Westfälische
Wilhelms-Universität
Münster

Department of
Information Systems and Statistics
Prof. Dr. Heike Trautmann

Project Documentation

Project Seminar: Balancing in Games

Supervised by: Dr. Mike Preuß,
Nicolas Pflanzl

Submitted by: Aleksandr Agureikin <a_agur01@uni-muenster.de>,
Alexander Anokhin <a_anok01@uni-muenster.de>,
Marlene Beyer <marlene.beyer@uni-muenster.de>,
Christoph Laenger <c_laen01@uni-muenster.de>,
Felix Nolte <felix.r.nolte@uni-muenster.de>,
Marcel Renka <marcel.renka@uni-muenster.de>,
Martin Rieger <m_rieg06@uni-muenster.de>,
Jonas Winterberg <jonaswinterberg@uni-muenster.de>

Deadline: 2016-03-18

Abstract

This documentation will show in detail the outcomes of a research oriented project seminar that investigated means of non-manual or semi-automatic methods of game parameter balancing. The findings are based on a game prototype, provided by Blue Byte GmbH, one of the leading computer and console games developers in Germany. The documentation will provide insights on the game itself as well as the applied modifications. Based on an extensive literature analysis in the field of Modern Gaming and Artificial or Computational Intelligence, an evolutionary algorithm was applied in order to discover an optimal set of game parameters. It will be shown that results from the algorithmic experiments are good candidates for a positive human player experience. A detailed data analysis will discuss the results delivered by the algorithm. Additionally, challenges and natural limitations of the conducted approach will be discussed and an outlook of potential future fields of research will be proposed.

Contents

List of Figures

List of Tables

Acronyms

2DR	2D Roguelike
AI	Artificial Intelligence
ANN	Artificial Neural Networks
ARPG	Action Role-Playing Game
BE	Balancing Environment
BPMN	Business Process Model and Notation
C#	C-Sharp
CI	Computational Intelligence
CS:GO	Counter-Strike: Global Offensive
CSV	Comma Separated Values
CT	Counter-Terrorist
D3	Diablo III
DDA	Dynamic Difficulty Adjustment
DDO	data-driven optimization
DVD	Digital Versatile Disc
EA	evolutionary algorithm
EC	Evolutionary Computing
FPS	First-Person Shooter
FSM	Finite State Machine
MCTS	Monte Carlo Tree Search
ML	Machine Learning
MMORPG	Massively Multiplayer Online Role-Playing Game
MSE	Mean Squared Error
NPC	Non-Player Character
PCG	Procedural Content Generation
PS	Project Seminar
PU AP	player unit attack power
PvP	Player vs. Player
R	R Programming Language
RTS	Real-Time Strategy
SC2	StarCraft II
T	Terrorist
ZU AP	zombie unit attack power
ZVG	Zombie Village Game

1 Introduction

Modern computer games grow in space and complexity. In order to ensure an efficient and profitable production of games, new approaches of automatic design and development are required. In this context, this project seminar investigated means of non-manual and algorithmic methods for game parameter balancing.

The research presented is both, theory- and data-driven. From the theoretical perspective an extensive literature analysis of relevant research streams has been carried out prior to the beginning of the practical part. The subsequent practical (data-driven) part is based on a game prototype provided by Blue Byte GmbH, one of the leading computer and console games developers in Germany.

Chapter 2 of this documentation introduces the research approach and working methodology of this Project Seminar (PS). Chapter 3 describes the conducted literature analysis. Chapter 4 briefly depicts the organizational aspects of the PS team. Chapter 5 offers a description of the games that were subject to research. Chapter 6 illustrates in detail the implemented methodology, the *Balancing Environment (BE)* and how it was constructed. Chapter 7 outlines technical restrictions of the game prototype and general challenges of the research setup. Chapter 8 documents the process of manual balancing that had been conducted in parallel as a comparison. Chapter 9 documents the conducted simulations that served as inputs for the data analysis. Chapter 10 provides analytic tools to find good, balanced solutions, get a deeper understanding about the game, and overcome time constraints for simulations. Chapter 11 provides a more generic process model which was derived from the conducted procedure and the experience gained. Chapter 12 discusses the main findings as well as limitations of the outcome of this PS and offers an outlook for future research.

All of the content which can be seen in this documentation can be found in the GitLab repository or the attached Digital Versatile Disc (DVD). There are four main directories in the repository. All of the content of the two different games are in the respective directories `2DRoguelike` and `ZVG`. All produced Comma Separated Values (CSV) logs can be found in the `CSVLogs` directory, once again ordered by the game and then by type of file. The `RCode` directory contains all code which was written in the R Programming Language (R) and used for the data analysis.

All the scripts can be found in git repository, however not all of them are relevant and developed during PS. Therefore, the below list outlines the main structure of the repository and describes briefly the content of developed folders.

- `ZVG/` - folder with ZVG Unity game
- `ZVG/Assets/GameInterface` - main folder with produced code

  `Algorithms/` - folder with optimization algorithms

    `EA/` - folder with different strategies for evolutionary algorithms

  `BalancingSuite/` - folder includes balancing suite script

  `Behaviors/` - folder includes developed AIs for the player

  `Games/` - folder includes developed `Game` entities

  `Goals/` - folder includes developed game goals

  `GUI/` - folder includes developed user interface objects

  `Misc/` - folder includes miscellaneous objects, like `Feature` or `Solution` entities

  `Misc Scenarios/` - folder includes miscellaneous scenes which were not used for balancing

  `Placement/` - folder includes placement scripts, they are used to place variable number of player units in a scene

  `Scenarios/` - folder includes scenes which were used for balancing

  `Utils/` - folder includes miscellaneous scripts, like developed statistical functions

- `2DRoguelike/` - folder with 2DR Unity game
- `2DRoguelike/Assets/Completed/Scenes` - folder with scenes which were used for balancing
- `2DRoguelike/Assets/Completed/Scripts` - main folder with adapted code

  `Algorithms/` - folder with adapted optimization algorithms

  `BalancingSuite/` - folder includes adapted balancing suite script

  `Controllers/` - folder includes developed AIs for player and zombies

`Games/` - folder includes developed `Game` entities

`Goals/` - folder includes developed game goals

`Misc/` - folder includes adapted miscellaneous objects, like `Feature` or `Solution` entities

`Utils/` - folder includes adapted miscellaneous scripts, like developed statistical functions

- `CSVLogs/` - folder with generated logs for both games, includes population and session logging
- `RCode/` - folder with produced R code

  `images/` - folder includes produced plots and images

2 Research Approach

According to the module compendium of the Master of Science in Information Systems study track at the University of Münster, a PS intends to apply material and methods that were introduced in former method tracks in a practice-oriented manner to solve a realistic and complex problem. The project is often performed in collaboration with a partner company from the industry, which varies from term to term [WWUa].

This PS was offered by the group of Information Systems and Statistics, which is focusing its primary research on continuous and combinatorial optimization problems with the integration of statistical techniques into multiobjective metaheuristics such as evolutionary algorithms [WWUb]. The collaboration partner from the industry was Blue Byte GmbH, one of the leading developers for computer and console games in Germany. The collaboration has been established by the supervisor prior to the beginning of the PS.

The project seminar itself consisted of several phases. During the first phase, each of the students was assigned with the literature search and analysis of a different research stream related to 'Modern Approaches for Balancing in Computer Games'. The findings of phase I are outlined in detail below in chapter 3. Based on the findings of the literature search, the PS group selected an approach to investigate an identified research gap.

The research environment (see chapter 5) was set around a provided prototype game that had been built prior to the beginning of the PS by Blue Byte GmbH. The prototype is bound to Unity, a game engine that allows to build 2D and 3D games for different platforms.

Based on this setup, a specific research design (working concept) was formulated, constructed and deployed during the second phase.

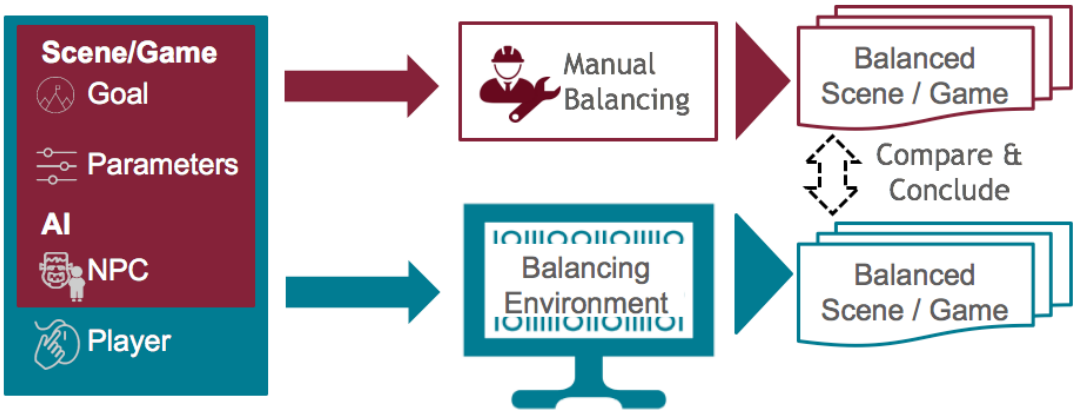


Figure 1: Working Concept: Automated and manual game balancing.

Based on a classical input-output-model, the working concept shows a set of inputs that are applied within two different methodologies of which both results were compared and analyzed. The inputs were conceptualized as general as possible and intend to match the provided prototype. On its highest level, the prototype build within Unity is an entire game construct or a subset of the game, i.e. a level or a scene. Each scene/game can have a different goal, which in the context of this PS is a quantification of the general purpose (e.g. challenge) in the environment of the game, i.e. winning condition (e.g. surviving for 60 seconds). The choice for a goal has wide implications on the remaining construct. The goal can vary, but constitutes in this sense as an independent variable.

Parameters are formalized values that define the workings of the game. The number of game parameters vary from game to game but can be very large even for small games. Most game parameters are directly or indirectly interrelated with each other. This complexity can bring about an unbalanced player experience. Hence, balancing a game can often involve tweaking the game parameters. Since a change of one parameter can propagate through all other parameters, some choice has to be made regarding which parameters should be subject for tweaking. This selected set of game parameters is the decision variable in the context of this research model and the value ranges of the respective parameters define the boundaries within the balancing can be tested.

Artificial Intelligence (AI) in the context of this PS refers to either Non-Player Character (NPC) behavior or simulated player behavior. The configuration of the

NPC behavior is attributed to the scene/game, while the player AI is an external representation that acts as an agent of how a human would play the game. Similar to the choice for a goal, both AIs have also wide implications on the remaining construct, but are also kept as independent variable, in the sense that the conducted research at hand did not seek to find an optimal human-like representation of player behavior.

The depicted set of inputs work in interrelation and are tested manually as well as non-manually by means of the BE. In both approaches, data is collected and analyzed. The entire approach is translated into a generic balancing process model that intends to guide future research and practitioners in reproducing this semi-automated methodology.

3 Literature Analysis

In the first phase of the PS, each of the participants was assigned with searching and analyzing literature in a particular field of research. Each of those research streams is targeted towards identifying relevant and applicable approaches for non-manual means to balance computer games. This research analysis had been conducted based on the following definition of game balancing:

Definition 1 *Game Balancing. Small changes to a game (basically parameter adaptation) in order to satisfy predetermined goals.*

The results of each literature analysis were presented and discussed in the context of weekly appointments over a period of four weeks.

The following sub-chapters provide a brief overview of the results for each field. They are structured in the chronological order of presentations held during the appointments. Table 1 gives an overview of the research streams (topic), the person who investigated the research stream, and the date of presentation.

Topic	**Group Member**	**Presentation Date**
Goals of Game Balancing	Christoph Laenger	10/29/2015
Data-driven Optimization Methods	Marlene Beyer	10/29/2015
Practical Examples: Unbalanced Games	Marcel Renka	11/05/2015
Practitioner's Perspective on Game Balancing	Aleksandr Agurekin	11/10/2015
Artificial Intelligence Methods: An Overview	Alexander Anokhin	11/10/2015
Procedural Content Generation	Martin Rieger	11/12/2015
Dynamic Difficulty Adjustment	Jonas Winterberg	11/12/2015
Balancing Applications in Other Domains	Felix Nolte	11/19/2015
Game Balancing by Computational Mechanism Design	Vanessa Volz (Tutor)	11/23/2015

Table 1: Presentation Topics per Student.

3.1 Goals of Game Balancing

This presentation focused on the twelve types of game balances featured in *The Art of Game Design: A Book of Lenses*[Sch14]. When a game designer intends to balance a game, each of these types should be taken into account. For each type, a goal can be formulated, e.g. with regards to the type *Challenge vs. Success* a game designer may decide to favor challenges instead of letting players be successful without much effort. In the following, all twelve types are explained in detail.

1. **Fairness**: It is necessary to distinguish between two kinds of games with regards to fairness. In symmetrical games it is simple to ensure fairness by giving equal resources and power to all players. The skills and strategies individual players bring to the game are the deciding factor. This is different in asymmetrical games, where it is possible and often desirable to give opponents different resources and abilities. For example, in order to level the playing field for two players of different skill levels, the less skilled player may be given access to more resources.

2. **Challenge vs. Success**: How challenging a game is, is related to the skill level of the player who is playing the game. Ideally, the game designer strikes a balance between these two characteristics in such a way that the player is neither bored nor anxious. However, a game designer may decide that their game provide players with a tough challenge on purpose.

3. **Meaningful Choices**: Choices are meant to have a real impact on what happens next and how the game turns out. Meaningless choices, e.g. different racing cars that all drive the same, as well as dominant strategies, e.g. a racing car that strictly is better than all others, may have a negative impact on game balance and should be avoided.

4. **Skill vs. Chance**: Games of skill tend to be more serious while games of chance tend to be more relaxed. It all comes down to the target audience and their expected preferences.

5. **Heads vs. Hands**: This is about how much of the game should involve doing a challenging physical task and how much should involve thinking. Again, it is important to know what the target audience is going to look like and to understand what they expect the game to be or not to be. Communicating

the balance between physical tasks and thinking involved is equally important as getting the balance right.

6. **Competition vs. Cooperation**: In most games this can be trivial, e.g. a single player game usually involves neither. However, this can be very complex in games that feature both, e.g. Massively Multiplayer Online Role-Playing Games (MMORPGs) featuring a Player vs. Player (PvP) environment and an economy where players may engage in trading.

7. **Short vs. Long**: If a game is too short, players may not get a chance to develop and execute meaningful strategies. If it is too long, players may get bored or may be put off by the time commitment required to play the game.

8. **Rewards**: Common types of rewards include points, powers, and resources. It should be noted that - to players - what felt rewarding before may not feel rewarding now. This can be avoided, e.g. by making rewards variable and adjusting them according to the progress of the player.

9. **Punishment**: Common types of punishment include the loss of points, powers, and resources. Taking risks can be exciting, especially if there is a reward to be gained. Players may value rewards more if there is a risk involved in gaining them.

10. **Freedom vs. Controlled Experience**: This is about the amount of control a player should be given. Game designers who want each player to have a similar experience may tend towards providing a controlled and therefore consistent experience.

11. **Simple vs. Complex**: The terms used here can be misleading. A simple game can be boring or elegant. A complex game can be confusing or rich. An example of how to make sense of this type of balancing is a simple set of rules which enables complex strategies: *easy to learn, hard to master*.

12. **Detail vs. Imagination**: Giving detail the imagination can use is of essence. The more details are provided, the less gaps can be filled by the imagination of the players. Omitting detail on purpose may give greater results than trying to relieve the players of their imagination.

The types of balancing mentioned above have been utilized during the balancing approaches conducted in this project seminar as can be seen in chapter 5.2 table 8

and chapter 5.3 table 9.

3.2 Data-driven Optimization Methods

This presentation gave an overview of core concepts of optimization techniques that might have to be considered during the PS. Stochastic and heuristic optimization as well as evolutionary algorithms (EAs) and practical implications of data-driven optimization (DDO) were discussed.

Being at variance to the original optimization logic where we determine the exact location of a local or global extreme (minimum or maximum values), DDO techniques enable decision makers to make informed decisions using the limited available historical data, and provide them with certain optimal guarantees. Optimization problems are concerned with finding the values for one or several decision variables that meet the objective(s) the best without violating the constraint(s). However, precise knowledge of real-life optimization problems rarely is available, inputs are not always reliably correct, and especially when it comes to analyzing big data sets performance becomes a challenge. Thus, a mathematical framework that is well-suited to the limited information and resources available is needed [Mar06].

Stochastic optimization techniques use probabilistic methods to solve problems. They cope with inherent system noise and models or systems that are:

- highly nonlinear,
- high dimensional, or
- otherwise inappropriate for classical deterministic methods of optimization [GHM12]

Stochastic optimization may be divided into three main approaches according to [Mad60]:

1. **Models with here-and-now decision**: The decision is made before the realization of parameters takes place. Optimality is reached mostly through the expected value of the *objective function* over all scenarios.

2. **Wait-and-see problems**: It is possible to wait for the decision until the parameters are realized. The goal is to calculate the optimal value for every possible scenario.

3. **Expected value**: Also here the decision is made before the realization of parameters. In contrast to the here-and-now decision the expected value is determined for each *parameter*. The problem is then solved using these values.

An example for a stochastic optimization problem would be the minimization of production costs of a company without knowing the exact demand for the product.

Heuristic techniques or *heuristics* refer to the iterative search for satisfactory solutions that are not necessarily optimal. The search process starts off with an arbitrary initial solution, then iteratively produces new solutions by some generation rule and evaluates these new solutions to finally report the best solution found during the search process [Mar06]. Different classes of heuristics exist, among which *construction methods* and *local search methods* belong to the best known. Construction methods use a step-wise approach to build a solution to the problem where the best choice at each step highly influences the solution. Contrarily, local search methods do not explore the search space systematically, but improve an initial solution in a progressive manner. Other well-known heuristics are: decomposition methods, inductive methods, and reduction methods [MR11].

Being part of Computational Intelligence (CI), evolutionary computation are heuristics that mimic the *survival of the fittest*-principle from biology by using a population-based approach: an initial population is created and evaluated and by means of recombination and mutation a selection of parents is used to produce a new generation of solutions. Different techniques of the evolutionary approach exist, using different representations for the individuals of a population, for example:

- Genetic Algorithms, using binary strings
- Evolutionary Programming, using Finite State Machines (FSMs)
- Evolution Strategies, using real-valued vectors
- Genetic Programming, using trees

Since evolutionary algorithms (EAs) have been used heavily in this PS, further details are outlined in 6.4.

Often the evaluation of parameter vectors is time-consuming and/or expensive as for example described in [MS14]. A *surrogate model* is a method used when an outcome of interest cannot be easily measured, so an approximation of the computationally expensive objective function is used instead [BDF+99]. The idea is to

describe the relation between control variables and the observation through a model or function. This is done by first creating the experimental design, gathering sample data with which the surrogate is constructed and afterwards validated. If this process was not satisfactory, it can be repeated until the desired results are achieved. Sample methods that can be used for this are: First Order Response Surface, Generalized Additive Model, Random Forests, and Kriging.

As a last topic, a general recommendation regarding the approach of new optimization problems was given: according to [WM97] there is no optimal optimization method. Hence, to tackle a new problem it makes sense to either adapt the algorithm or the problem design. In the first case, an existing algorithm is adapted to the problem in its current form. The latter case implies an appropriate formulation of the problem for an existing algorithm.

At the end of the presentation, the main takeaways were summarized: The techniques presented are used to guarantee certain optimality for decision makers. Mostly we do not even know all variables that determine the output to be optimized - this is where stochastic and heuristic methods can be applied. To improve or enable complex evaluation procedures, surrogate models are an appropriate tool to consider. Finally, there is no optimal optimization technique: one has to evaluate and adapt them.

3.3 Practical Examples: Unbalanced Games

This presentation covered examples of unbalanced game content from three different games:

- StarCraft II (SC2) (Real-Time Strategy (RTS))
- Diablo III (D3) (Action Role-Playing Game (ARPG))
- Counter-Strike: Global Offensive (CS:GO) (First-Person Shooter (FPS))

The different games were chosen in order to give a diverse overview of game types. Additionally the aim was to showcase popular games which are played in a competitive environment. This ensures that they're updated regularly in order to keep the number of players high and keep the tournaments interesting.

Two perspectives were analyzed for this case. The first one being the view of the game designer, who may ask himself:

"What changes need to be made for the **intended** way to play the game?"

The second view is the one of the player, who may ask himself:

"How can I **break** this game?"

Game Theory Basics In order to establish a more common understanding of what is balanced in a video game and what is not, game theory basics were used to showcase the difference. The prisoner's dilemma was used to introduce the idea of game theory. The prisoner's dilemma describes a situation where two individuals have to choose a strategy, whether to confess to a crime or to keep quiet and say nothing. To show the outcome of their individual strategies a payout matrix is used:

		Prisoner B	
		Keep quiet	Confess
Prisoner A	Keep quiet	$-1, -1$	$-10, 0$
	Confess	$0, -10$	$-5, -5$

Table 2: Prisoner's Dilemma Payoff Matrix.

The idea of this example is to show that there is a strictly dominant strategy, which is a strategy that will always guarantee a higher payoff then any other available strategy, meaning it doesn't matter which strategy the other person chose. In this case it would be the strategy *Confess* for both prisoners. The explanation for this is that no matter what strategy Prisoner B chooses Prisoner A will always have a higher payoff with *Confess*. The mathematical explanation of *Confess* vs. *Keep quiet*: $0 > -1$ and $-5 > -10$.

Strictly dominant strategies in video games would mean that a team or player will always do the same in a game, which could be seen as unbalanced, especially if the winning percentage of using this strategy is very high. An exemplary CS:GO round could be displayed by this matrix:

		Counter-Terrorists	
		Defend A	Defend B
Terrorists	Attack A	$0.25, 0.75$	$0.6, 0.4$
	Attach B	$0.6, 0.4$	$0.25, 0.75$

Table 3: Generic CS:GO Round.

Counter-Terrorists (CTs), who need to defend bombsites from Terrorists (Ts), have

a high win chance of 75% if the Ts focus on the same bombsite. However, when the Ts decide to attack the other bombsite the win chance decreases to 40%. In this case there is no strictly dominant strategy available for either side. But as we will later see depending on the exploitation of unbalanced game mechanics this matrix can change, enabling a strictly dominant strategy for one faction.

StarCraft II The RTS game *StarCraft II: Wings of Liberty* was released on the 27th July 2010 by Blizzard Entertainment. On the 12th March 2013 came the first expansion pack *Heart of the Swarm* and on the 10th November 2015 was the second one released, with the name *Legacy of the Void*. The game consists of a single-player campaign and a competitive multiplayer environment with a ladder-system. Three races are available for playing: Terran, Protoss, and Zerg. Blizzard Entertainment describes the game with the following words:

> "Wrangle some resources, build your base and raise an army to make your enemies run and hide [Blie]."

The unbalanced game mechanic in this case was a unit called *Warhound*. It was a unit that could be build by Terrans. It was introduced as an anti-ground unit with bonus damage against mechanical units with is special ability *Haywire Missiles*. The unit was intended to be released with the second expansion package *Heart of the Swarm*. Players used the unit in the early game since it had low costs, but still high damage output and a high amount of health. Blizzard Entertainment decided against the release of the unit into the game, after letting players test the beta version of the second expansion pack[Blid].

In a game theory context one could put the usage of the *Warhound* unit into this payoff matrix:

Build a Warhound?		Terran B: Yes	Terran B: No
Terran A	Yes	$1, 1$	$2, 0$
	No	$0, 2$	$0, 0$

Table 4: Should I build a Warhound?.

In this case it is quite obvious that not building a *Warhound* unit or multiple *Warhound* units is a mistake. Building the unit always results in a positive payoff for

either player. Therefore building the *Warhound* unit would be the strictly dominant strategy in this game.

