A Game Balance Optimization Framework

using Reinforcement Learning and Genetic Algorithms

Patrick Nicolai Andersen

Supervisor: Henrique Galvan Debarba

Master's Thesis
Medialogy, Copenhagen, Spring 2023

The Technical Faculty of IT and Design
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
A Game Balance Optimization Framework using Reinforcement Learning and Genetic Algorithms

Theme:
Game Design, Machine Learning, Reinforcement Learning, Genetic Algorithms

Project Period:
Spring Semester 2023

Project Group:
None

Participant(s):
Patrick Nicolai Andersen

Supervisor(s):
Henrique Galvan Debarba

Copies: 1

Page Numbers: 73

Date of Completion:
May 25, 2023

Abstract:

Game balance is an important aspect of good game design and can make or break a game. It can be a difficult problem to solve with game variables depending on each other in complex relationships. This is currently a long, complicated, and tedious process when developing a game, taking up a lot of valuable time and resources for game studios. This thesis proposes a framework to help automate the process of game balancing with machine learning. This is achieved using reinforcement learning to train agents to play the game, combined with a genetic algorithm to search the game balance space for optimal solutions. This is done in combination with user input to both ensure the creative input of the designer and to help reduce the search space and optimize the algorithm. In order to explore this approach a game was developed and the methodology was applied to optimize the game balance. Game balance is widely defined and for this thesis, the focus has been to find the parameters resulting in an equal win rate between two different factions in the game. The result is a methodology that can be applied in the game development cycle but with several shortcomings and caveats.

Contents

Chapter 1

Introduction

1.1 Motivation

Game balancing is arguably one of the most important aspects of good game design. Good game balance is a requirement for a fun and fair game. The immense complexity of video games makes balancing a challenging and tedious process. It is usually costly and time-consuming as play testing is often the only means of evaluation at a larger scale. In competitive multiplayer games, game balance is especially important as fairness plays a big role. If a single strategy is unbalanced, players will take advantage of it to gain unfair advantages over their opponents, destroying the experience of the other player. This leads to a less dimensional game where only a select few strategies will be viable options. This does not only create a game that is less fun, but it is also a waste of development resources.

"Balance is extremely important for multiplayer games, as a feeling of fairness is critical in not only keeping players interested but also keeping the game entertaining" (p. 24; Morosan 2019)

1.2 Introduction

There is a lot of research on balancing existing and published games (Morosan 2019), but the need for balancing games is especially during development. When the game is released and live there is an audience playing and essentially continuously play testing the game. It is however expensive to hire lots of game testers during development and it might not even be feasible for smaller game studios. Even in larger game studios this is often also done with coworkers, friends, and family and through public sourcing with for instance public or private alpha and beta testing. Regardless of play testing, game balancing will take valuable time away from game designers. Automation of the process could let game designers spend their valuable time on other tasks. There is a gap between academia and game studios. Only larger studios have the resources to pursue the algorithms and methods proposed in research (Morosan 2019). One of the foundations for this thesis is the hypothesis that one of the reasons this gap exists is because the research is done on completed games. This is of course a smart way to research the topic and has provided many valuable insights in the field. The methods are also usually applied to games with a lot of recorded logs and data or with existing AI that can be used for simulation, for instance as seen with the StarCraft games which have served as a foundation for much of the current research in automated game balance. This thesis will investigate how to optimize game balance during the development of a video game. The thesis will document both the development of a video game itself and the proposed process for automating game balancing and investigate how, and to what extent this process can help improve game balancing. The process proposed in this paper involves applying a machine-learning framework during development. By incorporating this workflow we allow for the training of an AI agent with reinforcement learning that can be retrained to adapt to a changing environment. This agent can be used to simulate and evaluate various balance states. A genetic algorithm can then be used with simulations to approximate the best state of the game balance by exploring different game states with different values. The main hypothesis is that this workflow can help automate the game-balancing process. In order to evaluate this, a game is developed and reinforcement learning models are trained during development. The hypothesis will be evaluated mainly through the exploration of this process and through an interview with a game designer.

1.3 Initial Problem Statement

"How can we automate the process of game balancing a multiplayer real-time strategy (RTS) video game during development?"

Chapter 2

State of the Art

2.1 Literature Review

The literature review will summarize and review several papers and research regarding the optimization of game balance in video games. The research covered can be divided based on evolutionary algorithms and machine-learning approaches.

2.1.1 Automating Game-design and Game-agent Balancing through Computational Intelligence (Morosan 2019)

"Automating Game-design and Game-agent Balancing through Computational Intelligence" is a Ph.D. written on the topic of computational intelligence-assisted game balance in 2019 (Morosan 2019). The report primarily investigates genetic algorithms and approximators as a means to explore and achieve game balance. Morosan, M targets several games primarily Ms. Pac-man, Starcraft, and TORCS proving the viability of the solution in various different games with their own sets of challenges (Morosan 2019). Morosan proposes that there is a gap between academia and game studios. He claims that only larger studios have the resources to pursue the algorithms and methods proposed in research. He wants to bridge the gap between academia and game studios (Morosan 2019). To modify game files from outside environments Morosan proposes a balance specification language.

Balance Specification Language

The balance specification language is a structured JSON file, containing all parameters related to the balance of the game, which makes it easy for both humans and computers to read the medium (Morosan 2019). He also specifies that these parameters should contain restrictions on range and precision. Additionally, he proposes an "enabled" boolean, which controls whether the variable is changed by the algorithm or not. Finally, he proposes a "minimize" flag, with the purpose to inform the algorithm to consider the total magnitude of all changes when calculating a score.

In order to evaluate the resulting balance of a game session, Morosan uses metrics and evaluators. Morosan refers to metrics as values defining a play session. These metrics can be anything from win rates to game duration. In order to evaluate the effect of the changes done by the algorithm, he proposes evaluators. An evaluator will access a single metric and contain a target metric value, an "enabled" boolean, and an optional additional parameter. There are several types of evaluators. From simple average and median evaluators to standard deviation and thresholds. Evaluators will compare the metric value of 1 or more games in a session, for instance, the average, and compare it to the target metric. The returned result is the absolute difference between the target average and the observed average multiplied by a weight. Threshold evaluators will compare a set of metrics from several games and return the number of games where the target metric was not reached or exceeded. The paper proposes a simple final score evaluation. A simple weighted sum is proposed because of the simplicity of the solution, which is more likely adopted outside of research. The final score is simply a weighted sum of all the absolute differences between targets and observations. This means that the optima of the resulting score is 0, and the closer the score is to 0 the better (Morosan 2019).

Genetic algorithms for game parameter balance

Genetic algorithms are based on evolution and how genes evolve in nature. The algorithm ranks several randomly generated solutions. The worst solutions "die" off and are omitted and the best solutions are used to spawn new solutions (Morosan 2019). These new solutions are formed based on the previous best solutions with an operator. There are three main categories of operators: crossover, mutation, and elitism.

With the crossover operator, the newly formed solution is based on the combination of two parent solutions. With the mutation operator, a solution is simply changed to create a new one. And finally, with the elitism operator, a solution is kept. There are several ways to conduct a crossover operation. The solution should be seen as an array of numbers representing the parameters used to achieve the evaluation result. The crossover operation can combine two arrays in various ways. A single-point crossover splits the array at a random point and swaps the two arrays resulting in two new solutions. Two-point crossover creates new solutions from swapping three parts of the array. The uniform crossover creates a new solution based on the parent solutions in a given distribution (Morosan 2019).

The strength of these algorithms is how fast and effortless they can find a solution that is approximating the optimal solution. In comparison to neural networks, this method requires much less data to produce results that are potentially just as good or at least closely approximate the best solution (Morosan 2019).

"GAs are capable of finding interesting, often innovative, ways of solving given problems. They do not always generate perfect solutions, but not all tasks require perfect optimality in the first place. Given that games can have many parameters, each with its own limits as to the values it can have, as well as a wide variety of relationships between them, the search space is immense. GAs thrive in these scenarios." (Morosan 2019)

Parameter configurations and results

In this paper, Morosan investigates generational GA using a combination of two-point crossover, a mutation operator, and elitism(Morosan 2019). If this does not suffice to create a population of new solutions, new solutions are generated using the original initialization process. Morosan uses a random seed in order to replicate results. The mutation was applied to a percentage of parameters adding a randomly generated displacement within a given range (Morosan 2019). Through exploratory testing based on the Ms Pac-Man game, Morosan found that mutation rates of 10% and 20% yielded the best results. Aggressive mutation rates of 50% and 40% yielded by far the worst results. Too low mutation rates of 5% also yielded bad results (Morosan 2019). Morosan also found that a reinitialization rate of 20% had a negative impact on results. He argues that a smaller rate of reinitialization could benefit results but because of the rather insignificant impact to the results, this was omitted in the configuration. Morosan also investigated different population rates and found in previous research that a population of 50 was optimal. Through tests on Ms. Pac-Man, he found that a population size of 50 or 100 gave the best results with a total evaluation budget of 2000 simulations. The final suggested parameter configuration is a population size of 50 evaluations consisting of 40% crossover, 20% elitism, and 40% mutation with a mutation rate of 20% or 10% of parameters. The weights and ranges of mutation will vary based on parameter and use case Morosan proceeds to evaluate the method on a more complex and realistic scenario with StarCraft and TORCS. The results of the optimized algorithm parameters yielded better results in less time. The algorithm not only proved to optimize the game balance but also proved to give the designer valuable insights into dynamics between parameters and potential balancing strategies (Morosan 2019).

"For the purpose of the StarCraft experiments, only a subset of the game's parameters was taken into consideration.. ..GAs have no major issues optimizing problems with many more than 12 variables and would not find the extra parameters overly problematic. Better hardware or access to a game's source code would also greatly increase the speed of fitness evaluation, allowing for more individuals in a population, more games to be played, or more generations to be run." (Morosan 2019)

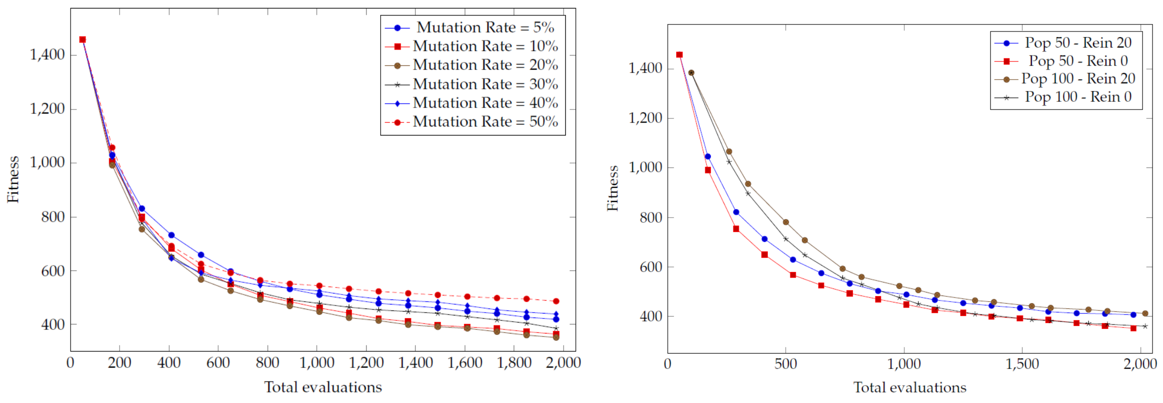


Figure 2.1: *Evaluation of various mutation rates (Left) and reinitialization rates (Right) in Ms. Pac-Man experiment using fitness over total evaluations (Morosan 2019)*

These methods proposed by Morosan in section 2.1.1 are created to work on existing games by modifying their source files using XML XPath. However the idea of using a medium such as a JSON file for modifying parameters from an outside environment, also makes sense during the development process. In order to adapt this method to apply during development, it is proposed to have a dedicated game variables class that can interface with the JSON file. The class will contain all variables that are used throughout the code and overwrite them from the JSON file when executed. The proposed evaluation process will serve as a good foundation for modeling simulations and evaluations in this project. Machine learning approximators however negatively affected the fitness levels and at best had a minimal influence on the computation time. These approximators will thus not be considered in the evaluation process.

2.1.2 An Integrated Process for Game Balancing (Beyer et al. 2016)

This paper investigates a process for game balancing that incorporates both manual game balancing and automated balancing. The authors advocate that while automation is required, it does not make sense to completely omit the perspective and intervention of the designers and manual game balancing (Beyer et al. 2016). The aim is a process that discovers several viable configurations and lets the designer influence the direction of the balancing during the process (Beyer et al. 2016). Beyer et al. apply a fitness function to determine the impact of changed game parameters. An optimization algorithm is chosen and applied, which changes the values of game parameters in order to see a result in the fitness (Beyer et al. 2016). The process is supported by two report documents. The pre-evaluation report and the configuration report. the pre-evaluation report goals of the game balancing is defined as well as other interesting aspects of the game. Such as for instance the target group. These can be more abstract goals related to how the game is meant to be played or what the game is meant to invoke in the player (Beyer et al. 2016). The second activity and configuration report details the technical setup of the game balancing. What scenes of the game is balanced, detailed information regarding the game parameters that are balanced, the fitness function, and the optimization algorithm (Beyer et al. 2016). Once these specifications are set and detailed, the process is carried out. The process is iterative and can switch between manual or automated balancing. The automated steps start by configuring and in some cases implementing the required AI agent, in order to simulate and automate the game. Finally, after every iteration, the data of simulations, manual or automated, are analyzed. The state of the balance and outcome of the test is analyzed, in order to inform decisions moving forward. This can for instance be to eliminate manipulation of certain variables (Beyer et al. 2016).

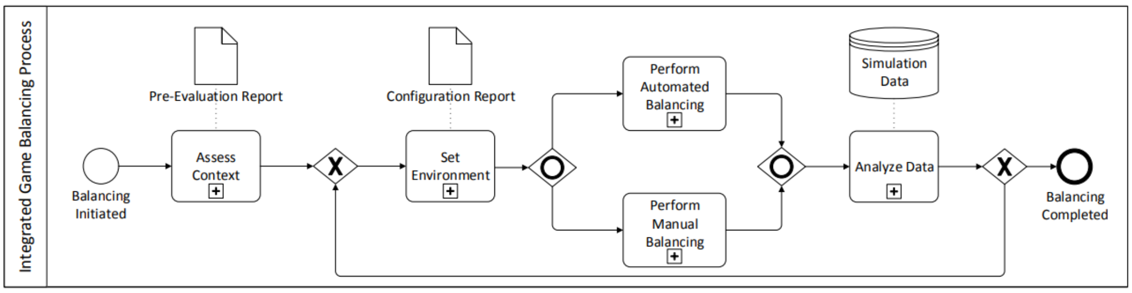


Figure 2.2: *The model representing the integrated game balancing process (Beyer et al. 2016)*

Beyer et al. evaluated the process on a tower defense game called "Zombie Village Game" by BlueByte GmbH. They apply several evaluators of fitness such as remaining health points and resources. This process overall proved promising. Automation was unsurprisingly faster than manual balancing. The automation provided valuable insights early on about the dynamics of the game parameters.

For future work, they propose to look into increasing the complexity of the evaluated game and using several fitness functions for several objectives of balancing. This could also be to allow for several player types and strategies. They also propose the direction of improving player AI agents to be more accurate (Beyer et al. 2016). This is assumed to discuss the accuracy of representing a human player.

This research was further developed in: **Integrated Balancing of an RTS Game: Case Study and Toolbox Refinement** In this paper, the process is applied to a more complex game, namely a clone of "Red Alert".

2.1.3 Dungeons & Replicants: Automated Game Balancing via Deep Player Behavior Modeling (Pfau et al. 2020)

Pfau et al. wrote a research paper on automating the game-balancing process using deep player behavior modeling. The aim of this approach here is to accurately represent human players with a neural network. This approach is a more viable solution as opposed to reinforcement learning strategies where the agent is not an accurate representation of a human player but is often vastly superior. This approach should also be able to more accurately represent the player population with several viable styles of play (Pfau et al. 2020). The team collected a dataset from an MMORPG called Aion of 213 players over 6 months and modeled a deep player behavior modeling (DPBM) agent for each of them (Pfau et al. 2020). In order to evaluate and measure balance in model simulations, Pfau et al consider 4 variables. A binary value of win or lose, the normalized duration of the encounter in time, and the percentage of health points remaining for both player and opponent (Pfau et al. 2020).

While this approach is promising modeling 213 players and spending 6 months on data collection is time and resources most cannot afford when developing a game. The approach however is very interesting after release, when such data can be widely available. While tuning the balance of the game is arguably better with the average players in mind, there are also advantages to account for the best possible strategies. The word of a strategy can quickly spread on the internet, and in this day and age, the potential of the individual players is as good as the common knowledge of the internet. Accounting for the best possible outcomes can help mitigate the balance that eventually skews when the best strategies are discovered. It should also be noted that the model is based on players with prior experience of the game and only applied to an already existing game where optimal play might already have been established.

The results were interesting and the team managed to discover an imbalance in the class design of Aion and regulate it using DPBM. Whether this balances the game can not be determined, as there are many factors of the game which was not represented in the model. It does however seems promising if tested on a larger slice of the game.

2.2 Applied Research

2.2.1 OpenAI and PPO

In 2017, OpenAI proposed proximal policy optimization (PPO) (OpenAI 2017) (Schulman et al. 2017) This algorithm has been responsible for the breakthroughs with reinforcement learning in video games and is the default at OpenAI. This is the algorithm used to train the agent famous for beating the best Dota 2 players in the world.(Berner et al. 2019) The algorithm performs better than the state-of-the-art approaches and is significantly simpler with regard to hyperparameter tuning and general implementation. The PPO algorithm and the field of reinforcement learning is further elaborated upon in section 3.1.

2.2.2 Square Enix at GDC 2010: Balancing Nightmares: An AI Approach to Balance Games with Overwhelming Amounts of Data

This section is based on the classic 2010 talk at GDC 2010 by Kazuko Manabe and Shigeru Awaji from Square Enix. (Kazuko Manabe 2010) The team at Square Enix used genetic algorithms to solve the balancing of an auto battler-type game called "Grimms Notes Repage". In this type of game, the player does not actively participate in battles. The player instead chooses the team, abilities, items, and upgrades to prepare the team for battle. The outcome of the battles is simulation-based. The team then runs battle simulations with balance states generated by a genetic algorithm to test different states. This allowed the team to balance a game with 10 to the power of 182 choices of equipment items for the battle. This helped them find game balance-breaking combinations of player options and is one of the few real-world examples of balance automation with machine learning, during development.

2.2.3 Bungie at GDC 2010: Changing the Time Between Shots for the Sniper Rifle from 0.5 to 0.7 Seconds for Halo 3

This section is based on the classic 2010 talk at GDC 2010 by Jaime Griesemer from Bungie (Jaime Griesemer 2010) Jaime provides a lot of arguments that balance is regarding a perceived state of flow. This talk is a testament to the huge workload this process can be and that a lot of these decisions are based on FEEL as opposed to direct measurable results. As he said in his talk about the broken sniper: "You're not gonna figure it out by looking at graphs or looking at data". This is a strong case against automation in some aspects of games and underlines the importance of keeping designers and players at the center of the design process. You have to be careful with accepting the results of automation. There are several ways bias can be introduced in these results. While automation may create a game that is flawlessly balanced, it might not necessarily be fun or feel great, which is arguably the most important aspect of, and the goal of games.

2.2.4 Modl.ai

Modl.ai[1] is a machine-learning platform that specializes in AI and ML bots for game testing. This is one of the current solutions attempting to help automate game testing and bots. The bots can be used both for testing and for playing. The bots can be used to fill in for players or to help smooth the launch of a multiplayer game lacking players. The most interesting part of this thesis is the testing automation. There are plugins for both Unreal Engine and Unity and an API for other development workflows. They are for instance used by the mobile games company King.

2.2.5 DeepMind

DeepMind is a British artificial intelligence (AI) research company that was acquired by Google in 2015. The company is known for developing cutting-edge AI technologies and algorithms, and it has made significant contributions to the field of machine learning.

DeepMind has worked on a variety of projects, including computer vision, natural language processing, and game-playing AI. One of the company's greater achievements was the development of AlphaGo, an AI system that became the first computer program to defeat a world champion in the board game "Go".

KataGo based on AlphaGo was recently defeated as a group of AI researchers discovered a flaw in its algorithm with an adversarial attack in 2022. (Wang et al. 2022) This is a testament to the imperfections that still reside in these machine learning models and flaws in the training that the developers were not aware of. Flaws in a model that defeats the best GO players alive.

[1]`https://modl.ai/`

Chapter 3

Analysis

The analysis chapter is focused on three central components of the project. A deep technical analysis of the development and training environments for machine learning in game development. Furthermore, the analysis will cover a section on the current game balance techniques used during development. Finally, the analysis will also cover a very brief overview of the game design research for the development of the game itself.

3.1 Technical Analysis

In order to evaluate the balance of the game, some gameplay data is required. This is often gathered through play testing with users or internal testing. This is expensive, time-consuming, and not sustainable during development. A playtest can be a means of evaluating the finished product and fine-tuning parameters once the game is already thoroughly balanced by developers. What if this game data can be artificially generated? This would be cheaper and faster, and if radical changes are made to the game, the data can be regenerated to fit the new revision. In order to generate this data, one proposed method is to train and use an agent instead of a human player. Once the agent is created there are several ways to evaluate different parameters and find the best game balance. For instance brute force testing of all possible scenarios (impossible), genetic algorithms, and decision trees. This thesis will focus on using genetic algorithms for this task.

3.1.1 Training: Deep Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning where an algorithm learns to make decisions by interacting with an environment. The algorithm receives feedback in the form of positive and negative rewards and observations regarding the state of the environment it operates within. Its goal is to learn the best set of actions to maximize the reward it receives.

To operate, an RL algorithm starts by observing the environment and outputting an action. The environment responds by providing feedback in the form of rewards and state observations, which the algorithm uses to update its behavior. The algorithm continues to interact with the environment, adjusting its behavior based on the feedback it receives, until it learns to make optimal decisions.

The goal of the RL algorithm is to maximize the total reward it receives over time. To achieve this, the algorithm explores through "trial-and-error," where it tries different actions and evaluates their outcomes. If the outcome is good, the algorithm will learn to take similar actions in similar situations in the future. If the outcome is bad, the algorithm will learn to avoid similar actions in similar situations in the future.

Overall, RL algorithms learn to make decisions through interaction with the environment, using a policy to guide their actions and feedback in the form of rewards or penalties to update their behavior. By learning through trial and error, the algorithm becomes better at making decisions, and over time, it can learn to make optimal decisions in complex environments. Deep reinforcement learning is a sub-field of reinforcement learning. Where traditional reinforcement learning relies on a direct decision process, Markov Decision Process (MDP), deep reinforcement learning relies on a policy, which is a set of rules that dictate the actions the algorithm should take in different situations. The policy can be represented in different ways, such as a decision tree, a neural network, or a set of rules. The algorithm tries different actions based on the policy. (Li 2017)

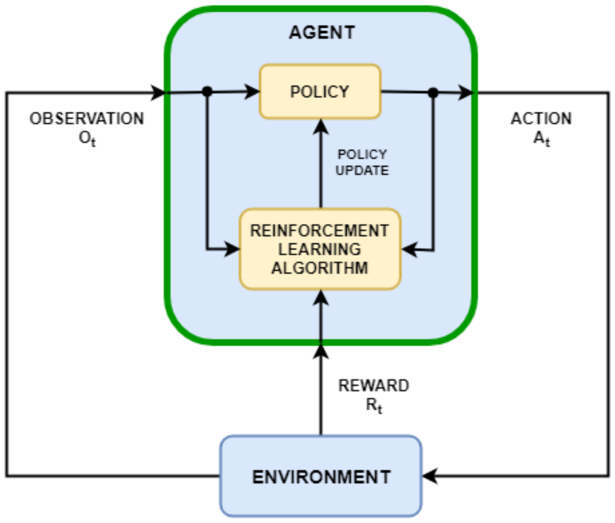


Figure 3.1: *Deep Reinforcement Learning Diagram*

ML-Agents is a framework developed by Unity for Unity, which allows Unity to interface with a Python environment for deep reinforcement learning with pyTorch and TensorFlow. ML-Agents allow for two deep reinforcement learning models, Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). Because of this, these are the two algorithms we will focus on for agent training in the next section on deep reinforcement learning. (Technologies 2023d)

Deep reinforcement learning (DRL) is a powerful approach for training agents to learn to interact with an environment through trial and error. One popular algorithm for DRL is Proximal Policy Optimization (PPO), which has shown impressive results on a wide range of tasks.

