both prototypes, as calculated previously for Figure 4.3.

The last graph, shown in Figure 4.5 explores a different aspect of the gameplay, the relationship between the participant's skill level and their performance across both prototypes. Each participant's skill level is calculated based on the average level reached across all of their gameplay sessions. The "Low" skill level corresponds to an average level reached of 1-3, "Medium" to 4-6, "High" to 7-8 and "Very High" to 9-10. Like the previous graph, Figure 4.3, each point on the graph represents the performance of an individual participant, with lines connecting their performance between both prototypes. The graph also includes dashed lines that represent the average level reached by the participants within each category, so that the differences between both prototypes can be more easily observed.

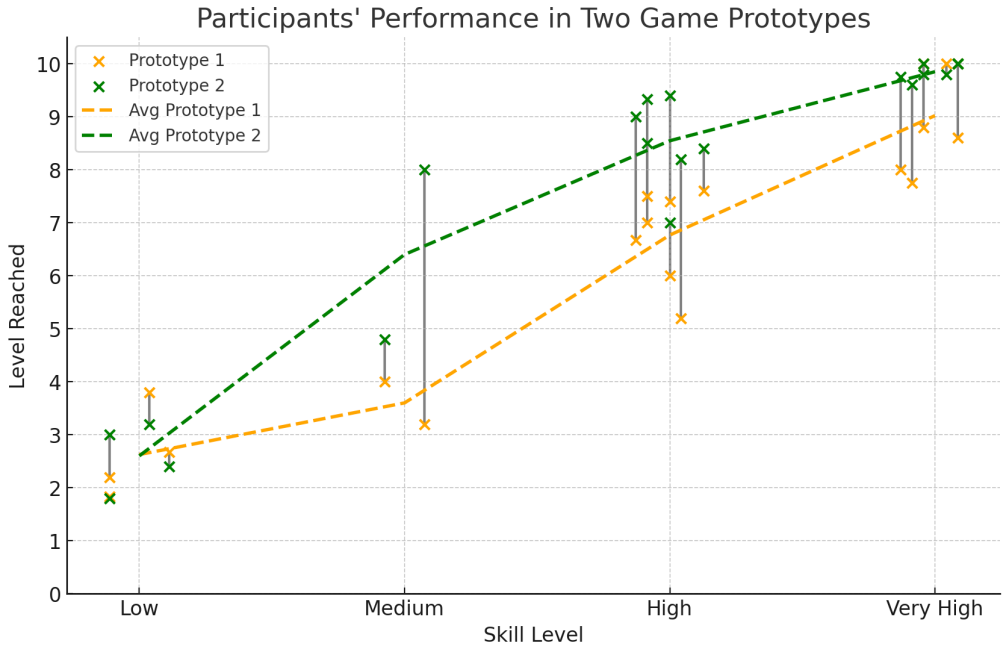


Figure 4.5: Participant's Skill Versus Average Level Reached Comparison

As could be observed, the graph shows that the participants generally had a better performance in prototype 2, except for certain data points in the "Low" skill level category and one in the "Very High" skill level. The average performance lines also show the increase in player performance as the skill levels of the player increased, with the line for prototype 2 showing a steeper rise that slows down when the average performance is closer to the maximum level. However, despite this difference, a paired t-test comparing these two lines revealed no significant statistical difference between them (p-value approximately 0.114, which is above 0.05).

The following subsection discusses the results presented in this subsection and their implication on the objectives of this research.

4.2.2 Discussion

The main objective of this thesis was to develop an innovative DDA system that builds upon existing methodologies and to assess its effectiveness by creating a video game prototype. The experiment was then done to evaluate whether the prototype with the DDA system integrated could improve player engagement and appropriately adjust the difficulty in comparison with another version of the prototype that uses a traditional static difficulty system.

The results of the experiment were mixed, regarding the strengths and limitations of the DDA system. One of the most significant findings was the statistically significant improvement in the Flow component for prototype 1, as shown in Figure 4.1. This suggested that the developed DDA system was successful in creating a more engaging gameplay experience in comparison to a traditional difficulty system, likely due to being more capable of adjusting the gameplay to keep the players in a state of Flow (Csikszentmihalyi 1990). This result

means that the DDA model succeeded in one of its objectives, which was to increase player engagement.

However, other components of the GEQ core module didn't show statistically significant differences between the two prototypes (see Figure 4.2, which suggests that while prototype 1 successfully improved the player's engagement, it did not significantly affect other aspects of the gameplay that the GEQ core module considers, including whether the experience was positive or not, the player's immersion and their perception on the challenge faced.

The results of the Post-Game module, which considers the player's feelings after playing the game, showed a statistically significant difference in the positive experience component between both prototypes in favor of prototype 2. This suggests that, while the DDA system kept the players more engaged in the gameplay, the players felt better after playing Prototype 2. This could be because the difficulty adjustment between levels when playing prototype 1 tended to be drastic, almost as if overcompensating-making the game too easy if the player struggled during the previous level, and vice versa. This issue was observed during the experiments and was even noted by certain participants after finishing the experiment. These fluctuations in difficulty could have thus been the factor that could have led to a lower Positive Experience score, due to the overall positive experience being disrupted.

However, no significant statistical differences were observed for the other components of this module, including Negative Experience or Tiredness, which suggests that despite the positive experience component being lower, it did not introduce additional fatigue or negative feelings in comparison with prototype 2.