Diablo III The ARPG was released in its first version on the 15th May 2012. Two years later on the 25th March 2014 the first expansion package called *Reaper of Souls* was released. For the purpose of discussing unbalanced game content the newest version of the game was used. The game features many different play modes like:

- Main campaign, which is the main story line of the game.
- Bounties, which are missions the player has to complete in order to get a reward.
- Rifts, which are randomly generated dungeon where the player has to kill enough monster to spawn a strong boss monster.
- Greater Rifts, which are similar to rifts but they have better rewards for the player and are more difficult to complete.

The game can be played in two different main modes. The non-seasonal and seasonal mode. In the seasonal mode the characters stay for three to five months and the player has to start over. The characters are then transferred to the non-seasonal mode, where characters and items are stored indefinitely. A player can choose to play the game alone or with a party and can compete on leaderboards for single-player mode or multiplayer mode. The main objective is to increase the paragon level of a character, which gives the character increased stats. Six classes are available for a player in Reaper of Souls. Three melee classes Barbarian, Crusader, and Monk, as well as three ranged classes Demon Hunter, Witch Doctor, and Wizard.

The main issue many players had with the game in fourth season of Diablo III was the competition between single-player and multiplayer game mode. Players who grouped up gained a massive amount of bonus experience for their characters and were therefore able to increase their paragon levels at a much higher rate than those players who played solo. In addition to the experience bonus groups also gained a bonus to finding gold and items in the game[Blia]. Additionally, players were able to equip items that further increase the gain of experience, as can be seen on Figure 2.

Once again this example can be put into the context of game theory. In this

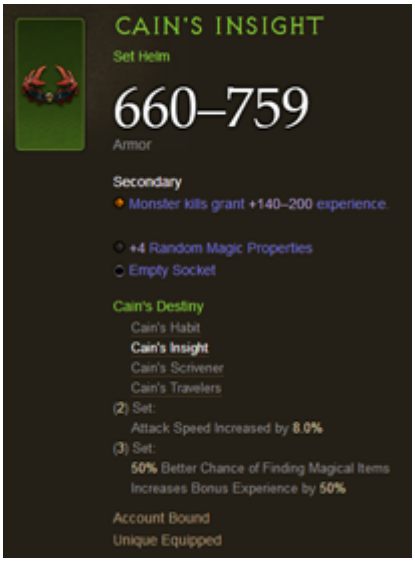


Figure 2: Diablo III Item: Cain's Insight.

payoff matrix 100 stands for a normal experience gain, i.e. 100%. 130 and 270 would in this case mean a +30% or +170% experience gain. The example in Table 5 shows that playing in a group of four players (Multi (4)) is beneficial in order to increase the experience. Using additional experience items (M4 w/ XP) can further increase the experience gain, almost doubling or tripling it.

Experience in %		Player B		
		Single	Multi (4)	M4 /w XP
Player A	Single	100, 100	100, 130	100, 270
	Multi (4)	130, 100	130, 130	130, 270
	M4 /w XP	270, 100	270, 130	270, 270

Table 5: How do I gain experience most efficiently?.

A second issue many players had with the game in season three was the diversity of classes used at the top of the leaderboards. In season three no Barbarian or Monk was present in the Top 100 of the 4-player leaderboards[Blib; Blic]. The issue was heavily discussed in the Diablo III forums and even Blizzard Entertainment employees talked about these issues. John Yang (Game Designer) said that he "will never be happy with build diversity until every build is completely equal in power level for both the top players and average players.[Yan]"

Counter-Strike: Global Offensive The FPS was released on the 21th August 2012. The game was developed by Hidden Path Entertainment and Valve Corporation. It is a round-based FPS in which two factions, Ts and CTs, have to fight against each

other. The objective of the Ts is to plant a bomb at one of the two bomb sites on the map, while the CTs have to hinder them. Ts and CTs have access to different kind of weaponry and the overall goal of a game is to win 16 out of the 30 rounds played. In general the maps which are played competitively are often sided towards one of the two factions.

One example for the abuse of an unbalance game mechanic was a so-called triple boost, where three players were climbing on top of each other. This way the player on top is able to overlook a huge part of the map *de_ overpass*. At the Dreamhack Winter 2014 tournament the Swedish team Fnatic was playing against the French team LDLC. The first half of the game ended 12-3 in favor of LDLC. Fnatic was able to turn it around and win the game with 16-13. They abused the game mechanic in order to kill the players of the French team without them being able to see the shooter[Dre].

Using this strategy in context of game theory can result in the following payoff matrix:

		Counter-Terrorists		
		Defend A	Defend B	Boost
Terrorists	Attack A	$0.25, 0.75$	$0.6, 0.4$	$0.01, 0.99$
	Attach B	$0.6, 0.4$	$0.25, 0.75$	$0.01, 0.99$

Table 6: de_overpass Payoff Matrix with Boost.

As one can see in Table 6 using the *Boost* strategy is strictly dominant for the counter-terrorists and they will therefore use it every chance they get. Leaving the terrorists with just a very slim chance to win the round and thereby changing the entire game. However, using this strategy was against the rules of the tournament and Fnatic had to withdraw from the tournament shortly after the game[Fna].

Additionally to map design the different available weapons of CS:GO also cause unbalanced games. Two pistols in particular were heavily changed over the course of a few months. The *CZ75-Auto* which is available to both Ts and CTs, and the *Tec-9* which is only available to Ts.

The first one was regarded as a good weapon to be used while running, because it had a high accuracy while running, as well as being a formidable starting weapon since it allowed for quick kills and rewarded a relatively high amount of money (300$)

compared to its costs (500$). Later the damage, draw time, reload time, and kill reward money were changed in order to accommodate for the other benefits of the weapon[Vala; Valb].

Take-Away Balancing games isn't just a one-way undertaking. Game developers and designers have to listen to their player base in order to keep their game up-to-date and fix bugs. Oftentimes designers rely on the feedback and are actively asking for it, e.g. in the form of alpha and beta tests. Players are looking at the game from another angle and try to actively see how they can destroy the game. Exploits, bugs, and problems with the game can be discovered this way. The dialog between game developers and players is really critical for the success of a game and the longevity. Abandoned games lose their player base because they're not fun to play.

3.4 Practitioner's Perspective on Game Balancing

Game balance is a topic for which many definitions often seem to fall at odds with one another [Jaf13]. For illustration, Technopedia.com defines game balance as *"a video game design concept where the strengths of a character or a particular strategy are offset by a proportional drawback in another area to prevent domination of one character or gaming approach"* [Tec]. Another example would be the definition from Wikipedia.org: *"game 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"* [Wik]. Moreover, at this project seminar game balance was defined as *"the small changes to a game (basically parameter adaptation) in order to satisfy predetermined goals"*. Nevertheless, all of the listed definitions view game balancing as the tuning or small changes to a game parameters, rather than vast changes to the game itself.

In addition to these definitions, there exist different perspectives on the process of game balancing. Existing perspectives on game balancing can be summarized as following:

1. **Game balancing as Options**. This perspective is inspired by David Sirlin, a fighting game and board game designer. Sirlin published a handout for his 2009 Game Developer's Conference talk *Balancing Multiplayer Competitive Games*, which probably serves as the most succinct description of his definitions and views of game balance [Sir09]. He goes so far as to offer a precise definition

of balance: *"A multiplayer game is balanced if a reasonably large number of options available to the player are viable - especially, but not limited to, during highlevel play by expert players"*. The core of this definition concerns *options*. This is clearly more broad than a single notion that a game should be 'fair'. In fact, Sirlin is careful to distinguish two varieties of balance:

a) Viable Options (during a game): The player must have many meaningful choices throughout a game. The choices must be materially different from each other, not worthless, and not dominated by other choices. To make the game deeper, the choices should not be exactly equal in value at all times.

b) Fairness (options before a game starts): Players of equal skill should have an equal chance at winning even though they might start the game with different sets of options / moves / characters / resources / etc.

A final noteworthy message from Sirlin is this: "*game balance is so complex as to be inherently unsolvable. If it were solvable, your players will solve it and stop playing. Intuition, not math, is the best tool to navigate high-complexity problems. Creating a game and seeing how its balance turns out takes years*" This is because players' own views of a game develop slowly over time, and the "true" balance of a game is always contingent on the players.

2. **Game balancing as Rapid Iterations**. This perspective is inspired by Jaime Griesemer, a designer of first-person shooters and action games. In a 2010 talk at the Game Developer's Conference, he captured some of the intricate challenge of the balance process of Halo 3, anchored by the story of a single balancing decision [Gri10]. In leading up to his discussion of balancing Halo 3 's sniper rifle, Griesemer describes many facets of the balancing process. In particular, he goes into lengths to disabuse his audience of the idea that balance is just fairness. In balanced games, each strategic element should have a distinct *role*. Such a role specifies not just its mechanics, but also the aesthetic feeling of using it. Griesemer's talk emphasizes the importance of frequent playtesting with rapid iteration. Only through extensive iteration and a focused design plan is it possible to let each strategic choice operate in synergy. Moreover, an insightful segment concerns the variety of players who might playtest a game, and how designers should take different forms of insight from each of them. He defines six example kinds of players, and the most useful kind of takeaway each can provide:

a) Optimizers: By "playing to win", these players will find discover dominant or strong strategies

b) Skeptics: These players' complaints will direct designers toward the most dissatisfying, unfair, or frustrating aspects

c) Specialists: These players will play in a particular fashion regardless of its effectiveness, thus providing an unbiased view of the strength of core components

d) Novices: Bad or new players reveal difficult or confusing components

e) Griefers: Just as optimizers seek the best way to win, these players seek the best way to frustrate other players, thus mapping the least pleasant edges of the design space

f) Professionals: Tournament-level players tend to value predictability, and thus these players will point designers toward segments of play that feel too unpredictable.

3. **Game balancing as Science**. This perspective is inspired by Alexander Jaffe, data and design Person for Spryfox and the author of *Understanding Game Balance with Quantitative Methods*. He defines game balancing as the *"meaningful diversity of gameplay experiences"* [Jaf13]. Furthermore, he stated that it is possible to think of each aspect of the experience (be it strategy, fairness, win conditions, duration, randomness, etc.) as defining an *axis* of game balance, often orthogonal. Meanwhile, the world *meaningful* is used to refer to the player's choices and consequences; meaningfully diverse experiences should be noticeably different in terms of choices and outcomes, not purely aesthetics. Under this definition, a certain kind of balance is not necessarily a desirable property of a game, it is simply a property to which designer attention should be paid. Jaffe states: *"most strategically interesting games are indeed not feasible to solve exactly. Yet this in no way prevents us from applying mathematical reasoning to game balancing [...] I do not claim that balance as a whole can be formalized, but rather that formal measures can deeply inform a designer's understanding of balance"*. Thus, game balance can be understood through the effectiveness of explicitly modeled players, and Jaffe states that many forms of game balance can be reframed as the success of some restricted mode or method of play.

4. **Game balancing as Art**. This perspective is inspired by Jesse Schell, a

game designer and educator, writer of *The Art of Game Design* [Sch14], a critically acclaimed guide to many facets of game design. Schell's book features a prominent chapter on game balancing, which presents balance in its broadest interpretation. He defines game balancing as *"nothing more than adjusting the elements of the game until they deliver the experience you want"*. The goals of game balancing that were defined by Schell and introduced in this paper at the chapter 3.1 Goals of Game Balancing, according to Schell, must be considered at once during the balancing process. Mechanical changes affect all twelve of the above axes, and design decisions must tradeoff between all of them simultaneously so as to capture the intended experience. This kind of nuanced experience is exactly what balance is all about.

The main takeaway from these four diverse characteristics of balance is that game balancing is indeed a difficult concept to pin down to a single definition. Most notably, it is possible to see that each of the four designers' understanding of balance is broader than the last, but still there is no common definition of game balancing in industry.

3.5 Artificial Intelligence Methods: An Overview

Artificial Intelligence (AI) plays an important role in automated game balancing. Indeed, the general concept of automated game balancing is to replace a human tester by a capable AI. However, one important assumption has to be made: the implemented AI is able to imitate the behavior of a real tester with a high degree of approximation. This assumption should stand true, otherwise the optimization process balances the scene against not real and possibly misleading behavior. From the other side, AI should be considered from the standard game perspective where AI is developed for NPCs.

The general concept of AI originates from the pioneering work of Turing [Tur50] where famous Turing Test is proposed to assess the quality of AI. The idea of the test is to fool an interrogator which decides either he communicates with a human being or with an AI. From that perspective the AI should be able to *think* and *act* humanly, however modern approaches do not only consider this from that perspective, but also add rationality of behavior. Indeed, according to Russell and Norvig [RNI95, pp. 5-7] an AI can be defined as any system, designed to perform human tasks, that *thinks* and *acts* both humanly and rationally. The above definition is rather

general and describes the generic AI, however modern games require more complex AI which can focus on player experience and is able to adjust appropriately. That leads to the concept of a "fun AI" that is highly utilized in games. It can be seen from Table 7 that the "fun AI" diverges from the general concept of a "good AI". Indeed, it focuses more on player experience and is aimed to be challenging enough for player. From that perspective it is not developed to pass Turing Test, but rather to be interesting for player.

"Good AI"	**"Fun AI"**
focus on human substitution	focus on player experience
tries to beat player	tries to compete with player
coherent with Turing Test	irrelevant in terms of Turing Test

Table 7: AI Concepts.

From the game perspective AI methods are special algorithms that are able to imitate behavior of humans or animals within the game [MF09, p. 4]. Since recent games are becoming more complex, where a lot of elements should have their own behavior, it would be unrealistic to have one AI method for the whole game. Therefore, normally the overall game AI model consists of a set of methods which together represent it. Figure 3 shows that the AI model includes different levels where a decision should be made in given computational time using the information provided. Thus, at every level one or another method is more appropriate which results in the model of AI rather being an ensemble of methods.

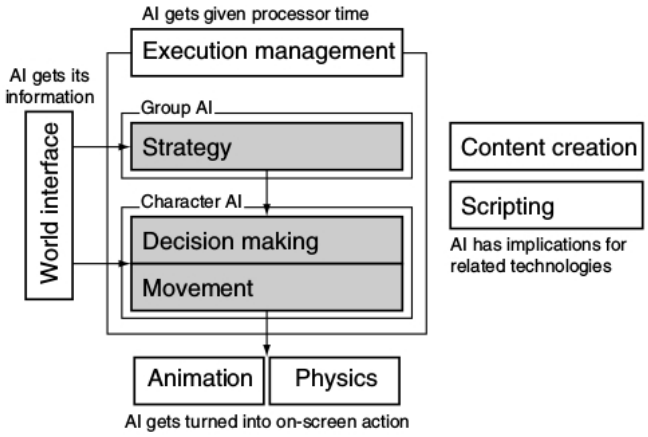


Figure 3: The Model of AI.
Source [MF09, p. 9]

According to Preuss [Pre16, p. 22] there are three primary research streams of AI methods: specialized algorithms, Machine Learning (ML) approaches, and Computational Intelligence (CI). Specialized algorithms are mostly designed to perform specific tasks such as pathfinding or movement. Typical examples of ML approaches are reinforcement learning and tree search. CI is a rather new research area which includes Evolutionary Computing (EC), Fuzzy Logic, Artificial Neural Networks (ANN), Monte Carlo Tree Search (MCTS) and others. Different techniques can be prevalent in different situations and at different levels of the AI model. Therefore, they are used jointly.

Strategy and decision making There are three primary AI methods for the decision making process which cover a wide range of arising game tasks: state machines, decision trees, and behavioral trees. These methods are rather simple, but extremely powerful and versatile. The central idea of the methods is, having exhaustive information about possible actions and the environment, construct a set of rules which define behavior of a unit. In order to represent these rules the methods use different data strcutures; for example, decision trees use acyclic graphs to capture behavior. These methods were applied in such complex games as *Halo 2*, *Anno 1404*, and *Bioshock*. Of course, the methods are still developing; for example, more complex hierarchical state machines are proposed. Lim et al. [LBC10] propose an adjustment of behavior trees for a game using recent optimization techniques.

Most board games, like chess or Go are deterministic with perfect information. Thus every player knows all the possibilities and can construct a game tree with all the variants of game outcomes. The tree can be further traversed in order to find the best action. The most simple way to traverse game tree is to use min-max tree search. This method uses the proposition that the game is a zero-sum game, therefore the opponent tries to minimize the outcome while the player tries to maximize it. Of course, this method cannot be applied in most of the cases because it considers all possibilities, therefore it is generally enhanced by the alpha-beta pruning technique which decreases the search space. The enhanced approach has been proven to be effective, famous chess program *Deep Blue* utilized it in order to beat world champion Garry Kasparov in 1997. However, this technique has a lot of disadvantages. First, it can be purely applied only for deterministic games with perfect information. Second, this method is not generally any-time, since final decision about the best action is done in the end of search. Third, the branching factor is still a bottleneck of the approach in most of the cases. These challenges can be relatively well solved by a new computational technique which is called MCTS. This method develops the concept of a multi-armed bandit in order to find the best actions. The central concept of MCTS is to repetitively explore "good" paths in game tree, randomly simulating and back-propagating game outcomes. This algorithm is very versatile and requires minimal expert knowledge. However, it can be enhanced in many ways. For example, the selection policy can incorporate patterns in order to prune weak paths. MCTS has become a good solution for a relatively strong AI, it is widely used in industry from board games (*AlphaGo*) to real-time strategy games (*StarCraft*, *Total War*). Theoretical studies of MCTS also can be found in literature [CBS+08; BPW+12]. However, the method has two main disadvantages: it should be properly parameterized and generally it is very intense computationally.

Strategy and decision making in games is a relatively generic process and can be supported by other methods which were originally developed for other purposes. The first method are EAs. Indeed, it is a robust problem solver which fits even as a game AI. Another method is Q-learning, it incorporates the concept of reinforcement learning and fit for the games where units should learn about the environment by themselves. Recently, Fuzzy Logic systems are tried in game industry [PKH+13], in addition ANN are commonly used to capture patterns and support main algorithms. One example of such support is Go AI *AlphaGo* which utilizes the concept of ANN

in order to use it for simulation and pruning within MCTS.

Movement The key movement task in most of the games is pathfinding. A commonly used pathfinding algorithm in game industry is A*, it originates from the work of Hart et al. [HNR68]. A* finds the shortest path between two points using heuristic which biases traversing to get solution faster. However, this algorithm does not guarantee optimality in case of inconsistent heuristic. The concept of influence maps is commonly used to support A* in navigation tasks. The general idea of influence maps is that every game element influences the game to some degree. These influences interfere with each other creating a final potential for a certain game spot. A similar approach is proposed by Hagelbäck [Hag12], he introduces potential fields which incorporates the same idea of interfering influences, but influence can change according to behavior of a unit. Combination of A* and influence maps or potential fields is relatively good approach to tackle with movement problem in games. Recent concerns about believable behavior pushed forward another movement related technique which is known as flocking. The main concept of flocking is that during movement in squad every unit is exposed to three forces: separation, cohesion and alignment. Recent studies prove that flocking seems to be rather promising for real-time games [DST+08].

AI methods are a relatively huge area which consists of a variety of techniques. These techniques do not substitute each other, but rather complement eliminating the mutual limitations. Depending on a context one or another method can be chosen, where in the end the whole ensemble of methods represents AI model for the game. Of course, AI methods develop very fast, however there are a lot of challenges have to be met and overcome.

3.6 Procedural Content Generation

> "Procedural content generation refers to the creation of game content automatically, through algorithmic means."[TYS+10]

Game content encompasses all aspects of a game that affect game-play, however it excludes NPC behavior or the game engine itself [TYS+10]. The method of Procedural Content Generation (PCG) can speed up game development, save human designer effort/cost, complement human creativity and extend the life-span of a game. The method of PCG can be applied either in the following way: [TYS+10]

1. Online vs. offline
2. Necessary vs. optional content
3. Random seed vs. parameter vectors
4. Stochastic vs. deterministic
5. Constructive vs. generate-and-test

The method of PCG can assist in balancing games in the form that new content is algorithmically generated based on some evaluation criteria that implicitly follows balancing goals. The evaluation can be a direct measure on the created content or an indirect measure that is delivered by a human or artificial player. An applied example from research shows that PCG has the potential to not fully solve the balancing problem but reduce manual labor [CR15].

3.7 Dynamic Difficulty Adjustment

Definition Dynamic Difficulty Adjustment (DDA) is a modern approach to modify difficulty during runtime of a game by evaluating user performance and changing key parameters accordingly in order to maintain a challenge for the player [Hun05].

Difficulty Flow As such, DDA resembles an evolving prediction problem, having to keep track of the player behavior. The goal of the adjustments during the game is to keep the difficulty in a conceptual *Flow Channel* where the player is alternated between over- and underperforming as depicted in the figure below:

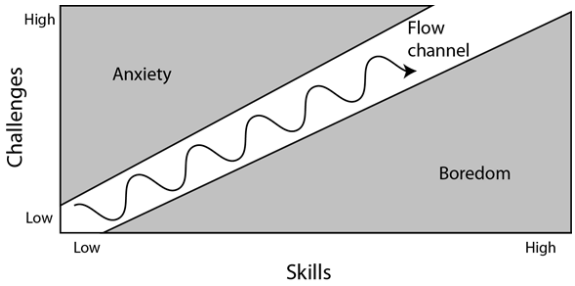


Figure 4: Difficulty Flow Channel.

Thus, the game is kept challenging even if the player does not follow a traditional difficulty curve [Hun05].

DDA Process Listing 1 presents a simple Pseudo-Algorithm for DDA. In the beginning, a starting difficulty is chosen by the player (e. g. easy, normal or hard). Then, the iterative difficulty adjustment process is called every time a "trigger" point is reached in the game. Trigger points are determined by the designer and can for example either be set after certain enemies, after a certain time period or after certain objectives are reached.

The difficulty adjustment process then places the player in an area of the flow channel. The three category areas are resembled by numerical ranges between 0 and 1: Underperforming (0-0.4), Challenging (0.4-0.6) and Overperforming (0.6-1). If the player results until the trigger are in the Underperforming area, the difficulty level is lowered. If the player results are in the Overperforming area, the level is raised.

Listing 1: Dynamic Difficulty Adjustment Pseudo-Algorithm.

```
1 INITIATE with predefined level (easy or normal or hard)
2 OBSERVE player results until trigger(result observe(t))
3 foreach (trigger)
4   {If (observe(t)n-1<0.4)
5       {lower difficulty in moderation;}
6   Else if (observe(t)n-1>0.6)
7       {raise difficulty in moderation;}
8   Else {difficulty is appropriate;}
9   OBSERVE new observe(t);}
```

3.8 Balancing Applications in Other Domains

Balancing does not only take place in the context of games, but also in other domains with human actors and systems. A brief exploration of applied balancing in non-game related cases was conducted to examine the application of balancing in different contexts and possible insights for the future project. Three exemplary cases were investigated, of which each one represented another field of application. The covered topics included load balancing [RDJ+06], computational mechanism [CCP99] and competition in networks [MAK+07]. Even though each case dealt with a different topic and used a different approach, the core characteristic of balancing was identified as an optimization problem connected to a limited amount of resources. Whether it was the limited bandwidth in the example of load balancing, the limited server capacity in the example of computational mechanism or the limited road capacity

in the example of networks, a limited amount of accessible resources lead to the search for a configuration or setup which provides the most efficient use or the fairest distribution of the resources.

All cases showed that balancing in form of an optimization problem needs certain preparations. Crucial is the representation of relevant and important information. Information therefore should be transformed in a quantifiable representation. For the representation of information different approaches can be used, as e.g. the Fisher-information in the example of load-balancing, which was used to represent the value of the carried information of each agent. Furthermore the approach should include an objective function to represent the degree of "goodness" of a possible configuration or setup. In the example of computational mechanism and server capacity such a function could be used to calculate the response time of server and the optimization problem would aim to minimize this response time.

Overall, balancing in other domains than game development is used to provide a fair and efficient way of distributing limited resources among participants by solving an optimization problem.