Proximal Policy Optimization (PPO)

PPO is a powerful and effective algorithm for reinforcement learning that has gained popularity in recent years due to its stability and efficiency. Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that updates the policy in an iterative manner to maximize the expected reward. PPO is a policy gradient algorithm, which means it directly optimizes the policy function to increase the expected reward. It was introduced by OpenAI in 2017 as a more stable and efficient alternative to other policy gradients algorithms like TRPO and ACKTR.

PPO works by using a surrogate objective function, which is a lower bound on the expected reward. The surrogate objective function is a clipped version of the original objective function, which helps to prevent large policy updates that could cause the policy to diverge. Specifically, PPO uses a clip function to ensure that the policy update is within a certain range of the previous policy, thereby limiting the magnitude of the update. (Schulman et al. 2017)

The PPO algorithm works in an iterative process with the following steps to update the policy:

1. **Batch collection**
   Collect a batch of experiences: The agent interacts with the environment by executing the current policy and collecting a batch of experiences. Each experience consists of the current state, the action taken, the resulting reward, and the next state. (Schulman et al. 2017)

2. **Advantage function**
   PPO estimates the advantage function, which measures how much better or worse the action taken was compared to the expected value. This is done by comparing the actual reward received with the expected reward estimated by the value function. (Schulman et al. 2017)

3. **Policy update**
   The policy is updated by maximizing the surrogate objective function, which is a clipped version of the original objective function. The objective function is typically the log-likelihood of the action taken multiplied by the advantage function. The clip function ensures that the policy update is within a certain range of the previous policy. (Schulman et al. 2017)

Action space

PPO is able to handle both discrete and continuous action spaces, making it a versatile algorithm for a wide range of tasks. (Technologies 2023d)

Policy

PPO is an on-policy algorithm. On-policy algorithms learn the optimal policy by interacting with the environment using the current policy. The data used to update the policy is generated using the same policy that is being updated. This can lead to more stable and consistent learning since the data used for training is more similar to the data that the policy will encounter during deployment. However, on-policy algorithms can be less sample efficient since they cannot use any old data from previous iterations of the policy. (Schulman et al. 2017)

Experience buffer

PPO does not use an experience buffer and samples directly from the batch of current experiences. SAC, for example, uses an experience buffer. This means that it can learn from prior experience samples, which makes it efficient when samples are scarce. (Technologies 2023d)

Sample efficiency

PPO is generally less sample efficient than SAC with fewer samples. SAC is very sample efficient, meaning it can learn from fewer samples than other algorithms, which is advantageous in an environment where samples are scarce. This is because SAC updates the policy and the value function using an experience buffer, which allows it to make better use of the data. (Technologies 2023d)

Speed

In terms of training speed, PPO is generally faster than for example SAC. This is because PPO updates the policy using a clipped objective function, which results in smaller policy updates and faster convergence.

Additionally, PPO can be more sample-efficient than SAC, meaning it can learn from fewer samples, which also contributes to its faster training speed. SAC can be slower than PPO due to the entropy regularization term in its objective function. This term encourages the policy to be more diverse, which can result in more exploration and slower convergence. However, SAC's ability to handle continuous action spaces can make it faster in certain tasks where discrete actions are not feasible. (Technologies 2023d)

3.1.2 ML-Agents PPO Hyperparameters

In order to optimize the training of the model it is important to evaluate the performance and to adjust the parameters of the model accordingly. This is commonly referred to as hyperparameter tuning. This section will serve as an explanation and analysis of the most important hyperparameters for a PPO model. (Technologies 2023d)

Batch Size, buffer size, and epochs

We collect a number of steps from the training (buffer size). We divide the buffer size into batches of the batch size and use these slices for training the model. The model is updated with these batches one at a time and repeats the process a number of times. How many times is defined by the epoch. The buffer size determines the number of learning steps to include when updating the policy. It can be important that this number is relative to the episode length of an experience and the number of parallel environments. This is because a buffer size that is too small won't include entire episodes in its policy update, but rather fractions of the episodes gathered from multiple parallel environments. On the other hand, a buffer size that is too large will require multiple episodes to update its policy and extra computational resources. (Technologies 2023d)

Learning Rate

The learning rate hyperparameter is a key parameter in machine learning algorithms that determines the step size taken by an optimizer when updating the weights of a neural network during the training process. The learning rate controls the speed at which the model learns from the training data, with a higher learning rate leading to faster learning but potentially unstable updates, and a lower learning rate leading to more stable updates but slower learning. (Technologies 2023d)

In neural networks, the weights of the model are updated during the training process by adjusting them in the direction of the negative gradient of the loss function with respect to the weights. The learning rate determines the size of the steps taken by the optimizer when adjusting the weights, with a higher learning rate resulting in larger steps and a lower learning rate resulting in smaller steps. (Technologies 2023d)

Choosing an appropriate learning rate is critical for the success of a machine learning algorithm. If the learning rate is too high, the model may not converge and the updates may be unstable, leading to poor performance on the training data. If the learning rate is too low, the model may converge too slowly or get stuck in a local minimum, also leading to poor performance. (Technologies 2023d)

Beta

In ML-Agents, the beta parameter is a hyperparameter used in the computation of the advantage function in reinforcement learning. The advantage function is used to estimate the advantage of taking a specific action in a given state, which is the difference between the expected cumulative reward of taking that action and the expected cumulative reward of taking the average action.(Technologies 2023d)

The beta parameter is used to control the amount of entropy regularization applied to the advantage function. Entropy regularization is a technique used in reinforcement learning to encourage exploration and prevent the agent from getting stuck in local optima by penalizing policies that are too deterministic or predictable.(Technologies 2023d)

The beta parameter in ML-Agents controls the strength of the entropy regularization applied to the advantage function, with a higher beta leading to more regularization and a greater emphasis on exploration, and a lower beta leading to less regularization and a greater emphasis on exploitation. (Technologies 2023d)

Epsilon

Epsilon is a hyperparameter in PPO that is used to clip the ratio of probabilities between the new and old policies. Specifically, the policy update is clipped by a factor of epsilon, ensuring that the update is not too large and does not significantly change the policy. Epsilon is usually set to a small value, such as 0.2 or 0.1, and is used to prevent the policy from changing too much between iterations. (Technologies 2023d)

Lambda

Lambda is another hyperparameter used in PPO that controls the trade-off between the bias and variance of the estimated advantage function. The advantage function is an estimate of how much better a certain action is compared to other actions in a given state. A high lambda value increases the bias of the advantage estimate, which makes the updates more stable but may result in a suboptimal policy. A low lambda value reduces bias but increases variance, which can make the updates more unstable but may result in a better policy. Typically, a lambda value of 0.95 is used in PPO. (Technologies 2023d)

Schedule

In PPO, the learning rate and other hyperparameters can be scheduled over time to improve training. For example, the learning rate can start high and then gradually decrease over time to allow the policy to converge to a better solution. Other hyperparameters, such as the entropy coefficient, can also be scheduled to change during training. The schedule can be fixed or adaptive, depending on the specific requirements of the problem being solved. (Technologies 2023d)

Time Horizon

The time horizon is a hyperparameter in PPO that determines how many timesteps are used to estimate the advantages and compute the policy updates. A longer time horizon can help capture more long-term dependencies and improve the stability of the updates but also increases the computational cost and memory requirements. Conversely, a shorter time horizon reduces the computational cost but may result in unstable updates and suboptimal policies. The time horizon is problem-dependent and can be set based on the length of the episodes or the specific requirements of the problem being solved. (Technologies 2023d)

3.1.3 Training Techniques

In addition to understanding the hyperparameter tuning parameters and process. It is also important to understand the way the agent is trained. Reinforcement learning employs several key training techniques to optimize agent performance. These techniques include value-based methods, where an agent learns to estimate the value of different actions or states, and policy-based methods, where the agent directly learns an optimal policy. Additionally, there are actor-critic methods that combine both value-based and policy-based approaches. Other techniques, such as exploration-exploitation strategies, replay buffers, and reward shaping, are utilized to improve learning efficiency and stability. These techniques facilitate the training process by enabling agents to iteratively improve their decision-making abilities based on rewards and experiences obtained from their interactions with the environment. But there are also approaches relating to how the training process itself is designed. The traditional approach with simple problems is generally to define hyperparameters, run the training, and then maybe refine hyperparameters as the agent stagnates in learning or if performance is low. This is generally the case for all types of training techniques. For more complex problems, however, it can be necessary to apply different techniques such as curriculum learning. (Li 2017)

Curriculum learning

Curriculum learning is a training technique used in reinforcement learning that involves gradually increasing the complexity of the training environment. The idea behind curriculum learning is to start with simple tasks and gradually increase the difficulty over time, allowing the agent to learn more complex behaviors. This can help the agent learn more efficiently and avoid getting stuck in suboptimal policies. Curriculum learning can be implemented in various ways, such as by increasing the difficulty of the environment, changing the reward function, or providing demonstrations. (Li 2017)

Self play

Self-play is a training technique used in reinforcement learning that involves having an agent play against itself to improve its performance. The idea behind self-play is that the agent can learn from its own mistakes and improve its performance over time. Self-play is commonly used in games, such as chess and Go, where the agent can play against itself to improve its strategies. Self-play can also be used in other applications, such as robotics and control, where the agent can learn from its own interactions with the environment. (Li 2017)

Imitation learning

Training a reinforcement learning model to handle simple tasks is an easy and efficient process, but as these tasks become more and more complex and comprehensive, so does the training of the model, at a certain point the chance of the model completing the task becomes infinitely low. This will often require training the model over multiple steps with separate rewards and observations to avoid this limitation. This is where imitation learning offers a real strength. Heuristic data can help the model learn to accomplish its task much more efficiently. (Li 2017)

Transfer learning

Transfer learning is a technique used in machine learning, including reinforcement learning, that involves transferring knowledge learned in one task to another related task. The idea behind transfer learning is that the knowledge learned in one task can be leveraged to improve the learning efficiency and performance of the agent on a related task. In reinforcement learning, transfer learning can be used to pre-train the agent on a related task or to transfer the learned policy or value function to a new task. Transfer learning can be useful in scenarios where the new task is similar to the original task, but may have different environmental conditions or goals. (Li 2017) (Torrey and Shavlik 2010)

Meta-learning

Meta-learning is a technique used in reinforcement learning that involves learning to learn and adapt to new tasks more efficiently. The idea behind meta-learning is to learn a set of initial conditions or hyperparameters that can be quickly adapted to new tasks or environments with minimal training. Meta-learning can be used to improve the learning efficiency and generalization performance of the agent, especially in scenarios where the agent needs to learn from a limited amount of data. Meta-learning can be implemented using various approaches, such as optimization-based methods or memory-based methods. (Vanschoren 2019)

Multi-task learning

Multi-task learning is a technique used in reinforcement learning that involves learning multiple related tasks simultaneously to improve the generalization performance of the agent. The idea behind multi-task learning is that the agent can learn common features or representations that can be shared across tasks, improving the overall learning efficiency and performance. Multi-task learning can be implemented using various approaches, such as using a shared network architecture or using a modular network architecture. Multi-task learning can be useful in scenarios where the agent needs to learn multiple related tasks or has limited data for each task. (Li 2017)

Evaluation

When analyzing the training in reinforcement learning it is important to monitor the learning progress and there are several graphs that can be useful to visualize this. Some common graphs include the learning curve, which shows the average reward or loss over time, and the entropy graph, which shows the entropy of the policy over time, which is an expression of how random the actions of the models are. The learning curve can be useful to see how the agent's performance improves over time, while the entropy graph can be used to monitor the exploration-exploitation trade-off of the policy. There are also other graphs such as loss and value loss that can be interesting to monitor how quickly the policy is changing. This will of course also depend on the settings in the hyperparameters and should help identify flaws and possible optimizations in the parameters and the state and reward functions.

3.1.4 Genetic Algorithms

Genetic algorithms (GA) are a class of computational optimization techniques that mimic the process of natural selection and evolution. They are particularly useful in solving optimization problems that involve large search spaces and multiple objectives.

The basic idea behind GA's is to create a population of candidate solutions, evaluate their fitness using a fitness function, and then select the best solutions to create new offspring through crossover and mutation operations. These offspring are then evaluated, and the process is repeated until a satisfactory solution is found.

GAs have been successfully applied in a wide range of optimization problems. One of the key advantages of GAs is their ability to find global optima in complex and non-convex search spaces.

There are several parameters that need to be carefully tuned when designing a GA, such as the population size, crossover and mutation rates, and selection method. The performance of a GA depends heavily on these parameters, as well as the fitness function used to evaluate the candidate solutions.

Although GAs have shown great promise in solving complex optimization problems, they also have some limitations. One major issue is that they can be computationally expensive, and require careful parameter tuning, especially when dealing with large populations and complex fitness functions. Additionally, GAs may get trapped in local optima, which can limit their ability to find the global optimum.

Crossover and mutation are essential operators in GAs, as they help to generate new candidate solutions and explore the search space. The balance between these operators and their respective probabilities is important for the success of the GA. Too much crossover can lead to premature convergence, while too much mutation can lead to a loss of good solutions. The specific implementation of these operators depends on the problem being solved and the specific GA being used. (Mirjalili and Mirjalili 2019)

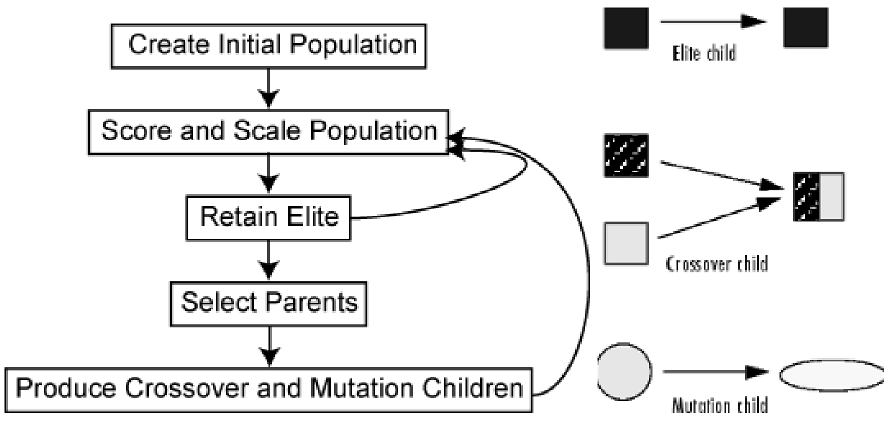

Create Initial Population
Score and Scale Population
Retain Elite
Select Parents
Produce Crossover and Mutation Children
Elite child
Crossover child
Mutation child

Figure 3.2: *Simple representation of the genetic algorithm (GA) with flowchart* (MathWorks 2023)

Crossover

Crossover is the process of taking two parent solutions from the current population and producing one or more offspring solutions by combining parts of each parent's genetic material. The goal of crossover is to create offspring that inherit the best traits of both parents and thus have a higher chance of being better than their parents. The process of crossover involves selecting a random crossover point along the chromosome (which represents the solution), and then exchanging genetic material between the parents on either side of that point. The resulting offspring inherit genetic material from both parents and are added to the population. (Mirjalili and Mirjalili 2019)

Mutation

Mutation is the process of randomly changing one or more genes in a gene. The goal of mutation is to introduce new genetic material into the population and explore regions of the search space that might not be reached through crossover alone. Mutation can help to prevent premature convergence of the GA by introducing diversity into the population. The process of mutation involves randomly selecting one or more genes in a chromosome and changing their values. The probability of mutation is typically set to a small value, as too much mutation can lead to the destruction of good solutions. (Mirjalili and Mirjalili 2019)

Fitness

The fitness function is a key component of genetic algorithms (GAs) that determines how well a gene solution performs for a particular problem. The fitness function assigns a numerical score to each candidate solution in the population, which indicates how well that solution solves the problem. The goal of the GA is to find the gene solution with the highest fitness score, as this solution is assumed to be the best solution to the problem. The fitness function will be tailored to the problem it solves and can be both a simple and very complex function to evaluate the solution. (Mirjalili and Mirjalili 2019)

3.2 Game Balance

3.2.1 Definition

In order to investigate game balance, the term itself must be defined. The definition of game balance is widely discussed and several different definitions and goals exist. One definition is as follows: "In game design, balance is the concept and the practice of tuning a game's rules, usually with the goal of preventing any of its component systems from being ineffective or otherwise undesirable when compared to their peers." This is the generally accepted definition that balance ensures a wide variety of strategies and general playability of the game. The goal is that all components of the game are valuable and that no strategy or component is vastly superior. Dan Felder, Game Designer at Blizzard Entertainment and Senior Game Designer at Electronic Arts defines balancing as an act that should help create a positive experience and remove broken gameplay. (Becker and Görlich 2020) (Felder 2015) Ultimately the game should be an enjoyable experience and this definition supports the general premise of balance that no strategy should be vastly superior. Marc Brown provides an interesting layer to the definition, that the perception of the game balance of a player is as important as the measurable game balance. (Becker and Görlich 2020) This is an interesting observation as some strategies like a rock-paper-scissors approach essentially create a fair and balanced approach, but do not necessarily create a feeling and perception of a balanced environment. At the same time, it is important that the players learn and understand how the game works in order to participate in a fair and balanced game. Without understanding, a game can feel impossible and unfair. In the paper "An Integrated Process for Game Balancing" Beyer et al. define game balance with more emphasis on the process itself: "Game balancing is the process of systematically modifying parameters of game components and operational rules in order to determine satisfactory configurations regarding predefined goals." (Beyer et al. 2016) This definition describes game balance as the process of adjusting the parameters and rules that constitute the game. Furthermore, the aspect of balancing is being viewed in relation to predefined goals. It is these predefined goals that are interesting. How we measure game balance is dependent on how these goals are defined. What constitutes a balanced game will always depend on the game itself and the vision of the designers. In most cases, perfect game balance is not achievable. However, imbalance in games can be used to the advantage of the designer in various ways. Perfect game balance might in fact not even be a desired result. A perfectly balanced game can also lead to a worse experience. If all options are equally good, there might be no reason to experiment with or even try other strategies (Becker and Görlich 2020). There can be a main goal in balancing which almost always relates to the wide definition of balance, but there can also exist sub-goals such as "pushing" and other strategies, defined in section 3.2.3.

3.2.2 Goals and Verification

The initial goal of balancing is described in the first definition proposed by Beyer et al. 3.2.1 In order to determine whether a game is balanced or not, it must be evaluated based on these predetermined goals. The overall goal is that no strategy is vastly superior. In order to achieve this goal sub-goals must be defined. A common sub-goal in multiplayer balancing could be that a hero or class should not be able to win every time. Another related sub-goal could be that no single component within this class should be able to win on its own. How can these goals be quantified? A common approach is to analyze the win rates of strategies. In a balanced setting, the average win rate should converge at 50%. This is however not always sufficient as will be discussed later in this chapter. An approach to the second goal could be to evaluate in play testing the damage outputs of each tower or the potential in modifiers. It can be difficult to evaluate crowd control (CC) effects like slowing opponents, but a potential solution could be to base it on average damage multiplied by the slowed percentage or the seconds of stun. However, setting up the parameters for evaluation is the lesser problem. Collecting the data and exploring the entire parameter space is a very tedious, time-consuming, and expensive process. Furthermore figuring out the initial parameters is also a very complicated topic, but there are strategies that can be applied to help approximate the optimal numbers. Lastly, it is important to raise the question of whether or not automating the process i desired. Balancing is also a design process and is a way the game is sculpted creatively. A goal of balancing can be to make a fun game which can be difficult to automate as it is a parameter of human perception.

3.2.3 Game Balancing Strategies

The power curve

Felder suggests that the first step to any game balancing is figuring out the power curve of the game. This means figuring out the formula or model for the power return on your investment resources. (Felder 2015) Game resources can for instance be energy, gold, or time. The relationship between the resource and measurable evaluators of power such as damage per second (DPS) can be modeled. (Felder 2015) By knowing the ratio of resource to power (for instance 1 gold = 1 damage per second) a baseline for the cost of components can be established. A helpful way to figure out the curve is by determining the ideal duration of a game session. At the 20-minute mark, a player should have enough resources to win the game. (Felder 2015) It is important that this curve is non-linear or the relative amount of resources and power level will not change throughout the game. It is important to note that this measure only gives an insight into components in isolation and the model will not be able to predict how the complex relationships between components affect the power level. (Felder 2015)

The Fermi solution and quick pointing

There are ways for game designers to better estimate these numbers. The Fermi solution suggests approximating the optimal results using an estimation chain. By doing several estimations to estimate a given relationship, the estimations often converge on the correct result as estimations are equally likely to over- and underestimate the correct figure (Felder 2015). The Fermi solution is the foundation of quick pointing which is the act of ranking components in a game, for instance, based on power levels. The averaged sum of these estimations will generally tend to approximate the correct ranking. The average sum of power can then be compared between different strategies and power levels. (Felder 2015)

Triple tapping

Triple tapping is a balancing approach that aims to drastically reduce the required number of game-balancing iterations. The approach aims to correctly identify the correct value in 3 steps. The initial value is determined by figuring out the highest and lowest numbers we think might work and using the average of these figures. In most cases, this value will still be incorrect if so triple tapping can be used to approximate it over 2 reevaluations. Triple tapping is the strategy of intentionally overshooting when reevaluating an unbalanced component. By overshooting the target value, we get an idea of both a minimum and maximum value and its effect on the game component. This gives us a much better chance of evaluating the correct result in fewer iterations of changing the value and evaluating the impact. (Felder 2015)

Safeguards

Safeguards are a way of dynamically adjusting the balance of the game. This is an approach that is difficult to generalize to all games. These are essentially ways the game and community will balance the game itself. This can for instance be by having components that are bought through bidding wars between players. Players themselves will then decide on the value of a component and in turn the game and players balance the game themselves. (Felder 2015) These could potentially be defensive options that can counter specific strategies ensuring that the potential of a strategy is finite.

Rock paper scissors

Some games like rock papers and Scissors are inherently balanced game if you ignore the psychological aspects that make some players excel. This method is popular in creating a balanced game by making strategically unbalanced matchups. This can be dangerous if win rates of specific matchups approximate 100% as a game can quickly actually start to resemble rock paper scissors and make people quit an unfavorable matchup. Therefore this approach is usually used to create slightly favorable matchups or rely on counter strategies to make unfavored matches winnable and thus playable. (Felder 2015)

Tier list

This approach to game balance describes the balance of a game divided into tier lists, where some strategies are stronger than others. This also relates to the strategy of emphasizing fun. By focusing on a few strategies that are generally better than others, the designers can focus on making sure that the viable options include a variety of different strategies and keep the game interesting. This is a way of structuring strategies in different tiers of viability, often with the S tier being the most viable and F tier the least viable. (Felder 2015)

Buffing and nerfing

When a root of a balance problem is discovered there are two main strategies to solve the issue. Nerfing describes the process of either decreasing the power level of a strategy or creating counter strategies or increasing existing counters. In turn, buffing describes the opposite effect of increasing the power level of a strategy or decreasing the power levels of counters (Becker and Görlich 2020).

Play testing

During development and especially in alpha or beta states of games play testing on a larger scale can be done. This is often done at a smaller scale of the studio itself or with friends and family. There are however also larger scale tests where the game is opened to the public or a select few, usually during the alpha or beta testing. This can significantly help discover problems with game balance and general issues like bugs.