Regarding the participant's performance, the participants generally reached higher levels in prototype 2, with the difference in the average levels reached between the two prototypes being statistically significant. However, despite this supposedly meaning that prototype 2 was easier, there was no statistical difference found in the subjective difficulty ratings given by the participants for both prototypes. This could be due to how the participant sample was distributed, which ended up being composed mostly of players of high or very high skill. Taking this factor into consideration, the DDA model likely ended up adjusting the difficulty to be harder more frequently, impacting the participant's overall performance, but not necessarily the perceived difficulty.

Further analysis of the relationship of the perceived difficulty and the average level reached found that there existed a weak positive correlation between these in Prototype 1, meaning that a higher perceived difficulty was weakly associated with reaching higher levels. In contrast, this correlation was moderately negative for prototype 2, meaning that a higher perceived difficulty was moderately associated with a decrease in the level reached. This reinforces the idea that prototype 1 was more effective in balancing the challenge offered with the player's performance.

One of the main limitations of this study was the skewed distribution of the participant's skill levels, with most participants ending up in the high or very high skill level category. So despite the results seemingly indicating that the system was successful in improving player engagement and adjusting the difficulty to be harder for players of higher skill levels, it remained unclear how effective the DDA system would be for players of lower skill levels since the existing data points for that category ended up being too limited to discover if the system was able to adjust the difficulty downwards accordingly.

In conclusion, the proposed DDA system showed effectiveness in improving player engagement, particularly by maintaining a flow state during gameplay. However, its sometimes drastic adjustments in difficulty led to a less positive experience post-gameplay compared to traditional static difficulty scaling. The performance data also demonstrated that the DDA system was also successful in adjusting the difficulty to be harder for skilled players in comparison to prototype 2. However, future research should aim for a more diverse and balanced participant sample to better evaluate the DDA system's effectiveness on the lower skill levels.

Chapter 5

Conclusion

5.1 Conclusion

This chapter was organized into three sections, each dedicated to considering different aspects of the proposed Solution: Evaluation of the Solution, Challenges, and Future Work. It's dedicated to summarizing the main findings and contributions of the thesis, while also identifying opportunities for future research in the field.

The first section, Evaluation of the Solution, evaluates the proposed DDA system and how it performed in practice, basing it on the findings from the results of the experiment. It also analyzes the extent to which the objectives of the thesis were met, such as improving player engagement and appropriately adjusting the challenge through the proposed solution.

The second section, Challenges, describes the main obstacles faced during the research and implementation of the solution, such as unexpected issues faced during implementation or other technical issues that delayed certain parts of the process.

Lastly, the Future Work section proposes directions for extending the research done in this thesis, based on the limitations and issues of the proposed solution discussed earlier. It also suggests other avenues of research that could be more deeply explored in the context of DDA in video games.

5.1.1 Evaluation of the Solution

The main goal of this thesis was to implement and test a DDA system through the usage of an underexplored technique or method in the existing state of the art. As mentioned in the conclusion of the State of the Art chapter (See section 2.3.2), Unity was chosen as the platform to develop the system, and the ML-Agents toolkit, which utilizes DRL, was selected as the method to implement the DDA system. This method represents an innovative contribution to the field since the existing state of the art mentions only one other work where the toolkit was utilized for implementation of DDA (Xiong 2019).

To confirm whether the developed DDA system was successful, it needed to demonstrate the capability of adjusting the difficulty of the game based on the player's performance, maintaining an appropriate level of challenge, and consequently improving the player's engagement.

The data gathered from the GEQ and player performance metrics demonstrated substantial evidence that the system was successful in fully, or at least partially, achieving the objectives defined to verify the success of the solution. More specifically, through analysis and comparison of the GEQ component scores, the DDA system has demonstrated a significant

increase in the player's concentration and engagement, as described by the score of the "Flow" component. This shows that the system was effective in keeping the participants in a state of engagement, which is one of the objectives of DDA.

When evaluating the system's effectiveness in adjusting the challenge level to be more appropriate to a player's skill level, the results were mixed, however. The system was successful in adjusting the difficulty to be higher for the participants of higher skill, evidenced by the fact that those participants generally reached lower levels in the prototype with the DDA system in comparison with the other. However, whether the system is capable of appropriately adjusting the difficulty downwards for lower skilled players is less clear. While the results of the small subset of participants that showed lower skill levels demonstrated some indications that the model seemingly adjusted the difficulty to be easier, the sample was too small to draw any statistically significant conclusions in comparison with the other prototype. Thus, while the system has shown to work as intended with players of higher skill, further testing with a more diverse pool of participants would be necessary to evaluate the DDA system's performance across different skill levels.

While the proposed solution met, or at least partially, all the objectives of a successful DDA system, there are some areas for improvement. Despite the overall success in adjusting the difficulty, there were some instances noted where the difficulty adjustment between levels was too abrupt, increasing or decreasing the difficulty too much between one level and another, which may have impacted the score of the positive experience component to be lower compared to the score of the static difficulty scaling version of the prototype.

Despite some issues found, the proposed solution developed in this thesis not only successfully achieved the main defined objectives of a DDA system, but also did so through an innovative use of the Unity platform and the ML-Agents toolkit, which uses DRL. Despite some issues found that would need to be resolved with further refinement, the approach has shown significant promise over traditional difficulty systems.