3.9 Game Balancing by Computational Mechanism Design

The usage of computational mechanism design for game balancing is part of recent research. Insights into an ongoing PhD study were provided to us in advance of the project's main work with the idea of giving a first understanding of computational mechanism and the possible application in the project. The key insights and take-aways are summarized in the following part.

There are different motivations for approaching game balancing by computational mechanism design. From the perspective of game developers computational mechanism design could face the optimization potential in the balancing process, which is currently an expensive and tedious process. A successful application in the context of game balancing could in addition provide researchers with the opportunity of transferring such application to other systems and real-world problems. Even though, there are potential improvements identified, there are several challenges and problems to face. The application of computational mechanism design in the balancing process should be robust in a way that it should only need small adjustments over time, has

to deal with multi-objectivity and has to go along with the rules and underlying physics of a game to go along with game-crucial aspect as immersion. Beyond this, the formalization of a system or player goals is necessary but difficult to realize. Talking about balancing in general, the motivations and problems of balancing have to be considered. The goals of balancing are motivated by different point of views, as for example gameplay concerns like difficulty adjustment, design decisions like immersion or economic reasoning like the relevant target audience.

Balancing itself can be viewed as an optimization problem with different objectives which can be game-related as well as more general. One way to face such problem is through game theory. Game theory is used as a forward model to determine actions or strategies in a game, but struggles with the limitations of manual optimization and is not automated. Using computational mechanism and the automated search with simulation data represents another way, but to face the idea of an automated search with simulation data, several challenges have to be faced. First, the design goals have to be defined. The goal can be a challenging game or a game that is fun or fair. Second, the problem itself has to be formalized. Game theory can help in this case to approximate utility functions and an optimization problem. Third, the computational mechanism design has to provide characteristics like robustness.

Overall, computational mechanism design shows to be a reasonable approach for game balancing, but goes hand in hand with additional preparations and adjustments that have to be considered.

3.10 Main Findings of the Literature Analysis

The above-provided set of research streams provide an extensive representation and overview of the current understanding in research about modern computer games and applied CI techniques to perform certain tasks. It does not raise the claim of being fully exhaustive. However, no canonical approach could be identified in which researchers dedicated their work towards finding an automated way in balancing game parameters. This finding is supported by [YT14] and [Luc08]. None of the below analyzed research streams revealed any methodology of applying CI techniques dedicated explicitly to game balancing.
The intention of balancing is implicitly included within each of the depicted game

Figure 5: Panorama of AI research, [YT14] Fig. 2.

AI/CI areas of figure 5. For example, in PCG it is the underlying the intention to generate game content that satisfies a positive player experience and is therefore balanced. However, the explicit intent of game balancing is not yet included.

In conclusion, this provides an opportunity to investigate a gap that may be of high interest for both researchers and practitioners. In practice the task of game balancing is often done manually, which holds true at least for the industry partner of this PS, who demonstrated their manual approach in a project meeting. It should be noted, that no information is available about the on-goings of the entire game industry (e.g. some sort of survey).

4 Project Organization

In projects with highly unpredictable development, including aspects like the direction of development or possible obstacles, flexibility is the core need for the applied management methods. A short explanation of the organization and the management of the project is given in the next chapter, to provide an understanding of the project team's working method.

4.1 Project Set-Up

The basic idea of the project seminar as a research seminar with mainly open development directions provided reasoning for an agile approach of project management. The project followed an agile approach of project management. As a reference served the Scrum framework, rooted in agile software development. The simple methodology, the distinct defined roles and artifacts enable an enactment and use of this framework without preliminary knowledge of agile methods. The project consisted of the project team, represented by eight master students from the field of information systems and three supervisors, two representatives from departments of the university, completed by a PhD student whose research overlaps with the projects topic. The project was set up for a time of one semester, around six month. Even though, the project seminar did not represent the typical setup for the Scrum framework, the methodology of Scrum was adapted to the circumstances of the project. While the main aspects of the Scrum framework were applied, other aspects where left out since they were seen as not feasible for the project.
The general project process is visualized in the following graphic.

4.2 Application of Management Methods

The basic idea to work in short iterations, so called sprints, was used during the whole project. Sprints were usually around a week long and weekly meetings with the supervisors encouraged the project team to break down work in smaller work packages and present their progress every meeting. The meetings served as an opportunity to inform the supervisors about the progress of the last week as well as to discuss recent developments together and set or change work packages for the next week. For the coordination and communication with the supervisors two students were chosen from the team to fulfill a variation the Scrum master role. These two students organized the coordination and communication with the supervisors. Structuring

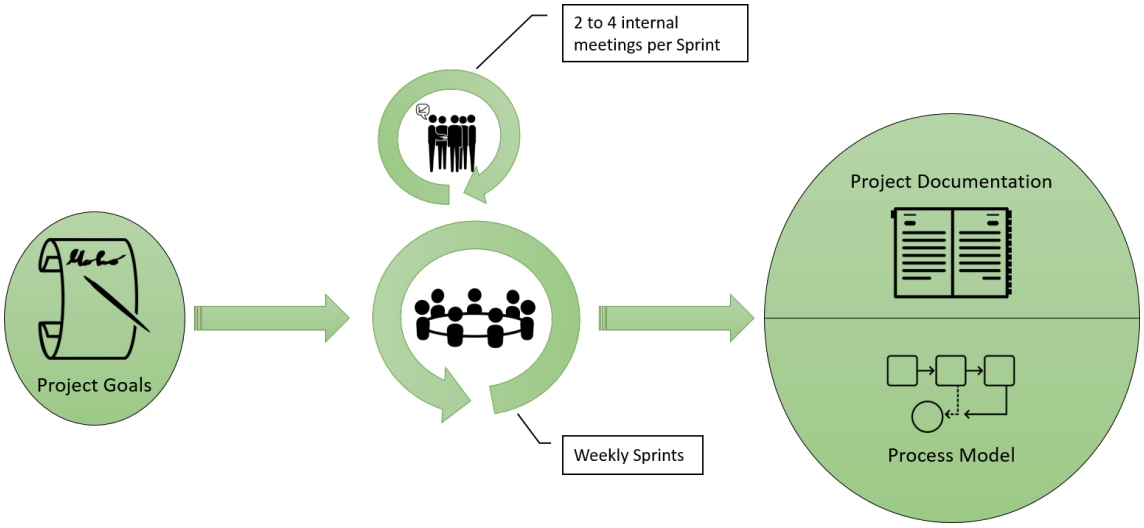


Figure 6: General Project Procedure.

the work helped the team to identify work packages and keep them as modular as possible. In retro perspective, the team divided the work into three main streams: The ZVG, the 2DR game and the data analysis. The team split up to work on the different stream and broke down the work in even more detailed work packages, as e.g. for the 2DR into AI and algorithm implementation, or for the data analysis into e.g. evolutionary algorithm and meta-optimization. Every internal meeting of the team began with a so called "daily scrum" where the team stands together for 10 to 15 minutes and everyone shortly talks about the open tasks on the Scrum board with following questions in mind:

1. *What have I done/achieved since the last meeting?*
2. *What is my plan for today?*
3. *Are there any impediments that hinder me in my current work?*

4.3 Tools Used

In addition the coordination and communication were supported by different tools. Trello served as a digital Scrum board, Dropbox and Google Drive were used for document sharing and Git was used to work on code in parallel. The selection of the tools was based on the students' experiences with the tools and the provision by external partners (GitLab provided by University of Muenster).

5 Game Description

5.1 Gameplay Environment Setup

In order to begin play-testing the game, it is required to download and install Unity, a development suite for creating multiplatform 3D and 2D games. A personal edition can be downloaded for free. This edition sufficed to fulfill all necessary development goals in the context of this project seminar.
After installation, Unity first requires to open a project. Within a project the user has access to scenes and assets of the game. An asset is a representation of any item that can be used in a scene. Types of assets are commonly classified into image files, model files, meshes and animations, audio files or other texture materials. Unity also offers to import complete asset packages.
For scripting, Unity uses C-Sharp (C#), a class-based and component-oriented programming language. The C# scripts are embedded and integrated within components of Unity. Components are the functional elements of game objects and define their behavior. Game objects do not perform behavior by themselves but are merely the container that can hold different elements that define the game object.

5.2 The Zombie Village Game

As introduced above, the data-driven analysis of this project seminar is based on a game prototype provided by Blue Byte. The game and its source code were provided via Gitlab and exclusively applicable within the Unity game engine. The provided prototype included main assets and scripts as well as a text-file that described the current and intended gameplay. The text-file informs that "the idea behind the game is to build and manage a camp invaded by enemies", which classifies the game as a Tower Defense (TD) game a sub-genre of Real-Time Strategy (RTS) games. However, the concept of placing static towers to kill creeps [Rum11]; [ATA+11] is replaced in the form of player units that can be placed freely on the map and attack Zombie-like units.

[Sch14] defines twelve types of balancing that are used here to describe balancing of the ZVG. The first of them is *Fairness*: Games can be asymmetrical or symmetrical. Symmetrical games "give equal resources and powers to all players" ([Sch14]), asymmetrical games respectively don't. The goal should be to balance the game s.t. it feels fair to the player. The next type is about keeping the player's experience balanced between *Challenge vs. Success*, for example by increasing the difficulty

for each level. The goal here is to keep the player from being bored (too much success) as well as frustrated (too much challenge). The *Meaningful Choices*-type is about providing choices to the player that have actual impact to the outcome of the game. Doing this, balancing should make sure that the player is feeling freedom and fulfillment by having exactly as many choices as desired. *Skill and Chance* are two opposing forces in a game, for which the balancing goal depends very much on the player, since the decision about how much skill versus how much chance should be needed to win a game is very subjective. In terms of how a game should be won it is also important to balance it with regards to *Competition vs. Cooperation*. A game can be either competitive or cooperative or a combination of both with certain emphasis of one of these forces. Another balancing type for keeping the player from being frustrated or bored is about the length of the game (*Short vs. Long*) which is very much self explaining. In order to achieve a certain retention of players by keeping them happy, *Rewards* are a good measure. Having many types of rewards within your game is generally desirable. In contrast to this, *Punishment* should be used carefully. It is useful however, to make the player use caution and evaluate risk in certain game situations. Another type of game balance is the degree to which a player gets *Freedom vs. Controlled Experience*. The emphasis here lies on the creation of a better experience for the player. Another goal is to achieve the right level of *Complexity vs. Simplicity*. The guideline here should be to create meaningful complexity by means of a simple system. Finally, it is important to note that the game does not create the experience alone but that it is about *Detail vs. Imagination*. The game is considered to be balanced regarding this type if provides enough details for the player to understand it and still leaves enough room to inspire imagination of the player.

Goal	Application
Fairness	"Zombie Village" is an asymmetrical game that should most importantly provide interesting challenge to the player. The player should however feel capable of beating the NPC which implies fairness of the game.
Challenge vs. Success	Players should consider "Zombie Village" to be a challenging game but should be successful (=win the game) in most of the cases.
Meaningful Choices	Meaningful choices are provided to players by letting them decide about the placement and removal of the player units with is the core action of the game and highly impacts its results.
Skill vs. Chance	The focus of the game lies on player skill rather than chance. The strategy together with the player experience should decide about the game outcome.
Competition vs. Cooperation	Since this game does not include multi-player modes so far, it is a purely competitive game.
Short vs. Long	"Zombie Village" is a mobile game designed with different levels each with different complexity. Generally, a player should be able to finish a level in a short amount of time, roughly meaning less than five minutes.
Rewards	Rewards are currently implemented by displaying a success message.
Punishment	The player has to collect junk in order to stay alive and be able to win the game, otherwise he/she gets punished by loosing.
Freedom vs. Controlled Experience	Placement of player unit is free but at the same time limited due to the boundaries of the scene map.
Complexity vs. Simplicity	The rules of the game are very simple, a certain complexity is given through the set up of the levels.
Detail vs. Imagination	At the current stage of the game it does not make sense to apply this.

Table 8: Balancing goals for the ZVG.

5.3 The 2D Roguelike Game

The used game is an exemplary Unity tutorial. The game can be classified as a mixture of infinity runner and dungeon crawler, in which the player has to try to reach the exit on the map to survive a day in the context of the game and thus reach the next level. As an overall goal the player should aim to survive as many level as possible. Starting from the bottom right corner of a 64 fields containing map, the player has to overcome obstacles in form of walls and enemies to reach the exit on the top right corner. The player starts the game with a certain amount of food, which serves as a health indicator of the player's unit. Each attempted move as well as each hit of an enemy reduces the amount of food. If the food count reaches zero, the game is lost. Food tiles are placed on each levels map to provide the opportunity for the player to refill his food count by a certain amount, by collecting such food tiles.

The main game elements are therefore:

1. The player unit, which is controlled by the player.
2. Enemy units, which are placed randomly on the map and move towards the player and attack him, if in distance.
3. Destructible walls, which are obstacles for the player on the way to the exit.
4. Food tiles, which are optional pickups for the player to regain food.

As for the ZVG, [Sch14] twelve types of balancing are used to describe the balancing of the 2DR game. An explanation of these twelve types can be found in chapter 5.2. The balancing goals for the 2DR game are:

Goal	Application
Fairness	The player should be able to identify some kind of increasing difficulty with levels and be able to reach further levels with learning and improving his gameplay.
Challenge vs. Success	Players should consider the game to be challenging. The player should aim to improve his reached level with every play through.
Meaningful Choices	Meaningful choices are provided to the player by letting him decide about the route he takes to the exit or the optional pickup of items on such a route.
Skill vs. Chance	The focus on the game lies on the player's skill rather than chance. The player's strategy and his experience should decide about the game outcome. There's partially provided chance in the game through the random placement of elements in each level.
Competition vs. Cooperation	Since this game does not include multi-player modes, it is a purely competitive game.
Short vs. Long	The game is supposed to be played in a short amount of time, roughly meaning in less than five minutes.
Rewards	Rewards are currently only given by displaying the reached level after losing the game.
Punishment	The player can be hit by an enemy unit (multiple times), if he does not plan his path best. Loosing food due to inefficient planning is used as punishment.
Freedom vs. Controlled Experience	The player can move freely on the map, but only in the given 8x8 field sized map.
Complexity vs. Simplicity	The rules of the game are very simple and is kept through the levels.
Detail vs. Imagination	At the current stage of the game and the given simplicity of the game it does not seem to make sense to apply this.

Table 9: Balancing goals for 2DR.

6 Build of the Balancing Environment

6.1 General Methodology

This chapter will outline the purpose and functionality of the BE. The BE performs tasks in a service-oriented manner. It represents the combined setup of a game scene, a set of game parameters, a game goal, a fitness function, a player AI and an optimization algorithm, which are loaded into the public class `BalancingSuite.cs`. The following table contains an example set for inputs.

Input	**Input Version**
Scene	Scene #006: Player(s) vs. Zombie(s) with Resources and Buildings
Goal (Winning condition)	Kill 40 zombie units
Player AI	AggressivePlayerBehaviour
Algorithm	Genetic Algorithm
Fitness function	Distance to sum of remaining player health

Table 10: Sample Set for Input of the Balancing Environment.

Based on the specified inputs, `BalancingSuite.cs` will initialize the operation. It first instantiates game parameters based on the predefined types of parameters and their permissible set of value ranges. Secondly, it initiates a loop that generates new solutions until a predefined condition has been reached. The loop runs and evaluates the current instance of the game scene by applying the player AI goal, current game parameters to the game scene and computing the result against the fitness function. After each run-time, the simulated instance reports the values in the form of the current solution. If the termination criteria have not been reached, the loop continues to redefine and replace the current game parameters with new ones based on the heuristics of the algorithm. The following Pseudo-code reflects the general behavior:

Algorithm 1: Pseudocode of the Balancing Environment

Data: game scene, set of game parameters, fitness function, set of player AI, optimization algorithm

Initialize balancing environment with game scene, set of game parameters, fitness function, set of player AI, optimization algorithm;

Set current game parameters to initial seed from set of game parameters;

while *Termination condition is not satisfied* **do**

  Set current solution to evaluation of game scene, current game parameters, fitness function, set of player AI;

  Report current solution;

  Set current game parameters to new game parameters of optimization algorithm

end

Return best solution of optimization algorithm;

The following sections will explain in detail each of the required components.

- Game Scene (chapter 6.2.1)
- Game goal (chapter 6.2.2)
- Player AI (chapter 6.3)
- Fitness function (chapter 6.4)
- Set of game parameters (chapter 6.4)
- Optimization algorithm (chapter 6.4)

6.2 Game Scene

6.2.1 Scenes

Scenes in the Zombie Game are realized by combining available Prefabs in Unity. A limited number of Prefabs are considered to be the main elements of the game, namely: player units, zombie units, resources, and buildings. A scene features at least two different game elements from this list. At most, all four different game elements are featured in a scene. The scene featuring all four game elements is called *Main* scene (`Main.unity` and was provided by Blue Byte GmbH together with the ZVG. Furthermore, Blue Byte GmbH provided a scene featuring player units and zombie

units. The following list comprises all scenes that can be created by combining game elements according to the aforementioned constraints. Please note that scenes not featuring player units have been omitted from the list, since scenes without player units are not suitable for approaches at non-manual game balancing using player AI. In total six scenes were created:

1. Player units and zombie units
2. Player units and resources
3. Player units, zombie units, and resources (`SceneToBalance.unity`)
4. Player units, zombie units, and buildings
5. Player units, resources, and buildings
6. Player units, zombie units, resources, and buildings (`Main.unity`)

Each scene features a certain number of non-game element Prefabs required for the scene to work properly. These non-game element Prefabs are the same for each scene. For this reason, a scene template was created which featured all the required non-game element Prefabs, namely:

- Cheats
- ClientGameLogic
- Directional Light
- EventSystem
- GridDisplay
- GuiResources
- IngameMenuCanvas
  - FastPlaceMenu
  - ObjectivePanel
  - PlaceObjectContainer
  - ResourceInfoPanel
  - ResultPanel

- Main Camera
- Missing Prefab
- ModifyGameParameter
- Zone Border Stone

This template has been reused several times, each time extended with different game elements according to the list mentioned above. The scene featuring player unit, zombie units, and building showed two bugs which could not be resolved.

1. If a player unit enters the detection range of a zombie while the zombie is moving towards a destructible wall, the zombie will "focus" the player unit and not attack and destroy the destructible wall. Instead, the zombie will get stuck in front of the wall. This is true for all three different zombie unit types (AStar, MoveDirect, and Navpoint).

2. If a player unit is within the detection range circle of a zombie before the zombie started to move towards a destructible wall, the zombie will "focus" the player unit and not move at all. This is true for all three different zombie unit types (AStar, MoveDirect, and Navpoint).

Keeping these bugs in mind it was necessary to use a scene without buildings, since the mentioned bugs often occurred. It was decided to use a scene with player units, zombies units, and resources. The scene is called *SceneToBalance* and is used for all simulation purposes, i.e. it is used for the optimization with evolutionary algorithms and for the two parameters algorithm.

The player has three player units available in the *SceneToBalance* with the ultimate goal to kill all eight zombies on the map, while not running out of food to keep playing. Additionally, there are four resource spots on the map which are guarded by varying groups of zombies. The first resource spot is guarded by three zombies, the second by two, the third by one, and the last one is not guarded at all. The resources can be gathered in order to extend the playtime, i.e. increasing the food amount available to the player.

6.2.2 Goals

In its original state, the ZVG Prototype did not consist any objectives for the player. We implemented several ideas to use in the *Balancing Environment*. These goals are

realized in C#-Classes that can be appended to a scene to `ModifyGameParameter.cs` in order to create winning and failing conditions.

Abstract goal In order to handle the goal creation process, the abstract superclass `Goal.cs` has been created. It consists all essential functions required as displayed below:

Listing 2: abstract goal variables.

```
1 public abstract class Goal : MonoBehaviour {
2
3   public static Goal instance;
4
5   public abstract bool IsAchieved ();      // victory condition
6   public abstract bool IsFailed (); // loss condition
7   public abstract void Restart (); // reinitializes the goal
8   public abstract double[] GetFitness (); // fitness calculation
9
10    // display functions for the GUI
11  public abstract string GetAchievedMessage ();
12  public abstract string GetFailedMessage ();
13  public abstract string GetObjectiveMessage (); }
```

Individual goals The following individual goals for the game were created:

1. Survive for a given amount of time (`TimeGoal.cs`)
2. Kill a given amount of zombies (`DeadZombiesGoal.cs`)
3. Reach a given amount of Score Points (`ScoreGoal.cs`)
4. Gather a given amount of Resources (`JunkGoal.cs`)
5. Gather food to survive while combating the zombies (`SurvivalGoal.cs`)

Winning and failing conditions and results are shown to the player through interface panels with flexible display messages. Furthermore, the fitness has to be returned for further calculations (example below).

Listing 3: goal fitness calculation for overall average player health.

```
1     public override double[] GetFitness () {
2         int totalHealth = 0;
```

```
3
4           //create a list of player units with enough health
5           List<ClientGridObject> playerUnits = Scopes.
                AliveActivePlayerUnitsWithEnoughHealth (0f);
6
7            //iterate over the list, summing up the health
8            foreach (ClientGridObject playerUnit in playerUnits) {
9                totalHealth += playerUnit.GetGridObjectDefinition ().
                    health;}
10
11          return new double[1] { Mathf.Abs((float)health -
                totalHealth) }; }
```

The states Restart, IsAchieved and IsFailed are later used by the *Balancing Environment*. New goals can be created by overriding the superclass functions with selected conditions to extend the game (short code example from the *SurvivalGoal* below):

Listing 4: override victory and failure conditions.

```
1   // victory condition (all zombies are dead & initial food)
2     public override bool IsAchieved () {
3         return Fitness.GetJunkAmount() > 0 && Scopes.
              AliveActiveZombieUnits().Count == 0;}
4
5     // failure condition (player units dead OR food is empty)
6     public override bool IsFailed () {
7         return Scopes.AliveActivePlayerUnitsWithEnoughHealth (
              healthLimit).Count < numAlivePlayerUnits && Scopes.
              PlayerUnitsInResidence().Count == 0 || Fitness.
              GetGameJunkAmount() == 0f;}
```

Our main goal in the balancing process is the *SurvivalGoal*. This goal has the objective to survive a level by gathering food (food is decreasing over time) and fighting zombies, thereby incorporates two strategic elements implemented, namely resource collection and fighting. Both aspects are measured by whether at least one or more player has survived as well as whether enough food is left to proceed with the game. With expiration of food before all Zombies are killed, the game is lost. On the other hand, there is no strategic advantage in collecting food, when there is still enough already collected food that allow to fight the Zombies. Those strategic aspects will be covered in more detail in the chapter about the player AI, but it shall be noted here to explain the setup of the Survival Goal. Hence, the

SurvivalGoal extends the original fitness function (overall health distance to 50) with the additional target parameter food (food distance to e.g. 5). To distinguish on the impact of the different variables, weights for both parameters are introduced, with food having a higher multiplicator (which displays the scarcity thereof).

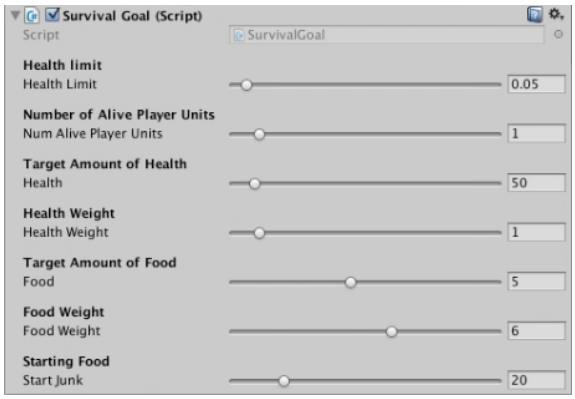


Figure 7: Survival Goal standard setup.