3.2.4 Considerations

Competitive play

Sometimes you are dealt a bad hand, but if you are playing competitively, you must perform as best as possible. This means that not forfeiting an unfavorable match still can have an impact overall. For instance by influencing a match-making rating. Game balance does not only refer to the state of numbers but is also the perceived state of the player. Matchmaking of players is a crucial step to a balanced game environment. There are several strategies for competitive play to achieve this. A popular strategy is matchmaking rating and often in combination with tiers relating to a specific skill level. This can for instance be advancing from bronze through silver and gold. This makes sure the game is fun for everyone and balances across different skill levels maintaining fun and flow. This is often also applied to noncompetitive/ranked games. This is often done through a hidden matchmaking rating. Players have a rating that is hidden from them, making sure players of equal skill are matched. The matchmaking rating is often based on the number of games played and the win rate.

The meta is the term most commonly used to describe the best strategies used by the player base. The meta is a somewhat evolving state of the game strategies that somewhat help with game balance. As players discover how to counter strategies or develop strategies that beat the existing meta of popular strategies. (Becker and Görlich 2020) This can also work as a strategy to keep the game interesting by creating variation in the game.

By changing the meta and prioritizing certain strategies, the game is constantly evolving and keeps being fun and interesting.

Perceived balance

The balance of a game will always relate to the skill of the player. This does not mean that there is no objective balance in the game, but that the perceived balance depends on the player's skill level. There is also an interesting angle on this which is related to the flow state of the player. Optimizing the balance state or difficulty to fit the skill levels of the player, could make a game more fun and engaging. This is also a field that is researched extensively with real-time game balancing.

This is also another clear case where game data on its own do not serve as a full picture of balance. It will depend on the perception of the player and the state of flow experienced, which will vary from player to player. Becker touches on the verification of game balance requiring feedback and statistical data. A global win rate can be influenced by various factors like usage, skill levels, or strategies being more or less fun (Becker and Görlich 2020). Game balance can also be utilized to make fun strategies more viable. This strategy prioritizes certain strategies or components of the game that are the most fun. This might mean that some strategies become nonviable, but this is a design decision where you must consider if it is worthwhile to increase the incentive to play the most fun strategies and in return make some strategies less viable. This is one of those balance goals that can be difficult to validate with an automated process but can help shape balance around a strategy or a set of parameters determined by the designer.

Fairness and luck

Luck can create a variation to a game that makes the game less deterministic. It is important to consider that luck should never be the determining factor. The win rate of two equally skilled players should thus always approximate 50%. (Becker and Görlich 2020)

There will always exist unbalanced states in a game. It can for instance be a result and reward of a player playing well through a feedback loop. This must however be controlled by the designer, to best avoid checkmate situations. This is a situation where a player lost the game, but the game is not over. This is not fun or engaging for the losing player.

Pushing

"Pushing" in-game balance is the act of deliberately including unbalanced components in the game to teach the player about the game mechanics. An example is the magma rager in Blizzard Activation's Hearthstone. This card initially might look strong to a new player due to its cheap cost and high attack damage, but they will quickly realize the weakness of the card having only 1 health point. This card is therefore terrible and virtually unplayed among experienced players. However, by learning the basic dynamics of the game a new player will perform better and this understanding will lead to a higher perceived balance of the game.

3.2.5 Game Balance Patterns

Achieving balance in a game is a complex process, and game designers must consider many factors, including player feedback, data analysis, and testing. Game balance patterns can provide a starting point for designers and can be adapted and combined to suit the specific needs of each game. (Becker and Görlich 2020)

This section will cover some of the most common game balance patterns. These are strategies and design principles used to achieve balance in games. These patterns can be found in various game genres, from strategy games to fighting games, and can be applied to both single-player and multiplayer games.

Some of the most common patterns include:

1. **Rock-paper-scissors**
   This pattern involves creating a set of mechanics or abilities where each one is stronger against one type and weaker against another. This pattern can be seen in games such as Pokémon, where different types of Pokémon are strong or weak against each other.
2. **Time-to-kill balance**
   This pattern focuses on balancing the time it takes to defeat a player or enemy. This pattern is commonly used in first-person shooters, where weapons are balanced based on their rate of fire, damage, and accuracy.
3. **Resource balance**
   This pattern involves balancing the availability and use of resources in a game. This pattern is commonly used in strategy games, where players must manage resources such as food, gold, and wood to build structures and units.
4. **Progression balance**
   This pattern involves balancing the progression of players through the game, ensuring that they do not become too powerful or weak as the game progresses. This pattern can be seen in role-playing games, where players gain experience points and level up to unlock new abilities and equipment.
5. **Risk-reward balance**
   This pattern involves balancing the risk and reward of different actions in the game. This pattern can be seen in games such as poker, where players must decide whether to take risks for a higher reward or play it safe.

Determinism

Deterministic games are games in which the outcome is entirely determined by the actions of the players. In these games, the player's actions are known, and the game's outcome is entirely predictable. The balance in deterministic games is typically achieved by ensuring that the game mechanics are fair and that no one strategy dominates the game. In these games, balance is typically achieved by designing a game that has a range of viable strategies that are all equally likely to succeed.

Non-deterministic games, on the other hand, are games in which the outcome is partially determined by chance or random events. In these games, the player's actions are not always known, and the outcome of the game is not always predictable. The balance in non-deterministic games is achieved by ensuring that the game mechanics are designed in such a way that chance events are balanced and that players have an equal opportunity to succeed, regardless of their luck.

Game balance is important in both deterministic and non-deterministic games, as it helps to ensure that the game is enjoyable and fair for all players. In deterministic games, balance is achieved by designing a game with a range of viable strategies, while in non-deterministic games, balance is achieved by ensuring that chance events are balanced and that all players have an equal opportunity to succeed.

Symmetry

Symmetrical and asymmetrical games are two types of games that have different design requirements and balance considerations. An asymmetrical game is one where all players have access to the same resources and abilities, and the game mechanics are identical for all players. In contrast, an asymmetrical game is one where players have different resources, abilities, or game mechanics, creating different play experiences and challenges.

Symmetrical games are often used in competitive settings, such as esports or board game tournaments, where players are expected to rely on their skills and strategies to win. The balance in symmetrical games is achieved by ensuring that all players have access to the same resources and abilities, with no player having an inherent advantage over others. This balance allows players to rely on their skills and strategies rather than on the game mechanics to win.

Asymmetrical games are designed to provide players with different experiences, challenges, and playstyles. Asymmetrical games can be cooperative, competitive, or a mix of both, and they often require players to adopt different roles or playstyles to succeed. The balance in asymmetrical games is achieved by ensuring that the different player roles are equally challenging and rewarding and that each player's actions have a meaningful impact on the game's outcome.

Balancing asymmetrical games can be challenging, as it requires game designers to carefully consider the impact of different player roles and abilities on the game's mechanics and balance. If one player has a significantly more powerful ability or resource than others, the game can quickly become unbalanced and unfair. Therefore, it is essential to ensure that each player's abilities and resources are balanced and complement each other, rather than creating an imbalance.

In conclusion, symmetrical and asymmetrical games have different design requirements and balance considerations. Symmetrical games rely on equal access to resources and abilities to ensure fairness and balance, while asymmetrical games provide different experiences and challenges for players, requiring careful consideration of the impact of different player roles and abilities on the game's balance. Achieving balance in both types of games is critical to ensure that the game is enjoyable and fair for all players.

When designing a game balance automation workflow it is important to consider the type of game and in extension symmetry of the design. A symmetrical design will require a different strategy than an asymmetrical one. It is likely that an asymmetrical game will benefit more from self-play as this can train multiple agents simultaneously. This however always depends on the game itself and there is no one fits all rules.

Complexity

Complexity is an essential aspect of game design, as it can influence the game's depth, replayability, and player engagement. Complexity can be categorized into two broad categories: discrete and continuous complexity. (Becker and Görlich 2020)

Discrete complexity refers to games that have a limited number of actions, outcomes, or variables. These games often have clear rules and defined strategies, making them easy to understand and analyze. Discrete complexity can be found in many classic games, such as Chess or Go, where the number of pieces and moves is limited, but the strategies can be deep and complex.(Becker and Görlich 2020)

Continuous complexity refers to games that have a large number of actions, outcomes, or variables, making them harder to analyze and understand. These games often have a more significant level of unpredictability, making them more challenging to balance and design. Continuous complexity can be found in games such as simulation or strategy games, where there are multiple factors to consider, and the outcome can depend on numerous variables and interactions.(Becker and Görlich 2020)

Both discrete and continuous complexity have their advantages and disadvantages in game design. Discrete complexity can make games more accessible and easy to learn, making them ideal for casual players or for games that require quick decision-making. However, discrete complexity can limit the game's depth and replayability, as players may quickly master the game's mechanics.(Becker and Görlich 2020)

Continuous complexity can provide players with a more significant level of depth and replayability, as the game's outcome can vary depending on multiple variables and interactions. This complexity can make the game more challenging and engaging, but it can also make the game harder to balance and design.(Becker and Görlich 2020)

Achieving the right level of complexity in a game requires careful consideration of the game's mechanics, player interactions, and goals. Game designers must balance the game's complexity to ensure that it is challenging but not overwhelming and that it provides a rewarding and engaging experience for players. (Becker and Görlich 2020)

3.3 Game Design

3.3.1 Real-time Strategy (RTS) Genre

Real-time strategy (RTS) is a genre of video games that involve players managing resources, building structures, and controlling units in real time to achieve objectives. In an RTS game, players typically start with a small base or set of units and must gather resources such as gold, minerals, gas, or food, which can be used to build structures, produce units, and research new technologies.

The objective of an RTS game can vary, but it usually involves defeating the opposing player or faction. Players must balance their resource management with building an army, upgrading units and structures, and exploring the map. They must also make strategic decisions about when to attack, when to defend, and when to retreat.

RTS games are often played from a top-down perspective, giving players a bird's eye view of the battlefield.

Some well-known examples of RTS games include Warcraft, Starcraft, and the Age of Empires games.

Tower defense genre

Tower defense is a subgenre of real-time strategy video games in which players must defend a territory or base against waves of enemy attacks by building and upgrading defensive structures called towers. The objective is to survive all, or as many of the waves of enemy attacks without letting them through the defenses. The player will lose lives and/or gold when an enemy gets through. When enough minions breach the defenses, the player loses.

Players can build different types of towers with various abilities. Towers are placed strategically along the path that enemies will take toward the player's objective. As enemies approach, the towers will automatically attack them, inflicting damage, crowd control abilities, and slowing their progress.

Tower defense games often feature different types of enemies with varying strengths and weaknesses, forcing players to adapt their strategies to each new wave of attacks. They may also include boss battles, special abilities or power-ups, and other features to keep the gameplay engaging and challenging.

Some well-known examples of tower defense games include Plants vs. Zombies and Bloons Tower Defense.

Chapter 4

Final Problem Statement

In the early stages of the research, several different directions from the initial problem statement were considered. This was mainly concerning approaches regarding data of human player recordings and alternative algorithms. In the end, deep reinforcement learning was proposed as the most optimal way to solve the problem. Especially with the added benefit of imitation learning to speed up the training process. While a game is in development, the game is often run several hundreds of times during development. With imitation learning, we can avoid spending hours simplifying the game into smaller more easily managed scenarios that the reinforcement learning model can use for training and then spend several hours training the model for each step. Why not use this run time and free heuristic data to help train the model? This approach could also potentially assist the AI in imitating human players, as the data it works with will be from real players. There is a potential bias here in that the data it uses comes from developers and there will need to be some consideration as to which test runs will benefit the model. Sometimes a developer might want to create a certain outcome in the game, and this data might not be helpful for the model and should be discarded.

Genetic algorithms should work better than for instance trees by approximating the optimal solution. Since we are doing tests of the balance parameters. It is important that the algorithm can find a good solution in a relatively short time. A GA can find an approximation of the best balance state and should relatively quickly be able to find a good solution. By indirectly applying the principles of the triple tapping approach 3.2.3 to game balance ranges we can further reduce the search space for the algorithm. Game balance is a widely defined term and for good reason. There are several different approaches and desired outcomes to the process. Moving forward the desired outcome of the game-balancing process is defined as an equal win rate between the two factions. This was done to pursue the most obvious and simple game balance criteria to avoid further complications of the process and introduce potential bias. The initial evaluation plan is to follow the defined process model 6.3 and to conduct a test to evaluate the resulting game balance state and whether the goal as defined by the fitness function will translate to real players. This leads to the following final problem statement.

Final Problem Statement

"How can we optimize the game balance of a multiplayer real-time strategy game during development using reinforcement learning and genetic algorithms?"

Chapter 5

Methods

5.1 Procedure

The project was conducted as an exploratory and experimental case study. This means exploring relatively uncharted academic territory and processes within the game balance and machine learning topic.

The planned procedure was to follow an agile approach, with user evaluations of usability and following balance evaluations, however, due to scrapping user evaluations the process ended up following a waterfall approach. This was due to the heavy workload of implementation, which did not require user evaluation. Instead, the methodology was proposed and evaluated with user interviews and validation of the resulting proposed game balance. There are many aspects of the project to improve and optimize. This would be the focus of the next iterations, to improve upon the process, both from the discoveries of this study and especially with feedback in mind from game developers and studios. There were of course many iterations within the development and evaluation on the technical side of things. Iterations of hyperparameters, reward, state, and action functions. Iterations of game design and development.

5.2 Project Management

The project was managed using an agile approach with Scrum. This includes several backlogs related to design and development as well as the iterations of model training and tuning. A Gannt chart was used as a backlog to keep track of general progress, sprints, and timeline. This was created following the functional requirements listed in the design section, section 6.1. The workflow consisted primarily of sprints of one or two weeks in duration. This was supported by weekly supervision meetings.

The game was developed as a vertical slice, including all functionality to play a slice of the game. This mainly concerns limiting the implementation to two classes with a limited range of possible towers and minions to select from. This also concerns the possible minions and bosses encountered in the waves sent to each player. The implementation was managed with git using gitHub. The RL workflow was also supported using TensorBoard for evaluation, and TeamViewer for remote control of the computer running the training and simulations. Game description in section 6.2;

It was the plan to code the game following the Hungarian notation. This however was not really executed. Code comments were also very scarce, which is not ideal. The code structure mainly consists of regular scripts and classes, with several singleton structures.

5.3 Validity and Reliability

The method used to explore this process is highly reproducible and reliable. The approach and results are covered in detail. However, when applying the methodology to other games the results will be very different due to the nature of machine learning. The process is lacking validity being focused on this specific game, but the exploratory approach aims to discuss the generalizability and to validate it through an interview. This interview explores how the process can translate to a different random game and genre to understand the implications. Unfortunately, the quality of the model was too low to produce valid results and introduced a huge bias. One problem with the reliability of the method is that the evaluation criteria and RL functions will change for different games and different problems. One way to solve this could be to run simple balance tasks with self-play where reward functions, for instance, won't become a confounding variable between studies. In the case of evaluating the methods for multiple games. There is a lot of bias conflicting with the validity.

Applying the proposed workflow to a game in development is of course prone to bias. This subjective research approach is therefore assisted by a qualitative interview with a game designer working in the industry to gain a qualified third-person perspective on the workflow. This also provides some information regarding the generalizability of the workflow in terms of other genres, game development pipelines, and developers. The subjective measures of testing the workflow were however very valuable in getting the initial understanding of the field. In future research getting objective measures of applying the workflow in game studios would of course be ideal.

It was the plan to conduct a test with real players to assess the validity of the method. To accomplish this several design goals were established in the requirements for the design, see section 6.1. However, since this test was not carried out, these requirements could have been left out. These requirements include database integration and game polish. Multiplayer was essential for understanding the process of experiencing the workflow around a networked game as well as single-player components which contribute to the game. In this way, the development provided insights into single and multi-player development.

It was also planned to conduct a usability test with an emphasis on learnability prior to the test mentioned above. This would help validate the learnability of the game, as defined as a non-functional requirement in the design requirements, section 6.1 The intent was to gain insights into how well players understood the game to eliminate this as a confounding variable. However, as this final user evaluation test was discarded, so was this initial evaluation of usability.

5.4 Data Collection

The process was partly quantitative, collecting and documenting the process with training data. However, the evaluation was also conducted in a qualitative manner through interviews. The interviews were conducted with a semi-structured approach. This was chosen because of the exploratory nature of the evaluation and the freedom to ask follow-up questions and talk about the topics most relevant to this specific industry professional. Another reason was the fact that only a single or a few interviews would be conducted, making coding of results and any quantitative features of the data redundant.

The target group of the interviews was industry professionals in game studios. Anywhere from programmers, to level designers and producers. This was mainly to get an insight into a game studio and its development pipeline regarding game balance. This also provided a random sample of a game studio and games that the process could be reviewed for. These were acquired with purposive sampling through LinkedIn and through the network of project supervisor Henrique.

Chapter 6

Design

6.1 Requirements

The requirements for the solution are defined below, categorized into functional and non-functional specifications. The functional requirements relate to the fundamental functionality and hard specifications of the system. The non-functional describe the soft specifications required to optimize the solution for the case.

Functional Requirements

1. **Game engine**
   The game should be developed using the Unity engine to take advantage of the ML-Agents library for reinforcement learning. 3.1
2. **Multiplayer**
   The game needs to support two players. This requires networking and matchmaking/lobby implementation.1.1
3. **Platform**
   The game must as a minimum be playable on Windows. Other platforms such as Mac, Android and iOS are second priority.
4. **Game balance class**
   All variables related to the balance of the game should be stored in a single class. The balance state of the game is quickly and easily modified and can be accessed outside the Unity environment with JSON if necessary.2.1.1
5. **Factions**
   The game should support minimum two factions, in order to evaluate the faction balance in a multiplayer scenario.
6. **Database**
   The game require the implementation of a database in order to store the data of play sessions for evaluation. (Not implemented)
7. **Python Environment**
   In order to work with the Unity and ML-Agents environment, a python virtual environment must be created with the necessary dependencies and GPU CUDA support, to speed up training times and capacity for parallel training environments. This also includes a connection to TensorBoard for reviewing training data. 3.1
8. **Game Design**
   There are several specifications related to the design of the game that must be accomplished: Matchmaking lobby, Class selection, Tower building, Minion sending, and Tower selling. This also includes game UI that accomplishes the above tasks as well as showing the game states such as gold, HP, Wave

duration/time, and minion/tower stats and costs.

9. **Genetic Algorithm (GA) and interface**
   Implementation of a genetic algorithm (GA) that interfaces with the game balance class, either within Unity or through JSON from an outside environment, and reads the resulting data from simulations such as win/loss, gold, and HP. This should likely run in the application in order to take advantage of ML-Agents to decrease the runtime of simulations. 3.1
10. **Offline mode**
    It is required for machine learning training to run the game without executing net code. This requires an implementation that can work both online and offline.

Non-functional Requirements

1. **Learnability** The game must be fairly simple and easy to learn, in order to reduce possible bias in results. This has the added benefit of an environment of reduced complexity for reinforcement learning and genetic algorithms to operate within.
2. **Duration** The duration of the game should be as short as possible and last no longer than 20 minutes.
3. **Balance** The goals of the balancing process should be clearly defined and serve as the foundation of an evaluation of the process.
4. **Polish** The state of the game must be fairly polished including art style, 3D models, animations, and audio in order to gain player interest for evaluation purposes. A better-looking game will be more likely of attracting players for testing.

6.2 Game Design

The game created is a multiplayer cross-platform game. It is a classic tower defense game in the real-time strategy genre. This game is inspired by the maul wars mods created for Blizzard's Warcraft III title.

The main objective of the game is to survive waves of incoming minions. These minions are both spawned on a regular basis, but a player can also send minions to the opponent to try and win the game. The players defend off the minions with towers they create in their base. Players create a path or a maze with their towers in order to control the way the minions move through the battlefield. As opposed to traditional tower defense games where towers are built along an already-defined path.

The towers can be placed in a hexagonal grid covering each player base. There are two available factions. Dragonmancer and Elementalist. Both factions have a shared tower unit, the archer, and 3 other unique towers. They also have 3 unique minions they can send to the opponent. Each player starts the game with the same amount of gold. Gold is used to buy towers (defense) or minions (offense). Each player also starts with a health pool. When a player loses all of their lives, due to minions breaking through the defense, they lose the game. Players will receive gold when killing units with their towers.

Each tower and minion have a different set of attributes making it excel at different scenarios. Some towers have an area of effect damage allowing them to hit multiple towers at once. Other towers will slow down units or do no damage but provide additional gold income.

It is important for interesting game design to have variation and this is often accomplished through the element of chance. This can for instance be a random variation in damage. This however was not implemented. There is a good chance that this will influence the complexity of the machine learning problem. For this reason, it can be argued that this is an important game design element to include. It was not included in this implementation.

The initial balance of the game will be based on some of the balance strategies defined in section 3.2.3. The initial balance of the game would

6.3 Graphic Design

A user interface was created to interact with the lobby and game. The user interface includes 7 main layouts. 6 of the layouts are included in figure 6.2. This includes a general home screen menu (1). When the player presses play, they will enter the menu for selecting the class (2). This menu includes a back button, the class options you can select, and a play button. There is also a test button for testing purposes. On clicking play the user is directed to the matchmaking screen (3). The user will be presented with messages regarding loading and finding opponents, once a match is made the user is directed to the game. The game UI (4) consists of 4 buttons. The tower button to open the tower select menu (5), the minion button for the minion select button, and two buttons for jumping the camera to the enemy base and player base view. There are also 4 information widgets showing health, gold, wave number, and wave timer. The select tower menu (5) consists of a scroll view of the available towers. This should also include stats like gold cost, damage, and range, but this has not yet been implemented. There is a similar menu for the minion button. Lastly, we have the settings menu which is just a simple drop-down menu to choose between graphics performance presets.

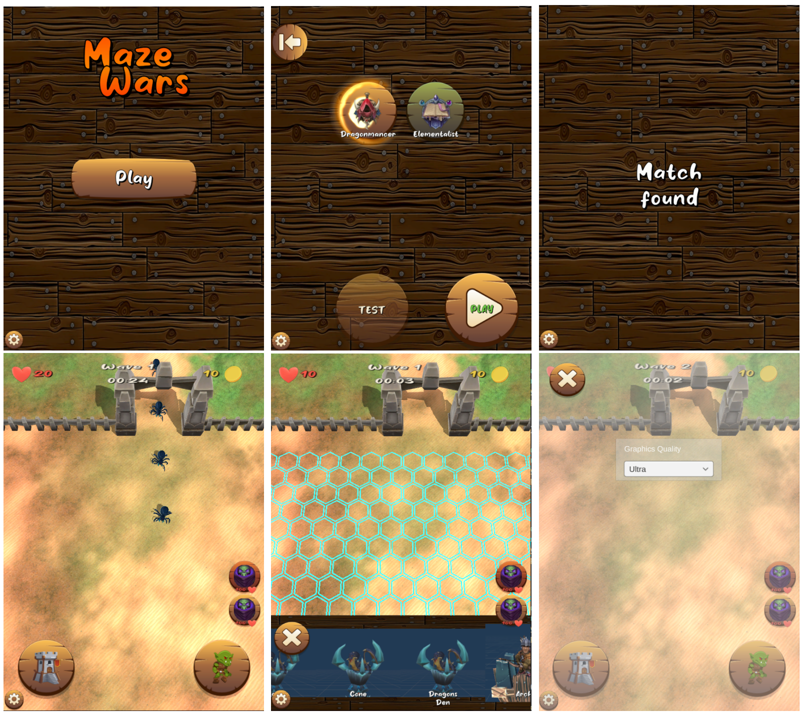


Figure 6.1: *(1) Home screen, (2) Play > Select class, (3) Finding match screen, (4) In-game UI, (5) Place tower menu, (6) Settings menu*

6.4 Game Art

The game art is kept in the stylized design. The design was created using third-party assets including 3D models and animations from stores such as the Unity asset store, Humblebundle and Turbosquid. The stylized design style is great for creating mobile games as it is lightweight both in terms of expression but also for performance. Since the style is low in polygons. Several shaders were acquired from the Unity asset store as well. Most images were also acquired from third-party sites such as Pngtree. The ground was created using the Unity terrain editor with several stylized materials. Everything is animated with Unity's animation toolkit and shaded using a stylized toon shader or the AllInOneVfx library.