5.1.2 Challenges

The development of the DDA system faced several challenges, mostly as a consequence of the computational complexity of training the model, and the inherent complexities in training such a system for real-time gameplay.

Due to computational limitations, training the different models became increasingly computationally heavy as the training scenarios became more complex, with more active agents involved like the player agent, several enemy agents, and eventually the DDA agent itself. This led to fewer parallel training environments at a time, until it was only possible to simulate a single training environment when training the DDA model, which significantly extended the training duration for each iteration of that model. Even with a single training environment, initial training of the model led to several crashes of Unity, which interrupted the training. Optimizing the game's physics and resolving existing memory leaks only partially mitigated this issue, significantly reducing the rate of crashes when training the model, but still now allowing more than one training environment to be used at a time. This challenge accentuated the need for more robust computational resources that could handle a larger number of active agents in multiple training environments.

Additionally, the complexity of the DDA model itself led to the necessity of a significantly larger number of training steps compared to the training of simpler models, such as the models for the enemy or player agents. Some of the later iterations required nearly a week

of continuous training to show more significant results, which represented a significant time investment for this research. The real duration of training these iterations was longer in reality due to the need to periodically pause the training sessions in the night and resume the following morning. This was necessary due to not being able to leave the computer running continuously for such extended periods. However, the ML-Agents toolkit used mitigated the inefficiency of this process, since it allowed the saving and loading of checkpoints to resume the training process without any loss in the process.

Another challenge faced occurred during the training of the player models, which were used to simulate the different skill levels of the players. Despite some success in training the models to simulate the behaviours of players of lower skill, the models representing players of higher skill struggled to imitate their behaviour. This issue likely occurred due to the increasingly complex and strategic behaviour demonstrated by players as their skill level increased and the lack of sufficient demonstration files to train the more advanced models. To resolve this issue, several helper methods had to be integrated to mechanically assist these agents during gameplay by adjusting their aim, deflection and dodge timing. The implementation of these methods made sure that the agents could perform effectively across all the defined skill levels despite the limitations identified.

Lastly, as mentioned in the previous section, another limitation found was the size of the subset of the participants representing players of lower skill, which was too small to draw any statistically significant conclusions regarding the effectiveness of the DDA model in adjusting difficulty for players of lower skill.

These obstacles found during development show the need for future research and development of the solution, which is described in the following subsection.

5.1.3 Future Work

The use of the ML-Agents toolkit, which utilizes DRL, has shown potential avenues for further research and refinement on the methodology used for implementing DDA for a video game prototype.

One of the primary areas of the research that could be further developed is the training methodology used for developing the agent responsible for the DDA in the prototype, which was responsible for generating each level of the game, choosing how many, and which enemy or hazard types to instantiate. While the current implementation has shown promising results, future research could focus on optimizing the training scenarios and parameters for better results or finding better methods to more efficiently use the computer's resources. These approaches could significantly optimize the training process, reducing the training duration and improving the stability of the models, leading to potentially more effective and responsive DDA systems.

Another area that could be further researched is the optimization of the parameters of the DDA model or the negative or positive reinforcement defined for training. Future research could involve fine-tuning these parameters to find the optimal configuration that maximizes the results of the model in adjusting the difficulty appropriately and maximizing the player's engagement.

In addition, expanding the application of the ML-Agents toolkit for DDA beyond the genre of the specific game prototype developed for this research could significantly enhance the versatility of this system, potentially resulting in a DDA framework that can more easily

be adapted to different video game genres. Doing this could involve standardizing the performance metrics or the observations collected by the model to allow the DDA system to be better integrated into different game environments with minimal adjustments.

In this research, the developed DDA system is responsible for procedurally generating each level, but future research could explore further how DDA and PCG can be more integrated to dynamically change other aspects of the game environment based on the performance of the player.

In conclusion, while the thesis has established a strong foundation for using the ML-Agents toolkit for implementing DDA systems, there are various opportunities for further exploration and refinement of the solution. Currently, a scientific article regarding the findings of this thesis is being prepared for submission to IEEE Transactions on Games.

Bibliography

Andrade, K. de O. et al. (May 2016). "Dynamic difficulty adjustment with Evolutionary Algorithm in games for rehabilitation robotics". In: *IEEE Xplore*. url: https://doi.org/10.1109/SeGAH.2016.7586277.

Caldas, O. I. et al. (Jan. 2023). "Behavioral and Psychophysiological Measures of Engagement During Dynamic Difficulty Adjustment in Immersive Virtual Reality". In: *JUCS - Journal of Universal Computer Science* 29.1, pp. 16–33. url: https://doi.org/10.3897/jucs.89412.

Carew, J. (Feb. 2023). *What is reinforcement learning? A comprehensive overview*. url: https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning.

Charles, D. and M. Black (2004). "Dynamic Player Modelling: A Framework for Player-centred Digital Games". In: url: https://www.cp.eng.chula.ac.th/~vishnu/gameResearch/AI_november_2005/DynamicPlayerModelling.pdf.

Csikszentmihalyi, Mihaly (1990). *Flow: The Psychology of Optimal Experience*.