The figure above shows the standard setup of the *SurvivalGoal*. The 'Health Limit' describes the lower threshold of a player unit that at least needs to be survived to win the game. The 'Number of Alive Player Units' marks the minimum number of player units that need to survive to win the game. In the *SceneToBalance*, one usually begins with three player units. The 'Target Amount of Health' is the target health value for the optimization algorithm. The 'Health Weight' is the weight with which the fitness function measures this value. The 'Target Amount of Food' is the target food value for the optimization algorithm. The 'Food Weight' is the weight with which the fitness function measures this value. The 'Starting Food' is the amount of food at the beginning of a playtime. The values of figure 7 are the standard values applied during simulating (see chapter 8).

Furthermore, the `SurvivalGoal.cs` class offers three different variants (*fitness-variant* parameter) for the array output:

1. health result, food result, weighted result (default)
2. food result, health result, weighted result
3. weighted result, health result, food result

6.2.3 Interface Structure

The original ZVG interface has been extended by new graphic panels to display objectives and (as soon as the condition is reached) victory or failure screens. Furthermore, relevant information are shown in the top bar called *ResourceInfoPanel* (containing Player Units alive, Zombies alive, Food left and Time elapsed).

Graphic Panels The *Objective* and *Result Panels* both utilize an overlay to darken out the current scene. Also, flexible text and image fields are grouped in a dialog panel which allows direct manipulation through the goals. The panels are attached to the *IngameMenuCanvas* which contains all GUI elements. While the *Objective Panel* only has a Continue option, the *Result Panel* also offers a Restart and an Exit function.

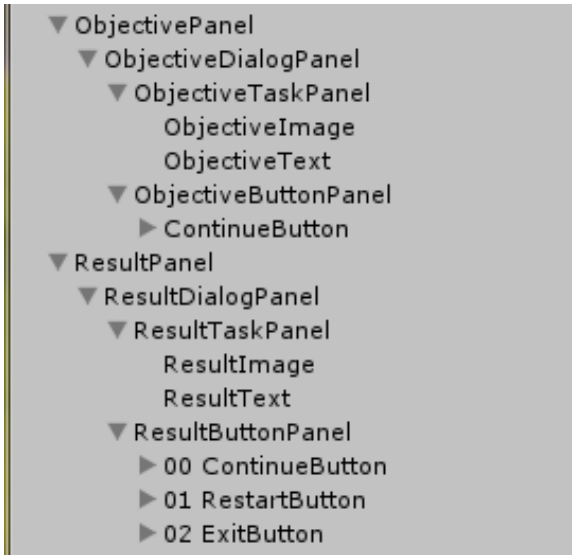


Figure 8: Graphic Panels Structure.

Counters Relevant game information is attached to the already existing *ResourceInfoPanel* by utilizing the *Counters* script. If needed, additional information can be added by creating a new text label and including the function in the script (example below) - values have to be initialized and continuously updated.

Listing 5: counters script example for junk data.

```
1 public class Counters : MonoBehaviour {
2     Text junkText;
3         // initialization - search for the GameObject and assign
4         void Start () {
```

```
5          GameObject resourceJunkChildGameObject = ClientMisc.
               GetChildGameObject(this.gameObject, "Text_Junk");
6          if (resourceJunkChildGameObject != null)
7          {
8          junkText = resourceJunkChildGameObject.GetComponent<Text
               >();}}
9
10     // update - get relevant information and update textfield
11         void Update() {
12              if (ClientGameLogic.instance.resourceData != null &&
                    junkText!=null) {
13                  junkText.text = "Food: " + Fitness.
                        GetGameJunkAmount();}}}
```

6.3 AI Development

6.3.1 Development Techniques

One of the main inputs for the BE is AI that includes both NPC AI and player AI. The difference between the last two is that NPC AI is the conventional type of AI for games that almost every game possesses. In contrast, player AI is the specific AI that is able to play the game instead of a human player. While human testing is also possible for the proper work of BE, this type of producing simulations is not automated. The goal of this PS is to create an automated balancing tool, so there is a need to develop a player AI in order to balance a game automatically.

For the development of the player AI for the ZVG and the 2DR games various techniques were used, including behavior trees, A* search algorithm, and influence maps. Behavior trees have become a popular tool for creating AI characters after the release of Halo 2 [MF09]. Behavior trees have a lot in common with Hierarchical State Machines but, instead of a state, the main building block of a behavior tree is a task. There are different types of tasks; for illustration, looking up the value of a variable in the game state, or executing an animation. Moreover, tasks are composed into sub-trees to represent more complex actions. In turn, these complex actions can again be composed into higher level behaviors, and that gives behavior trees their power. Since all tasks have a common interface and are largely self-contained, they can be easily built up into hierarchies without having to worry about the details of how each sub-task in the hierarchy is implemented.

The behavior trees for the ZVG and the 2DR include following types of tasks:

1. Leaf tasks are a type of the most basic task in behavior tree:

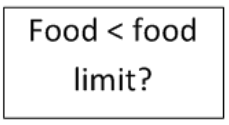
a) 

Figure 9: Behavior tree element: condition.

Condition. It tests some property of the game and returns the success status code if the Condition is met or returns failure otherwise.

b) 

Figure 10: Behavior tree element: action.

Action. It alters the state of the game and most of the time it will succeed.

2. Composite tasks are a type of task that keeps track of a collection of child tasks (leaf tasks or other composite tasks), and their behavior is based on the behavior of their children:

a) 

Figure 11: Behavior tree element: selector.

A Selector. It will return immediately with a success status code when one of its children runs successfully. As long as its children are failing, it will keep on trying. If it runs out of children completely, it will return a failure status code.

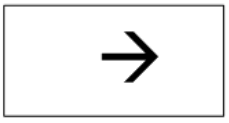
b) 

Figure 12: Behavior tree element: sequence.

A Sequence. It will return immediately with a failure status code when one of its children fails. As long as its children are succeeding, it will keep going. If it runs out of children, it will return in success.

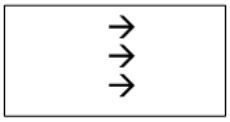
c) 

Figure 13: Behavior tree element: parallel.

A Parallel. It has a set of child tasks, and it runs them until one of them fails. At that point, the Parallel task as a whole fails. If all of the child tasks complete successfully, the Parallel task returns with success.

3. Decorator tasks are a type of task that has one single child task and modifies its behavior in some way:

a) 

Figure 14: Behavior tree element: until fail.

Until fail. It keeps running child task until it fails.

On the other hand, the A* algorithm is usually used for pathfinding [MF09]. Given a graph (a directed non-negative weighted graph) and two nodes in that graph (start and goal), there is a task to generate a path such that the total path cost of that path is minimal among all possible paths from start to goal. Any minimal cost path will do, and the path should consist of a list of connections from the start node to the goal node. Rather than always considering the open node with the lowest cost-so-far value, the algorithm chooses the node that is most likely to lead to the shortest overall path. The notion of "most likely" is controlled by a heuristic. If the heuristic is accurate, then the algorithm will be efficient. Furthermore, the calculation of heuristic for the A* algorithm incorporates the concept of an influence map. This concept keeps track of the current balance of enemy or resource influence at each location in the level. The A* heuristic considers the current balance calculated by an influence map while choosing the node. The practical implementation of behavior trees, the A* algorithm, and the influence map for the ZVG and the 2DR is described in the following sections.

6.3.2 Zombie Village Game Player AI

The ZVG includes both NPC AI and player AI, but while NPC or zombie AI was already implemented in the initial prototype that was provided by Blue Byte GmbH, player AI was not included in that prototype. Thus, the main task of the development of the AI for the ZVG was to develop player a AI as one of the parts of the final BE.

Since the main balancing scene in the ZVG includes two game objects (zombies and food) that the player can interact with, the player AI should be able to interact with these objects in the same way as human player would. For most of the time the player AI should fight with zombies, but when it runs out of health or food it should move to the safe spot or the resource spot accordingly.

The conceptual development of player AI was performed by using behavior trees, and the resulting behavior tree is presented in the following figure.

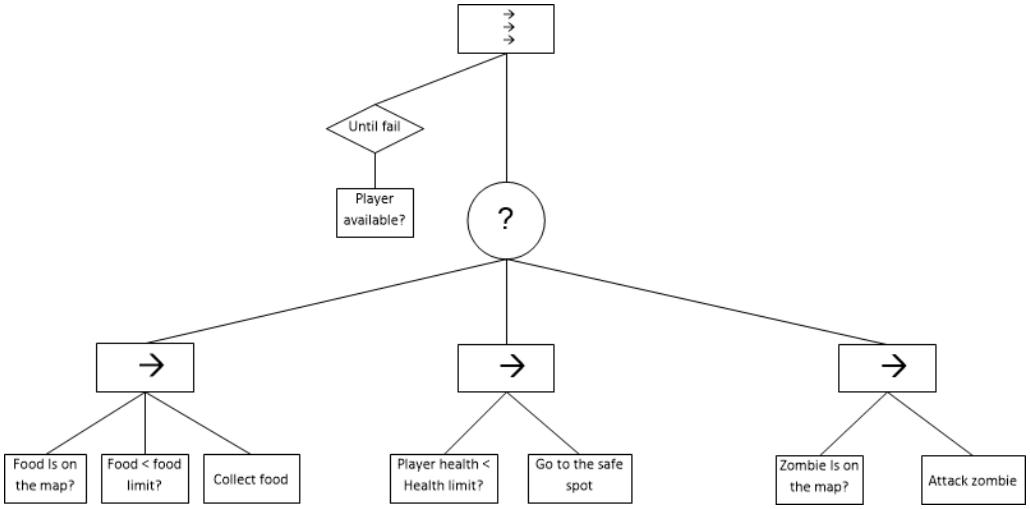


Figure 15: Behavior tree for the ZVG.

1. This tree starts with the parallel task that checks if there is a player unit available. If it is not available, then there are no player units on the map and AI does not perform any further actions.

2. If there is a player unit on the map that is available, then the AI starts to perform actions with this player unit. The next task is the selector that starts with the first sequence task. This task checks two conditions: are there any resource spots (i.e. food) on the map and has the player AI reached the food

limit. If the conditions are met, then the player AI should place a player unit near the resource spot in order to collect food.

3. Nevertheless, if the conditions of the first sequence task have not been met, then the selector task chooses the second sequence task that checks the amount of health of player units. If the amount of health of a player unit has fallen below the health limit threshold, then the player AI should place that player unit in the safe spot.

4. If the last condition has not been met, then the selector task chooses the last sequence task that finally checks for the zombies on the map. As the result, if there are any zombies on the map, then the player AI places a player unit near zombie in order to start the fighting. However, if there are no zombies on the map, then the player AI does not perform any actions with a player unit.

The practical implementation of the described concept required the development of four C# classes: `PlayerBehaviour.cs`, `AggresiveAndResourcePlayerBehaviour.cs`, `Scopes.cs` and `Predicates.cs`. First two of the listed classes can be found in the GameInterface – Behaviors folder, while the second two can be found in GameInterface – Utils folder. `PlayerBehaviour.cs` and `AggresiveAndResourcePlayerBehaviour.cs` directly implement the concept described by the behavior tree and should be described in more detail. In contrast, `Scopes.cs` and `Predicates.cs` are supportive classes that primary consist of return methods for `AggresiveAndResourcePlayerBehaviour.cs`, so they are excluded from the detailed analysis.
The first class, called `PlayerBehaviour.cs`, inherits from `MonoBehaviour.cs` and contains player AI parameters as well as methods to move player unit to certain object. The parameters of the player AI include:

1. Bool variable `active` that specifies if player AI is activated or not

2. Integer variable `decisionDelay` that specifies how often the player AI should make a decision

3. Float variable `differentDistance` that specifies the distance of the placement of player units from the safe spot

4. Protected integer `nextDecisionTime` that is used for the calculation of time for the next decision

5. Protected bool `canPlay` that specifies when the player AI is able to make the next decision

Listing 6: public and protected variables of PlayerBehaviour.cs.

```
public abstract class PlayerBehaviour : MonoBehaviour
{
    [Header(("AI active"))]
    [Header(("############ UNCHECK FOR MANUAL PLAY ############")
        )]
    public bool active = true;

    [Header(("AI decision delay interval"))]
    [RangeAttribute(1, 10)]
    public int decisionDelay = 2;

    [Header(("Indifferent distance"))]
    [RangeAttribute(5.0f, 15.0f)]
    public float indifferentDistance = 10.0f;

    public static PlayerBehaviour instance;

    protected int nextDecisionTime = 0;
    protected bool canPlay = false;
```

The calculation of time for the next possible decision is performed by using `decisionDelay`, `nextDecisionTime` and `canPlay` variables in the methods `UpdateCanPlay(int logicTicksSinceStart)`, `SetNextDecisionTime(int nextDecisionTime)` and `CanPlay()`:

Listing 7: The calculation of time for the next possible decision in PlayerBehaviour.cs.

```
    public void UpdateCanPlay(int _logicTicksSinceStart)
    {
        if (_logicTicksSinceStart >= nextDecisionTime)
        {
            nextDecisionTime += decisionDelay;
            canPlay = true;
        }
    }

    public void SetNextDecisionTime(int nextDecisionTime)
    {
        this.nextDecisionTime = nextDecisionTime;
```

```
13          canPlay = false;
14      }
15
16      public bool CanPlay()
17      {
18          return canPlay & active;
19      }
```

The placement of player unit is performed by the methods `MovePlayerUnitTo (ClientGridObject playerUnit, int _x, int _z)`, `MovePlayerUnitToSafePlace (ClientGridObject playerUnit, int _x, int _z)` and `MovePlayerUnitToObject(ClientGrid playerUnit, ClientGridObject zombieOrResourceUnit)`. The difference between `MovePlayerUnitToObject` and `MovePlayerUnitToSafePlace` is that the first one uses game objects such as zombie or resource spot as an input for the placement, while the second one uses coordinates for the placement. Both of the methods call `MovePlayerUnitTo` in order to perform actual placement, and this method performs placement by calling `PlaceNewObjectsOnMap` method:

Listing 8: The calculation of placement in PlayerBehaviour.cs.

```
1       protected void MovePlayerUnitTo(ClientGridObject playerUnit,
            int _x, int _z)
2       {
3           PosIntXZ newPosition = Utils.GetFreePositionAroundPoint(
                _x, _z);
4
5           //UpdateGridObjectsOnMap.MoveGridObjectToResidence (
                playerUnit);
6           PlaceNewObjectsOnMap.instance.StartPlaceGridObject(
                playerUnit, _x, _z);
7           PlaceNewObjectsOnMap.instance.PlacementRequesterOkPressed
                ();
8       }
9
10      protected void MovePlayerUnitToSafePlace(ClientGridObject
            playerUnit, int _x, int _z)
11      {
12          MovePlayerUnitTo(playerUnit, _x, _z);
13      }
14
15      protected void MovePlayerUnitToObject(ClientGridObject
            playerUnit, ClientGridObject zombieOrResourceUnit)
```

```
16      {
17          GameBlockingMap gameBlockingMap = ClientGameLogic.
                instance.GetBlockingMap();
18
19          PosIntXZ placementPosition = gameBlockingMap.
                FindAttackPoint(playerUnit, zombieOrResourceUnit);
20
21          MovePlayerUnitTo(playerUnit, placementPosition.x,
                placementPosition.z);
22      }
```

The second class, called `AggresiveAndResourcePlayerBehaviour.cs`, inherits from `PlayerBehaviour.cs` and contains player AI parameters as well as methods to perform actions according to the behavior tree. The parameters of player AI include:

1. Integer variables `safeX` and `safeZ` that specifies the position of the safe spot
2. Float variable `healthLimit` that specifies the threshold of health when a player unit should be moved to the safe spot
3. Float variable `junkLimit` that specifies the threshold of food when a player unit should be moved to the resource spot

Listing 9: public variables of AggresiveAndResourcePlayerBehaviour.cs.

```
1 public class AggresiveAndResourcePlayerBehaviour :
      PlayerBehaviour
2 {
3       [Header(("Safe coordinate X"))]
4       [RangeAttribute(-1000, 1000)]
5       public int safeX = 100;
6
7       [Header(("Safe coordinate Z"))]
8       [RangeAttribute(-1000, 1000)]
9       public int safeZ = -110;
10
11      [Header(("AI Bot health limit"))]
12      [RangeAttribute(0.0f, 1.0f)]
13      public float healthLimit = 0.05f;
14
15      [Header(("AI Bot junk limit"))]
16      [RangeAttribute(0, 100)]
17      public float junkLimit = 10;
```

The main calculations are performed in the `Play()` method. It starts with the declaration of the instances of the methods from the `Scopes.cs`. Basically they are used to find a player unit, resource spot, or zombie according to certain conditions:

Listing 10: Declaration of methods from Scopes.cs in AggresiveAndResourcePlayerBehaviour.cs.

```
1 public override void Play()
2     {
3         ClientGridObject attackingPlayerUnit = Scopes.
              FirstAliveActivePlayerUnit();
4         ClientGridObject notAttackingPlayerUnit = Scopes.
              FirstAliveActivePlayerUnitNotInCombat();
5         ClientGridObject notAttackingWithHealtPlayerUnit = Scopes
              .
              FirstAliveActivePlayerUnitWithEnoughHealthNotInCombat
              (healthLimit);
6         ClientGridObject gatheringPlayerUnit;
7         List<ClientGridObject> notEmptyResourceSpots = Scopes.
              NotEmptyResourceSpots();
8
9         ClientGridObject notSafePlayerUnit = Scopes.
              FirstAliveActivePlayerUnitWithNotEnoughHealth
              InCombatNotInSafePlace(healthLimit, safeX, safeZ,
              base.indifferentDistance);
10        ClientGridObject attackingZombieUnit = Scopes.
              FirstAliveActiveZombieUnitInCombat();
11        ClientGridObject notAttackingZombieUnit = Scopes.
              FirstAliveActiveZombieUnitNotInCombat();
```

Then this method incorporates the logic from the behavior tree and performs the actions according to the first sequence task: The first action is to check if are there any resource spots on the map and if the player has reached the food limit. If the conditions are true, it picks up a player unit and puts it near resource spot:

Listing 11: First action (try to collect some food) in AggresiveAndResourcePlayerBehaviour.cs.

```
1 //Action 1. If not enough food - collect resources
2         if (Fitness.GetGameJunkAmount() <= junkLimit &&
              notEmptyResourceSpots.Count != 0)
3         {
4             //Choose gathering player unit: one that is not in
                  combat with enough health, just not in combat or
```

```
            even unit in combat
5           if (notAttackingWithHealtPlayerUnit != null)
6           {
7               gatheringPlayerUnit =
                    notAttackingWithHealtPlayerUnit;
8           }
9           else if (notAttackingPlayerUnit != null)
10          {
11              gatheringPlayerUnit = notAttackingPlayerUnit;
12          }
13          else {
14              gatheringPlayerUnit = attackingPlayerUnit;
15          }
16
17          ClientGridObject notNearZombieResourceSpot = Scopes.
                FirstNotOccupiedResourceSpotNotNearZombie(
                gatheringPlayerUnit.gridObjectDefinition);
18          ClientGridObject bestResourceSpot = Scopes.
                BestResourceSpot(gatheringPlayerUnit.
                gridObjectDefinition);
19          //Action 1.1. Move player unit to resource spot
                without zombies
20          if (notNearZombieResourceSpot != null)
21          {
22              MovePlayerUnitToObject(gatheringPlayerUnit,
                    notNearZombieResourceSpot);
23          }
24          //Action 1.2. Move player unit to resource spot with
                zombies
25          else if (bestResourceSpot != null)
26          {
27              MovePlayerUnitToObject(gatheringPlayerUnit,
                    bestResourceSpot);
28          }
29      }
```

If the conditions are false, then this method checks if a player unit has reached the health limit. If yes, it moves player unit to the safe spot:

Listing 12: Second action (try to move player unit to the safe spot) in AggresiveAndResourcePlayerBehaviour.cs.

```
1 //Action 2. If enough food but not enough health - go to safe
      place
```

```
2           else if (notSafePlayerUnit != null)
3           {
4               MovePlayerUnitToSafePlace(notSafePlayerUnit, safeX,
                    safeZ);
5           }
```

If a player unit has not reached the health limit yet, then this method tries to place the player unit near zombie in order to start the fighting:

Listing 13: Third action (try to fight zombies) in AggresiveAndResourcePlayerBehaviour.cs.

```
1 //Action 3. If enough food and enough health - fight zombies
2           else {
3               // Action 3.1. Attack attacking zombie
4               if (notAttackingWithHealtPlayerUnit != null &&
                    attackingZombieUnit != null)
5               {
6                   MovePlayerUnitToObject(
                        notAttackingWithHealtPlayerUnit,
                        attackingZombieUnit);
7               }
8               // Action 3.2. Attack not attacking zombie
9               else if (notAttackingWithHealtPlayerUnit != null &&
                    notAttackingZombieUnit != null)
10              {
11                  MovePlayerUnitToObject(
                        notAttackingWithHealtPlayerUnit,
                        notAttackingZombieUnit);
12              }
13          }
14
15          base.canPlay = false;
```

6.3.3 2D Roguelike Player AI

The 2DR includes both NPC AI and player AI, but as well as in the ZVG NPC or zombie AI was already implemented in the initial game from the Unity Tutorial website, the player AI was not included in that game. However, the standard zombie AI was too simple and also produced some bugs. Thus, the main task of the development of AI for the 2DR was to develop a player AI and a zombie AI as the parts of the final BE.

Since the 2DR includes three game objects (zombies, food, and walls) that the player can interact with, the player AI should be able to interact with these objects in the same way as human player would. For most of the time the player AI should avoid zombies and at the same time try to collect as much food as possible, but with every step the player loses one point of food, and when he tries to destroy a wall he loses 4 points of food in total, so this game involves resource management.

The conceptual development of the player AI was performed by using A* algorithm and an influence map, while for the development of the zombie AI a behavior tree was used. At the first stage of the development of the player AI the concept of influence maps was applied. Every object on a map influences it with a certain power, and such a power is given by a possible change of food produced by a certain game object. For example, one move takes one food, any enemy can steal 10 or 20 food if player is nearby, any food game object can bring 10 or 20 food. Influences interfere with each other. Example of such a map is shown below.

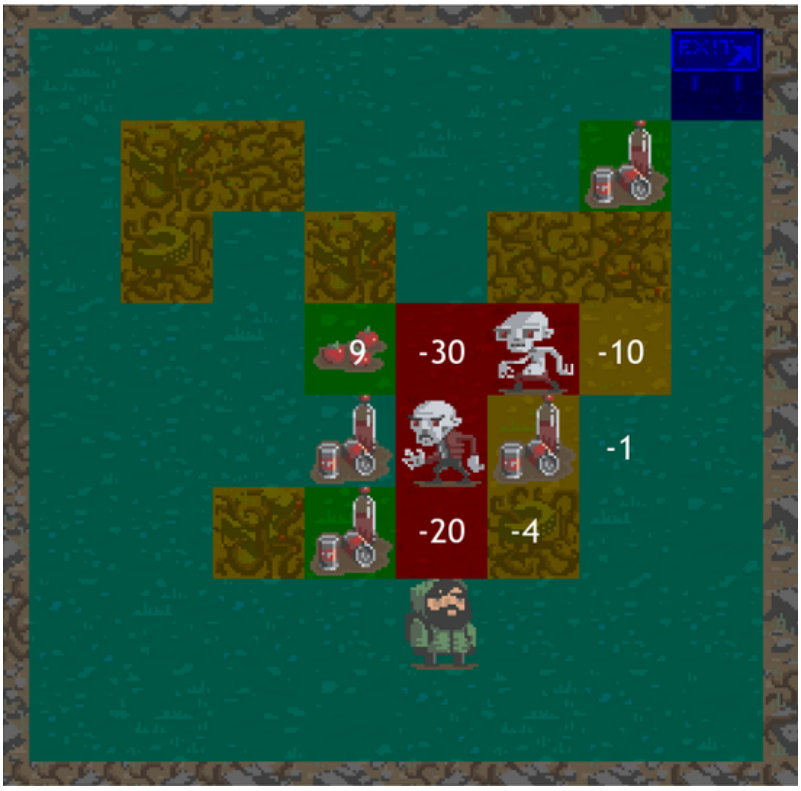


Figure 16: Influence map for the 2DR.