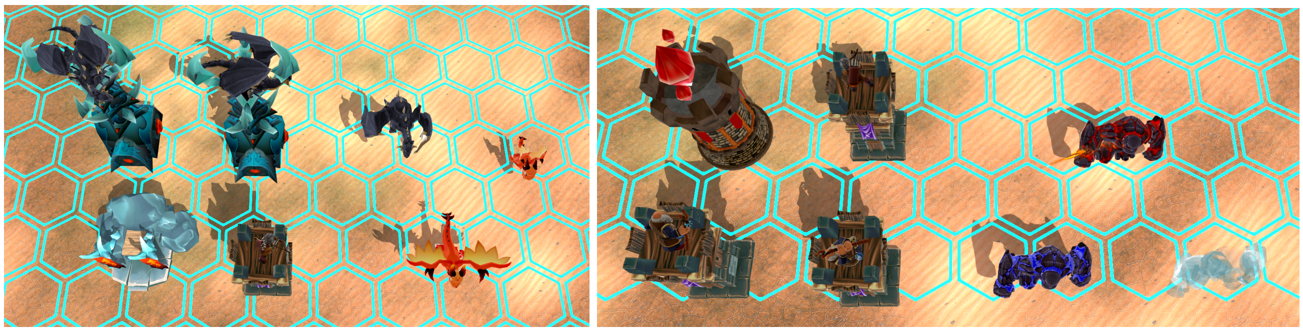


Figure 6.2: *(1) Dragonmancer Class Art, (2) Elementalist Class Art*

6.5 Process Design

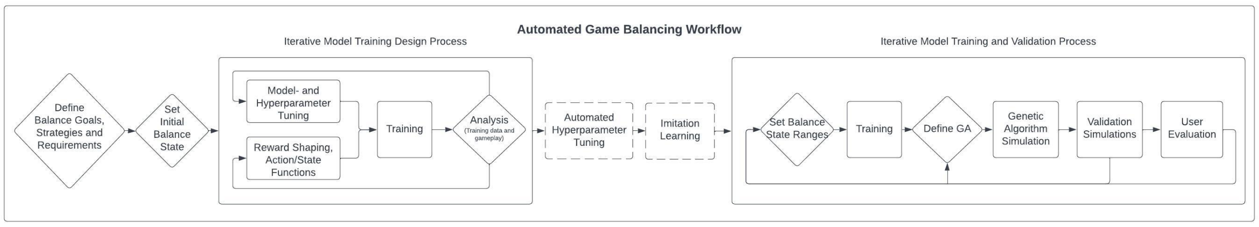

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 6.3: *Automated Game Balancing Workflow*

The process of the workflow will be described in this section and is visualized in figure 6.3. The model primarily consists of two iterative workflows. An initial iterative design process of the model with emphasis on hyperparameter tuning, reward shaping and state/action functions. The second iterative process relates to training the model for a wide variety of balance states to learn the various balance parameters. This is coupled with several evaluation processes to validate the model.

The process consists of 13 steps:

1. **Define Balance Goals, Strategies, and Requirements**
   Before beginning the design of the model it is important to establish the goals of the balancing process and the strategy to accomplish this. The goal should be supported by requirements or sub-goals as described in section 3.2.2. It is important to consider what variables are being changed and to make sure the implementation can interface with these balance variables.
2. **Set Initial Balance State**
   It is up to the developer to define the initial state of the balance when training the early version of the model, these parameters will be optimized later but is very important that the model is based on sensible parameters from the beginning. Otherwise, the agent will be unable to learn to navigate the environment. It is a good idea to apply some of the balancing strategies from section 3.2.3 to reach a good starting point in very few iterations. This depends on the problem and intuition of the designer.

3. **Model- and hyperparameter tuning**
Step 2 through 4 is part of an iterative design process, where the reinforcement learning model is designed and optimized. In this step, the model and hyperparameters are changed to best fit the problem the method is solving. Is it a PPO or SAC model, or something different? What are the appropriate hyperparameters like batch size and learning rate? This process will also vary depending on the problem. For more complex problems it can be required to train the model with
4. **Reward and state shaping**
This step reflects upon the design of the reward system and how the model sees the state. Is it given enough information to understand the game or too much, can it be simplified? In the case of strict training with self-play, this step is in most cases omitted.
5. **Analysis**
Training is run with new parameters of the model or a new reward system or state observations. The results are reflected with graphs regarding rewards, episode length, entropy, etc, or ELO in the case of self-play. In some cases actual inference of the model should be analyzed visually, to better estimate the actual gameplay performance of the model.
6. **Automated Hyperparameter Tuning**
An optional step is to run automated hyperparameter tuning. This can run iterations of different hyperparameter settings in order to estimate the best hyperparameters to solve the problem. This is a lengthy process, but can likely be worthwhile for some applications, to avoid the long and tedious process of manually optimizing the parameters. This has not been explored during this thesis but is likely a good place to optimize, depending on the problem that is being solved.
7. **Imitation Learning**
Once the hyperparameters, actions and observation vectors, and reward functions are locked in. Imitation learning can optionally be used to speed up the training process. This is estimated to roughly increase the speed by a factor of 4. (Arts 2023)
8. **Set Balance State Ranges**
This is the first step in the next iterative process of training and applying the RL model with the genetic algorithm. Once the model is trained and reward systems and hyperparameters look promising, the model training should be continued using varying states of game balance within the ranges defined in this step. This ensures that the model understands the differences in game balance states. For instance, if a tower deals 10 damage or 100 damage. To limit the complexity of this problem it is up to the designer to specify the limits of the balance state. This can for instance be to lock a variable to a set value or to enable values between 1 and 2 or 1 and 100. This is where the designer sets the limits of the state and influences the final design.
9. **Training**
The model is trained using varying data. Randomization of states can be accomplished using the genetic algorithm or a simple randomization algorithm. The model is trained until it is stagnant over various states of game balance.
10. **Define Genetic Algorithm**
Define the fitness function and the parameters to optimize the genetic algorithm (elitism, mutation, population size, etc.)
11. **Genetic Algorithm Simulation**
Once the model has an understanding of the game states, the genetic algorithm can be used to find the optimal solution. The simulations are run using a fitness function describing the optimal game balance. For instance the win rate over x amount of games. The best solution is logged and validated.
12. **Validation Simulations**
Once the best solution has been found, it is validated with a large number of simulations. This will validate whether the balance values are truly indicative of the desired optimal fitness, which in this case is the win rate.

13. **User Evaluation**
    The final step in the process is to validate the result with human players. This both validates if the game balance is valid in the context of real human players and if the results are fit for further use during development.

Chapter 7

Implementation

This section covers the various development environments used to implement the game and balancing process as well as the development and integration of the game and machine learning environments.

7.1 Environments

7.1.1 Game Development Environment

Game engine: Unity

The game engine used to create the game is Unity version 2021.3.12f1. The game is created using the Universal Render Pipeline (URP) which is great for both pc and mobile games. Unity uses the C# coding language internally. The engine also offers a reinforcement learning library that integrates with a Python environment. (Technologies 2023b)

3D Software: Blender

Most 3D objects will be outsourced from places like Humblebundle, Unity asset store and Turbosquid. Some models will work great out of the box, but some require 3D work. The 3D software suite used for this will be the open-source suite Blender. Blender is free and open source and provides all the tools needed to create and edit 3D files. (Roosendaal 2023)

Pathfinding library: A* Pathfinding Project Pro

The A* Pathfinding Project is an extensive pathfinding library for the Unity game engine. It uses the A* algorithm and multithreading to provide optimized pathfinding. It provides several tools and different approaches like navmeshes, grid graphs, and point graphs. It also supports dynamic obstacles and navmesh cutting. It comes both as a free version and a pro version. This project will use the pro version for all pathfinding needs. (Granberg 2023)

Networking library: Netcode for Gameobjects

"Netcode for Gameobjects" is Unity’s new mid-level networking library created specifically for the Unity game engine. It is the official upgrade to the deprecated Unet network. That was Unity’s previous library, which was unofficially replaced by Mirror by 3rd party developers. The library is very well documented and also built to interface well with Unity’s new gaming services "Unity Gaming Services" (UGS) which among many other things provides matchmaking service and api. This library supports both a host/listen server model

(P2P) and a dedicated game server (DGS) model. DGS would be the desired model to better handle cheating. However, DGS is expensive and P2P is virtually free and easy to scale as most computation is handled on the client's system and not on a hosted server. This library will be used to build the net code for the game with a P2P model.(Technologies 2023a)

Lobby/Relay library: Unity Gaming Services

Unity Gaming Services (UGS) is a set of tools provided by Unity for multiplayer solutions, monetization, and much more. This library will be used to create the game backend. This utilizes the lobby and relay services to create lobbies and connect players securely together with a peer-to-peer model. (Technologies 2023c)

Unity ML-Agents

The ML-Agents library for Unity allows for deep reinforcement learning algorithms (PPO, SAC, MA-POCA, self-play). It allows for training, imitations (heuristics), and evaluating (inference) within the editor. Imitation learning is supported by GAIL and BC learning algorithms. The library also allows for running training in executables exported from the editor. The library uses PyTorch and Tensorflow for models and training, which can be accessed through Python in the command prompt terminal. It supports training single-agent, multi-agent cooperative, and multi-agent competitive scenarios. (Technologies 2023d)

Grid

In order to facilitate tower placement and maze building, a simple hexagonal grid was created. Most of the code to generate this grid was provided by GitHub user 'Sunny Valley Studio'. [1]

Additional libraries

Alongside the above-listed primary applications, libraries, and frameworks there are some smaller additional libraries that will be used frequently throughout the game implementation. 'Feel' is a library for quickly creating small movements and FX of UI and game objects. 'All in 1 Vfx Toolkit' is a toolkit including several shaders for creating visual effects in Unity. This library will be the primary asset for effect shaders in the game. These libraries are both available in the Unity Asset Store.

7.1.2 Machine Learning Environment

Python

Python is used in order to run reinforcement learning models and neural networks. There are several Python libraries and their dependencies used here including pip, numpy, and more. Some of the more important libraries are described below.

ML-Agents

ML-Agents (Machine Learning Agents) is an open-source toolkit developed by Unity Technologies for implementing and training machine learning models within the Unity game engine. This powerful toolkit allows developers to create intelligent agents that can learn to interact with and navigate within complex environments, making it ideal for building realistic and immersive games.

[1] `https://github.com/SunnyValleyStudio`

The ML-Agents toolkit provides a set of tools and workflows for creating, training and evaluating intelligent agents in Unity environments. The toolkit includes a Python API for interacting with Unity, as well as a number of pre-built example environments and models that can be used as starting points for custom projects.

One of the key features of ML-Agents is its support for deep reinforcement learning (RL) algorithms, which are ideal for training agents to learn complex behaviors through trial-and-error interactions with their environment. These algorithms use neural networks to approximate the optimal policy for a given task, based on the agent's observations and rewards.

Another important feature of ML-Agents is its support for multiple agents within the same environment. This allows developers to create complex scenarios where agents must cooperate or compete with each other to achieve their objectives. For example, developers could use ML-Agents to create a game where multiple agents must work together to solve puzzles or defeat enemies.

In addition to its support for RL algorithms, ML-Agents also includes a number of other machine learning techniques, such as imitation learning, which allows agents to learn from human demonstrations, and curriculum learning, which gradually increases the difficulty of a task as the agent becomes more proficient.

pyTorch

PyTorch is a popular open-source machine learning framework that is widely used for developing and training deep neural networks. PyTorch provides a flexible and dynamic programming interface that allows developers to quickly prototype and experiment with various machine learning models.

ML-Agents provides seamless integration with PyTorch, allowing developers to easily use PyTorch for developing and training their intelligent agents. The integration allows developers to take advantage of PyTorch's powerful GPU acceleration and automatic differentiation capabilities.

To use PyTorch with ML-Agents, developers can define their neural network models using PyTorch's standard interface, and then integrate these models with the ML-Agents training pipeline. The training pipeline handles the communication between the Unity environment and the PyTorch model, allowing the agent to learn from its interactions with the environment.

The PyTorch integration also allows developers to leverage the rich ecosystem of PyTorch tools and libraries, such as TorchVision for image processing and TorchText for natural language processing. This makes it easy to develop agents that can learn from complex sensory inputs, such as images or text.

One key advantage of using PyTorch with ML-Agents is the ability to easily scale up training using distributed training techniques. PyTorch supports distributed training across multiple GPUs or even multiple machines, which can greatly accelerate training times for large models.

TensorBoard

TensorBoard is a web-based visualization tool that is built into TensorFlow. It provides a suite of visualization tools for monitoring and debugging the training process of deep neural networks, including ML-Agents models. TensorBoard makes it easy to visualize and analyze the various metrics and data that are generated during the training process.

TensorBoard supports a wide range of visualization types, including graphs of the neural network architecture, histograms of weights and biases, and charts of various performance metrics such as accuracy, loss, and learning rates. Developers can use TensorBoard to track the progress of their models over time, and to identify and diagnose issues that may arise during the training process.

One key feature of TensorBoard is its support for real-time monitoring. As the model is being trained, TensorBoard updates the visualization in real time, allowing developers to monitor the progress of the training process and quickly identify any issues or anomalies.

TensorBoard also provides support for distributed training, which is important for scaling up training across multiple machines or GPUs. Developers can use TensorBoard to monitor the progress of training across multiple machines and to aggregate the results for analysis.

Another useful feature of TensorBoard is its support for hyperparameter tuning. Developers can use TensorBoard to visualize the performance of their models across a range of hyperparameters, such as learning rates or batch sizes, allowing them to identify the optimal hyperparameters for their specific problem.

Genetic algorithm

The genetic algorithm (GA) will be implemented with C# and run within the Unity editor and application. This makes it easy to interface with the game state to change values and utilize the ML-Agents implementation to speed up the simulation process. There is however limitations to this approach which does not allow for parallel environments. Another approach would be to handle the GA outside the Unity environment and interface with the balance values through JSON. This is the strongest approach but does require significantly more development time as communication is required between the Unity applications and the outside environment.

Database API: Firebase Realtime

Firebase is Google's database service with API's for several different frameworks and platforms including Unity. (Google 2023) This database library will be used to store game logs, in order to store and review game stats for evaluation. This can be used to gather any relevant data for evaluating the balance state of the game. During these evaluations, a database was not needed, but it will be when collecting user data in the final step of the evaluation. This was not implemented, since no game-play sessions were recorded.

7.2 Game Implementation

This section will cover the implementation of the game. The section will not be very detailed as the implementation and code is very extensive. Overall the implementation consists of two general pipelines. Online play and offline play for RL model training. There is a dedicated lobby and game scene with online play and a dedicated game scene for model training. Most of the scripts and functions are shared between the two environments and are differentiated by accessing a variable in the TestMode class.

TestMode

This class is used to handle how tests are carried out, containing enums for choosing the tested classes or whether an agent is trained or tested with inference. One of the most important aspects of this class is the isTestMode boolean which is accessed in several other classes to differentiate whether the functionality of the program should run as networked or in offline mode. The class is implemented as a Singleton structure allowing for this variable to be easily accessed from the entire codebase.

Networking

The game is networked using Unity's 'Netcode for gameobjects' netcode library and matchmaking and relay services provided by Unity Gaming Services (UGS). This allows the game to run on a P2P model where one player is hosting a game that another play can join.

Matchmaking

Matchmaking and opening connections are handled in the matchmaking class. This class is mostly accessed from the UIManager class to call for opening connections when trying to enter a game, by for instance the FindGame() function. The matchmaking class is implemented as a Singleton allowing for easy access from the entire codebase. The class consists of mostly asynchronous calls for user authentication and to search for or create lobbies asynchronously. The shortcoming here is that reconnecting to a lobby if the connection was lost has not yet been implemented.

User Interface

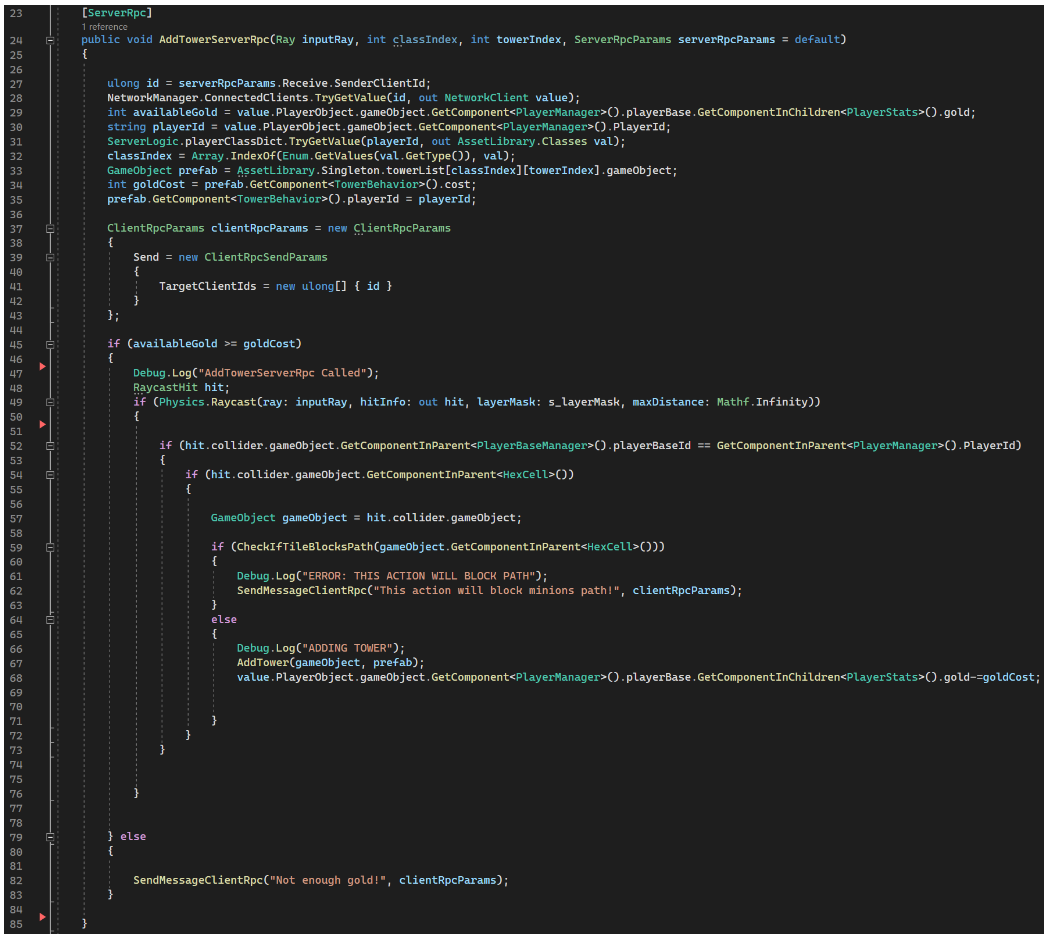
In order to connect to a game and interact with the game, a user interface was created. The UI mostly consists of one large script the UIManager class. This class contains references to all components of the UI and methods relating to hiding and showing these, as well as creating the correct inputs based on user input like class selection, unit selling, and so on.

Hexagonal Grid

A grid was created to place towers on the map. In order to accomplish this a GridManager class and a HexCell class was created. The grid manager class creates all the cells depending on the specified size. The HexCell class manages each individual cell of the grid and contains information such as its number, references to the neighbors, whether the cell is occupied and methods related to shading.

Tower Manager

The tower manager class is responsible for managing tower creation. It mainly consists of the AddTowerServer-Rpc() method which is used for creating towers over the network. The method takes as input the index of the player tower and runs several checks to validate the action. It checks if the player has enough gold to build and whether the tower will block the minion's path which is not allowed.

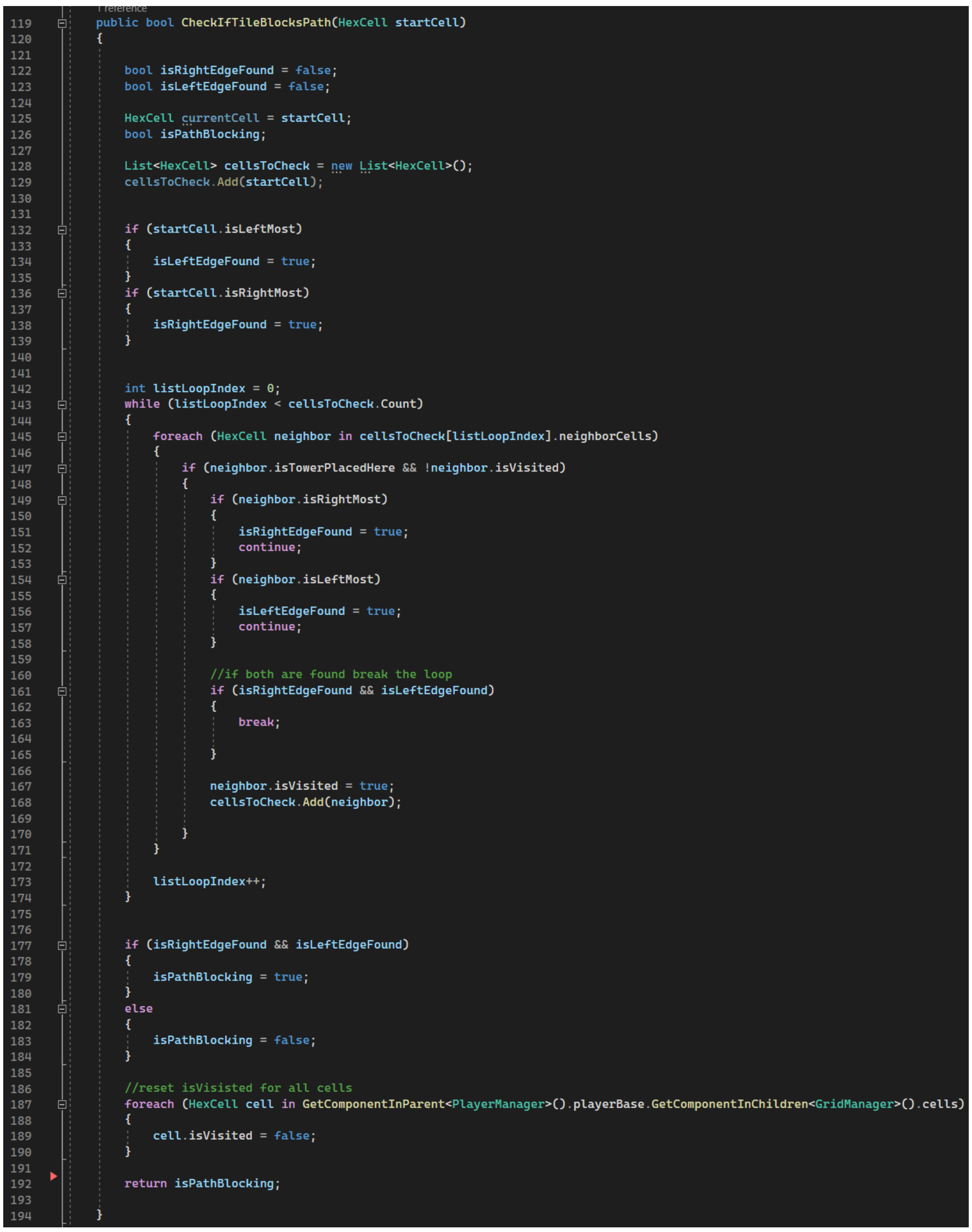


```
[ServerRpc]
1 reference
public void AddTowerServerRpc(Ray inputRay, int classIndex, int towerIndex, ServerRpcParams serverRpcParams = default)
{

    ulong id = serverRpcParams.Receive.SenderClientId;
    NetworkManager.ConnectedClients.TryGetValue(id, out NetworkClient value);
    int availableGold = value.PlayerObject.gameObject.GetComponent<PlayerManager>().playerBase.GetComponentInChildren<PlayerStats>().gold;
    string playerId = value.PlayerObject.gameObject.GetComponent<PlayerManager>().PlayerId;
    ServerLogic.playerClassDict.TryGetValue(playerId, out AssetLibrary.Classes val);
    classIndex = Array.IndexOf(Enum.GetValues(val.GetType()), val);
    GameObject prefab = AssetLibrary.Singleton.towerList[classIndex][towerIndex].gameObject;
    int goldCost = prefab.GetComponent<TowerBehavior>().cost;
    prefab.GetComponent<TowerBehavior>().playerId = playerId;

    ClientRpcParams clientRpcParams = new ClientRpcParams
    {
        Send = new ClientRpcSendParams
        {
            TargetClientIds = new ulong[] { id }
        }
    };

    if (availableGold >= goldCost)
    {
        Debug.Log("AddTowerServerRpc Called");
        RaycastHit hit;
        if (Physics.Raycast(ray: inputRay, hitInfo: out hit, layerMask: s_layerMask, maxDistance: Mathf.Infinity))
        {

            if (hit.collider.gameObject.GetComponentInParent<PlayerBaseManager>().playerBaseId == GetComponentInParent<PlayerManager>().PlayerId)
            {
                if (hit.collider.gameObject.GetComponentInParent<HexCell>())
                {

                    GameObject gameObject = hit.collider.gameObject;

                    if (CheckIfTileBlocksPath(gameObject.GetComponentInParent<HexCell>()))
                    {
                        Debug.Log("ERROR: THIS ACTION WILL BLOCK PATH");
                        SendMessageClientRpc("This action will block minions path!", clientRpcParams);
                    }
                    else
                    {
                        Debug.Log("ADDING TOWER");
                        AddTower(gameObject, prefab);
                        value.PlayerObject.gameObject.GetComponent<PlayerManager>().playerBase.GetComponentInChildren<PlayerStats>().gold-=goldCost;


                    }
                }
            }



        }


    } else
    {

        SendMessageClientRpc("Not enough gold!", clientRpcParams);
    }

}
```

Figure 7.1: *AddTowerServerRpc() method*

Currently, the towerManager is implemented to work in networked mode using a passed Ray variable inputRay. This is however not very efficient and should be optimized with a simple integer reference of the hex id.