Hiremath, Shivayogi V. et al. (2015). "Brain computer interface learning for systems based on electrocorticography and intracortical microelectrode arrays". In: *Frontiers in Integrative Neuroscience* 9. url: https://doi.org/10.3389/fnint.2015.00040.

Hullett, K. and J. Whitehead (2010). "Design patterns in FPS levels". In: pp. 78–85. url: https://doi.org/10.1145/1822348.1822359.

Hunicke, R. (2005). "The case for dynamic difficulty adjustment in games". In: *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology - ACE '05*, pp. 429–433. url: https://doi.org/10.1145/1178477.1178573.

IJsselsteijn, W.A., Y.A.W. de Kort, and K. Poels (2013). *The Game Experience Questionnaire*. English. Technische Universiteit Eindhoven.

Jagoda, P. (2018). "On Difficulty in Video Games: Mechanics, Interpretation, Affect". In: *Critical Inquiry* 45.1, pp. 199–233. url: https://doi.org/10.1086/699585.

Juliani, Arthur et al. (2020). "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627*. url: https://arxiv.org/pdf/1809.02627.pdf.

Li, X. et al. (2010). "To create DDA by the approach of ANN from UCT-created data". In: url: https://doi.org/10.1109/iccasm.2010.5620008.

Massoudi, P. and A. H. Fassihi (2013). "Achieving dynamic AI difficulty by using reinforcement learning and fuzzy logic skill metering". In: *2013 IEEE International Games Innovation Conference (IGIC)*. doi: 10.1109/igic.2013.6659136. url: https://doi.org/10.1109/igic.2013.6659136.

Miettinen, Kaisa and P. Neittaanmaki (1999). *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, GE*. USA: John Wiley Sons, Inc.

Missura, O. and T. Gärtner (2009). "Player Modeling for Intelligent Difficulty Adjustment". In: *Discovery Science*. Ed. by J. Gama et al. Vol. 5808. Lecture Notes in Computer

Science. Berlin, Heidelberg: Springer. url: https://doi.org/10.1007/978-3-642-04747-3_17.

Noblega, Ashey, A. Paes, and E. Clua (2019). "Towards Adaptive Deep Reinforcement Game Balancing". In: *International Conference on Agents and Artificial Intelligence*. Vol. 2, pp. 693–700. url: https://doi.org/10.5220/0007395406930700.

Or, Dvir Ben, M. Kolomenkin, and G. Shabat (2021). "DL-DDA – Deep Learning based Dynamic Difficulty Adjustment with UX and Gameplay constraints". In: *ArXiv*. url: https://doi.org/10.48550/arxiv.2106.03075.

Pedersen, C., J. Togelius, and G. N. Yannakakis (Sept. 2009). "Modeling player experience in Super Mario Bros". In: *IEEE Xplore*. url: https://doi.org/10.1109/CIG.2009.5286482.

Pfau, Johannes, Jan David Smeddinck, and Rainer Malaka (2019). "Deep Player Behavior Models: Evaluating a Novel Take on Dynamic Difficulty Adjustment". In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. New York, NY, USA: Association for Computing Machinery, LBW0171:1–LBW0171:6. doi: 10.1145/3290607.3312899. url: https://doi.org/10.1145/3290607.3312899.

– (2020). "Enemy Within: Long-term Motivation Effects of Deep Player Behavior Models for Dynamic Difficulty Adjustment". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, pp. 1–10. url: https://doi.org/10.1145/3313831.3376423.

Rahimi, M. et al. (2023). "Continuous Reinforcement Learning-based Dynamic Difficulty Adjustment in a Visual Working Memory Game". In: *ArXiv*. url: https://doi.org/10.48550/arxiv.2308.12726.

Rayhan (Feb. 2023). *Difference Between Deep Learning And Neural Network*. url: https://differencebetween.io/deep-learning-and-neural-network/.

Rodriguez, Guillermo Romera (2020). "Player modeling for dynamic difficulty adjustment in top down action games". In: doi: 10.17760/d20335162. url: https://doi.org/10.17760/d20335162.

Romero-Mendez, E. A. et al. (2023). "The Use of Deep Learning to Improve Player Engagement in a Video Game through a Dynamic Difficulty Adjustment Based on Skills Classification". In: *Applied Sciences* 13.14, p. 8249. url: https://doi.org/10.3390/app13148249.

Sekhavat, Y. A. (2017). "MPRL: Multiple-Periodic Reinforcement Learning for difficulty adjustment in rehabilitation games". In: *Proceedings of the International Conference on Serious Games and Applications for Health (SeGAH)*. doi: 10.1109/segah.2017.7939260. url: https://doi.org/10.1109/segah.2017.7939260.

Spronck, Pieter, Ida Sprinkhuizen-Kuyper, and Eric Postma (Sept. 2003). "Online Adaptation of Computer Game Opponent AI". In.

Wang, J.-Y. and Y.-R. Tseng (2013). "Dynamic difficulty adjustment by fuzzy rules using in a neural network controlled game". In: url: https://doi.org/10.1109/icnc.2013.6817985.

Weber, M. and P. Notargiacomo (2020). "Dynamic Difficulty Adjustment in Digital Games Using Genetic Algorithms". In: *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. Recife, Brazil, pp. 62–70. doi: 10.1109/SBGames51465.2020.00019.