Consequently, A* algorithm uses influence map as a graph to traverse. Every edge to a point on map corresponds to an influence of that point. Heuristic function is

Manhattan distance to the exit, i.e. H^1=|x^e-x^p |+|y^e-y^p | where (x^e, y^e) and (x^p, y^p) coordinates of the exit and a point on map respectively. This algorithm gives significant improvement in terms of average number of survived days. The main issue with that algorithm is that if food game objects are not on shortest path player unit does not pick them up. An xxample of A* Controller is shown below.

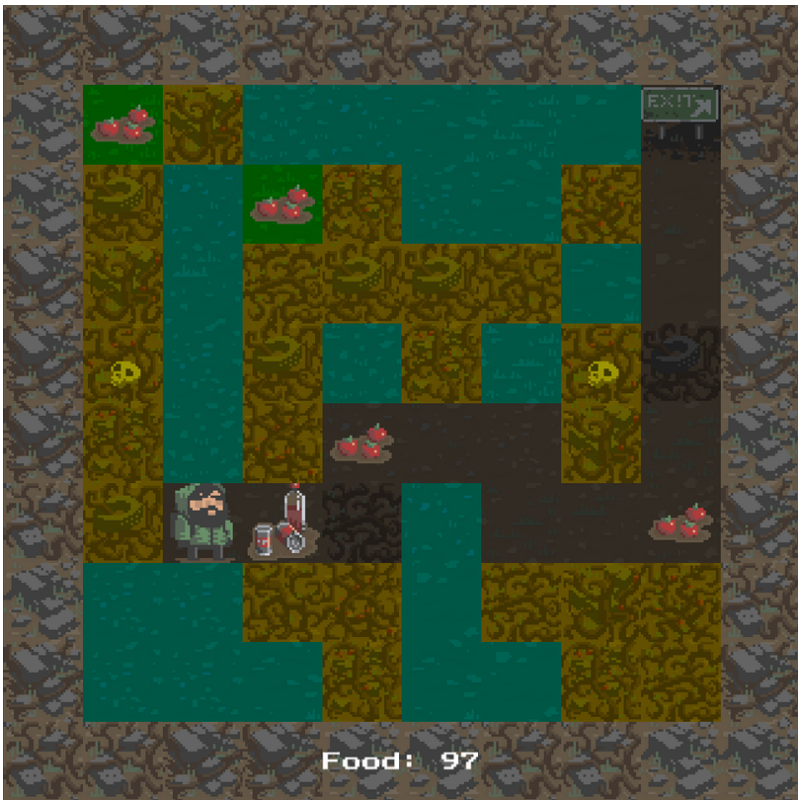


Figure 17: A* pathfinding for the 2DR.

As the result of another project for the "Modern Game AI Approaches" class, there are other player AI algorithms including different types of A* Controller, Random Controller, Euclidian Distance controller, Manhattan Distance Controller, Decision Tree Controller and Monte Carlo Tree Search Controller, but for the evaluation of BE performance for 2DR only A* controller was used. For that reason all other controllers are excluded from the detailed description.

On the other hand, the development of the zombie AI incorporated the concept of behavior trees. The reasons for changing the standard zombie AI that was implemented in the game are: It produced unstable results when the interaction with A* happened, and sometimes zombies blocked the exit and it was not possible for the player to proceed to another level. The standard zombie AI was changed to behavior

tree described below:

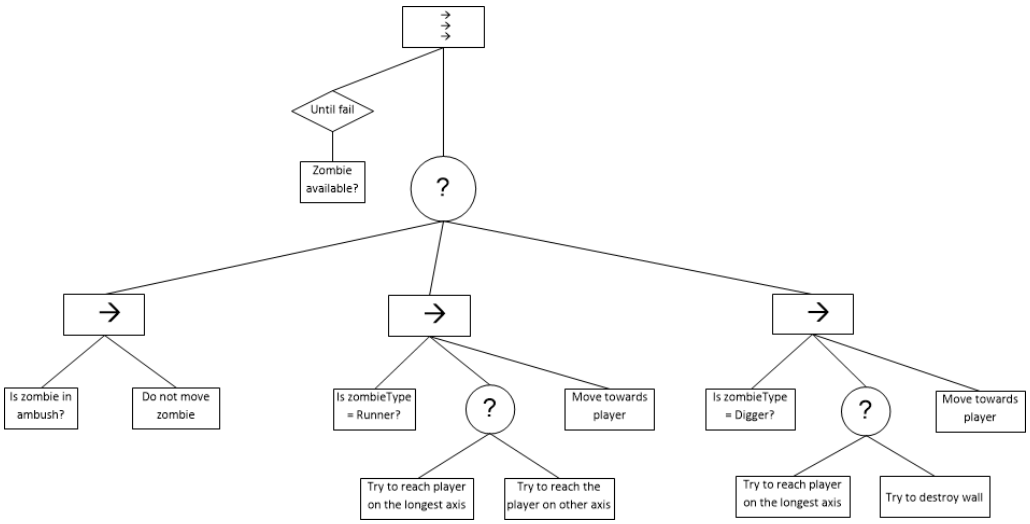


Figure 18: Behavior tree for the 2DR.

1. This tree starts with the parallel task that checks if there is a zombie unit available. If it is not available, then there are no zombie units on the map and AI does not perform any further actions.

2. If there is a zombie unit on the map that is available, then the AI starts to perform actions with this zombie unit. The next task is the selector that starts with the first sequence task. This task checks is this zombie unit in ambush. A zombie unit is in ambush when it stays near exit and there are other zombies on the map that are closer to the player. If the zombie is in ambush, it does not move until the player will be near the exit and the zombie can start fighting.

3. Nevertheless, if the conditions of the first sequence task have not been met, then the selector task chooses the second sequence task that checks if the type of this zombie is a runner zombie. The choice of the zombie type depends on the sprite model of the zombie, and since there are two types of sprites for the zombies in 2DR, there are two types of zombies and runner is the first type. If the zombie is considered to be a runner zombie, then it tries to move towards player avoiding any obstacle on its way.

4. If the last condition has not been met, then the selector task chooses the last sequence task that finally checks if the type of this zombie is a digger zombie. In contrast to runner zombie, digger zombie tries to move towards player destroying walls on its way.

The practical implementation of the described concept required the development of two C# classes: `AStarController.cs` and `EnemyController.cs`. Both of the listed classes can be found in the Completed – Scripts - Controllers folder. `AStarController.cs` implements the concept of the A* algorithm, while `EnemyController.cs` implements the concept of the behavior tree for the zombie AI.

6.4 Optimization Algorithm

6.4.1 Concept

The BE incorporates CI techniques. At this point, it is important to distinguish between CI and AI within the context of this PS, because "there is no agreement on the exact meaning of the terms Artificial Intelligence (AI) and Computational Intelligence (CI)."[YT14] In context of this PS and in line with chapter 3.5, AI refers to the either the NPC behavior or simulated player behavior (player AI). On the other hand CI refers to the methodology applied to optimize game content. In this sense, the BE incorporates CI techniques where the goal of automated game balancing can be reformulated as an optimization task in which the optimization problem is to find the best set of parameters that offers a balanced game experience, which is measured by a fitness function. More specifically this optimization problem is of discrete combinatorial nature defined by the configuration possibilities of the types of game parameters and their discrete value ranges. Depending on the value ranges and the number of game parameter types, the concurrent search space can be calculated by multiplying all of the absolute values of the ranges. The following table contains the applied standard set of game parameters, which translates into a search space of 220,000 possible configurations.

Game parameter	**Value range**
Time to consume	$\in [1, 5]$
Player unit attack power	$\in [1, 20]$
Player unit attack distance	$\in [1, 20]$
Enemy unit health	$\in [90, 100]$
Enemy unit attack power	$\in [1, 10]$

Table 11: Standard set of Game Parameter.

In order to find a good configuration in a search space where the problem size leads to a super-polynomial running-time of an exact method, it is canonical conception in research to apply non-deterministic algorithms that deliver robust results within reasonable i.e. polynomial time [ES08]. According to [Tal09], metaheuristics fall into the category of approximate algorithms and break down into the two sub-categories, namely single-solution based and population-based methods (see figure 19).

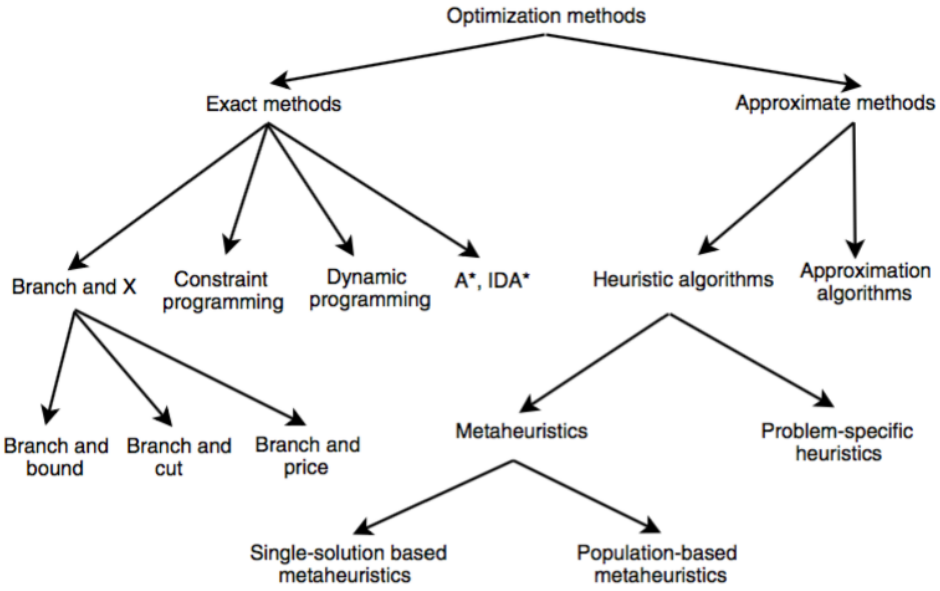


Figure 19: Optimization methods [Tal09] .

Both methods represent a special heuristic or in other words certain strategy to search the problem space efficiently. If applied correctly, these methods are known to generate high-quality solutions, however "there is no guarantee of finding a global optimal solution." [Tal09]

The outlined search space of table 11 with 220,000 possible solutions translates into a run-time of approximately 70 days. In the case of this PS, this is mainly due to the necessity of simulating every single solution in real game-time speed, where one solution is equal to an assessed average playtime of half a minute. If this necessity had been obsolete, the simulation time could have been reduced immensely. Therefore, a search space of 220,000 configurations may have theoretically been solved by an exact method. However, in order to offer a scaleable approach and circumvent the given restriction of simulating in real game-time speed, the use of population-based metaheuristics provides an appropriate methodology to continuously deliver high-

quality solutions even when the amount of parameters and their value ranges will be increased. Among the different variants of population-based metaheuristics exist EC, which has established its applicability to discrete optimization problems. For this reason, the optimization algorithms that have been applied within the BE are built on the principles of EA problem solvers.

The general workings of an EA is inspired by the idea of mimicking Darwin's theory of 'survival of the fittest', a natural selection mechanism that favors those individuals within the population that are adapted best to given environmental conditions [ES08]. Based on this nature-intrinsic principle the underlying concept behind all variants of EAs is similar; given a population, i.e. a multiset of individuals (candidate solutions) and some form of environmental pressure (problem), it is expected to see an iterative evolution (development towards solving the problem) in the fitness (value expressed by an objective function). Generally, the process of recombination and mutation (variation operators) creates diversity and novelty (exploration) and the process of selection pushes towards quality (exploitation). Together, selection and variation, form the basis of an EA. While the selection operators only act on the population, the variation operators only act on each individual. The general working of an EA can be seen in pseudocode and figure below.

Algorithm 2: Pseudocode of an evolutionary algorithm

```
Initialize population from random candidate solutions;
Evaluate each candidate;
while Termination criterion is not satisfied do
    Select parents;
    Recombine pairs of parents;
    Mutate resulting offspring;
    Evaluate new candidates;
    Select survivors for the next generation;
end
```

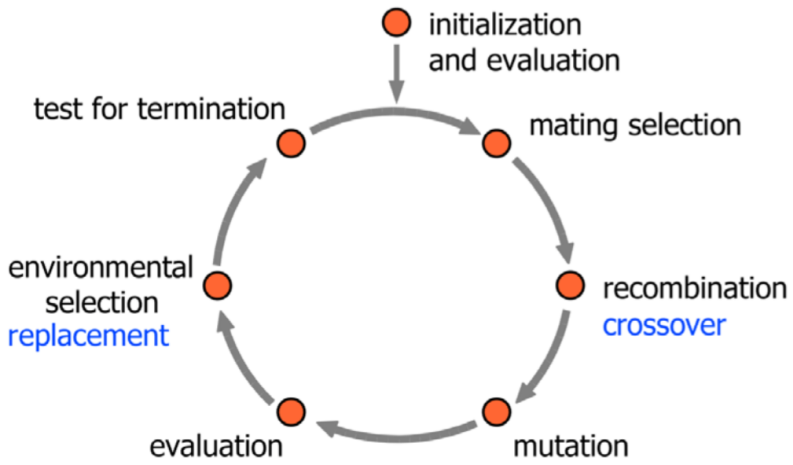


Figure 20: Working cycle of an evolutionary algorithm [Pre15].

By mimicking nature in a computational environment, a set of terminologies derived from nature can be mapped to their corresponding meaning of algorithmic problem solving (see table 12).

Any EA consists of a set of components. These components specify the methodology and operation technique of the EA. They interrelate with each other and their adjustment is influenced by the type of problem.

At first, it is required to define a representation. It is the encoding construct of the individuals' form that serves as modifiable data structure to apply the variation and selection operators as well as the fitness assessment. The scheme of this data structure is often numerical or a binary string of fixed length. In this sense, vectors can be Boolean, real-valued or integer. In the phenotypic approach, individuals are represented internally exactly as they are represented externally. The phenotypic approach has been applied in the case of the PS as it is in itself a useful representation for optimization problems that involve evolving a combinatorial path or permutation [ES08]. It is therefore of integer nature due to the discrete value ranges. Importantly, a representation must be complete (entire search space), connected (search path among solutions) and efficient (reduced time complexity), of which the first two are guaranteed by the phenotypic approach. Working with the integer values also satisfies the efficiency clause. The table 11 gives an overview of the standard setting

Evolution	Problem solving
Environment	Problem space OR search space
Individual	Candidate solution
Population	Current set of candidate solutions limited by a predefined size
Generation	Number of any given population during algorithm runtime
Parent	Candidate solution that is part of the population
Child	Changed copy of a parent
Fitness	Quality
Fitness function	Objective function OR utility function OR cost function
Chromosome OR genome OR genotype	data structure of the candidate solution
Gene OR locus	A particular position in a chromosome
Allele	Value of the gene
Selection	Selection of candidate solutions based on their quality
Mutation	Probabilistic variation of a candidate solutions
Recombination OR crossover	Copying and swapping parents' information

Table 12: Evolutionary algorithm metaphor.

of the representation at hand.

Based on this representation the EA generates a random set of candidate solutions serving as initial population. The recorded run-time 289 serves here as an exemplary case with the following setups:

BE Component	Setting
Game	Zombie village game
Scene	Scene to balance
Goal	Survival goal
Player AI	Aggressive and resource player behavior
Optimization algorithm	Evolutionary algorithm with single-point crossover
Fitness function	Weighted fitness function

Table 13: BE setup of run-time 289.

AI Parameter	Setting
AI active	YES
AI decision delay interval	2
Indifferent distance	10
AI health limit	0.05
AI junk limit	10

Table 14: Standard setup of the Aggressive and resource player behavior (also for run-time 289).

SG Parameter	Setting
Health limit	0.05
Number of alive player unit	1
Target amount of health	50
Health weight	1
Target amount of food	5
Food weight	6
Starting food	20

Table 15: Standard setup of the Survival goal (also for run-time 289).

EA Component	Setting
Population initialization	Uniform random
Population size μ	10
Number of offspring λ	20
Simulation time limit (max generations)	21
Random number seed (positive integer)	361480
Parent selection operator	Best two of μ
Crossover operator	Single-point OR 1-point
Mutation operator	Gaussian perturbation ($\sigma = 1.95$)
Survivor selection	$(\mu + \lambda)$

Table 16: Setup of evolutionary algorithm with single-point crossover for run-time 289.

Individual	Solution ID	Gene values	Fitness 1	Goal achieved
μ_{i_1}	Sol_1	(4,6,19,97,10)	116	False
μ_{i_2}	Sol_2	(5,7,20,90,2)	296	True
μ_{i_3}	Sol_3	(1,9,1,91,8)	82	False
μ_{i_4}	Sol_4	(1,19,13,93,8)	262	True
μ_{i_5}	Sol_5	(5,4,17,98,7)	112	False
μ_{i_6}	Sol_6	(3,14,19,91,5)	311	True
μ_{i_7}	Sol_7	(1,5,7,98,6)	58	False
μ_{i_8}	Sol_8	(5,10,20,99,9)	109	False
μ_{i_9}	Sol_9	(5,7,13,92,5)	53	True
$\mu_{i_{10}}$	Sol_{10}	(3,18,10,97,1)	385	True

Table 17: Initial population data (generation 1) of run-time 289.

At this point in the simulation, an initial population of ten individuals has been randomly generated from the set of permissible value ranges and their fitness values have been computed. Each individual has a unique Solution ID defined by the birth-time of the simulated playtime. The table 17 shows a high variance in the fitness, due to the randomized initialization procedure. The fitness values are computed by the objective fitness function $g(x) = |T_h - \sum P_{h_{t=end}}| + (6 * |(T_f - P_{f_{t=end}})|$ where $|T_h - \sum P_h|$ expresses the absolute distance of the sum of all player units' health $(\sum P_h)$ remaining at the end of a simulated playtime $(t = end)$ to the target health value $((T_h)$ see table 15) and where $|(T_f - P_{f_{t=end}})|$ is the absolute distance of the player's remaining food $(P_{f_{t=end}})$ to the target food value $((T_f)$ see table 15). The expression $|(T_f - P_{f_{t=end}})|$ is weighted times six in order to even it out with the absolute values of $|(T_h - \sum P_{h_{t=end}})|$ and assign both an equally important weight when measured. The objective is to minimize the distances and consecutively the overall result. The target values are set manually based on manual player experience and serve as optimal values for a player experience that should neither be too hard nor too easy and serves the balancing goals of table 8. It is however important to also take into account whether a goal has been achieved or not. In the survival goal, the game is lost either when all player units have died or all food has been consumed. Since the BE is computing with absolute positive values only, it is possible to get a very low result even when the game has been lost. In our initial population, (Sol_7) marks an example of this case.

From the initial population (μ) the best two parents are selected for recombination. In the case above, (i_7) and (i_9) are selected to produce twenty offspring (λ) as depicted in the table below.

Individual	**Solution ID**	**Gene values**	**Fitness 1**	**Goal achieved**
λ_{i_1}	Sol_{11}	(4,18,13,95,10)	230	True
λ_{i_2}	Sol_{12}	(3,10,12,91,8)	82	True
λ_{i_3}	Sol_{13}	(5,6,15,96,8)	112	False
λ_{i_4}	Sol_{14}	(2,15,15,90,1)	363	True
λ_{i_5}	Sol_{15}	(5,20,10,92,10)	316	True
λ_{i_6}	Sol_{16}	(1,8,14,93,4)	88	True
λ_{i_7}	Sol_{17}	(5,19,12,93,8)	334	True
λ_{i_8}	Sol_{18}	(1,10,12,92,6)	64	True
λ_{i_9}	Sol_{19}	(3,9,11,93,8)	74	True
$\lambda_{i_{10}}$	Sol_{20}	(1,20,14,91,1)	235	True
$\lambda_{i_{11}}$	Sol_{21}	(4,6,13,91,5)	111	False
$\lambda_{i_{12}}$	Sol_{22}	(3,20,12,90,7)	111	True
$\lambda_{i_{13}}$	Sol_{23}	(5,9,11,91,5)	139	True
$\lambda_{i_{14}}$	Sol_{24}	(1,20,11,92,5)	289	True
$\lambda_{i_{15}}$	Sol_{25}	(5,6,14,93,6)	100	False
$\lambda_{i_{16}}$	Sol_{26}	(4,16,12,94,6)	322	True
$\lambda_{i_{17}}$	Sol_{27}	(5,20,12,93,8)	334	True
$\lambda_{i_{18}}$	Sol_{28}	(1,28,1,4,16)	94	False
$\lambda_{i_{19}}$	Sol_{29}	(5,5,13,96,5)	87	False
$\lambda_{i_{20}}$	Sol_{30}	(1,20,14,90,6)	87	True

Table 18: Generated offspring from generation 1 during run-time 289 .

The given offspring ($\lambda_{i_1,i_{20}}$) have at this point also already been subject to the mutation operator. Therefore, in the next step the algorithm will now select those individuals for the next generation that have best fitness values from the pool of $(\mu + \lambda)$.

Individual	Solution ID	Gene values	Fitness 1	Goal achieved
μ_{i_1}	Sol_{11}	(4,18,13,95,10)	230	True
μ_{i_2}	Sol_{30}	(1,20,14,90,6)	87	True
μ_{i_3}	Sol_{23}	(5,9,11,91,5)	139	True
μ_{i_4}	Sol_4	(1,19,13,93,8)	262	True
μ_{i_5}	Sol_{12}	(3,10,12,91,8)	82	True
μ_{i_6}	Sol_{24}	(1,20,11,92,5)	289	True
μ_{i_7}	Sol_{16}	(1,8,14,93,4)	88	True
μ_{i_8}	Sol_{18}	(1,10,12,92,6)	64	True
μ_{i_9}	Sol_9	(5,7,13,92,5)	53	True
$\mu_{i_{10}}$	Sol_{19}	(3,9,11,93,8)	74	True

Table 19: Population data (generation 2) of run-time 289 .

From the table above it becomes clear that due to the fact that ($\mu < \lambda$) the selection pressure is quite high and the majority of parents from the generation 1 has been replaced with produced offspring. While generation 1 exhibited a mean fitness value of 178, the mean fitness value of generation 2 has decreased to 156 while the condition whether the goal has been achieved is now always True. Since the optimization problem is a minimization problem, this is an improvement. This optimization cycle continues until we have reached the termination criterion which is a maximum number of generations of 21 in this case.

Individual	Solution ID	Gene values	Fitness 1	Goal achieved
μ_{i_1}	Sol_{156}	(5,7,12,93,5)	53	True
μ_{i_2}	Sol_{60}	(5,7,11,93,5)	53	True
μ_{i_3}	Sol_{50}	(5,7,12,92,5)	53	True
μ_{i_4}	Sol_{210}	(5,7,10,91,5)	53	True
μ_{i_5}	Sol_{195}	(5,7,15,98,5)	46	True
μ_{i_6}	Sol_{378}	(4,7,10,92,5)	41	True
μ_{i_7}	Sol_{234}	(4,7,10,95,5)	41	True
μ_{i_8}	Sol_{273}	(5,7,10,95,5)	53	True
μ_{i_9}	Sol_{9}	(5,7,13,92,5)	53	True
$\mu_{i_{10}}$	Sol_{179}	(5,7,9,94,5)	53	True

Table 20: Population data (generation 21) of run-time 289 .

The final population demonstrates that almost every individual from the initial population has been replaced with a better version. The mean value is 50, which is a clear indicator that the algorithm provides convergence. However, a common limitation of metaheuristics is that the optimization algorithm may get stuck at a local optima. This may have been the case in run-time 289. As can be seen from the figure below the last generations do not continue to further evolve significantly and seem to be stalled at around a value of 50. From other run-times (e.g. run-time 294) it can be learned that it is indeed possible to further converge towards a value near or equal to 0. On the other hand, run-time 289 may still evolve even further if the termination criterion had been set for a longer run-time.