The CheckIfTileBlocksPath() method implements a grassfire path algorithm to calculate whether a hex cell is a candidate for a tower. If placing a tower results in blocking the path the method returns false, otherwise it returns true.



```
public bool CheckIfTileBlocksPath(HexCell startCell)
{

    bool isRightEdgeFound = false;
    bool isLeftEdgeFound = false;

    HexCell currentCell = startCell;
    bool isPathBlocking;

    List<HexCell> cellsToCheck = new List<HexCell>();
    cellsToCheck.Add(startCell);


    if (startCell.isLeftMost)
    {
        isLeftEdgeFound = true;
    }
    if (startCell.isRightMost)
    {
        isRightEdgeFound = true;
    }


    int listLoopIndex = 0;
    while (listLoopIndex < cellsToCheck.Count)
    {
        foreach (HexCell neighbor in cellsToCheck[listLoopIndex].neighborCells)
        {
            if (neighbor.isTowerPlacedHere && !neighbor.isVisited)
            {
                if (neighbor.isRightMost)
                {
                    isRightEdgeFound = true;
                    continue;
                }
                if (neighbor.isLeftMost)
                {
                    isLeftEdgeFound = true;
                    continue;
                }

                //if both are found break the loop
                if (isRightEdgeFound && isLeftEdgeFound)
                {
                    break;

                }

                neighbor.isVisited = true;
                cellsToCheck.Add(neighbor);

            }
        }

        listLoopIndex++;
    }


    if (isRightEdgeFound && isLeftEdgeFound)
    {
        isPathBlocking = true;
    }
    else
    {
        isPathBlocking = false;
    }

    //reset isVisisted for all cells
    foreach (HexCell cell in GetComponentInParent<PlayerManager>().playerBase.GetComponentInChildren<GridManager>().cells)
    {
        cell.isVisited = false;
    }

    return isPathBlocking;

}
```

Figure 7.2: *CheckIfTileBlocksPath() method*

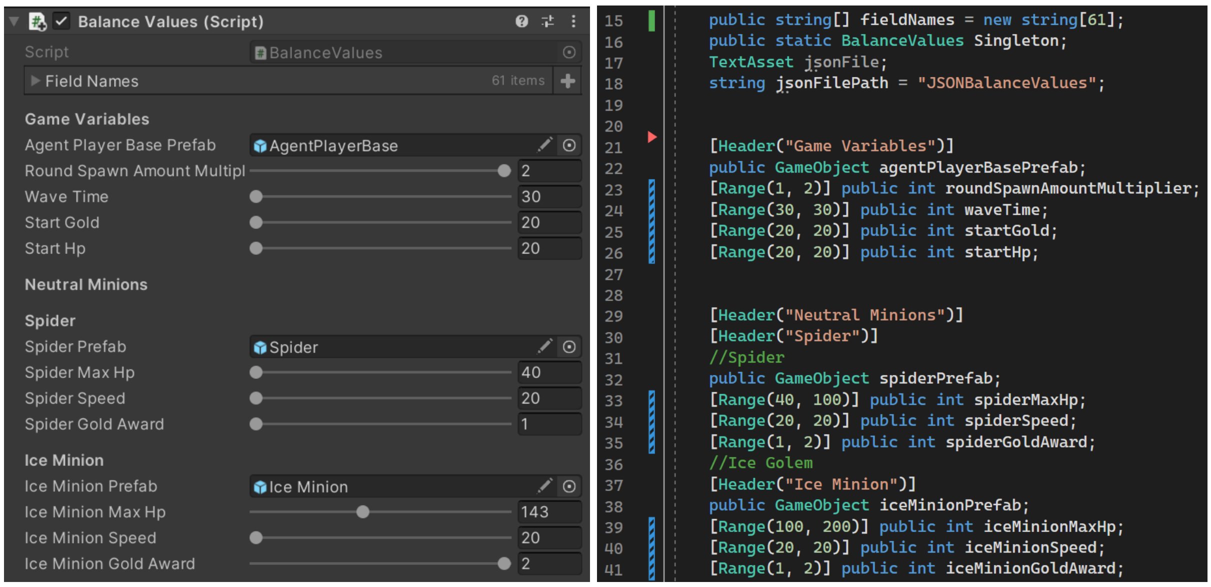
Tower behavior

The tower behavior class is responsible for all behaviors related to player towers. This is mostly code related to observing and attacking minions on the map. The tower will check for colliders within its given radius and attack minions on cooldown. The damage, radius, and cost of towers are defined by the BalanceValues class.

The class uses an enum to select the type of tower attack in order to share core tower functionality between multiple different towers. Tower attacks could for instance be a single target damage projectile or an area of effect attack damaging units around the target as well or slowing the units.

BalanceValues

The BalanceValues class contains references to all important balance values of the game. It consists of a function to assign all values through the inspector. This creates an overview in the inspector of all balance values and with the possibility to assign ranges and lock to a specific range as proposed in Morosan's balance specification language (Morosan 2019). This view can be seen in figure 7.3 This function is also used to set all values generated by the genetic algorithm. BalanceValues is implemented as a Singleton structure for easy access throughout the codebase.



Balance Values (Script)

Script	BalanceValues
Field Names	61 items
Game Variables	
Agent Player Base Prefab	AgentPlayerBase
Round Spawn Amount Multipl	2
Wave Time	30
Start Gold	20
Start Hp	20
Neutral Minions	
Spider	
Spider Prefab	Spider
Spider Max Hp	40
Spider Speed	20
Spider Gold Award	1
Ice Minion	
Ice Minion Prefab	Ice Minion
Ice Minion Max Hp	143
Ice Minion Speed	20
Ice Minion Gold Award	2

```
    public string[] fieldNames = new string[61];
    public static BalanceValues Singleton;
    TextAsset jsonFile;
    string jsonFilePath = "JSONBalanceValues";


    [Header("Game Variables")]
    public GameObject agentPlayerBasePrefab;
    [Range(1, 2)] public int roundSpawnAmountMultiplier;
    [Range(30, 30)] public int waveTime;
    [Range(20, 20)] public int startGold;
    [Range(20, 20)] public int startHp;


    [Header("Neutral Minions")]
    [Header("Spider")]
    //Spider
    public GameObject spiderPrefab;
    [Range(40, 100)] public int spiderMaxHp;
    [Range(20, 20)] public int spiderSpeed;
    [Range(1, 2)] public int spiderGoldAward;
    //Ice Golem
    [Header("Ice Minion")]
    public GameObject iceMinionPrefab;
    [Range(100, 200)] public int iceMinionMaxHp;
    [Range(20, 20)] public int iceMinionSpeed;
    [Range(1, 2)] public int iceMinionGoldAward;
```

Figure 7.3: *The balance values in the Unity inspector (L) and in the script where ranges can be edited (R)*

Minion Behavior

NPC Behavior is controlled by the NPCStats class. This is a simple class responsible for the state of a minion and holds all variables related to his, such as health, speed, and cost. This continuously checks the health and calls a Death() method when health reaches zero, to start animations and destroy the object. This handles different kinds of deaths when killed by a tower or reaching the enemy base. It also handles effects like speed reduction.

Pathfinding is done using the library A*. The implementation consists of the Seeker class and AIPath class on the minion gameobjects. When a minion is spawned the seeker class is accessed and SetDestination() method is called. Another relevant variable is the maxSpeed which is also accessed when the speed of a minion is altered.

Player Manager

The PlayerManager class contains variables like the player ID and references to the player base and other relevant classes or instantiated objects. The class also contains several methods for handling variables and objects over the network like IDs, player and enemy objects.

Player Stats

The Player Stats class contains variables like the gold and hp of the player. This class is responsible for running collision checks to see if any minions made it through the defense. The method will destroy the minions and subtract the health from the player with networked remote procedure calls.

Spawn Manager

The SpawnManager class is managing the spawning of NPCs, both minions sent by players and regular waves of spawns. The main functionality is an asynchronous function that runs for a set amount of waves increasing the health points, amount, and types of minions each round.

InputManager

The InputManager class is used to get input from the player. This mostly concerns camera control, mouse/-touch input, and hotkey actions. The InputManager also ensures that actions are handled differently depending on the platform.

Asset Library

The asset library script loads all minions and towers directly from the resources folder on start with the InitializeLibrary() method. It holds references in two-dimensional lists to the minions in minionList and towers in towerList. This class is structured as a Singleton for easy accessibility from the entire codebase.

Other

There are many other scripts that were implemented to run the game. These are less important and will not be covered. The classes are listed here: PlayerBaseManager, CameraParent, StopSpell, TextPopUpManager, DistanceCalculator, AnimatorManager, AudioManager, GameSettings, NetworkManagerUI and SceneManager.

7.3 Reinforcement Learning Implementation

The reinforcement learning implementation (and imitation learning) environment was set up using ML-Agents with Unity and a Python virtual environment. The tests ran using CUDA cores of an Nvidia RTX 3080. Experimentation with varying amounts of parallel environments proved stable performance running 20 simultaneous environments. With 30 or 40 parallel environments, crashes and anomalies in the run time of the computer would start to occur. Performance graphs were viewed with TensorBoard which allowed for tracking the learning process in detail. Hyperparameters of the models were accessed through .yaml files in the virtual environment. Inference runs to check the performance of the models was done inside the editor. Recordings of demonstrations (heuristic runs) for imitation learning were also done in the editor. This was done through the 'Behavior Parameters' script by changing the behavior type between 'default' for training, 'heuristic only' for imitation, and 'inference only' for testing. The 'MazeWarsAgent' class was created to control rewards and observations for the models. This class inherits the 'Agent' class and overrides certain methods. The 'OnActionReceived' method takes the actions produced by the model to produce an action within the game. The 'CollectObservations' adds observations to the model regarding the state of the game. When the EndEpisode() method is called. The game episode ends and the OnEpisodeBegin() is called. This method resets the game state and all parameters to start a new episode and continue training. The 'Heuristic' method was also over-written to set the actions through user input instead of from the model. In order to collect the inputs from the user a 'UserInput' method was defined, which would allow a player to manually set these parameters through interacting with the game. Rewards were controlled through a 'RewardSystem' method.

7.3.1 Reward System

The reward system was implemented in a way that emphasizes placing towers that increase the travel distance of enemies and proximity to those towers. Rewards depend on settings in TestMode class of whether cumulative rewards (reward shaping) were enabled and if win/loss rewards were enabled for that particular run.

The agent is rewarded for:

- **Damage done to towers**
- **Tower placed with 1 or 2 neighbors**
- **Whenever the travel distance of enemies is increased**
- **Enemy health points lost**

The agent is rewarded negatively for:

- **Loosing health points**
- **Tower placed with more than 2 neighbors**
- **Trying to place a tower in an occupied spot**
- **Trying to place a tower in a path-blocking spot**
- **Trying to place a tower it cannot afford**
- **Tie/Loss**

7.3.2 State and Observations

The agent is informed of the game state through a total of 118 observations. 1 of which is discrete, the faction, and the remaining 117 continuous:

- **Faction (Discrete)**
- **Health points**
- **Enemy health points**
- **Array of tower placements and type (49)**
- **Gold**
- **Damage done**
- **Minions killed amount**
- **Towers placed amount**
- **Distance through maze**
- **All balance parameters (61)**

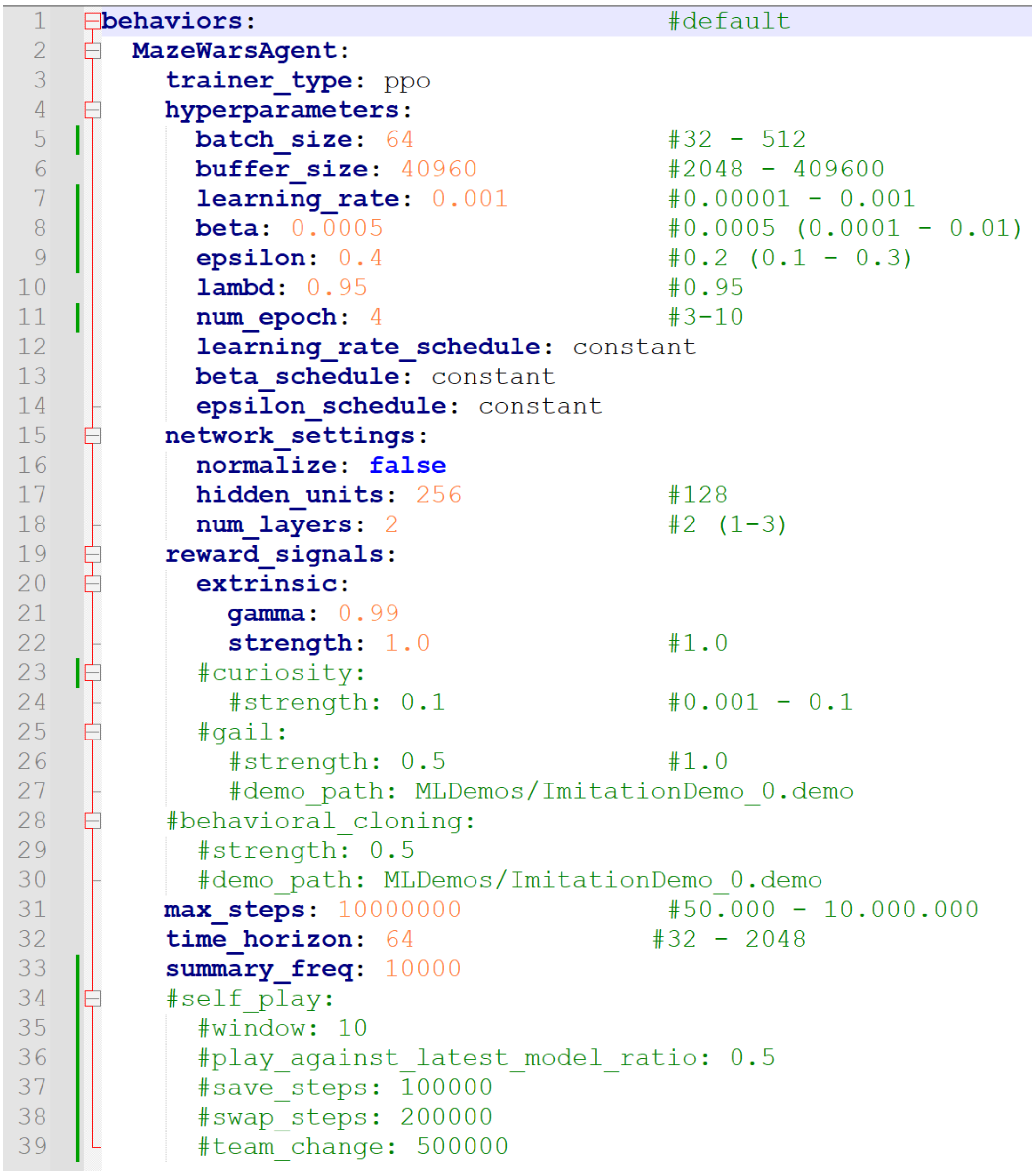
7.3.3 Actions

The agent utilizes a total of 6 discrete action parameters:

- **Tower position X axis selector**
- **Tower position Y axis selector**
- **Spawn tower boolean**
- **Spawn minion boolean**

- **Tower type selector**
- **Minion type selector**

7.3.4 Final Hyperparameters



```yaml
behaviors:                          #default
  MazeWarsAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64                #32 - 512
      buffer_size: 40960            #2048 - 409600
      learning_rate: 0.001          #0.00001 - 0.001
      beta: 0.0005                  #0.0005 (0.0001 - 0.01)
      epsilon: 0.4                  #0.2 (0.1 - 0.3)
      lambd: 0.95                   #0.95
      num_epoch: 4                  #3-10
      learning_rate_schedule: constant
      beta_schedule: constant
      epsilon_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 256             #128
      num_layers: 2                 #2 (1-3)
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0               #1.0
      #curiosity:
        #strength: 0.1              #0.001 - 0.1
      #gail:
        #strength: 0.5              #1.0
        #demo_path: MLDemos/ImitationDemo_0.demo
    #behavioral_cloning:
      #strength: 0.5
      #demo_path: MLDemos/ImitationDemo_0.demo
    max_steps: 10000000             #50.000 - 10.000.000
    time_horizon: 64               #32 - 2048
    summary_freq: 10000
    #self_play:
      #window: 10
      #play_against_latest_model_ratio: 0.5
      #save_steps: 100000
      #swap_steps: 200000
      #team_change: 500000
```

Figure 7.4: *.yaml file with final hyperparameters*

7.3.5 MazeWarsAgent Class

The class inherits the Agent class of the ML-Agents library. This class is responsible for the interfacing of the neural net model running in Python and the agent in the Unity environment. This involves feeding the model the state of the game and requesting an action to execute. This class also contains the entire reward system. There are several booleans in this class to manage how the training behaves. These are variables that control the reward behavior between cumulative or episode rewards. Whether or not both sides are training or an inference model, or which class or both we are training. It can also control if demonstrations are being recorded for imitation learning.

The CollectObservations method manages feeding observations of the game state to the model. These include stats like health points, tower positions, or the state of the balance values. Balance state observations are fed to the model using the CollectBalanceVars() class which accesses BalanceValues and returns all integer values of the class in a balanceVars list. The OnActionReceived function manages the execution of actions, once an action is requested and the current values are returned from the neural net. This function contains the logic that executes actions based on these values.

7.3.6 BehaviorParameters Class

The BehaviorParameters class is the standard interface provided by the ML-Agents library to manage the general settings of the agent and RL environment. This class contains settings for whether the agent is running with inference, heuristic, or default training states. It also is where we define the vector lengths of the observation data and action data, and define if the data is discrete or continuous.

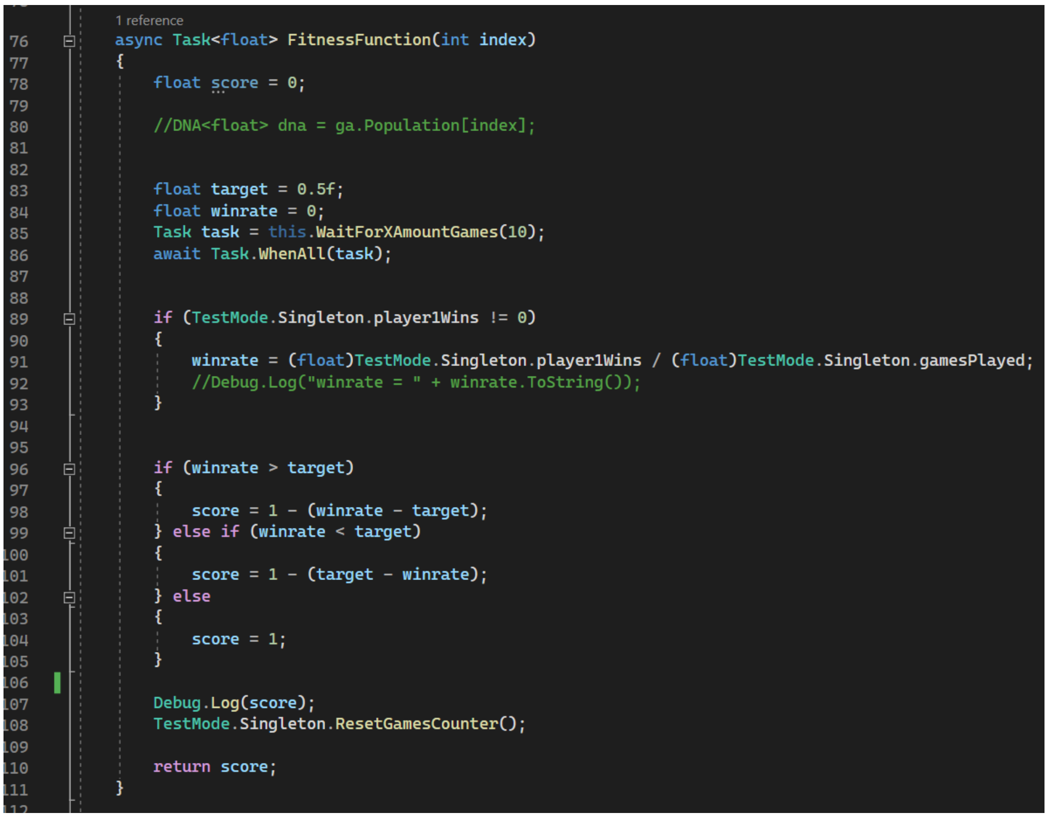
7.4 Genetic Algorithm Implementation

A lot of this algorithm was implemented using code from Tiago Figueiredo.[2] The code was rewritten to run asynchronously and to work for this specific task.