Xiong, Hongyou (2019). "Dynamic Difficulty Adjustment: Developing an Adaptive Game AI with Machine Learning". In: url: http://hdl.handle.net/10211.3/210640.

Zohaib, M. (2018). "Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review". In: *Advances in Human-Computer Interaction* 2018, pp. 1–12. url: `https://doi.org/10.1155/2018/5681652`.

Appendix A

Consent Form

Before you start the game, we need to inform you about the data we collect during your gameplay and how we use it. By accepting these terms, you consent to the collection and use of your data as described below:

1. Data Collection:

During your play sessions, we will collect the following information:

- Timestamps of your gameplay sessions.
- Generic ID: A unique, anonymized identifier assigned to your game profile to distinguish between different players.
- Player Performance Metrics: Data related to your in-game performance.
- Player Demonstrations: Recordings of your gameplay actions and decisions.

2. Purpose of Collection:

The collected data will be used exclusively for the following purposes:

- Academic Analysis: Analyzing gameplay metrics to contribute to the development of an academic thesis related to the comparison between traditional and dynamic difficulty systems.
- Player Modeling: The gameplay recordings will be utilized to train player models of different skill sets, and to develop and refine a mechanism within the prototype that adjusts the game's difficulty based on the player's performance.

3. Data Sharing and Privacy:

- Your data will be kept confidential and stored securely.
- It will be automatically deleted within 6 months.
- Data may be used for purely academic purposes in analyzing the gameplay metrics or for training machine learning models.
- You have the right to access the data collected about you and request its deletion.

4. Consent Withdrawal:

You can withdraw your consent at any time by contacting us at dspjorge@gmail.com. Upon withdrawal, we will stop collecting data from that point forward and will delete your personal data from our records if requested.

5. Agreement:

By clicking "I Accept," you confirm that you have read and understood these terms and agree to the data collection and use as described. If you do not agree, unfortunately, we will not be able to start the game.

Appendix B

Game Prototype Tutorial

The tutorial for the game prototype was designed to introduce the players to the basic game mechanics. The environment of the tutorial imitates the main game environment but does not include the level generation logic. Instead, text instructions appear on the screen as the player progresses through the tutorial.

Upon starting the tutorial, the player was first introduced to the basic movement controls, with the first text prompt explaining how to use the WASD keys to move around. Once the game detected that the player was moving, the text changed to prompt the player to use the left shift to dash while moving. Upon a successful dash, the player was then instructed to use the left mouse button to shoot in a certain direction. Once the player did so, the text then explained how to deflect projectiles by using the right mouse button.

After showing how to use these basic mechanics, the tutorial spawned a basic enemy into the tutorial environment that the player had to defeat. After the enemy's defeat, the tutorial congratulated the player and returned them to the main menu, where they could start the real game.

Appendix C

Flowcharts for Reward Mechanisms

This appendix contains a series of flowcharts that visually represent the processes used to calculate and apply rewards for the player models and the DDA model.