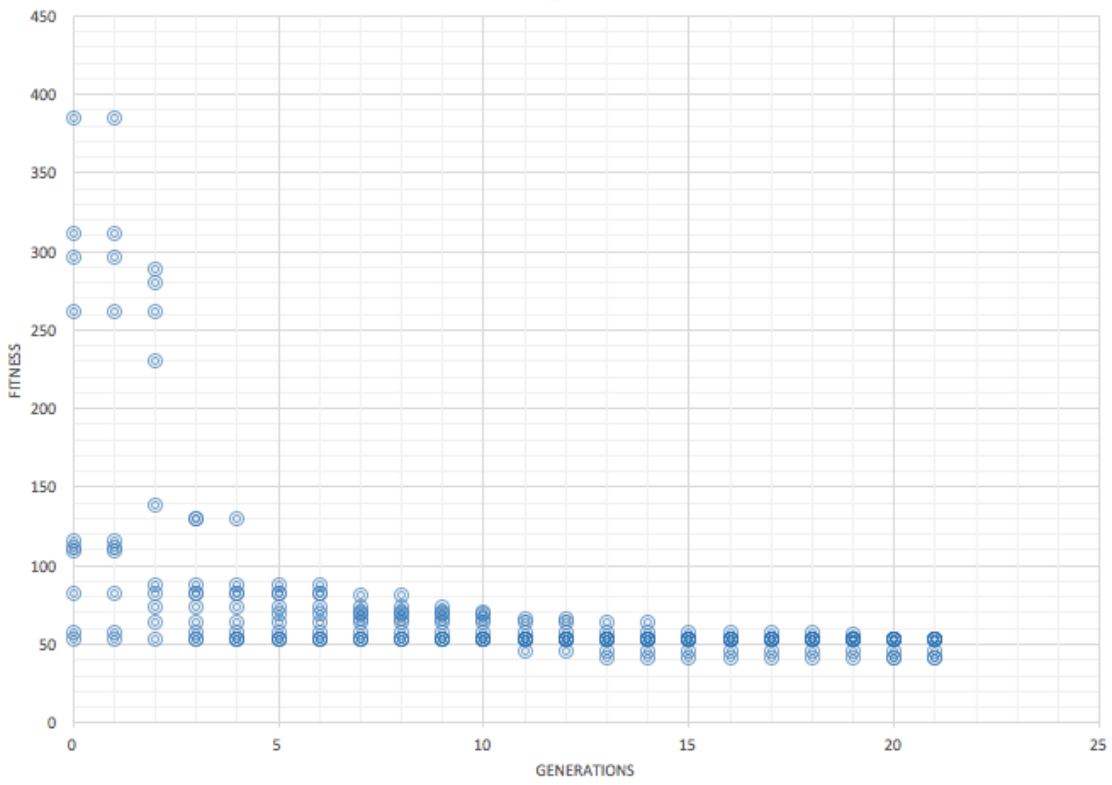


Figure 21: Development of the generations' fitness of algorithm runtime 289.

On the other hand, solutions that seem to be a representation of local or even global optima may be suitable enough candidates for providing a balanced player experience. For this reason, a selection of optimal or near-optimal solutions have been manually tested and assessed. The results are seen in the chapter manual balancing.

6.4.2 Evolutionary Algorithm Variants

Table 16 shows one specific variant of an EA. This specific variant has a strong bias towards the best solutions given the parent selection strategy, which may have quite exploitative impacts on the working of the algorithm. On the hand the standard deviation of the Gaussian perturbation is set to high value, which may provide enough exploration and counter balance the parent selection strategy and indeed the ranges of the produced offspring show a high variance while the final population demonstrates that a solution from the initial population can continue to exist if it is good enough. Together this could be interpreted as an indicator for enough exploitation and exploration. However, in order to substantiate this finding, several other variants of EA were built for comparison (see table 21. In chapter 5.5, the F-Race seeks out to find the best configuration in terms of validity, reliability and time

complexity. All of those variants have a uniform random population initialization with $(\mu = n)$, where (n) is a variable that is set manually before run-time. Secondly, the number offspring (λ) is set to be higher than the number of parents (μ) with $(\lambda = m)$, where (m) is a variable that is set manually before run-time with $(m > n)$. Thirdly, the termination criterion is always set to a maximum number of generations (g), where (g) is a variable that is set manually before run-time with $(g = z, \frac{z}{\lambda} \approx 400)$. Finally, it is necessary to provide a random number seed which is a variable that is set manually before run-time with positive integers.

Component	**EA0**	**EA1**	**EA2**
Parent selection	Best two of μ	Best two of μ from random r, $r \subset n$	Fitness-proportional with sigma scaling
Crossover	Uniform random	Uniform random	Uniform random
Mutation	Gaussian perturbation ($0.1 \leq \sigma \leq 4$)	Gaussian perturbation ($0.1 \leq \sigma \leq 4$)	Gaussian perturbation ($0.1 \leq \sigma \leq 4$)
Survivor selection	$(\mu + \lambda)$	$(\mu + \lambda)$	$(\mu + \lambda)$

Component	**EA3**	**EA4**	**EA5**
Parent selection	Best two of μ	Best two of μ	Best two of μ
Crossover	Single-point	Uniform random	Uniform random
Mutation	Gaussian perturbation ($0.1 \leq \sigma \leq 4$)	Gaussian perturbation ($0.1 \leq \sigma \leq 4$) with mutation probability ($0.01 < p_m \leq 0.3$)	Gaussian perturbation ($0.1 \leq \sigma \leq 4$)
Survivor selection	$(\mu + \lambda)$	$(\mu + \lambda)$	Fitness-proportional with sigma scaling

Table 21: Different applied variants of the evolutionary algorithm.

General conception of designing the different variants is to only change the strategy of one component at the time and keep the remaining components of the base variant EA0 constant. This allows comparison of the different techniques and helps to identify which of the strategies serves best compared to the base variant. Firstly, the parent selection of EA1 is reduced in exploitation by randomly setting up a pool of candidates for mating of which the best two are then selected, which is a tournament selection where (r) defines the selection pressure [ES08]. The closer the value of

(r) is to the value of (μ) the higher the selection pressure. A value of 4 provides a reasonably small decrease in selection pressure that. The Fitness-Proportional method in EA2 selects the parents based on the probability of their absolute values. In this sense, very good solutions take over which may lead to premature convergence, however, in comparison with the GENITOR method of the base variant, it offers a higher exploration. The applied sigma scaling incorporates information about the mean fitness and is the constant c is set to the standard value of 2 [ES08]. The single-point crossover method in EA3 reduces exploration as it maintains most of the parents' information with the downside of a strong bias towards the first and last gene of the chromosome. The mutation probability (pm) in EA4 is applied to further control the standard deviation of the Gaussian perturbation towards a mutation that in average changes on gene per chromosome [ES08]. The Fitness-Proportional method in EA5 has the same impact as in EA3, only that it occurs during the survivor selection.

6.4.3 Balancing Environment Instructions

In the following the steps to set up the balancing environment for a specific scene are explained. These instructions serve as a guideline to apply the balancing environment to the two used games during this project in Unity. Exemplary pictures for the purpose of visualizing the instructions are taken from the ZVG. Even though, the instructions are also valid for the 2DR game.

1. Chose and load the scene that is to balance.

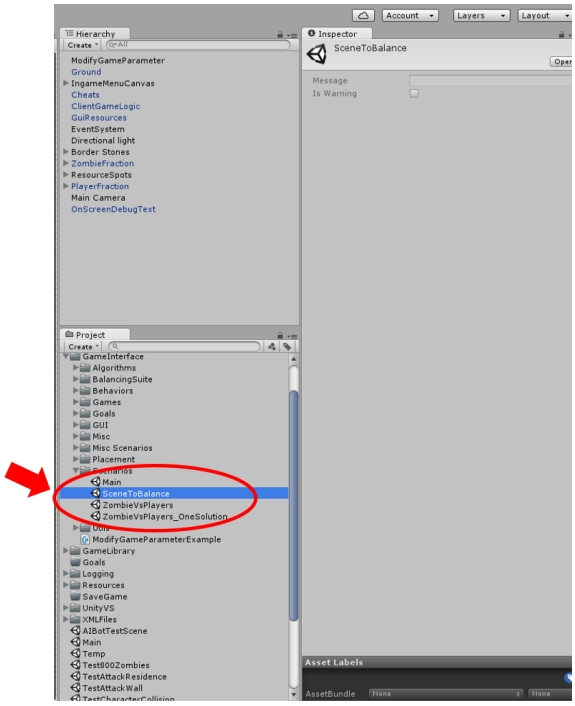


Figure 22: Choosing a scene to balance.

2. Make sure the chosen scene has the game object *ModifyGameParameter* to attach your scripts to. All following scripts will be attached to this game object.

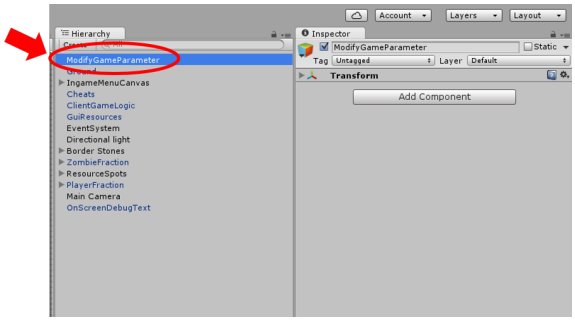


Figure 23: Necessary game object *ModifyGameParameter*.

3. Next, add the `ComplexGame.cs` script to the game object *ModifyGameParameter*.

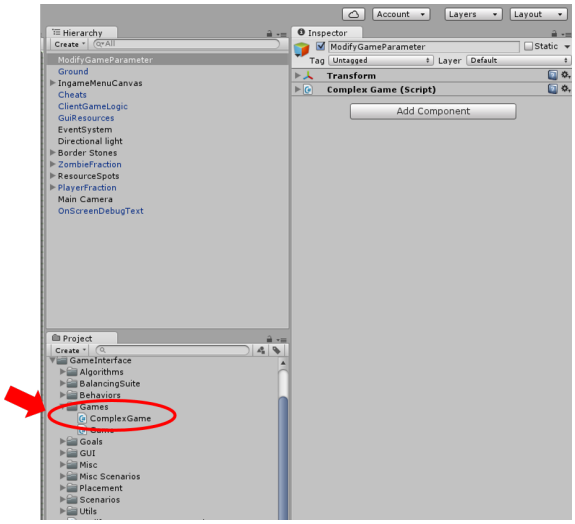


Figure 24: Adding ComplexGame.cs.

4. Attach the frame of the Balancing Environment through attaching `BalancingSuite.cs` to *ModifyGameParameter*.

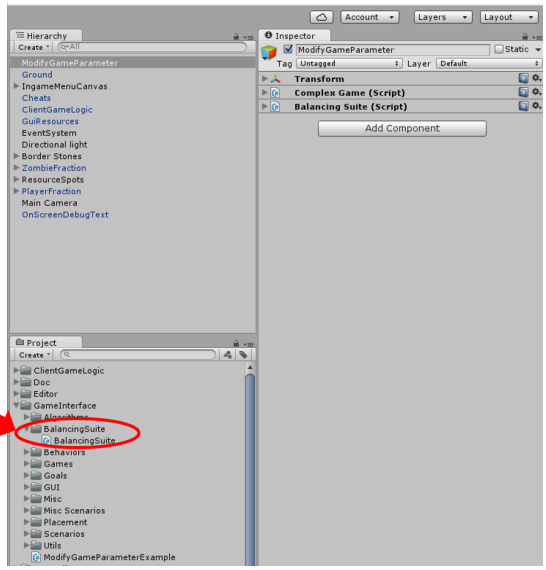


Figure 25: Attaching the Balancing Environment frame.

5. For logging future results give the sessions log file a name.

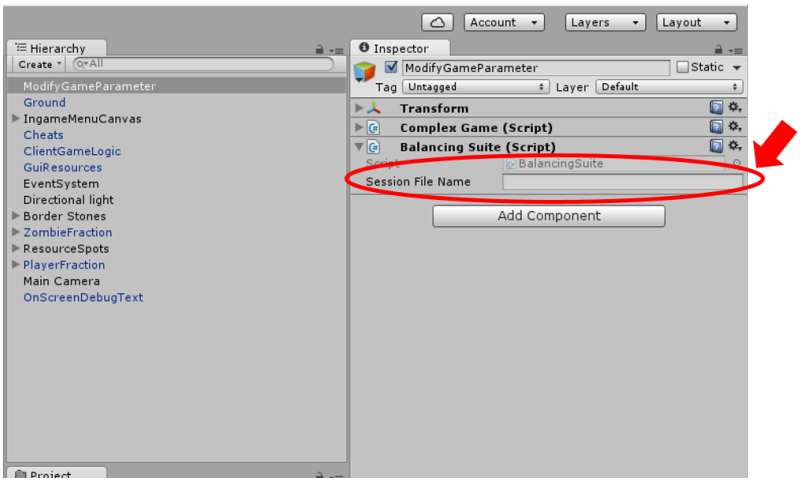


Figure 26: Name for session log file.

6. Enable the Player AI through attaching the relevant script (here: `AggressiveAndResourcePlayerBehaviour.cs`) to *ModifyGameParameter* and set the AI relevant parameters.

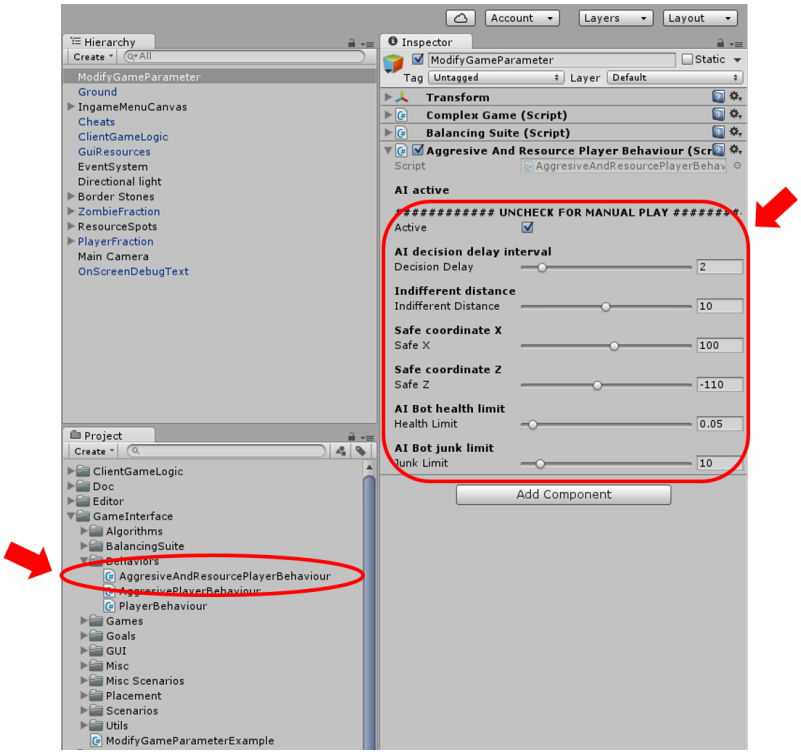


Figure 27: Player AI configuration.

7. Chose a goal for the current scene, attach the relevant goal script to *ModifyGameParameter* (here: `SurivalGoal.cs`) and set the goal relevant parame-

ters.

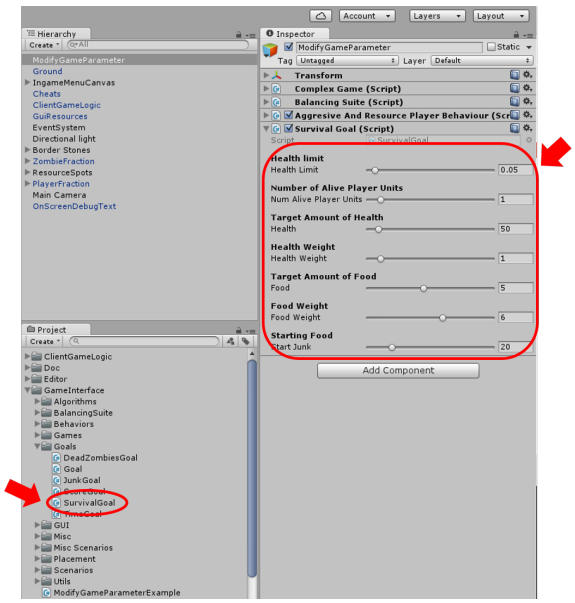


Figure 28: Goal configuration.

8. Choose an algorithm for parameter optimization by attaching one of the algorithm scripts (here: `EA.cs`) and set the algorithm relevant parameters. Give the population log file a name for future documentation of results.

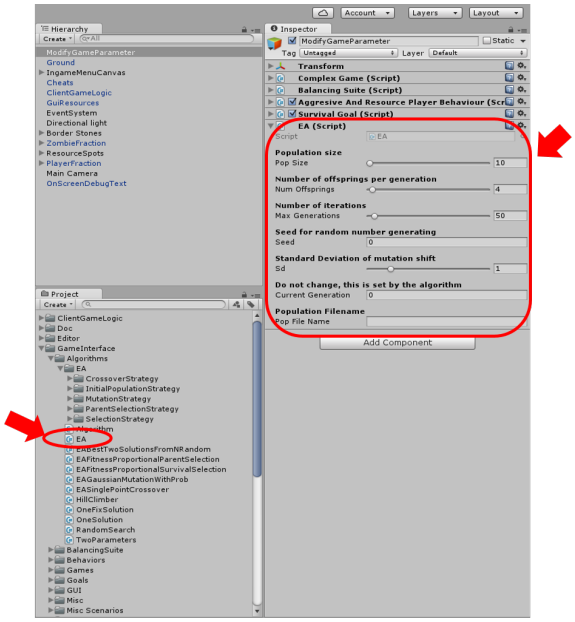


Figure 29: Algorithm configuration.

The BE is now set up. The simulation can be started by pressing the Unity "Play"-

Button. The above-shown example of setting up the BE shows the case of the ZVG. The methodology is the same for the 2DR game.

6.4.4 Technical Implementation

From the technical perspective the BE is a set of C# scripts which are developed using principles of object-oriented programming. The scripts can be separated into two levels: level of abstraction and level of realization. The level of abstraction includes abstract classes which define the main structure and functionality of the BE. This level can be reused as a first sketch of a technological standard for automated game balancing. Indeed, it includes all the required interfaces to perform the latter. Developers can use the implemented code and properly extend functionality of provided abstract classes in order to adapt BE for a specific game. The level of realizations shows how this functionality can be extended on the examples of two games ZVG and 2DR. The BE is developed in a modular way, so every component is minimally depended on other components and can be seen as a separate unit. Development highly utilizes the Singleton Pattern, allowing instantiation of only one object of a particular class. This is mostly done due to Unity specificity, however this can be omitted in future versions.

The BE is mostly anchored around two main entities: *Feature* and *Solution*. The feature-entity encapsulates information about one dimension of the search space; for example, it can be player attack distance or player health in ZVG. This entity includes the following fields:

1. `name`: Textual name of the feature.
2. `value`: Numerical value of the feature.
3. `lowerBound, upperBound and isDiscrete`: Values which define the configuration of the search dimension, used further on by the algorithm in order to satisfy optimization prerequisites.

The solution-entity represents one point in the search space with a fitness vector attached. It includes the following fields:

1. `id`: Identifier of the solution.
2. `features`: Array of features which represents one point in the high dimensional space.

3. `fitness`: Vector of numerical fitness values which represent the "goodness" of the point in terms of being balanced.

4. `isGoalAchieved`: Boolean value that indicates "true" whenever the player AI has achieved the goal of the game during a simulation run.

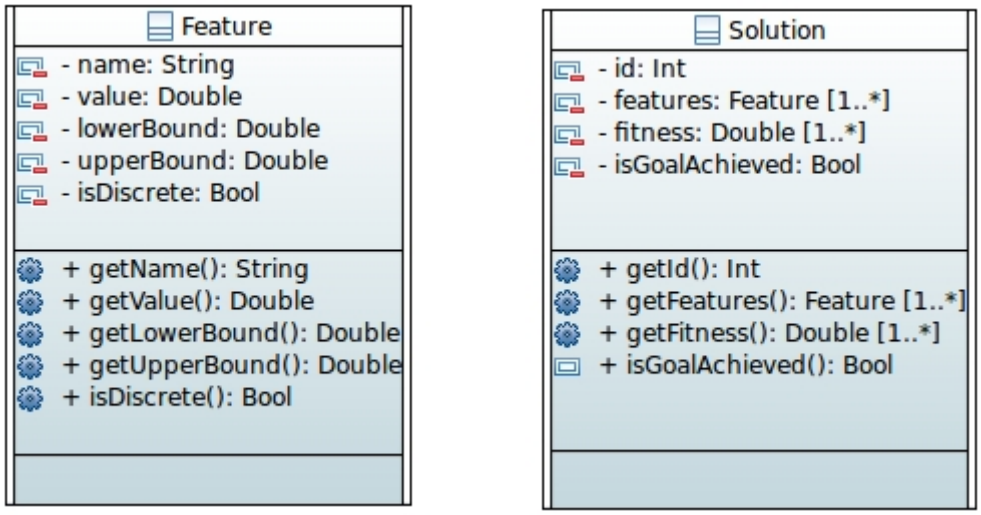


Figure 30: Feature and Solution Entities.

Both entities include only private fields, since after instantiation they cannot be modified, however the basic getters are defined for all the fields. Figure 30 shows these entities, basic constructors are omitted for both entities, since they take all the fields for instantiation.

An important component of the BE is an object that supports communication between itself and a game. The BE checks different parameters of the game and thus these parameters should be passed and accordingly set within the game. The game-entity represents this object and defines required functionality for such communication. It incorporates the following fields and methods:

1. `instance`: Instance of this class - it realizes Singleton Pattern.

2. `awake`: This function is used instead of a constructor to instantiate the object and keep only one instance of the class.

3. `getFeatures`: Abstract method that returns current game features.

4. `setFeatures`: Abstract method that sets current game features for a simulation.

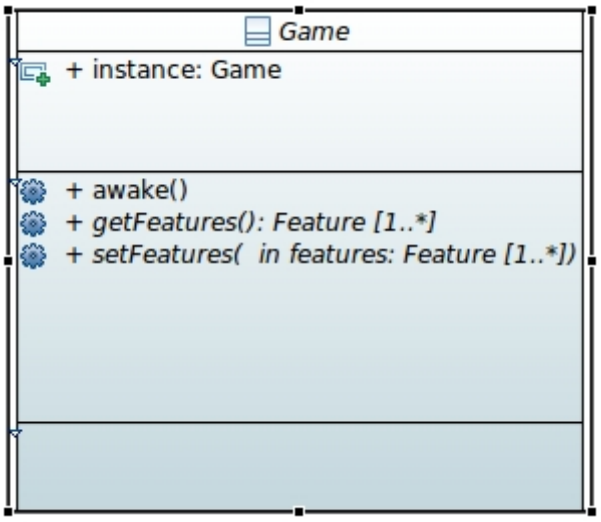


Figure 31: Game Entity.

Game developers should appropriately extend this class in order to adapt the BE for a particular game. Accordingly, functionality of getting and setting game parameters should be realized. Figure 31 depicts the *Game* entity.

The core of the optimization process is the entity *Algorithm*, it represents the optimization algorithm used to find the optimal parameter combination for a game. The abstract class `Algorithm.cs` includes the following fields and methods:

1. `instance`: Instance of this class, it realizes Singleton Pattern.
2. `awake`: This function is used instead of a constructor to instantiate the object and keep only one instance of the class.
3. `getCurrentFeatures`: Method that returns the next features for a simulation.
4. `setCurrentSolution`: Method that delivers the simulated solution.
5. `isTermial`: Method that returns "true" whenever the algorithm reaches the terminal state - for example, when an optimal solution is found.
6. `getBestSolutions`: Method that returns a set of best solutions which are sorted appropriately.