7.4.1 BalanceGA Class

The BalanceGA class is where most of the custom code for this implementation is written being the fitness function and gene generator. The fitness function simply calculates the win rate over a set amount of games. 7.5

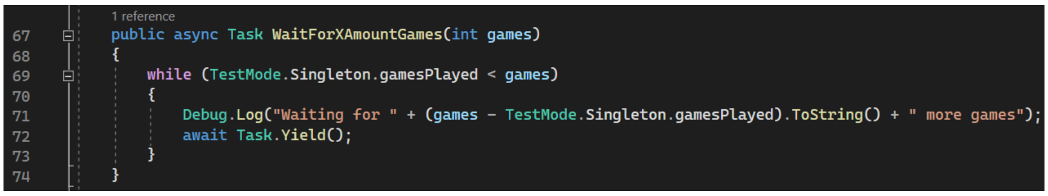
[2]https://bitbucket.org/kryzarel/generic-genetic-algorithm/src/master/



```
1 reference
async Task<float> FitnessFunction(int index)
{
    float score = 0;

    //DNA<float> dna = ga.Population[index];


    float target = 0.5f;
    float winrate = 0;
    Task task = this.WaitForXAmountGames(10);
    await Task.WhenAll(task);


    if (TestMode.Singleton.player1Wins != 0)
    {
        winrate = (float)TestMode.Singleton.player1Wins / (float)TestMode.Singleton.gamesPlayed;
        //Debug.Log("winrate = " + winrate.ToString());
    }


    if (winrate > target)
    {
        score = 1 - (winrate - target);
    } else if (winrate < target)
    {
        score = 1 - (target - winrate);
    } else
    {
        score = 1;
    }

    Debug.Log(score);
    TestMode.Singleton.ResetGamesCounter();

    return score;
}
```

Figure 7.5: *Fitness function*

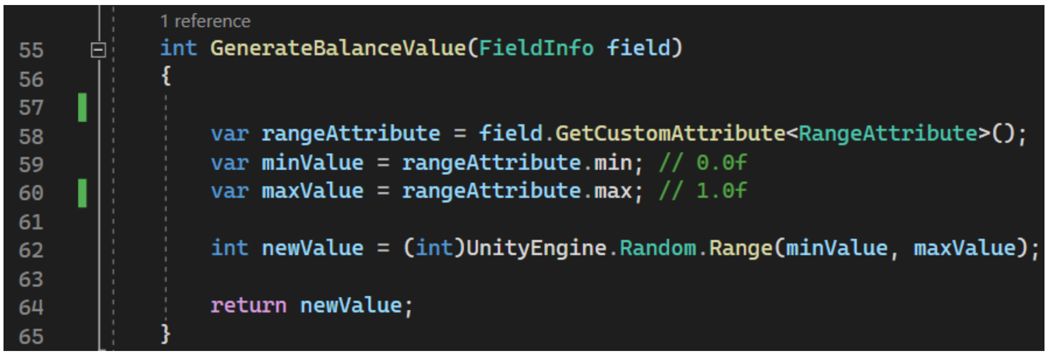
The WaitForXAmountGames method is a task that is awaited within the fitness function, which allows the algorithm to suspend while waiting for simulations to complete.7.6



```
1 reference
public async Task WaitForXAmountGames(int games)
{
    while (TestMode.Singleton.gamesPlayed < games)
    {
        Debug.Log("Waiting for " + (games - TestMode.Singleton.gamesPlayed).ToString() + " more games");
        await Task.Yield();
    }
}
```

Figure 7.6: *Asynchronous task to await simulations*

The GenerateBalanceValue function works as the gene generator. Whenever a new value for a balance variable needs to be generated this is the function used. It uses the FieldInfo class of the variables of the BalanceValues class to get the RangeAttribute which contains the set range for the specific variable. This is used to generate a new random value within the specified range. 7.7

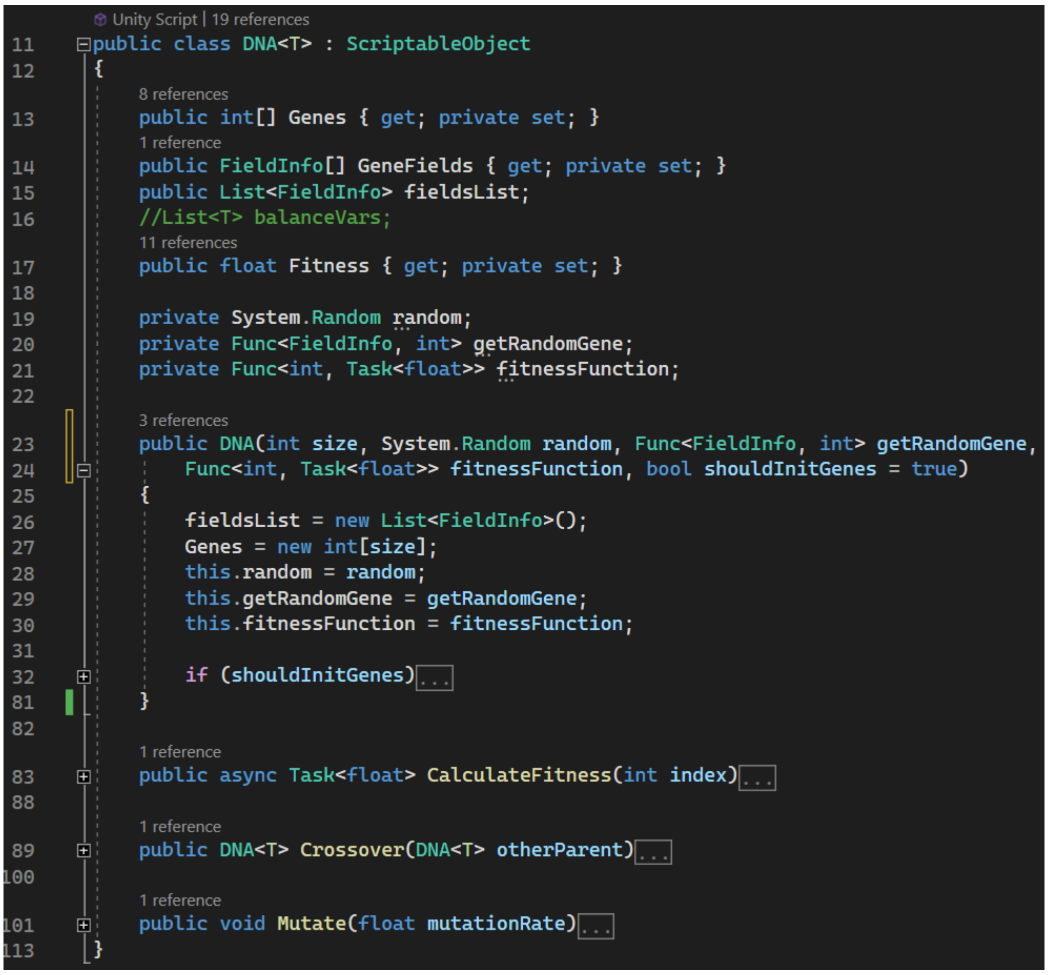


```
1 reference
int GenerateBalanceValue(FieldInfo field)
{

    var rangeAttribute = field.GetCustomAttribute<RangeAttribute>();
    var minValue = rangeAttribute.min; // 0.0f
    var maxValue = rangeAttribute.max; // 1.0f

    int newValue = (int)UnityEngine.Random.Range(minValue, maxValue);

    return newValue;
}
```

Figure 7.7: *Generation of random balance values within set ranges*

7.4.2 DNA Class

The DNA class holds the information of a gene. It holds an array storing the 61 balance values. It also contains the function for calculating the fitness and functions for calculating crossover and mutation. 7.8

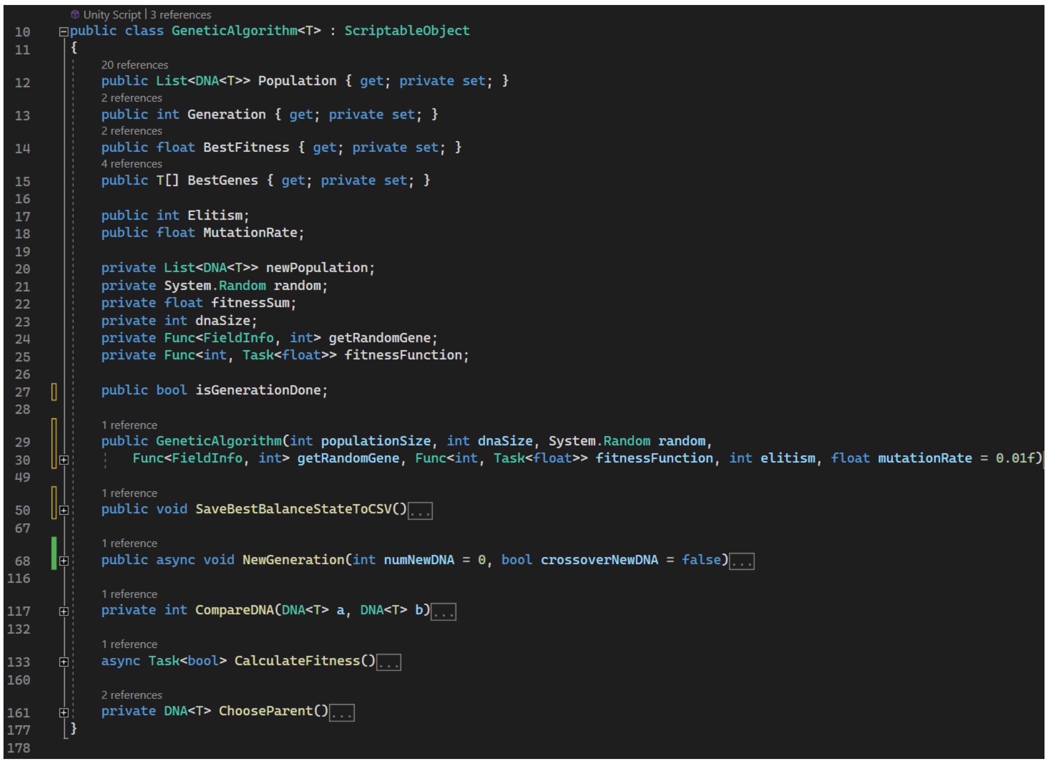


```
Unity Script | 19 references
public class DNA<T> : ScriptableObject
{
    8 references
    public int[] Genes { get; private set; }
    1 reference
    public FieldInfo[] GeneFields { get; private set; }
    public List<FieldInfo> fieldsList;
    //List<T> balanceVars;
    11 references
    public float Fitness { get; private set; }

    private System.Random random;
    private Func<FieldInfo, int> getRandomGene;
    private Func<int, Task<float>> fitnessFunction;

    3 references
    public DNA(int size, System.Random random, Func<FieldInfo, int> getRandomGene,
        Func<int, Task<float>> fitnessFunction, bool shouldInitGenes = true)
    {
        fieldsList = new List<FieldInfo>();
        Genes = new int[size];
        this.random = random;
        this.getRandomGene = getRandomGene;
        this.fitnessFunction = fitnessFunction;

        if (shouldInitGenes)...
    }

    1 reference
    public async Task<float> CalculateFitness(int index)...

    1 reference
    public DNA<T> Crossover(DNA<T> otherParent)...

    1 reference
    public void Mutate(float mutationRate)...
}
```

Figure 7.8: *DNA Class*

7.4.3 GeneticAlgorithm Class

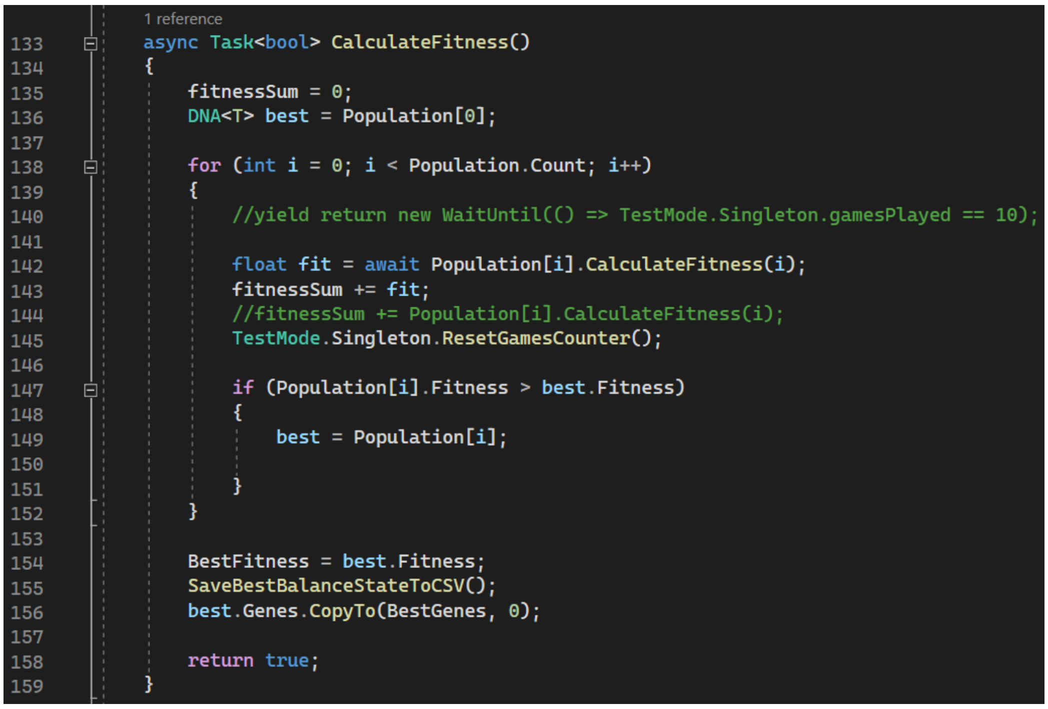
The GeneticAlgorithm class is mostly responsible for creating generations and handling choosing of parents and DNA comparison. This class also manages saving progress to a CSV file.



```
public class GeneticAlgorithm<T> : ScriptableObject
{
    public List<DNA<T>> Population { get; private set; }
    public int Generation { get; private set; }
    public float BestFitness { get; private set; }
    public T[] BestGenes { get; private set; }

    public int Elitism;
    public float MutationRate;

    private List<DNA<T>> newPopulation;
    private System.Random random;
    private float fitnessSum;
    private int dnaSize;
    private Func<FieldInfo, int> getRandomGene;
    private Func<int, Task<float>> fitnessFunction;

    public bool isGenerationDone;

    public GeneticAlgorithm(int populationSize, int dnaSize, System.Random random,
        Func<FieldInfo, int> getRandomGene, Func<int, Task<float>> fitnessFunction, int elitism, float mutationRate = 0.01f)

    public void SaveBestBalanceStateToCSV()...

    public async void NewGeneration(int numNewDNA = 0, bool crossoverNewDNA = false)...

    private int CompareDNA(DNA<T> a, DNA<T> b)...

    async Task<bool> CalculateFitness()...

    private DNA<T> ChooseParent()...
}
```

Figure 7.9: *Genetic Algorithm Class*

The class also holds a CalculateFitness function, not to be confused with the function in the BalanceGA class. This method manages the calculation of the fitness sum and saves the best state to the CSV file. This is also an asynchronous Task as it must await each fitness calculation of each population to finish before adding up the sum. 7.10



```
1 reference
async Task<bool> CalculateFitness()
{
    fitnessSum = 0;
    DNA<T> best = Population[0];

    for (int i = 0; i < Population.Count; i++)
    {
        //yield return new WaitUntil(() => TestMode.Singleton.gamesPlayed == 10);

        float fit = await Population[i].CalculateFitness(i);
        fitnessSum += fit;
        //fitnessSum += Population[i].CalculateFitness(i);
        TestMode.Singleton.ResetGamesCounter();

        if (Population[i].Fitness > best.Fitness)
        {
            best = Population[i];

        }
    }

    BestFitness = best.Fitness;
    SaveBestBalanceStateToCSV();
    best.Genes.CopyTo(BestGenes, 0);

    return true;
}
```

Figure 7.10: *Calculate Fitness Sum*

7.4.4 Training

To train the agents to learn the meaning of different states consisting of different balance state parameter values, the agent must be trained on a wide variety of different parameters. Once the agents know how to play given different circumstances, the agents can adapt and perform given any set of balance data and optimization can begin.

Chapter 8

Evaluation

8.1 Interview

To gain some insight regarding the implementation of balance automation in the industry and to further the understanding of game balancing in general, an interview was conducted. This hour-long interview provides qualitative insights regarding a random game studio and the challenges and benefits relating to the automation of game balance. Alberto Giudice works as a level designer at Triband in Copenhagen but also works on several other aspects of the game including programming and bug fixing and more.

The general takeaway of the interview is that a lot of the focus of game balancing at their studio revolves around player feel. It is also clear how game testing and balancing take up a ton of time for the studio but much of this time will be difficult to automate. Alberto estimated that roughly 70% of the balancing is based on player feedback. They use a cloud testing service with 6-8 playtesters and test the game every few weeks with an hour of gameplay. This adds up to a lot of data and it usually takes roughly a week just to evaluate all of this data. This 70% of the balancing both have to do with making the game exciting and creating small epic moments for the player, but this also requires tweaking variables to make these moments perfect. For instance, perfectly making it over a gap in a car, feel rewarding, but to get this right, it takes tweaking and evaluation that players will make the gap. At least most of the time. This again has to do with feelings of being skillful or the game being exciting and not boring. They can also make these changes based on the engagement of the player. When doing play testing they get both video feed of the gameplay and the player. This allows the studio to help assess what feelings the game provokes. This is where automation will struggle. There might be some small applications for automation but the most interesting part here will be how automation can serve as a tool for the remaining 20%. Alberto argues that he definitely sees room for automation but that it might be difficult in the current game they are working on for example. This is because of the random nature of the game design itself. Because there are so many random mechanics that does not translate to the rest of the game, it will be difficult to train a model to learn it as a lot of the common strategies like transfer learning will be redundant to apply. Alberto also expresses concerns if the agents can actually simulate real player behavior and this is also where generative AI was mentioned and discussed briefly as having possible future benefits and contributions to the research.

In general, he believes it makes more sense for games like the developed tower defense game. This likely also has to do with differences in continuous and discrete games I would add. He suggests that maybe a good way to use the workflow, in general, could be to search and prune the strategies and mechanics that break the game and to find what does not work, instead of finding what works the best. One of the advantages he sees is removing the guesswork from parameter ranges. Maybe the automation can be used to discover what ranges of parameter values are relevant and that work in a greater context. From there the designer will have an easier time tweaking the values in a creative way. It was important for Alberto that he could be able to intervene with the model at all times, to use it in the way he wants or change what he wants. It should work as a tool and only work on its own 50% of the time.

When asked if he thinks the studio and the employees could adapt to a process like this, it would much depend on whether it would make their life harder. The process would have to be seamless, otherwise, it would be shut down. This also extends to concerns if the time spent on machine learning is paid back with dividends. This is also one of the concerns that was discovered during this project. Especially in small teams, there is a concern that all the time spent training the model and setting up the systems, etc, will be more work and time-consuming and even more expensive than game testing and manual balancing. Alberto argues for at least a 2x increase in efficiency to make it worthwhile.

In conclusion, implementing automation in game balancing presents challenges and potential benefits. The interview with Alberto Giudice highlights the focus on player feel and the time-consuming nature of game testing. While 70% of balancing relies on player feedback, the remaining 20% holds promise for automation in this random example of game design. Automation may be better suited for specific game genres, offering advantages such as removing the guesswork from parameter ranges. Balancing automation has the potential to serve as a valuable tool alongside human creativity in game development, but the conclusion is that it still has some ways to go.

Chapter 9

Results

The results section provides an overview of the researcher's process of following the proposed workflow to balance the developed game. The results section first presents the established balance goals, strategies, and requirements as defined in the flow chart, figure 9.16, and continues to follow the workflow. There are experiments with self-play and imitation learning not following the proposed workflow. This is because of experiments throughout the process in order to learn and discover how these strategies can be integrated and the challenges it poses. The section is summarizing roughly 100 training runs of data which will always be unique to the designed game. Therefore much of the detail is omitted, instead focusing on summarizing the process instead of the detailed results. The accompanying test numerations are mainly for the figures and sections to refer to the data.

9.1 Training Results

9.1.1 Balance Goals, Strategies, and Requirements

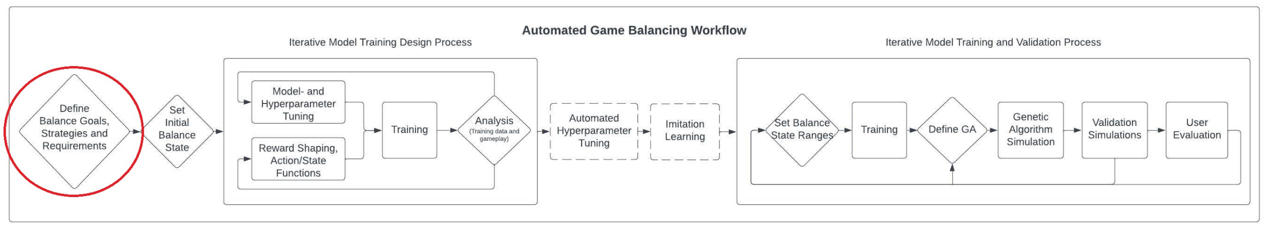

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.1: *Automated Game Balancing Workflow Step*

Goals

1. Equal win rate for both classes
2. Viability of all towers and minions

Requirements

1. **50% win rate Dragonmancer/Elementalist**
   Because there are only two classes, we only need to check for the win rate of one class.

2. **Relevance of all towers**
   Sub-goals were never implemented in the GA.

3. **Max 20-minute game**
   Games never lasted long enough for this time cap to be relevant, and it was never implemented in the GA.

Strategies

1. **Transfer learning** 3.1.3
   This strategy is applied by leveraging the model trained as one class, to train new classes.

2. **Imitation learning** 3.1.3
   The strategy was experimented with and the goal was to help speed up the training process. However, this technique and step in the model were skipped.

3. **Self-play** 3.1.3
   The self-play learning strategy was experimented with but not applied to the actual model.

4. **Resource balance** 3.2.5
   The resource balancing strategy was used by locking the health and gold variables, to ensure equal resources and starting conditions for both classes.

5. **Triple tapping** 3.2.3
   The triple tapping technique was used to generate the initial balance state for training and indirectly in the method through minimum and maximum values for gene generation.

9.1.2 Test 0.1-0.11: Experiments with Reward Shaping, Hyperparameters, and Imitation Learning

These are the initial results from training the RL model to play the game. Prior to these training results, several experiments were conducted training with the CPU. This was before the correct versions of the libraries were installed and GPU CUDA utilization was supported.

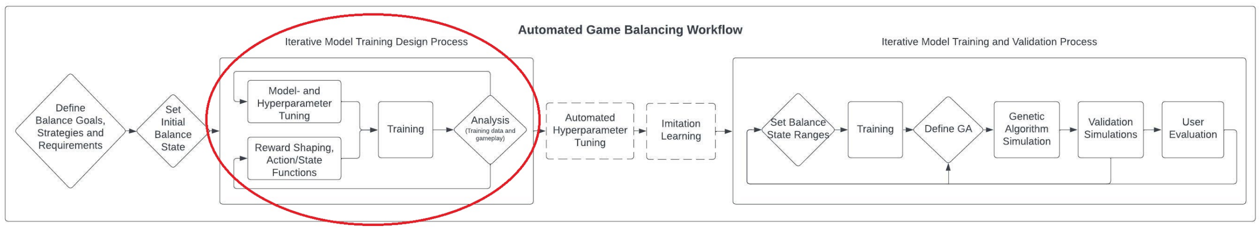

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.2: *Automated Game Balancing Workflow Step*

Figure 9.3 documents the initial experimentation with reward shaping and hyperparameters and with RL and IL (Imitation learning)

Test .6 (green) in figure 9.3 was a decent result and an indication that the model could somewhat learn and improve. The learning was however very fluctuating and unstable with a sudden steep drop in performance, indicating a learning rate that was too high. This was the case for all different sets of hyperparameters, with the best results stagnating in learning, indicating problems with the underlying reward, state and action functions.