C.1 Player Model Flowcharts

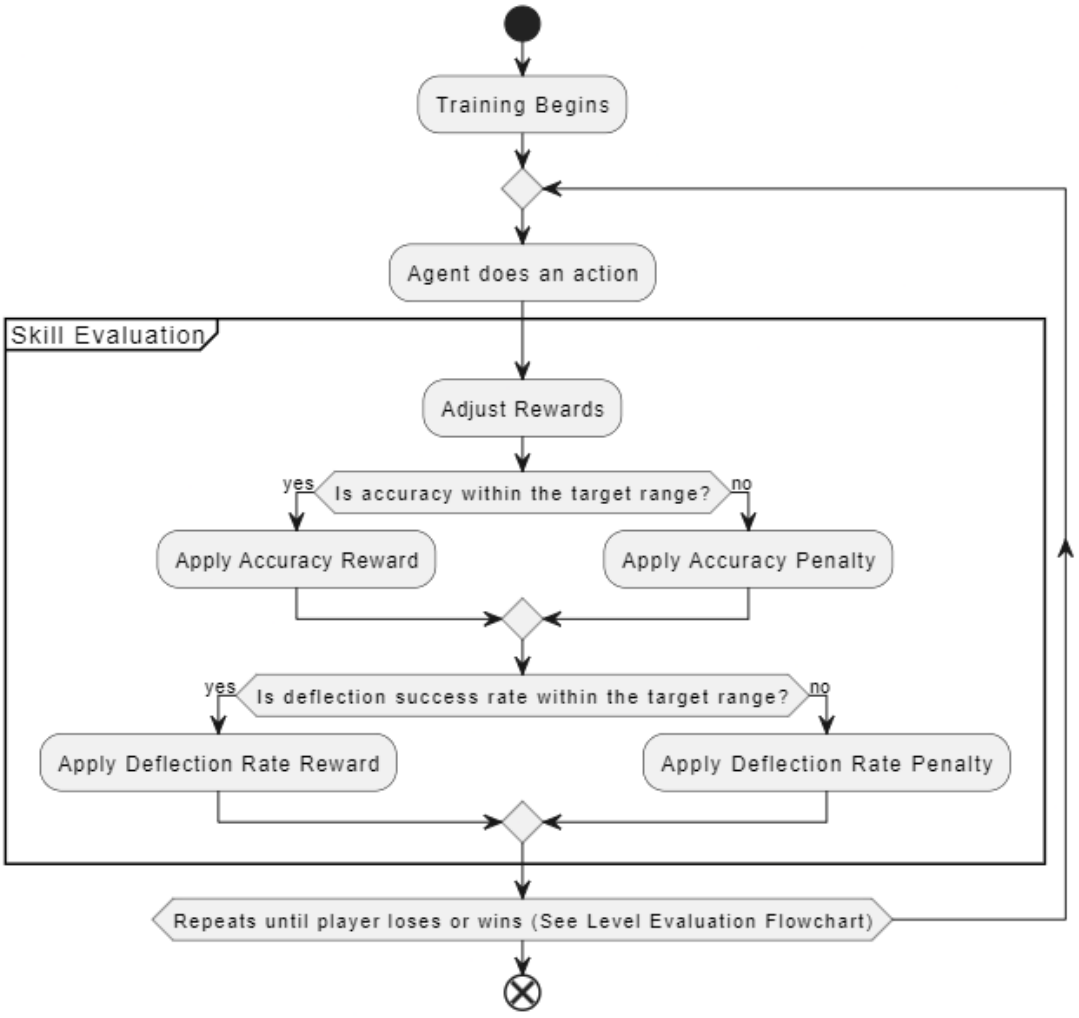


Figure C.1: Player Model Flowchart Skill Evaluation

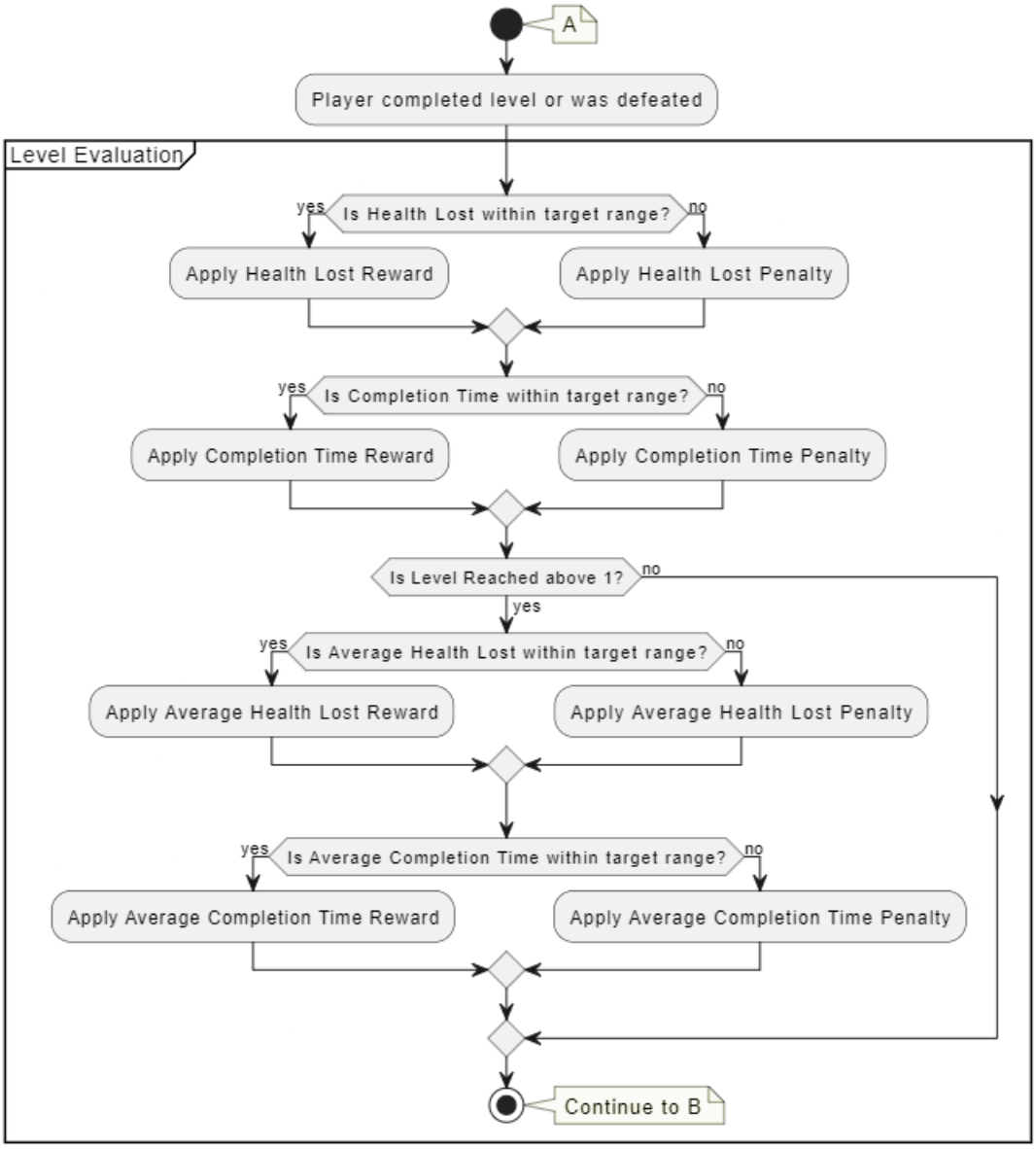


Figure C.2: Player Model Flowchart Level Evaluation