It can be seen from figure 32 that the communication with the optimization algorithm is messaging oriented. This approach makes the BE generic and avoids putting the main emphasis on one entity, which makes the environment more modular.

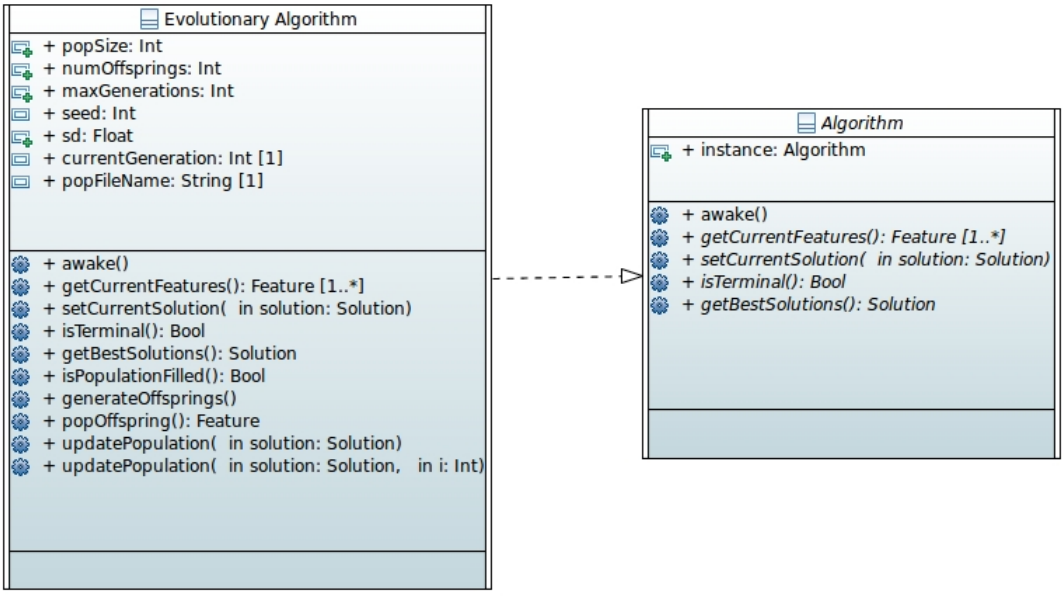


Figure 32: Algorithm Entity and Exemplary Extension Evolutionary Algorithm.

Figure 32 shows an exemplary extension of the algorithm class. It realizes the logic of EAs. On the figure only public fields are shown to present the main logic of the approach. This algorithm is used for optimization during automated game balancing, therefore it should be outlined in more details. From the development point of view most of the evolutionary algorithms are different only in strategies that they use for generating initial population, parent selection, crossover, mutation and selection. Therefore, technically all developed EAs are different only in a way how these strategies are instantiated. For each strategy there is an abstract class (`InitialPopulationStrategy.cs`, `ParentSelectionStrategy.cs`, `CrossoverStrategy.cs`, `MutationStrategy.cs` and `SelectionStrategy.cs`) to implement required functionality, which should be extended further. Thus, having implemented different strategies, various EAs can be constructed just calling the respective class constructors with required parameters. Listing 14 shows the instantiation part of the strategies.

Listing 14: Technical realization of different Evoluationary Algorithms.

```
1 ...
2
3 private InitialPopulationStrategy initialPopulationStrategy;
4 private ParentSelectionStrategy parentSelectionStrategy;
5 private CrossoverStrategy crossoverStrategy;
```

```
6  private MutationStrategy mutationStrategy;
7  private SelectionStrategy selectionStrategy;
8
9  ...
10
11 protected void Awake()
12 {
13     ...
14
15     // determines how initial population is generated
16     initialPopulationStrategy = new RandomPopulationStrategy();
17
18     // determines how parents are selected
19     parentSelectionStrategy = new BestTwoSolutionsStrategy();
20
21     // determines how new offsprings are generated
22     crossoverStrategy = new ProbabilityBiasedCrossoverStrategy
           (0.5);
23
24     // determines how new offsprings are mutated
25     mutationStrategy = new GaussianMutationStrategy(sd);
26
27     // determines how new offsprings are combined with population
28     selectionStrategy = new ReplaceWorstSolutionsStrategy();
29
30     ...
31 }
32
33 ...
```

During the development of the EAs in this PS various strategies were implemented. Based on these strategies different algorithms were constructed, which are described in chapter 6.4.2. However, considering all the combinations of strategies, a higher number of algorithms can be simply developed and applied.

The central entity that brings together all the components: *Goal*, *Game*, *Algorithm*, and *AI*, is the *Balancing Suite*. This entity controls the achievement of the goal, reports to the algorithm game outcome, gets and sets new parameters and restarts the simulation process. A representation of this entity can be seen in figure 33. Class `BalancingSuite.cs` includes the following fields and methods:

1. `instance`: Instance of this class, it realizes Singleton Pattern.

2. `sessionFileName`: File name for the simulation log.
3. `currentFeatures`: Parameters that are simulating at the moment.
4. `runID`: Identifier of the simulation run.
5. `logger`: Logger object which creates logging files and writes logs.
6. `awake`: This function used instead of a constructor to instantiate object and keep only one instance of the class;
7. `eval`: The main method that controls the balancing process. It gets and sets parameters and repetitively restarts the simulation process.
8. `stopSimulation`: The method that closes all logger connections and stops the balancing process.

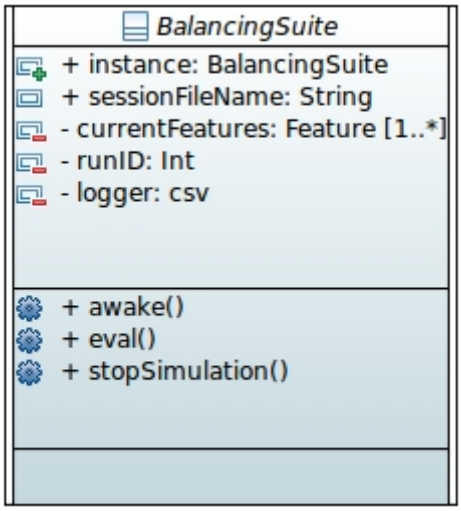


Figure 33: BalancingSuite Entity.

So far the technical implementation of the BE is not exhaustive. Moreover, it is generic and versatile. Developed and applied firstly for balancing ZVG game, it was then translated and easily adapted for 2DR game - a first proof of the flexibility and reuseability of the implementation. The developed approach can thus be enhanced further towards a generic framework for automated game balancing.

6.5 The "Optimal Optimization"

Prior to any computational and human simulation effort, the selection of the best algorithm to the problem should be approached. Even if the type of algorithm has

already been selected, this most likely still has a certain set of parameters itself to determine how it will proceed - like the mutation and crossover methods for example in EAs. We call a specific setting of such parameters the *algorithm configuration*.

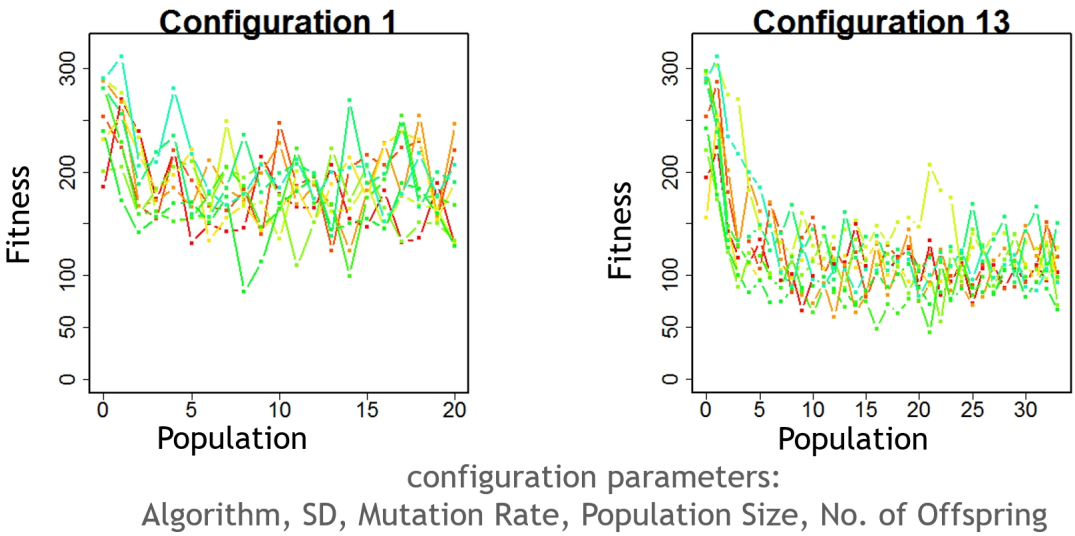


Figure 34: Sample Configurations for F-RACE.

The two graphs shown in figure 34 each display ten simulation runs with the same algorithm configuration applied on the ZVG. What is the configuration that should be preferred, keeping in mind that the minimum fitness value is best? In this case configuration 13 on the right side obviously achieves smaller fitness values and improves faster than configuration 1. One can conclude from this example, that it is beneficial to invest time in "optimizing the optimization" in order to be more efficient in the long run.

Depending on which algorithm parameters are chosen and the value ranges allowed, many possible algorithm combinations exist. The algorithm parameters for the F-RACE were generated randomly from the permissible set of value ranges given in table 22. Firstly, the value ranges for the number of offsprings and the population size were derived from literature in the way that in $(\mu + \lambda)$ selection it is canonical conception of Evolutionary Strategies to have an offspring surplus so that $(\lambda > \mu)$. This induces a larger selection pressure [ES08]. This strategy is applied here in the way that λ has ranges from $11 - 20$, while μ has ranges from $4 - 10$, where the minimum is bound by the necessity of having at least 2 individuals to perform the recombination operator. Secondly, the mutation rate is only applied to creep

mutation of EA4 where each gene is independently being added a small positive or negative value with a mutation probability (p_m), which is usually set so that in average between one gene per offspring is mutated. In the case of balancing the ZVG an individual has a chromosome with 5 genes so that a value of 0.2 for p_m should bring one mutation in average [ES08]. Therefore, for the algorithm parameter search it was chosen to set of range between $0.01 - 0.3$. Thirdly, the standard deviation determines the extent to which a given value is perturbed. This is set between 1 - 4 to match the integer search space of the representation.

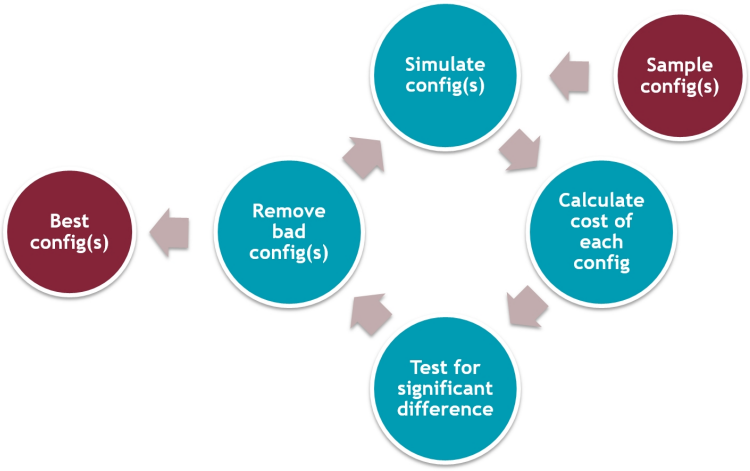


Figure 35: F-RACE Overview.

The method applied to find a good algorithm to minimize the fitness in ZVG is an adapted version of the F-RACE algorithm described in [BCP+10]. The algorithm, as outlined in figure 35, receives an initial set of algorithm configurations from which it determines the best one through the assignment of cost values and calculation of statistical tests.

The adapted F-RACE was implemented in R. An R-package (`race`) exists and was examined first, but owing to the unavailability of an interface between R and Unity in the PS system landscape, it was necessary to implement the algorithm from scratch. Supporting R-packages used for this are: `gsheet` (access the production schedule in Google Drive), `zoo` (calculate the rolling mean in cost function 3), and `dplyr` (additional operations like counting and substitution). Algorithm 3 describes the logic of the implemented code in a simplified way. To be able to perform the actions described in algorithm 3, the configuration files are first read into R using the details provided in the production schedule spreadsheet. Using these details, each con-

figuration run can be assigned to a step looking up its specific seed value. By checking for how many seeds all simulations are available, the code derives the total step size k. A significance level α has to be set prior to running the code which is relevant for the outcome of the Friedman and Conover test. Ideally, α should be between 0.05 and 0.1.

Parameter	**Range**	**Step size**	**Type**
Algorithm	EA1 to EA5	1	character
Standard Deviation	1 to 4	1	integer
Mutation rate	0.01 to 0.3	0.01	decimal
Population size	4 to 10	1	integer
Number of offsprings	11 to 20	1	integer

Table 22: Algorithm configuration parameters for F-RACE.

Algorithm 3: Adapted F-RACE for algorithm selection

```
input  : Configuration files, α
output : Configurations to be removed

k ← No. of steps available;
for i ← 1 to Total Number of Configuration files do
    Calculate cost value for current file;
end
for j ← 1 to k do
    currentBlock ← cost values current step;
    allBlocks ← Union(allBlocks, currentBlock);
    if j > 1 then
        Calculate Friedman test value T;
        if T > 1 − α quantile of χ² then
            Calculate Conover;
            if Conover > t_{1−α/2} then
                Add to configurations to be removed;
            end
        end
    end
end
Return configurations to be removed;
Return best configuration;
```

Both, Friedman and Conover, are rank based tests, using the rank of configurations in the *currentBlock* as well as the sum of the ranks over *allBlocks*. The algorithm

can therefore use this information to return the best ranked configuration. At any given point in time this results in the best configuration found and those removed so far. Applying the *Wilcoxon matched-pairs signed-rank test* when only two configuratiosn are left as described in [BCP+10] will then deliver the overall best configuration.

Different cost functions have been implemented as the idea about cost definition evolved. They can flexibly be interchanged. Specifically, these are:

- **ID of third best**: This cost function accounts for finding at least three good solutions very fast. If at least three solutions below the cost limit are found, the solution ID of the third best solution is assigned as a cost value. In case the algorithm did not produce more than two good solutions, this is penalized by assigning a cost value of 500.
- **Fitness quantiles**: This cost function rewards those configurations that find good solutions fast and reliably and at the same time penalizes those that don't a bit more smoothly. It calculates the cost depending on the number of solutions with a fitness value below the previously assigned cost limit. If at least three solutions below the cost limit are found, the cost for the current configuration is set to the 50% quantile of the fitness. To penalize those configurations where not many good solutions are found, the cost value is set to the 75% quantile of the fitness for these.
- **AUC of top 3 so far**: This cost function looks at how the mean of the best three solutions found so far evolves during the simulation. It calculates the area under the curve of the graph with the solution ID on the x-axis and the mean fitness out of the best three solutions so far on the y-axis.

Table 23 lists the results achieved with the different cost functions after ten steps, meaning ten ZVG simulation runs with the same configuration. The *ID of third best* could not remove configurations using a cost limit of 10 or 20. The *Fitness quantiles* function is very sensitive to the cost limit. For a low cost limit, it seemed very effective, however looking at the remaining configurations, the results do not seem reliable enough. For example, configruations 14 and 15 (see figure36) seem very unpredictable. The cost function *AUC of top 3 so far* does not require any cost limits and it does reliably remove those configurations first, that seem unsatisfactory

also when looking at the visualization. Consequently, this cost function is most recommendable.

Cost function	Cost Limit	To be removed	Best so far
ID of third best	20	NA	NA
ID of third best	10	NA	NA
Fitness quantiles	20	8	17
Fitness quantiles	10	1, 2, 3, 6, 7, 8, 11, 16, 17, 18, 20	12
AUC of top 3 so far	NA	10, 14, 15	19

Table 23: Results of different cost functions for F-RACE.

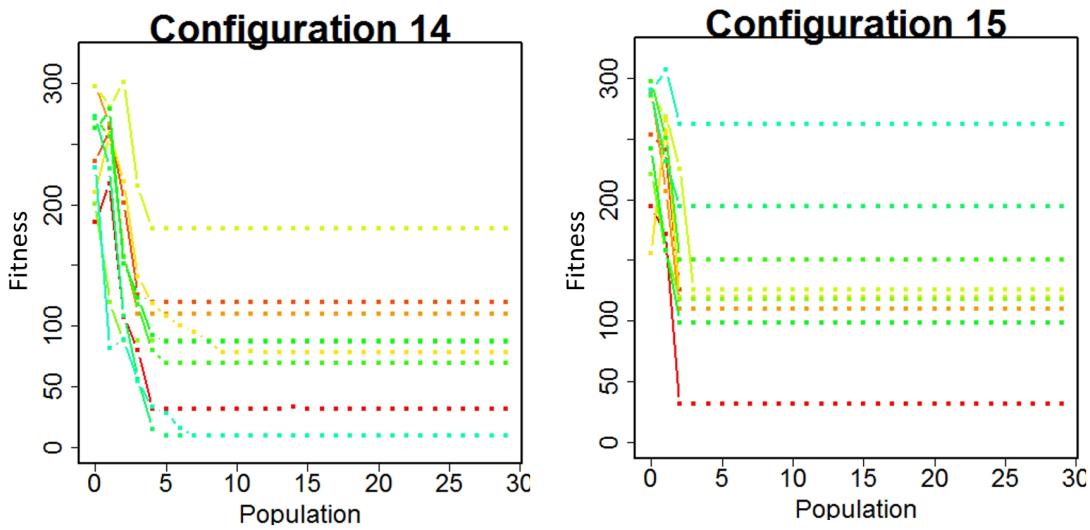


Figure 36: Sample configurations expected to be removed by F-RACE.

To sum up this section, a statistical approach to find a good algorithm configuration for the problem to be tackled is useful. However, this comes with high upfront effort regarding simulation of the game and should rather be considered for problems of higher complexity. The logic implemented in R for ZVG can be applied to other games and problems. Nevertheless, a more enhanced level of automation for the simulation should be envisioned.

7 Balancing Showstoppers

7.1 Communication

The communication with Blue Byte GmbH proved to be very restricted and sometimes unreliable. The game developer responsible for the ZVG prototype did not have time to update or work collaboratively on the prototype, leaving necessary adjustments (e.g. implementation of a winning condition) to the project group. Due to the fact that the developer had commitments of higher priority within Blue Byte GmbH, he could only spend a very limited amount of time on support in resolving the reported bugs and problems. The only way to contact the game developer was via e-mail. However, the feedback often did not help to resolve major bugs that prevented the BE from working reliably.

Initially, it was agreed to have the opportunity to call the game designer via Skype at least once. However, this opportunity was often declined or delayed by the developer due to the other commitments. When the Skype session had finally been arranged it was restricted to text chat instead of a voice call. While this was unfortunate it helped resolving the major bugs, which ultimately led to a reliable working BE, in combination with the ZVG. In conclusion this kept the PS from progressing for several weeks and reduced the time available for simulation and data analysis.

7.2 Bugs: Zombie Village Game

In the process of working with the prototype, several bugs hindered the progress of the implementation. As part of the game logic was hidden due to property rights, fixing the bugs was a tedious procedure and required direct communication with the game creator. In the following is a list of the bugs faced by the team:

Logic Tick Mismatch - Status: fixed Occurring client/server asynchronisation broke the game. This issue arose when there were no zombies in range of the player or when the zombies could not be respawned, i.e. the predefined limit of zombies to spawn was reached.

Out-of-Sync Bug - Status: fixed Sometimes the client and server ticks were restarted equally, but often they got out-of-sync, e.g. "*3 seconds on client side vs. 9 seconds on server side*".

Respawn Object Bug - Status: fixed Respawn objects were not restarted after the first restart, i.e. they did not spawn zombie units.

Zombie Health Bug - Status: fixed Zombie units sometimes remain on the map with a small amount of health points left but are no longer attackable by player units.

Disappearing Interface Objects - Status: fixed With activated player AI, User Interface elements disappear. As a result, the player AI has to be deactivated for manual play mode. We included a checkbox in the interface to switch between manual and automated interaction.

Pathing Through Wall Bug - Status: not fixed Due to attack preferences, Zombies will try to reach player units beyond walls, unable to target the wall first - thus, they are stuck in front of the wall. As a result, the scene we used did not feature any walls or structures.

Simulation Speed Bug - Status: not fixed Increasing the game speed results in behavior bugs and erroneous data due to client/server asynchronisation.

7.3 Bugs: 2D Roguelike

Zombie Blocking Goal Quadrant - Status: fixed In the original game a zombie was able to move to the exit and block it, so that the player was not able to proceed to the next level and would lose the game in the current level.
This was fixed by extension to the zombie unit AI. In any case the exit is only be accessible for the player unit.

Player and Zombie stuck in one field - Status: not fixed Units can move to the same field at the same time and after that do not continue to move. The player can therefore not continue the game and will die on that position.
This enforces a simulation with a high decision delay for the player AI and a low game speed or in case of manual balancing a slowly acting player.

Zombies stuck in one field - Status: not fixed Units can move to the same field at the same time and after that do not continue to move. The zombies can therefore not further chase the player, which lowers the difficulty of that level by accident.

This enforces a simulation with a high decision delay for the player AI and a low game speed or in case of manual balancing a slowly acting player.

8 Manual Balancing

In the following section the manual balancing process, created results and gained insights are described and explained.

8.1 Zombie Village Game

The ZVG was the first game used for manual balancing in the same setup as for the automated approach. Manual balancing was conducted for the scene `SceneToBalance.unity`.

8.1.1 Preliminaries

[Sch14] was read in order to get an idea of how a manual balancing process may look like. Furthermore, insights gained during the Blue Byte presentation at the WWU Münster were taken into account.

Gained useful information:

- Figure out the audience's desire
- A lot of playtesting is necessary
- Adjust elements of the game until they deliver the preferred experience
- Understand relationships between elements of the game
- Understand which elements need to be balanced
- There are twelve common types of game balance (as described in [Sch14])
- Some game balancing methodologies are introduced
  - Doubling and halving
  - Train your intuition by guessing exactly
  - Document your model
  - Tune your model as you tune your game
  - Plan to balance
  - Let the players do it

– The Lens of the Problem Statement

For the ZVG, the player should face a challenge (the type/goal of game balance). It had to be determined which parameters are supposed to be balanced and which ones are supposed to have a fixed value. Regarding balancing methodologies, a combination of the ones known so far was used: doubling and halving as very useful to find boundaries of values for parameters. Within those boundaries guessing played a big role. Document as much as possible in order to keep track of the process and progress, make adjustments wherever they are needed.

8.1.2 The Balancing Process

A number of assumptions were made before working on a preliminary manual balancing process:

1. A specific scene featuring player units, zombie units, and resources is used
2. The player (NOT player units) loses resources over time; has to gather resources to stay in the game
3. Certain parameters are fixed, e.g. player unit health (100), zombie unit health (100), zombie unit attack distance (1), and starting amount of resources (20)

Keeping these assumptions in mind, a preliminary manual balancing process was developed. In the following, all steps and their explanations are listed:

1. Create an Excel sheet showing all relevant parameters
   a) Fixed: player unit amount, player unit health, zombie unit amount, zombie unit health, zombie unit attack distance, resource spot amount, resource spot junk amount, player junk at game start amount
   b) Variable: player unit attack power, player unit attack distance, zombie unit attack power, resource reduction time interval
2. Add default parameter values provided by Blue Byte GmbH to Excel sheet: player unit attack power (9), player unit attack distance (5), zombie unit attack power (3)
3. Add parameter value for resource reduction time interval: intuitively (2) was chosen

4. Play game using these parameters
5. Analyze logs created during play through:
    a) Check if humans survived
        i. If >0, player won
        ii. If 0, player lost
    b) Check if zombies survived
        i. If >0, player lost
        ii. If 0, player won
    c) Check time played
    d) check resources remaining
6. Repeat steps 4-5 until five solutions without major errors/bugs occurring are generated
7. Adjust parameters based on observations
    a) One parameter at a time
    b) Use doubling/halving in order to find boundaries
8. Document parameter changes in Excel sheet
9. Repeat steps 3-6 until doubling/halving does not yield useful result anymore
10. Use "brute force" approach in order to find appropriate parameter values between the previously identified boundaries
11. Repeat steps 4-6 and 8 and continue to use "brute force" approach
12. Successfully identify a "good solution"

8.1.3 Parameter Setting

In order to determine which parameters should be fixed and which parameters should be variable (the game elements to be balanced/adjusted), a group discussion was conducted. The discussion yielded the following classification:

Fixed	Variable
Player Unit Amount	Player Unit Attack Power
Player Unit Health	Player Unit Attack Distance
Zombie Unit Amount	Zombie Unit Attack Power
Zombie Unit Health	Resource
Zombie Unit Attack Distance	Reduc-
Resource Spot Amount	tion
Resource Spot Junk Amount	Time
Player Junk Amount at Game Start	Interval

Table 24: Overview of fixed and variable parameters for the ZVG.