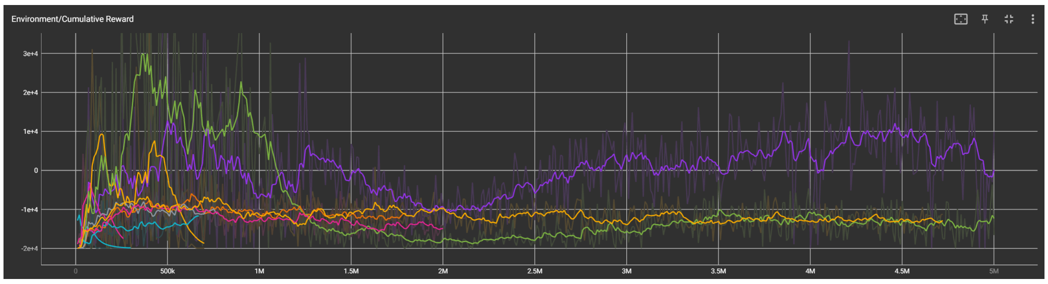


Figure 9.3: *0.1-0.11: Experiments with reward shaping and hyperparameters*

Test 0.12: State Simplification and Distance Reward Shaping

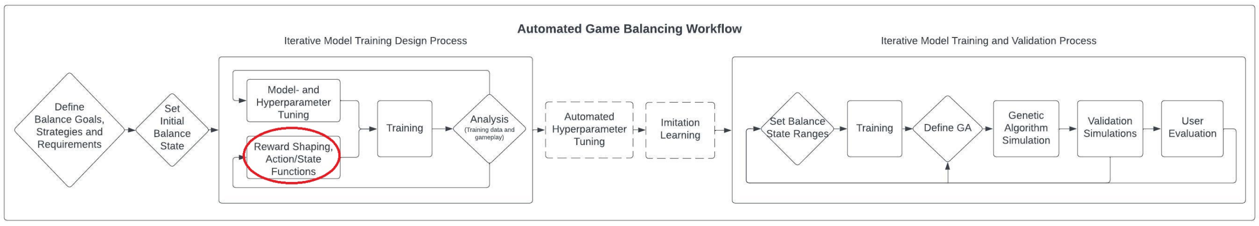

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.4: *Automated Game Balancing Workflow Step*

To help the agent solve the problem and increase learning, a distance reward was added and the state was simplified. This helped the agent to survive longer but did not have enough information on the state now. The result, shown in figure 9.5 was more steady learning, but still not to the desired extent.

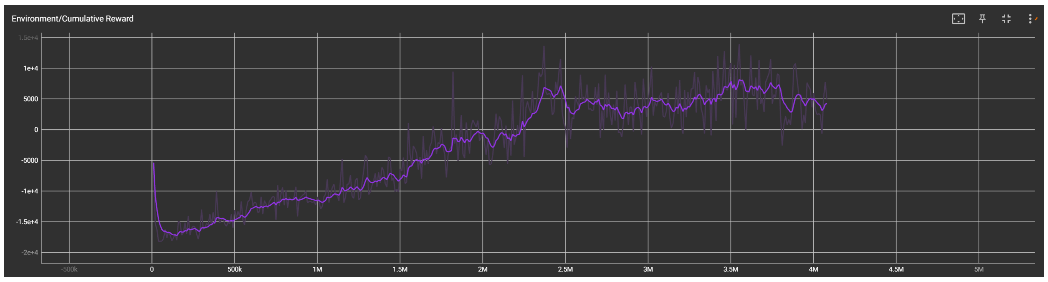


Figure 9.5: *0.12: State simplification and distance reward shaping*

9.1.3 Test 0.13 - 0.32: Experiments with Hyperparameters and Imitation Learning

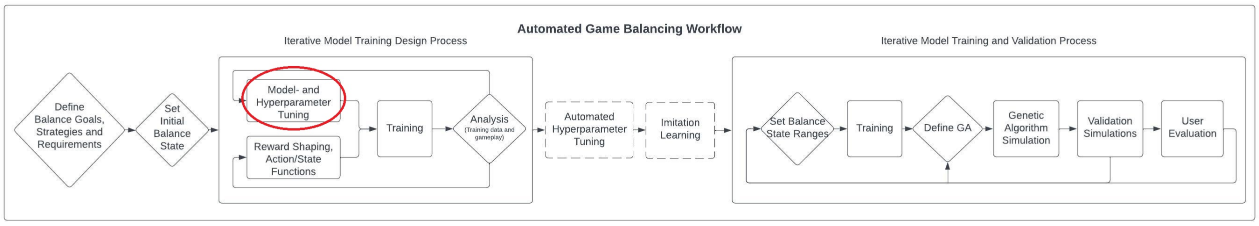

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.6: *Automated Game Balancing Workflow Step*

The next experiments were focused on utilizing imitation learning to speed up the learning, in combination with refining the hyperparameters for this training strategy. This yielded some decent results, but from evaluation of the agent play. The agent still was not learning an optimal policy but was still mostly relying on seemingly random actions. Tt was realized that the problem could be simplified and that the model might need more information regarding state and nudging with further reward shaping. Most of the state information removed in the early stage of training was added back into the model, but with less data as the problem was simplified.

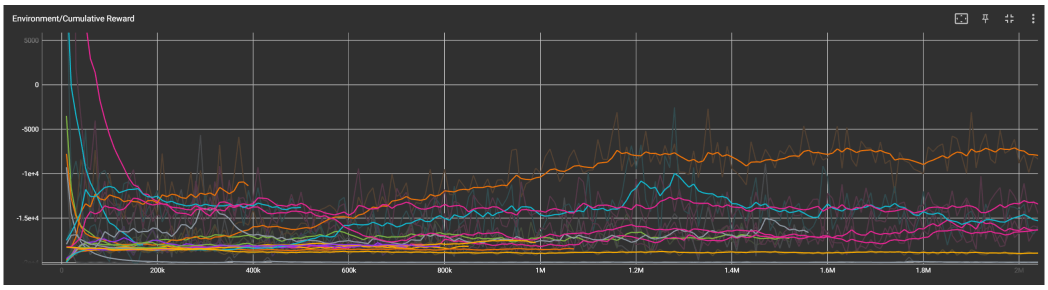


Figure 9.7: *0.13-0.32: Experiments with hyperparameters and imitation*

9.1.4 Test 2.3: New Reward, State and Action Functions and Simplification of Problem

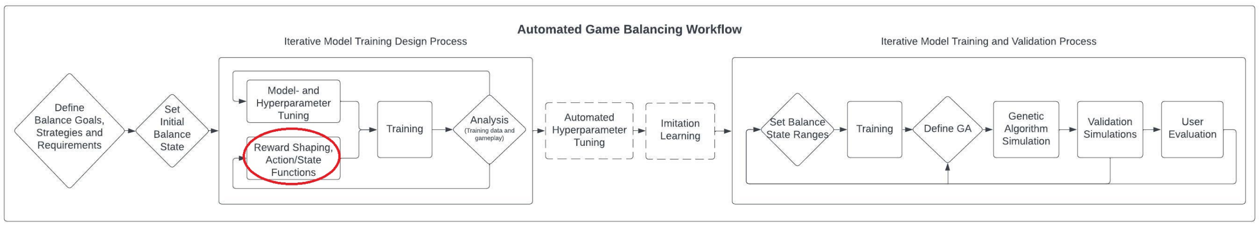

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.8: *Automated Game Balancing Workflow Step*

Test 2.3 showed significant improvement with the new refined model. The problem was simplified and the state and reward functions were optimized. These optimizations include:

- Negative reward for placing tower in already placed position.
- Count of the number of towers placed
- 2 axis representation of the position space as opposed to 1 axis.
- Array of map state, indicating where towers are placed and the type of the towers.
- Reducing board size from 15x15 to 7x7.
- Negative reward for placing a tower in an occupied spot, in a spot blocking the minion path, or trying to buy a tower without enough gold.

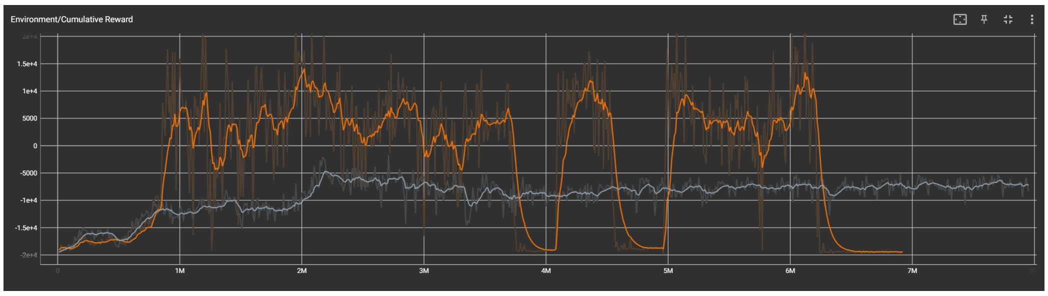


Figure 9.9: *2.3: Problem simplification, hyperparameters, and reward shaping*

The reward graph still showed very fluctuating results and steep increases and drops in learning, indicating too high a learning rate. Over the next couple of iterations, 2.4 to 2.9 several hyperparameters were tuned including specifically a reduction in learning rate and an increase in buffer size to reflect the 20 simultaneous environments in training.

9.1.5 Test 3.1: Stable Learning

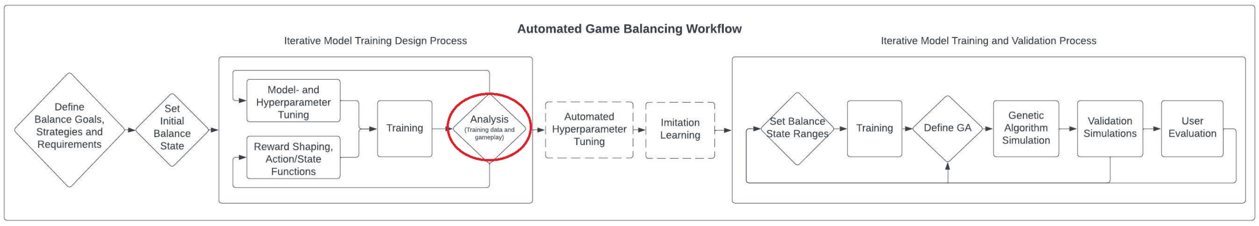

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.10: *Automated Game Balancing Workflow Step*

All these efforts from tests 2.3-2.9 resulted in a much more stable learning rate but with some stagnation. See figure 9.11 However, looking at gameplay data it was now clear that some sort of pattern learning was happening and the agents were able to start creating something resembling a maze. See figure 9.12 This is nowhere near the desired level of intelligent maze patterns, but it was decided to move on here, to investigate the self-play as this should be the superior strategy for training asymmetrically balanced games.

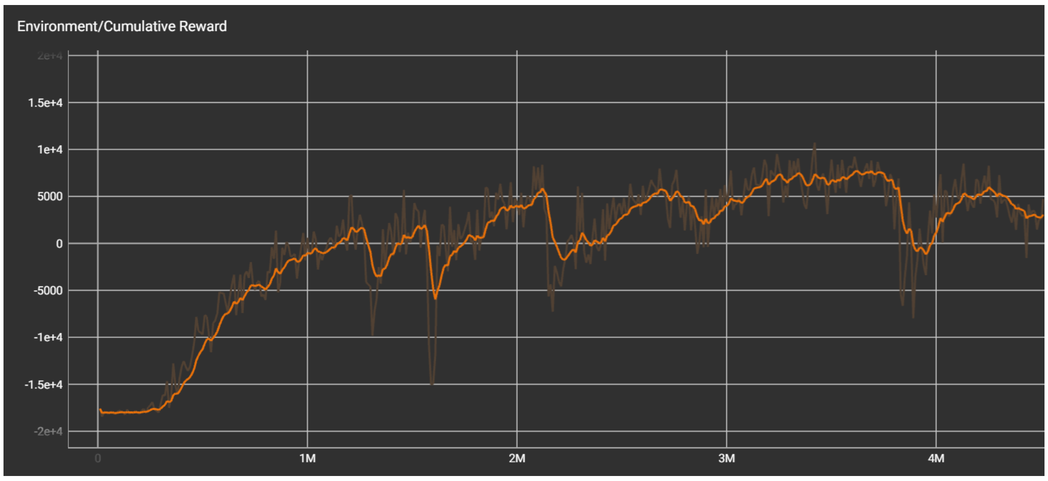


Figure 9.11: *3.1: Stable learning rate*

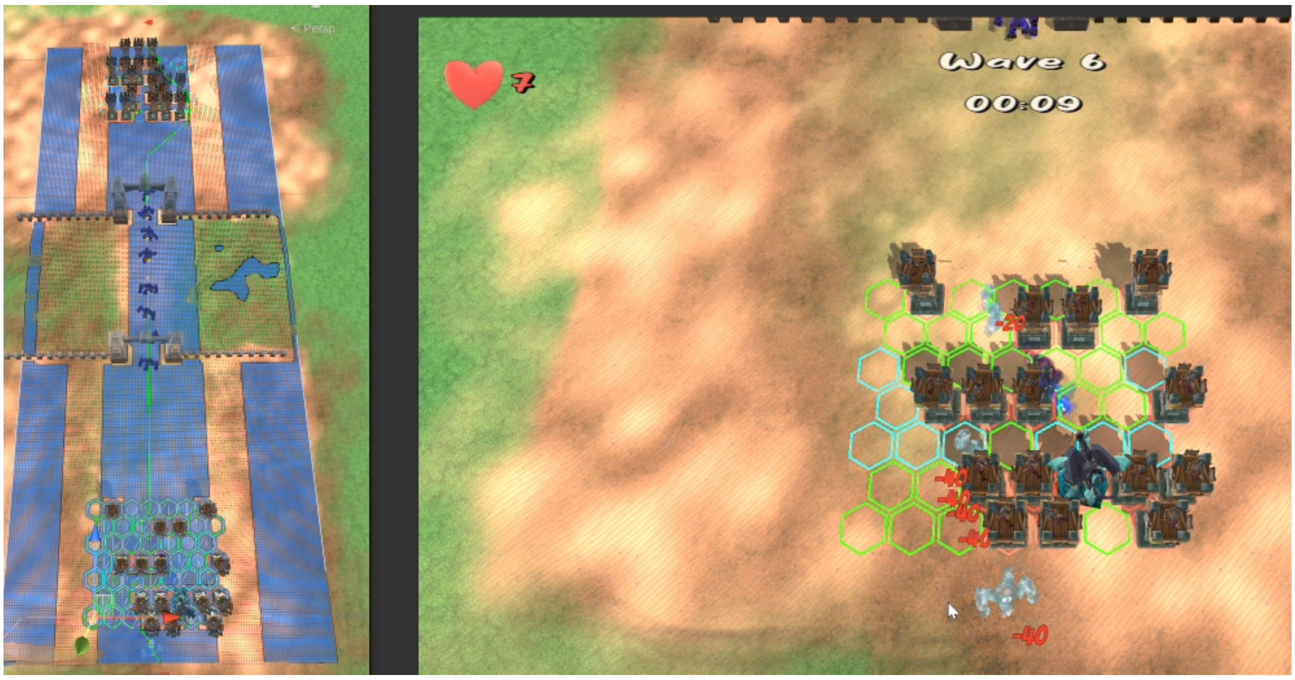


Figure 9.12: *Emerging maze patterns from the model after test 3.1*

9.1.6 Test 3.2-12.2: Experiments with Self-play

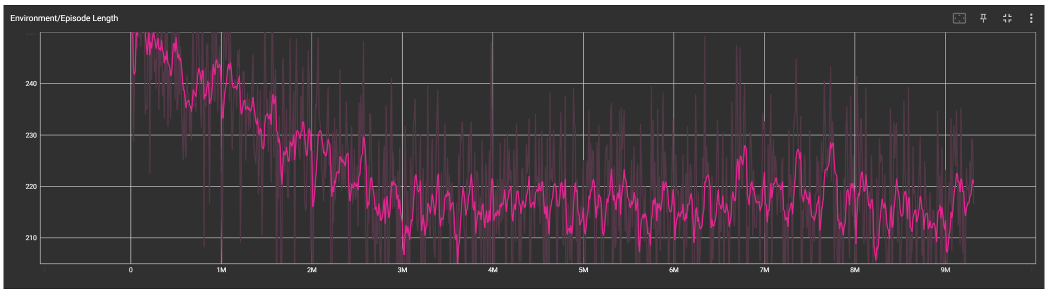


Figure 9.13: *9.4: Selfplay training episode length*

Up until test 9.4, there were several experiments with self-play, which did not provide any meaningful results. After several attempts with self-play and various hyperparameters and mixed learning strategies combining reward shaping and self-play, there was no real progress, and after a final effort with 15 million steps of training, the model still did not show any signs of learning and it was decided to move on and continue learning with reward shaping without self-play which showed some promise. The general takeaway here is that self-play can be a problematic approach to complex problems. To succeed with this process it would probably be wise to train using curriculum learning and change the complexity of the task during training. This approach was however avoided due to the additional workload it will create for the designers and developers, to create multiple versions of the game with varying amounts of mechanics and complexity in general.

9.1.7 Test 12.3: Learning Elementalist Faction (Transfer learning)

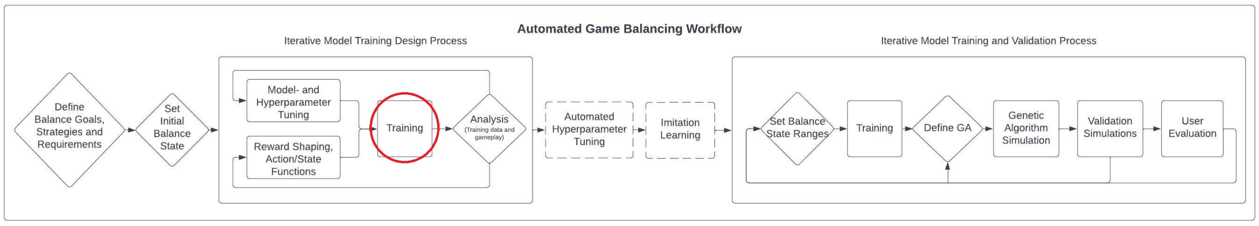

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.14: *Automated Game Balancing Workflow Step*

It is important to note that for the following tests, the scale of rewards was changed, when experimenting with self-play final rewards need to be 1, 0, and -1 for proper ELO measurement. Therefore when experimenting with reward shaping these rewards were scaled to work within a normalized range of -1 to 1.

After having no success with self-play and being pressured on time, it was decided to move away from this framework and focus on what was working - reward shaping.

Up until this point, only the Dragonmancer was trained in a specific balance configuration. To get the other class, the Elementalist, up to speed, this was trained against an inference model of the Dragonmancer class to great success. The model was now playing as the other class and the new state was reflected in its observation data. The test was initialized from test 8.2 to capitalize on existing knowledge from the Dragonmancer.

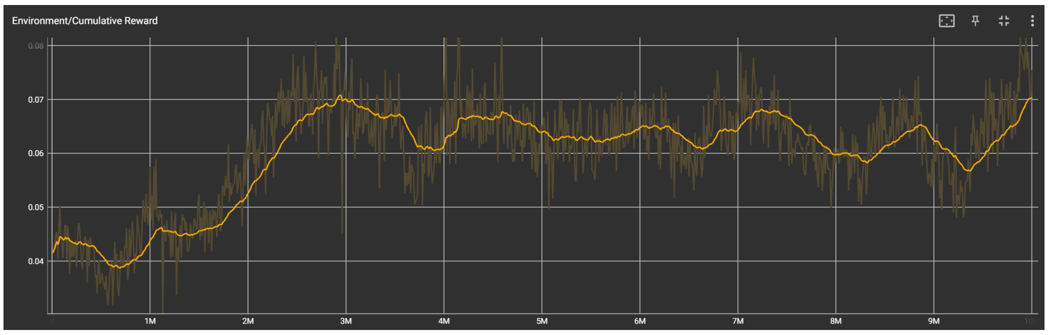


Figure 9.15: *12.3: Learning to play as a different faction*

9.1.8 Test 13.1-16.6: Training Both Classes with GA

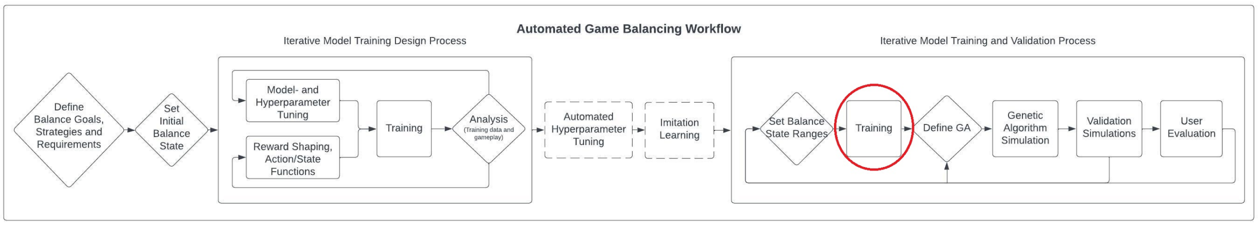

Automated Game Balancing Workflow
Iterative Model Training Design Process
Iterative Model Training and Validation Process
Define Balance Goals, Strategies and Requirements
Set Initial Balance State
Model- and Hyperparameter Tuning
Reward Shaping, Action/State Functions
Training
Analysis (Training data and gameplay)
Automated Hyperparameter Tuning
Imitation Learning
Set Balance State Ranges
Training
Define GA
Genetic Algorithm Simulation
Validation Simulations
User Evaluation

Figure 9.16: *Automated Game Balancing Workflow Step*

Figure 9.17 shows the training of the model for the Elementalist class for various states of game balance using the genetic algorithm for state variation. The orange graph represents the initial training run which dramatically increased progress. The next several runs were mostly stagnant. The expected result of these graphs was much more variation in the data. This might be because the GA was converging too quickly on balance parameters.

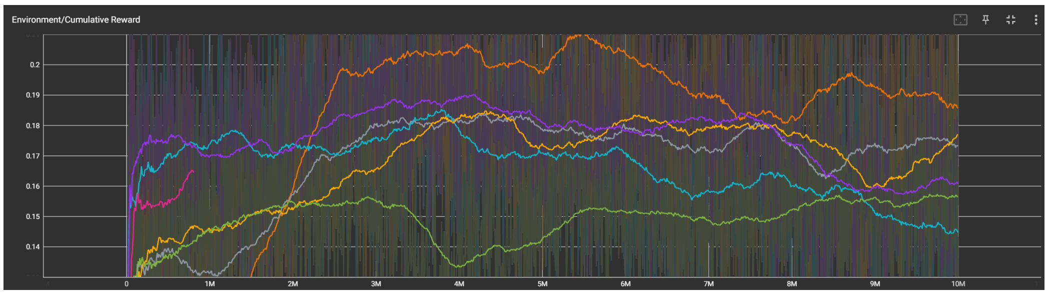


Figure 9.17: *tests 13.1-15.7*

Figure 9.18 shows the training for the Dragonmancer with varying states of game balance.

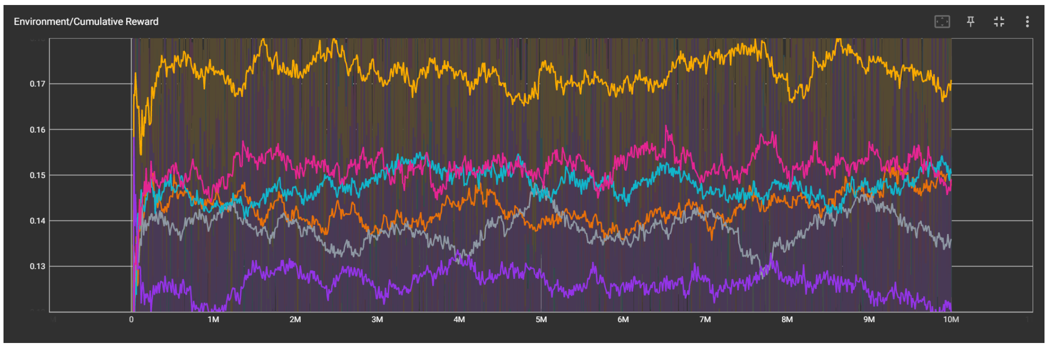


Figure 9.18: *tests 16.1-16.6*

9.1.9 Test 20.0: Conclusion and Search for Optima

When searching for the optima it became clear that the result would not be a representation of a valid balance state that translates to real gameplay. This is because the quality of the RL model was nowhere near resembling desired gameplay. Because of the randomization of the previous step in the training, the model had converged on a local minima, where it would continually create one type of tower to fill up all possible positions. This was in part an error in training the model without supervision for too many steps. This unfortunately means that the balance values that the genetic algorithm will find are not by any means an accurate representation of a valid balance state.