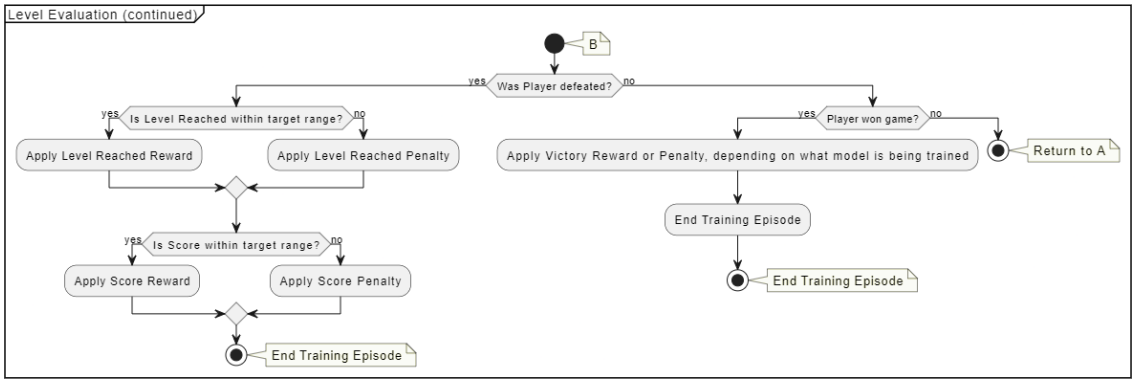


Figure C.3: Player Model Flowchart Level Evaluation (Continued)

C.2 DDA Model Flowcharts

Figure C.4: DDA Model Flowchart (Part 1)

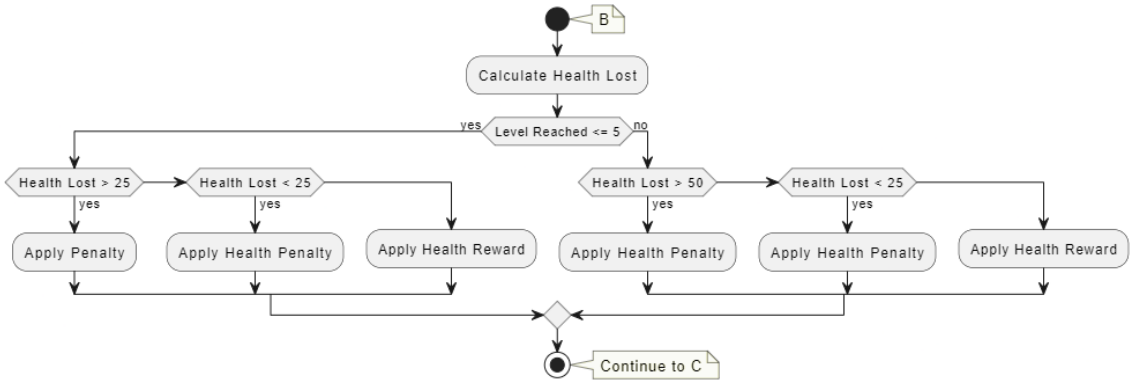


Figure C.5: DDA Model Flowchart (Part 2)