The genetic algorithm did generate a result by running a population of 100 genes over 10 generations, with a 0.1 mutation rate and elitism of 0.2 based on the findings in section 2.1.1 (Morosan 2019). The resulting gene from the simulations is unfortunately invalid. The result did produce a game state approaching the desired win rate. But this is mainly due to the two factions using the same starting tower and strategy resulting in an equal outcome regardless of the chosen faction. The final part of the workflow in the form of evaluations to validate the resulting game balance, was not carried out due to the invalid results and the redundancy of further validation. This is also in part due to using the GA to simulate varying balance states. Since the GA was actively sorting through balance states and favoring states resulting in an equal win rate this also shaped the training and thus created a bias in the model performance.

In figure 9.19 the episode length can be seen for all training runs. The length is very dynamic at first, this is mainly due to running fresh training runs starting from scratch every time. But this is also due to game mechanics parameter changes like wave times and speeds, etc, and also due to constant large changes in hyper-parameters, observations, and reward functions. As the initial RL model design and the game implementation approaches its final state, we can see the length converging on a time at around 250 steps on average. From here on there is a small increase at a very slow pace, ending at around 600 steps. The training was conducted over a month from around April 7th to May 10th. In figure 9.20 it can also be seen that the average reward performance of the model was stagnant at best. The randomness that can be seen in the average during May is due to the random state that the genetic algorithm is introducing. The results are fairly inconclusive but point to a lack of time and computational power.

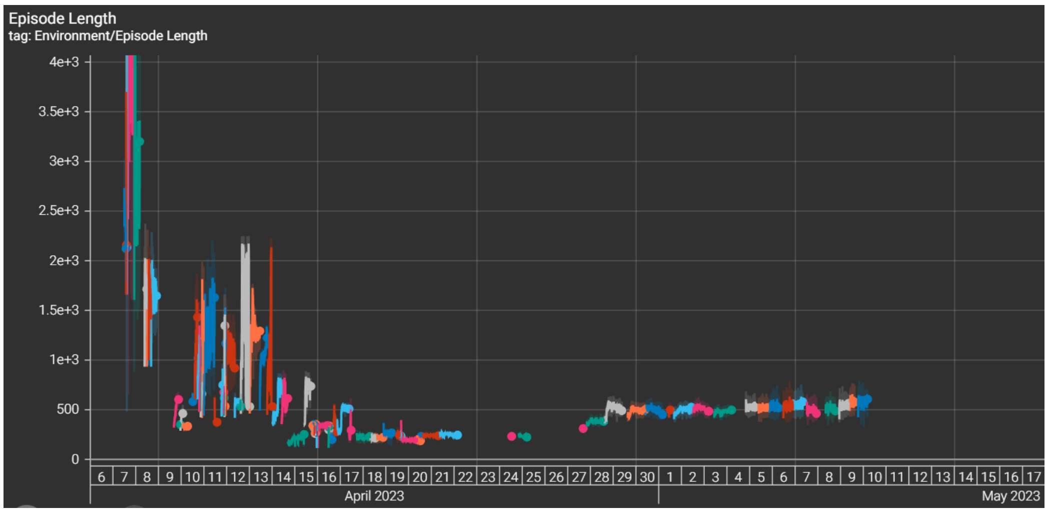


Figure 9.19: *Episode lengths for all training runs*

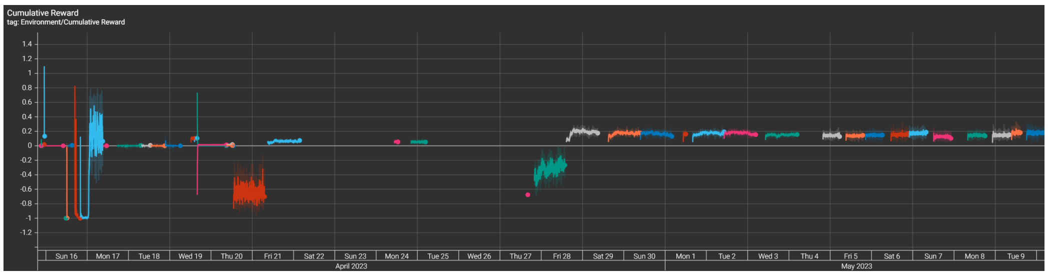


Figure 9.20: *Cumulative reward for all normalized training runs*

Chapter 10

Discussion

10.1 Development

10.1.1 Workflow and Process Integration

Despite the rather problematic results referenced in section 9, the methodology turns out to work great for testing during development. Several issues and exploits were discovered through the extensive testing that the methodology relies on. This is mainly due to running simulations at very high speeds. This part of the process can however also be achieved directly using the timescale function in Unity. It does however also create an environment that automatically simulates the game. This replaces a lot of manual debugging labor and a lot of time forcing certain scenarios. The problem however is that setting up the environment correctly can take a lot of time. This includes both integrating this process into the game development environment as well as tuning the hyperparameters and functions.

There are certain challenges to integrating this workflow in a multiplayer game. Most of these problems will not be encountered with a single-player game as it has to do with net code. The way this was solved in this case was through duplicate methods and conditionals. Mainly one conditional that determines if the test is run in training mode or in multiplayer game mode.

There is however one problem that can also translate to a single-player game, which is handling scenes. Sometimes scenes will rely on parts of other scenes and are required to be launched from that specific scene. You cannot run simulations with a game that requires user input to begin with, which is problematic for games with lobbies. You either need to work with conditionals or duplicate methods again to for instance log in or start the game without user input. A different solution was used for this development. In this case, a duplicate scene of the main game was used to avoid dealing with re-scripting the lobby. This is however also problematic as two almost identical scenes will need to be updated. Sometimes that is less work than potentially having to debug a ton of lobby net code.

An important discussion is whether or not this actually is an optimization of developer time. The manual labor of game balancing can quickly be replaced with hyperparameter tuning and reward shaping instead. This was the case during the development of this project. Reward shaping and hyperparameter tuning easily took away several weeks of development time during this project. To be fair, with years of experience this time would likely be decreased significantly. This of course also has to do with training iterations taking up a minimum of several hours causing progress in hyperparameter tuning and reward shaping to be very time costly.

10.1.2 Creativity and Optimization

How much of this process do we actually want to optimize? Part of game balancing is a creative endeavor and much like the discussion on generative AI and art, there are some similar valid concerns regarding human creativity. The process relies heavily on human input and only optimizes the balance for the vision of the designer. This should be viewed as a tool for the developer and a replacement for the manual labor of the task, and not the creative decision-making. The process should assist the designer in accomplishing the design goals. Without the human input of fitness functions and gene guidelines, the process does not accomplish anything. Finally as will be discussed in the limitations section, section 10.2, there are many problems that this framework will not work great at solving, where only humans can figure out the solutions - for now.

10.2 Limitations

10.2.1 Development Environment

There were some limitations discovered relating to standard development with GitHub and Unity LFS (Large File Storage). These problems relate to sharing large machine-learning models over the network. This might be much easier to handle in a CI/CD environment with a server. There might however be problems relating to accessing and modifying the model from several different users. This was not investigated nor documented during this project.

10.2.2 Imitation Learning

Imitation learning is a very powerful strategy, but it comes with certain limitations in this methodology. It can be difficult to optimize the training when using this framework. Demonstrations only work with models that are using the same size and composition of state and action vectors. This means that existing demonstrations become useless if it is decided to change the action or observation data. If these vectors are decided and never change, imitation learning should definitely be utilized to speed up the process significantly. Game testing can be used to record demonstrations for the training. The problem with vectors is however also a general issue with regular deep reinforcement learning with the ML-Agents framework. Once these vectors are defined it is not possible to continue developing existing models with new vectors. Unity is allegedly working on resolving this.

10.2.3 Generalizability

Training of OpenAI's DOTA 2 agent was 40.000 years in computation time and required millions of dollars in expenses. As is also reflected in the problems encountered in this project, this does raise the concern that this balancing method requires either a lot of resources or a very simple discrete problem to be beneficial. Imitation learning should help speed up the process significantly by an estimated factor of 4 according to EA. (Arts 2023) The hassle might however only be worth the effort on larger projects which at the end of the day favors larger game studios. This does not exclude indie companies from utilizing such a process but it will be important to assess the complexity of the problem and whether this will actually save time. This should however show accumulated returns over several games, as the models can either be applied to different but similar problems. Additionally becoming familiar with the workflow and the algorithms should make part of the process easier and faster.

There is also the concern regarding generalizability, this was tested in a solo development environment with a real-time strategy game with a fairly discrete action space. This automation process and workflow might not apply to games of other genres and in fact, the interview with Alberto from Triband, section 8.1, showed that the game they are currently working on would be problematic for several reasons. There will also be entirely different requirements for first-person shooters and massively multiplayer online games for instance. This should be explored further once a more solid foundation has been established in the field.

Another missing component regarding the genres and generalizability relates to the strategy design. This is not as crucial with only two factions, but as several more factions are introduced, more options are needed for each class to adapt to the opponent. This mainly relates to power curves and rock-paper-scissors game balance designs as discussed in the game balance section in section 3.2.3. Implementation of this might also have provided insights regarding the workflow. For instance, an evaluation process that evaluates several criteria. Having several more factions and several more towers, minions, and abilities would also significantly increase game complexity, which is already problematic in this case.

10.2.4 Game Engines

This methodology works great and seamlessly with Unity. But does it apply to other engines? There is definitely more work involved with applying it to other engines. You would have to manually set up an environment with a state, reward, and action loop that interfaces with a custom implementation of a neural net. This could of course still use libraries like PyTorch or Keras to perform the heavy lifting, but is much more problematic by far. Another layer on top of this would be limited documentation of the process. This adds another layer of work and problems to an already complex implementation that requires much time and effort. There are however tools like modl.ai 2.2.4 that can help with part of the process in other environments, like with their Unreal Engine plugin or their general API.

10.2.5 Action and Observation Vectors

Once the model has been trained using a specific action vector and observation vector, you cant extend the vector space. At least not to my knowledge. You can maybe use the model to teach another model (with imitation) but once this process is started you have to be careful that the game is close to its final stage. This can also be problematic with expansions to the game, if a completely new feature is added, this will not be reflected in the action and observation vectors, and it is likely that retraining the model is required. This however should not change the factors such as hyperparameters and reward shaping too much and the initial exploratory iterative process will be much faster. This is also where resources spent on imitation and training cannot be reused. This is however something that the team at Unity is looking into, allegedly.

10.2.6 Game Mechanics

Some balance problems have to do with the design of the game mechanics itself. This can be something that these algorithms will not have control over and the ability to change. In this case, the designer will have to intervene and solve the problem. This could for instance relate to the sniper problem from section 2.2.3, however, the process might still indicate this problem and help solve it faster by discovering it early and discovering that it is unsolvable with balance parameter optimization. Other problematic environments could be puzzle games, where the complexity of the tasks requires cognition or some type of more advanced computational intelligence. This is an area of the problem where the recent advances in computational intelligence via generative intelligence and large language models like OpenAI's GPT-4 could prove beneficial.

10.3 Bias

10.3.1 Representation of Human Behavior

One of the most important pitfalls of this method and a very important consideration is how well the agent translates to the real world. If the agent is not an accurate representation of human players, optimizing the game balance parameters with the agent is not relevant. This will yield settings that balance the game for agents but will give completely different results in the real world. Additionally, if the agents are not trained well enough, this is also an inaccurate representation of the problem and the GA will converge on parameters that are biased towards agents. In all cases, it will be difficult to train agents to behave exactly like humans, and there will always exist some sort of bias. This is one of the interesting topics to pursue in further work in this field.

10.3.2 Reward Shaping

It is difficult to provide the right set of rules that encourages an agent to learn. Another issue regarding reward shaping is the possibility of introducing bias in the agent. This is the reason why reward shaping can require many reiterations.

An example of biased and unwanted behavior from experience during this project was a rule that a match is tied if the round is less than 6. This was an effort to encourage early gameplay and avoid a result where agents win by spending all economy on minions. Agents however end up playing worse to never reach round 6. This is because the reward of getting a tie instead of a loss is 1. And the reward for getting a win instead of a tie is also 1. It is therefore easy for an agent that never wins to optimize for a tie instead. Self-play is a great alternative and provides a much more unbiased agent without shaping the rewards or creating conditions.

10.4 Evaluation

10.4.1 Balance Directions

There are many definitions, goals, and ideas about what good game balance is and should be as discussed in section 3.2.1. In this case, the algorithm is designed to optimize for an equal win rate between two factions. Nothing more and nothing less. There are several important game balance goals that are left out of this fitness function. For instance, this does not ensure that all towers are equally viable options, or that a sensible power dynamic exists between the towers. There are definitely ways to design the GA to take these factors into account. It was however not implemented in this project.

10.4.2 Using GA for State Variation

One issue with the data can be seen in figure 9.17 is that the graphs are much more stagnant and less dynamic than anticipated. This could be the result of either too shallow ranges for the genetic algorithm to explore or it could be that the algorithm converges too quickly on a stable set of genes. The rewards go up and down as expected but do not clearly indicate changes in the state. This could more easily have been analyzed if the change in state was synchronized to specific steps. This should be considered moving forward using this methodology to make analyzing and interpreting the results easier.

One of the largest concerns about the process is the time it can take to train a model to learn, not only the game itself which can be a challenging task on its own, but the entire search space of possible variables. This can become an incredibly large number of different possibilities to evaluate. While the model can definitely gain an understanding of the parameters without seeing all possibilities, this can still be a process that will take way too much time and computation power to be viable. This was for instance the case with this relatively simple game with around 60 balance variables.

Chapter 11

Conclusion

The thesis provided many insights into the process of automating game balancing and in this way served its exploratory purpose. It highlighted several issues and concerns regarding the workflow, and also several positive takeaways that benefit development, for instance contributing to debugging and general game testing workflows. There were however also several negative although beneficial insights and concerns.

One concern about the premise of this methodology is that the time spent optimizing game balance might instead be spent optimizing the training of agents' hyperparameters, reward functions, and training in general. It will likely depend on many factors such as the size of the project and team and the complexity of the game design and mechanics. The point of optimizing the game balancing with machine learning is mainly to reduce developer workload and time and to be able to achieve the same or better results at the same time. It is however a problem if a developer instead spends days tuning and optimizing hyperparameters and reward functions. This is one of the main concerns from the perspective of a solo developer exploring this workflow during the development of a game and this thesis.

Another large concern is how accurately the behavior of agents translates to real human gameplay. The evaluation assessing the resulting game balance with real players was not carried out. This thesis therefore unfortunately does not directly address the validity of the method. It can however be concluded that without an extensive amount of training, there is a large risk that the learned behavior is overfit to the reward function that was designed. This is a problem that self-play helps relieve, but as can be seen with the recent demise of DeepMinds's GO agent AlphaGo that was trained in a similar fashion, even in this extreme case of training the agent turns out to be deeply flawed.

Another problem with the automation of this process is, as discovered from the interview, that much of the game balancing has to do with the perception of the player and specifically how the game 'feels' in relation to play testing. This is a very difficult criterion to optimize for with automation, and for now, this will almost certainly require playtesting.

There are however many ways that automation can serve as a tool, for instance, to help narrow down the relevant parameter ranges or to search for unbalance in mechanics, rather than to search for the perfect state. This is likely a state that will depend on the perspective of human players and not on optimization. It all heavily depends on the type of game being developed.

It was not possible to validate the methodology because of the quality of the final model. It is however very likely that the process would yield a useful result with enough training in the case of this game. There are certain techniques that could help speed up the training times such as imitation learning and self-play with curriculum learning, but these processes also come with their own pitfalls and conditions as documented in this thesis.

Chapter 12

Future work

In the case of the experiment in this thesis, there simply was not enough time and available computational resources to solve the problem effectively. Moving forward this would be the most important direction to go to understand and validate the effectiveness of this automation process. In this context, it would also be very valuable to conduct this in the setting of a game studio. This would allow for a further understanding of how this process will integrate into a professional game workflow. Generally, future work in this field would also be more heavily focused on the process based on much more feedback from the industry, as this is important to create the foundation for experiments with studios adapting the process.

Usually, reinforcement learning is applied to very simple problems. Training time scales with game complexity and quickly becomes difficult to manage. It is beneficial to look for places where the problem can be simplified. Through game tests, the designers can learn some patterns from the game tests to make machine learning more efficient. Based on the proposed methods, we can use a reinforcement learning model to train an AI to play but gather minimal data about patterns of play in order to simplify the model. For instance, removing the objective of evaluating maze positions in this case. Initial tests could discover the most common maze builds. This way we reduce the decision space significantly and thus the training time. By learning how players generally place their units we can generalize and assume similar patterns. This can reduce the number of possible scenarios for the machine learning model to look through drastically. The remaining search space is however still huge and of course, this also limits the model in finding problematic outliers and introducing a possible bias from the players in the game tests.

It is a problem if a developer spends days tuning and optimizing hyperparameters and reward functions instead of working on the game balance. One of the links in this chain that could very well be optimized is the hyperparameter tuning. There are several papers done on optimizing this part of the process for several machine learning problems using an automated process for hyperparameter tuning (HPO). (Zhang et al. 2021) (Yang and Shami 2020) This was also included as a potential step in the workflow, but was not explored in this thesis. This is therefore an obvious and interesting next step.

Bibliography

Arts, Electronics (2023). *SEED: Deep Learning - Imitation Learning with Concurrent Actions in 3D Games*. Accessed: 2023-05-01. URL: https://www.ea.com/seed/news/seed-imitation-learning-concurrent-actions.

Becker, Alexander and Daniel Görlich (2020). "What is game balancing?-an examination of concepts". In: *ParadigmPlus* 1.1, pp. 22–41.

Berner, Christopher et al. (2019). "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680*.

Beyer, Marlene et al. (2016). "An integrated process for game balancing". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, pp. 1–8.

Felder, Dan (2015). *Design 101: Balancing games*. Accessed: 2023-02-04. URL: https://www.gamedeveloper.com/design/design-101-balancing-games.

Google (2023). *Firebase Realtime*. Accessed: 2023-02-19. URL: https://firebase.google.com/.

Granberg, Aron (2023). *A* pathfiding Project*. Accessed: 2023-02-19. URL: https://arongranberg.com/astar/front.

Jaime Griesemer, Bungie (2010). *Bungie at GDC 2010: Changing the Time Between Shots for the Sniper Rifle from 0.5 to 0.7 Seconds for Halo 3*. Accessed: 2023-05-01. URL: https://www.youtube.com/watch?v=8YJ53skc-k4&ab_channel=GDC.

Kazuko Manabe Shigeru Awaji, Square Enix (2010). *Square Enix at GDC 2010: Balancing Nightmares: An AI Approach to Balance Games with Overwhelming Amounts of Data*. Accessed: 2023-05-01. URL: https://www.youtube.com/watch?v=X8nnCPl_uwc&ab_channel=GDC.

Li, Yuxi (2017). "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274*.

MathWorks (2023). *What Is The Genetic Algorithm*. Accessed: 2023-02-14. URL: https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html.

Mirjalili, Seyedali and Seyedali Mirjalili (2019). "Genetic algorithm". In: *Evolutionary Algorithms and Neural Networks: Theory and Applications*, pp. 43–55.

Morosan, Mihail (2019). "Automating game-design and game-agent balancing through computational intelligence". PhD thesis. University of Essex.

OpenAI (2017). *OpenAI: PPO*. Accessed: 2023-02-14. URL: https://openai.com/research/openai-baselines-ppo.

Pfau, Johannes et al. (2020). "Dungeons & replicants: automated game balancing via deep player behavior modeling". In: *2020 IEEE Conference on Games (CoG)*. IEEE, pp. 431–438.

Roosendaal, Ton (2023). *Blender*. Accessed: 2023-04-13. URL: https://www.blender.org/.

Schulman, John et al. (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Technologies, Unity (2023a). *Netcode for Gameobjects*. Accessed: 2023-02-19. URL: https://unity.com/products/netcode.

— (2023b). *Unity*. Accessed: 2023-04-13. URL: https://unity.com/.

— (2023c). *Unity Gaming Services*. Accessed: 2023-02-19. URL: https://unity.com/solutions/gaming-services.

Technologies, Unity (2023d). *Unity ML-Agents*. Accessed: 2023-04-13. URL: `https://unity.com/products/machine-learning-agents`.

Torrey, Lisa and Jude Shavlik (2010). "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, pp. 242–264.

Vanschoren, Joaquin (2019). "Meta-learning". In: *Automated machine learning: methods, systems, challenges*, pp. 35–61.

Wang, Tony Tong et al. (2022). "ADVERSARIAL POLICIES BEAT SUPERHUMAN GO AIS". In.

Yang, Li and Abdallah Shami (2020). "On hyperparameter optimization of machine learning algorithms: Theory and practice". In: *Neurocomputing* 415, pp. 295–316.

Zhang, Baohe et al. (2021). "On the importance of hyperparameter optimization for model-based reinforcement learning". In: *International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 4015–4023.

Appendix A

Appendix

A.1 Evaluation

A.1.1 Interview Questions

Introduction

- Ask permission to record.
- Permission for using his points in the thesis.
- Explain the interview process and what we will talk about.
- Talk a bit about who you are and what you do

General Game Balance

- How do you manage game balancing (you or studio)?
- How much time and resources are spent on balancing?
- How many variables are you often tuning, do you have an overview of this?
- Do you have any special workflow you follow?
- Is this an area that is problematic for the studio or for you?

The Process Workflow

- Explain the workflow I am proposing here.... Any questions?
- Basically I have been researching this process on automating game balancing with machine learning. I've done so while creating a small tower defense game to explore the workflow. This is the model I have created and followed. I use a combination of reinforcement learning and genetic algorithms to simulate games with different balance states that are generated by the genetic algorithm.
- Show and explain the workflow model.
- What are the challenges with this workflow from your perspective? The positives?

- Could you see a process like this being used in a development pipeline? What would it look like in one of your games maybe?
- Do you see any specific problems or needs this can solve?
- Are you still balancing once the games are released?
- Do you think this is something the dev team could be used to working with? Or do you see any challenges here?

AI in general

- General stance on AI tools in game development and other creative fields?
- Opinion on the level of influence from the designer vs optimization?

A.1.2 Test Design Validity

This test will be outside the scope of this thesis due to time and computation limitations, but it is an important aspect of validating the process. This section will cover some of the important considerations when evaluating the proposed method.

There are four critical categories to consider when designing a test of this proposed process:

1. Users

- Some users are more skilled than others in the genre - Some users might not understand the concept at all and some have hundreds of hours sunk into similar game concepts.

2. Multiplayer and participants

- Multiplayer game consideration - there might not be an opponent when they want to test the game.
- Playing with malicious intent to ruin tests
- Polish. The game must be polished to get people to play, maybe even a trailer.
- The problem of matching skill levels without matchmaking rating. This could be based on own players reporting familiarity with the genre and matching against others with the same rating. However, this is likely not a viable option as it requires even more players when matchmaking.

3. Exploitation

- Early tests to account for game-breaking bugs and exploits to ensure reliable results in balancing. This is in part done through reinforcement learning. The agent should likely discover these.

4. Test goals

- Duration. Game duration of approximately 15-20 minutes?
- Winrate. Balanced winrate? Or rock-paper-scissor without 100% win rates? only two factions.
- Data. Hp left, gold left, towers placed, some sort of player identification, GAME ID

Test Design

- Usability test to account for learnability?
- Maybe evaluate the single-player aspect of the game first?
- Questionnaires of people who played to evaluate the user
- Experience of the game in relation to balance
- Expert interviews with game designers?
- Engagement?
- We should know if the game was unbalanced before. In order to know if the balance was improved?