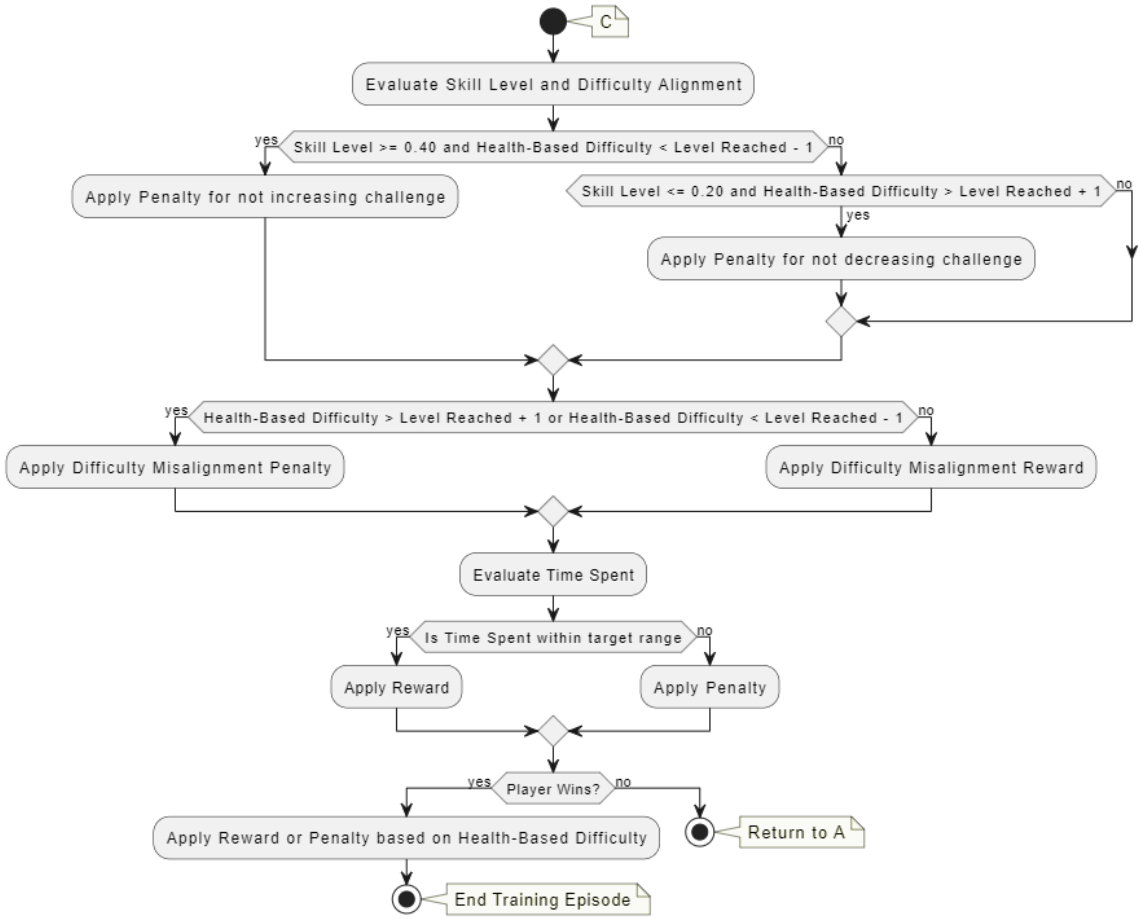


Figure C.6: DDA Model Flowchart (Part 3)

Appendix D

Experiment Survey

This Appendix contains the questionnaire sent to the research participants. The first section gathers basic information related to the participant's gaming background. The second section utilizes the core and post-game module of the GEQ (IJsselsteijn, de Kort, and Poels 2013) to evaluate each prototype, ending with a question on the subjective difficulty of the game.

D.1 Questionnaire on Player Engagement

We are conducting a survey to gather player feedback on two versions of a 2D top-down shooter game prototype. Your participation will help us understand how different methods of difficulty adjustment impact player engagement and experience.

D.1.1 Section One

In this first section, we will measure the participant's experience and background with video games.

1. **What is your age?**

    - 18-24
    - 25-34
    - 35-44
    - 45-54
    - 55-64
    - 65+

2. **How long have you been playing video games?**

    - Less than 1 year
    - 1-2 years
    - 3-5 years
    - 6-10 years
    - More than 10 years

3. **What's your experience with the video game genre of the prototype?**
   The genre is 2D Top-Down Shooter: A 2D top-down shooter is a game where the player controls a character viewed from above, typically moving in all directions and shooting at enemies.

   - No experience
   - A little experience
   - Moderate experience
   - A lot of experience
   - Extensive experience

4. **For the past year, how many hours do you play video games per week, on average?**

   - None
   - Less than 1 hour
   - 1-5 hours
   - 6-10 hours
   - 11-20 hours
   - More than 20 hours

D.1.2 Section Two

This second and final section is dedicated to measuring the player's engagement by utilizing parts of the GEQ (Game Experience Questionnaire), ending with a question on the perceived difficulty of both prototypes.

1. **Prototype 1 - In-Game Player Engagement (Core Module)**

   Please indicate how you felt while playing the game for each of the items on the following scale:
   1 - not at all
   2 - slightly
   3 - moderately
   4 - fairly
   5 - extremely

   1 I thought about other things

   2 I felt content

   3 I felt skilful

   4 I was interested in the game's story

   5 I thought it was fun

   6 I was fully occupied with the game

7 I felt happy

8 It gave me a bad mood

9 I found it tiresome

10 I felt competent

11 I thought it was hard

12 It was aesthetically pleasing

13 I forgot everything around me

14 I felt good

15 I was good at it

16 I felt bored

17 I felt successful

18 I felt imaginative

19 I felt that I could explore things

20 I enjoyed it

21 I was fast at reaching the game's targets

22 I felt annoyed

23 I felt pressured

24 I felt irritable

25 I lost track of time

26 I felt challenged

27 I found it impressive

28 I was deeply concentrated in the game

29 I felt frustrated

30 It felt like a rich experience

31 I lost connection with the outside world

32 I felt time pressure

33 I had to put a lot of effort into it

2. **Prototype 1 - Post-Game Player Engagement (Post-Game Module)**

   Please indicate how you felt after you finished playing the game for each of the items, on the following scale:
   1 - not at all
   2 - slightly
   3 - moderately
   4 - fairly
   5 - extremely

1 I felt revived

2 I felt bad

3 I found it hard to get back to reality

4 I felt guilty

5 It felt like a victory

6 I found it a waste of time

7 I felt energised

8 I felt satisfied

9 I felt disoriented

10 I felt exhausted

11 I felt that I could have done more useful things

12 I felt powerful

13 I felt weary

14 I felt regret

15 I felt ashamed

16 I felt proud

17 I had a sense that I had returned from a journey

3. **Prototype 2 - In-Game Player Engagement (Core Module)**

   (Same as the module for prototype 1)

4. **Prototype 2 - Post-Game Player Engagement (Post-Game Module)**

   (Same as the module for prototype 1)

5. **Please rate the difficulty of each prototype on a scale from 1 to 5, where 1 means "Very Easy" and 5 means "Very Difficult"**.

   - Prototype 1:
   - Prototype 2:

   Responses for each:

   - 1 - Very Easy
   - 2 - Easy
   - 3 - Neutral
   - 4 - Difficult
   - 5 - Very Difficult