8.1.4 Documentation of the Manual Balancing Process

For documentation and supporting the process an excel sheet was created with the following columns:

- Date
- Run ID
- Player Unit Health
- Player Unit Attack Power
- Player Unit Attack Distance
- Zombie Unit Health
- Zombie Unit Attack Power
- Resource Reduction Time Interval
- Player Units Alive (after the current playtest ended either by winning or losing)
- Zombie Units Alive (see above)
- Resources Remaining (see above)
- Win (Boolean)
- Time Elapsed (after the current playtest ended either by winning or losing)
- Comment

First Iteration: The player followed the preliminary balancing process and documented each playtest using the Excel sheet. The parameters were set to the default values for the first two playtests in order to get a feeling for the game and get used to the controls. Then, a single parameter got adjusted using the doubling and halving method. After identifying boundaries, the player started guessing values for the parameter that is currently being adjusted within the identified boundaries.

This first iteration of manual balancing didn't yield any useful results. The two main reasons identified are:

1. Bugs that could not be resolved yet had a negative impact on the playtesting. The communication between the server and the client did not work flawlessly, which resulted in a number of playtest that couldn't be interpreted in a meaningful way

2. The amount of "Player Units Alive" can't be interpreted in a meaningful way even for playtests not ruined by bugs, since the amount of health left for each player unit alive was not tracked. For example, all three player units being alive after a playtest ended could mean that the game did not provide a challenge. However, if all three units barely survived (average health remaining of 20) the game did indeed provide a challenge.

Second Iteration: Before starting the second and final iteration of manual balancing, the bugs mentioned above had to be fixed. Furthermore, the columns featured in the Excel sheet were adjusted. The column "Run ID" was renamed to "Solution" in order not to confuse it with "Run ID" of the (automated) Balancing Environment, where a run features multiple solutions. The most important adjustment was the addition of the column "Sum Health Left". After a playtest ended either by winning or losing, the sum of the health left of all three player units (dead or alive) is displayed and can now be documented in the Excel sheet. This allowed an interpretation of how challenging the game actually is – similar to the Balancing Environment. Lastly, in order to make it easier to distinguish playtests with and without issues like minor bugs, a new column "Issue" which is filled with "YES" and "NO" depending on whether the playtest was flawless or not, was added. Playtests where issues occurred could not be interpreted properly and were disregarded. For the second and final iteration of manual balancing, the preliminary balancing process explained earlier was used, too. It has to be noted that for each parameter combination, solutions

were created until a total of five solutions without any issues was reached. The scene `SceneToBalance.unity` was used and the following playstyle was adapted by the player:
A defensive playstyle where the layer focuses on the resource spots with zero to one zombies next to it first, next going for the group of two zombies and the resource spot next to them and finally, after gathering a resource buffer, attacking the last group of zombie consisting of three zombie units.
A total of 142 solutions were created by the player over several days. In the end, a satisfying solution was found (see table 25) :

PU ATK Power	PU ATK Distance	ZU ATK Power	Reduction Time
6	4	8	2

Table 25: Good Solution Parameter Combination Manual Balancing ZVG.

This solution featured an average "Sum Health Left" of 57 over a span of 16 flawless playtests (all featuring the parameter combination of the good solution). This was interpreted as a fitness value of seven with regards to the Balancing Environment. The solution provided a challenge for the player but at the same time allowed a considerate player to win frequently. 13 out of the 16 flawless playtests resulted in a win. Three out of the 16 flawless playtests resulted in a loss. Two out of the three losses were a result of running out of resources (the player lost because of starvation). One out of the three losses was a result of all the player units dying, leaving the player unable to reach the goal of killing all zombies.

8.1.5 Playtesting solution identified by Balancing Environment

The player conducting the manual balancing was then provided with the following three good solutions identified by the Balancing Environment (see table 26):

PU ATK Power	PU ATK Distance	ZU ATK Power	Reduction Time
17	1	10	2
14	1	7	2
13	2	10	2

Table 26: Good Solution Parameter Combination Manual Balancing ZVG.

The first two good solutions were challenging and enjoyable (average "Sum Health Left" of 55 and 72.3 over the span of ten playtests each). The pacing was faster using the first solution compared to the second solution which is related to the higher values of Player Unit Attack Power and Zombie Unit Attack Power. The third solution did not provide a challenge at all (average "Sum Health Left" of 135.7 over a span of seven playtests).

8.1.6 Result and Insights

- A satisfying solution was found by the manual balancing process, which was considered challenging yet enjoyable.
- Good solutions found automatically by the Balancing Environment can be challenging and enjoyable, too. Differences in playstyle between a human and the AI may render some good solutions found useless, though.
- During manual balancing, it became obvious that *Player Unit Attack Distance* can be exploited by the player and therefore has to be adjusted with caution, generally speaking it should not be set to a high value.
  - The third solution identified by the BE the player was provided with seemed to confirm the following statement: *Player Unit Attack Distance* of two in the hands of the player already tilts the fights in favor of the player (units). The AI tends to place player units very close to the zombie units, not (ab-)using the *Player Unit Attack Distance*.
- It seemed that out of all the possibly existing satisfying solutions the manual and automatic balancing looked and found very different ones.

8.2 2D Roguelike Game

The 2DR game and Unity tutorial was the second game used by us for manual balancing in the same setup as for the automated approach. Manual balancing was conducted to the main scene without leaving out game elements to reduce complexity as in the ZVG.

8.2.1 Preliminaries

As for the manual balancing approach for the ZVG the manual balancing process was oriented on the impressions from *The Art of Game Design – A Book of Lenses*

by Jesse Schell (see [Sch14] and the insights gained during the presentation by Blue Byte GmbH. Basically the same information as for the ZVG were taken into account (see 8.1.1 Preliminaries). For the 2DR game, the goal was to provide the player with a reasonable challenge. A player should strive to reach a higher level with each playthrough. It had to be determined which parameters were to balance and which ones were fixed values in the game setup. The same balancing methodologies as for the ZVG were used, as doubling and halving or simple guessing.

8.2.2 The Balancing Process

As for the manual balancing process for the ZVG a number of assumptions were made before working on the manual balancing process:

1. Cases where bugs occur with a major influence on the outcome of the playthrough are to disregard.
2. The fitness function is set up in a way that a new player is able to achieve level 15. Any playthrough not ending in level 15 is punished in a way that the punishment is harder the greater the distance between the actual level the player died from level 15. In this way a playthrough in which a player reaches level 13 or 14 can still be considered as "good", but not as "good" as if the player reached level 15.
3. The player starts with 100 food points. The fitness function as well as the ranges for the parameters are in line with this. Using this way we avoid e.g. food tiles that give the player a ridiculous amount of food or enemies that deal a too high damage to the player.

These assumptions in mind, the manual balancing process from the ZVG was adapted. The adapted steps are:

1. Create an Excel sheet showing all relevant parameters
2. Add default parameter values as an initial set of parameters
3. Play the game using the parameters
4. Analyze the playthrough
    a) Check the number of survived days

i. If a player survived around 15 days, mark the playthrough as good.

ii. If a player survived more than 18 days or less than 13, mark the playthrough as not good.

b) Check the fitness

i. If fitness is very high, consider that levels in playthrough are unevenly difficult.

ii. If fitness is very low, consider difficulty as conform to the wanted increasing difficulty over the levels.

5. Repeat steps three and four until three solutions without major issues are generated

6. Adjust the parameters based on observations

a) One parameter at a time

b) Use doubling and halving in order to find boundaries

7. Document the parameter changes in the Excel sheet

8. Repeat steps four and five until doubling and halving does not yield useful results anymore

9. Use “brute force approach” in order to find appropriate parameter values between the previously identified boundaries

10. Repeat steps three to five and seven and continue to use “brute force approach"

11. Successfully identify a “good" or satisfying solution

8.2.3 Parameter Setting

In order to determine which parameters should be fixed and which parameters should be variable and part of the balancing process, a group discussion was conducted. The discussion yielded the following classification:

Fixed	Variable
Start Food Point	Player Damage to Wall
Location of Start and Exit	Value of Food Tiles
Hit Points of Destructible Walls	Enemy Damage to Player
Map Size	Number of Food Tiles
Number of Enemies	Number of Destructible Walls
	Food Loss per Action

Table 27: Overview of fixed and variable parameters for 2DR.

8.2.4 Documentation of the Manual Balancing Process

For documentation and supporting the process an excel sheet was created with the following columns:

- Date
- Solution ID
- Starting food
- Map size
- Wall hit points
- Enemy unit number
- Player unit attack power
- Food loss per action (for player unit)
- Zombie unit attack power
- Food tiles amount
- Walls amount
- Days survived (indicates the reached level)
- Fitness (value of the fitness function)
- Issue (indicates if issue occurred during playthrough)

- Comment

All three chosen testers followed the preliminary balancing process and documented each playtest using the Excel sheet. The parameters were set to the default values for the first two playthroughs in order to get a feeling for the game and get used to the controls. Next, a single parameter got adjusted using the doubling and halving method. After identifying boundaries, the player started guessing values for the parameter that is currently being adjusted within the identified boundaries.

8.2.5 Insights

Three players generated each one around 100 playthroughs and approximately 30 solutions. The players were able to identify different trends. One player discovered that adjusting walls while keeping everything else at default values can result in satisfying solutions, while another player identified a correlation between zombie attack power and food item value. Zombie attack power should approximately stay at double the food item value, so that the average taken damage by a zombie is compensated by the average collected food. Furthermore, all players agreed that a large amount of food items with a low value each is preferred over a small amount of food items with a high value each, because this provides the player with more flexibility when choosing a path or lowers the disadvantage of discarding hard to reach food items. In the end, only the *"Wall and Default"* approach came close to satisfying solution, although no specific wall amount can be given, rather a range was identified. A wall amount between 19 and 22 was identified as satisfying.

Even though, a clear result and identification of "good" solutions were difficult to define. Identified reasons are:

1. Each level can be different, due to the fact that all game elements are positioned randomly on the map. This resulted in a high variation in the solutions of one set of parameters.

2. The three generated solutions for each parameter set may not be sufficient enough, as the high variation of the solutions indicates. A higher amount of solutions, preferably ten or more, would have been a better choice in regards to the variation, but also too time consuming at that state of the project.

9 Simulation

9.1 Data Logging

The parameter pairings which are created with the BE need to be logged in order to use them for any kind of data analysis. For this purpose CSV files were created. In order to log data for both games as well as a session and population file two, two classes were created. One for the ZVG and one for the 2DR game, both are called `WriteToCSV.cs`. Listing 15 and Listing 16 show both class constructors for the session and population file. The logic for both files is almost completely identical, which is the reason why the shorter version is chosen for the given examples here. Most of the time this will be the 2DR version. The fully commented code can be found in the Git repository, as well as the attached DVD.

The session file is used to log every single solution of a run. Therefore we can pass two things: First of all the file name and secondly the first set of features. The first set of parameters are our default values for the parameters. We extract the names of the features in order to write the first (title) row later into the CSV. After that we call the function which creates the session file (cf. Listing 15).

Listing 15: Class Constructor for the Session File (ZVG/2DR).

```
1 public WriteToCSV(string sessionFileName,Feature[] features) {
2   this.sessionFileName = sessionFileName;
3   foreach (Feature feature in features) {
4     rowTitles.Add(feature.GetName()); }
5   CreateSessionFile(); }
```

The population file only logs the solutions which are in the current population. Passing the first set of features is not necessary in this case, since the first initial solution is not relevant for the population. We only pass the file name for the population file and call the method which creates the population file (cf. Listing 16).

Listing 16: Class Constructor for the Population File (ZVG/2DR).

```
1 public WriteToCSV(string populationFileName) {
2   this.populationFileName = populationFileName;
3   CreatePopulationFile(); }
```

The creation of the session file (Listing 17) and population file (Listing 18) are almost identical. They only differ in two minor points. The first one is that they are saved

in different directories depending on the game (*ZVG* and *2DR*) and the type of document (session and population). The second difference is that the session file gets an extension to its name. The current date and time are appended to the file name. This was mainly done for testing purposes, so that it was not necessary to delete old files. Otherwise they work the same, which means they use the main directory of the repository and create files in their dedicated *CSVLogs* directories. Additionally, a `FileStream` and a `StreamWriter` are used to open up a connection to the files and to write into the files. As long as the simulation is running a connection is opened in order to instantly write new lines.

Listing 17: Create Session File (ZVG/2DR).

```
1 public void CreateSessionFile() {
2   DirectoryInfo workingDirectory = new DirectoryInfo(Environment.
        CurrentDirectory);
3   string savePath = workingDirectory.Parent.FullName + "/CSVLogs/
        ZVG/Session";
4   string dateTime = DateTime.Now.ToString("yyyy-MM-dd HH-mm-ss");
5   sessionFilePath = string.Format("{0}/{1}_{2}.csv", savePath,
        sessionFileName, dateTime);
6   session = new FileStream(sessionFilePath, FileMode.Append,
        FileAccess.Write, FileShare.ReadWrite);
7   writerSession = new StreamWriter(session); }
```

Listing 18: Create Population File (ZVG/2DR).

```
1 public void CreatePopulationFile() {
2     DirectoryInfo workingDirectory = new DirectoryInfo(
          Environment.CurrentDirectory);
3     string savePath = workingDirectory.Parent.FullName + "/
          CSVLogs/2DRoguelike/Population";
4     populationFilePath = string.Format("{0}/{1}.csv", savePath,
          populationFileName);
5     population = new FileStream(populationFilePath, FileMode.
          Append, FileAccess.Write, FileShare.ReadWrite);
6     writerPopulation = new StreamWriter(population); }
```

Now that the logging files are created the next step is to write content into them. The first line is always the title row, which contains all the different features, parameters, and other important values that need to be logged. An overview of the contents can be seen in Table 28. The title row contents were already collected for the session file

Both	**Content**
Time	Current date and time at the moment of logging
Generation	Current generation of the EA
SolutionID	ID of the current solution
ZVG	**Content**
Time To Consume	The time interval between food is consumed
PU Attack Power	Attack power of player units
PU Attack Distance	Attack distance of player units
ZU Health	Health value of zombie units
ZU Attack Power	Attack power of zombie units
Fitness 1	First fitness value depending on chosen fitness function
Fitness 2	Second fitness value depending on chosen fitness function
Fitness 3	Third fitness value depending on chosen fitness function
Is Goal Achieved	Boolean value which states if the goal is achieved
Player Alive	Number of alive player units
Zombies Alive	Number of alive zombie units
Junk Gathered	Amount of junk gathered in the game
Food	Amount of overall food in the game
2DR	**Content**
Enemy Attack Power	Attack power of enemy units
Food Value	Amount of health gained from food tiles
Player Wall Damage	Amount of damage dealt to walls
Number of Walls	Amount of wall tiles on the map
Number of Food	Amount of food tiles on the map
Food Loss per Step	Health lost per action
Fitness	Fitness value at the end of the game
Days survived	Number of days played

Table 28: Title Rows of ZVG and 2DR CSV Logs.

from the default parameters (cf. Listing 15). Therefore, the method checks if the file is empty, since the title row should only be written once into an empty file. If an EA is being logged the generation column will be added, otherwise the title row will be written without the generation column (cf. Listing 19).

Listing 19: Appending Titles to the Session File (2DR).

```
1 public void AppendTitles() {
2     if (new FileInfo(sessionFilePath).Length == 0) {
3       if (Algorithm.instance.DoWeNeedGeneration()) {
4         StringBuilder titleText = new StringBuilder("Time,
              Generation,SolutionID,");
```

```
5           titleText.Append(String.Join(",", rowTitles.ToArray()) +
                ",");
6           titleText.Append("Fitness,Days survived");
7           writerSession.WriteLine(titleText.ToString());
8           writerSession.Flush(); }
9       else {
10      StringBuilder titleText = new StringBuilder("Time,
            SolutionID,");
11      titleText.Append(String.Join(",", rowTitles.ToArray()) + ",
            ");
12      titleText.Append("Fitness,Days survived");
13      writerSession.WriteLine(titleText.ToString());
14      writerSession.Flush(); } } }
```

For the population file the titles need to be extracted from the passed population array, since the first default solution cannot be utilized. Otherwise the logic of both title appending functions is the same. All necessary feature names are being written into the first row of the corresponding CSV files (cf. Listing 20).

Listing 20: Appending Titles to the Population File (2DR).

```
1 public void AppendPopulationTitles(Solution[] population)  {
2     foreach (Feature feature in population[0].GetFeatures()) {
3         rowTitles.Add(feature.GetName()); }
4     if (new FileInfo(populationFilePath).Length == 0) {
5       StringBuilder titleText = new StringBuilder("Time,
            Generation,SolutionID,");
6       titleText.Append(String.Join(",", rowTitles.ToArray()) + ",
            ");
7       titleText.Append("Fitness,Days survived");
8       writerPopulation.WriteLine(titleText.ToString());
9       writerPopulation.Flush(); } }
```

Logging a solution for the session file or the population file is quite similar (cf. Listing 21 and 22). The only major difference is the fact that the population file always has a generation column. This means a check if the column is needed is not necessary. Otherwise the writing of a solution is quite straight forward. The needed content (cf. Table 28) will be included into a string in which every word is separated by a comma. After building the entire string it will be written into a new line. This will be done for every single solution. In the case of the population file we do this for every solution in the passed population. For the population file the number of written solutions depends on the population size.

Listing 21: Appending a Solution to the Session File (2DR).

```
1 public void AppendSolution(Solution solution) {
2   AppendTitles();
3   string generation;
4   string dateTime = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
5     if (Algorithm.instance.DoWeNeedGeneration()) {
6       generation = Algorithm.instance.GetCurrentGeneration().
            ToString();
7       StringBuilder solutionText = new StringBuilder(dateTime + "
            ," + generation + "," + solution.ToString());
8       writerSession.WriteLine(solutionText.ToString());
9       writerSession.Flush(); }
10    else {
11      StringBuilder solutionText = new StringBuilder(dateTime + "
            ," + solution.ToString());
12      writerSession.WriteLine(solutionText.ToString());
13      writerSession.Flush(); } }
```

Listing 22: Appending a Solution to the Population File (2DR).

```
1 public void AppendPopulation(Solution[] population) {
2   string generation;
3   string dateTime = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
4   foreach (Solution solution in population) {
5     generation = Algorithm.instance.GetCurrentGeneration().
          ToString();
6     StringBuilder solutionText = new StringBuilder(dateTime + ","
           + generation + "," + solution.ToString());
7   writerPopulation.WriteLine(solutionText.ToString());
8   writerPopulation.Flush(); } }
```

After the algorithm reached its termination criterion or criteria the opened `FileStream` and `StreamWriter` need to be closed, so that no problems arise regarding the file access or file locking. This is done by two very simple methods. Once again one for each file (cf. Listing 23 and 24). The `Close()` methods of the `FileStream` and `StreamWriter` classes are called to close the current streams and to release the file and any resources associated with the files.

Listing 23: Closing Streams to the Session File (ZVG/2DR).

```
1 public void Stop() {
2   writerSession.Close();
```

```
3   session.Close(); }
```

Listing 24: Closing Streams to the Population File (ZVG/2DR)).

```
1 public void PopulationStop() {
2     writerPopulation.Close();
3     population.Close(); }
```

9.2 Simulation Description

Simulation refers to the the actual run-times of the BE in which the BE is repeatedly set-up manually on different games with different scenes and goals, player AIs, fitness functions, game parameters and with different optimization algorithms and their respective configurations. The simulations (run-times) were collaboratively documented within a shared online spreadsheet containing the information of the entire configuration set, as introduced in chapter 5 for run-time 289. The simulation consisted of three phases.

Phase 1 was dominated by constructing a reliable version of the BE for the ZVG. For several weeks, the reliability could not be confirmed due to the above-outlined showstoppers. It was unclear whether the general workings of the BE were fully functional. Moreover, in situations of error occurrences, it could not be determined whether the errors should have been attributed to the inherited game bugs or the general working of the BE. Eventually, this was overcome, however it continued to be necessary to run the game in real-time game-speed.

In Phase 2, after resolving the major bugs, the BE was tested several times with the EA to see if it provided valid and reliable results. This could be confirmed by reading out the CSV logs as well as testing the results manually (see above). A certain setup of the BE for the ZVG turned out to work most reliable.

BE Component	Setting
Game	Zombie village game
Scene	Scene to balance
Goal	Survival goal
Player AI	Aggressive and resource player behavior
Optimization algorithm	Evolutionary algorithm
Fitness function	Weighted fitness function

Table 29: Standard BE setup for the ZVG.

AI Parameter	Setting
AI active	YES
AI decision delay interval	2
Indifferent distance	10
AI health limit	0.05
AI junk limit	10

Table 30: Standard setup of the Aggressive and resource player behavior.

SG Parameter	Setting
Health limit	0.05
Number of alive player unit	1
Target amount of health	50
Health weight	1
Target amount of food	5
Food weight	6
Starting food	20

Table 31: Standard setup of the Survival goal (also for run-time 289).

Phase 3 was dedicated to finding the best algorithm configuration and consumed most of the simulation resources due to its inherit necessity of comparing a larger number of run-times with different setups. In total this accounted for 353 run-times

out of 379 total simulations. When simulating the ZVG the standard setup was applied at all times. According to the F-RACE parameterization each of the following 20 algorithm configuration that were generated randomly from the permissible set of ranges (see table 22) have been tested 10 times with different random number seeds: 928300, 720732, 574453, 920235, 846611, 398564, 276979, 519523, 455329, 316480.

No.	**EA variant**	μ	λ	p_m	σ
1	EA3	7	20	-	3.57
2	EA1	6	19	-	2.89
3	EA5	9	13	-	0.85
4	EA3	10	12	-	0.91
5	EA4	10	15	0.28	0.56
6	EA1	10	11	-	1.92
7	EA2	9	20	-	1.92
8	EA5	6	18	-	3.87
9	EA5	10	14	-	0.57
10	EA4	9	17	0.19	3.82
11	EA3	9	18	-	1.78
12	EA2	8	14	-	0.24
13	EA3	6	12	-	1.11
14	EA2	9	14	-	0.13
15	EA3	6	14	-	0.06
16	EA3	10	20	-	1.95
17	EA3	6	14	-	2.39
18	EA5	8	17	-	2.4
19	EA2	6	15	-	1.6
20	EA5	6	15	-	1.59

Table 32: Fixed algorithm configuration parameters.

10 Data Analysis

The previous sections amplified how manual and automatic balancing produce data and how the best algorithm for this can be chosen. Data Analysis is now applied on these results in order to efficiently find the best solutions.

The analysis was driven by practical questions that might be of interest for game designers, like:

- How long does the algorithm need to get good results?
- Which parameters are more important in a scene?
- What are good parameter combinations?
- For a given parameter combination, can we predict how good a game is?

For all of these questions data analytics techniques deliver answers fast and reliably. This section will now elaborate on the main outcomes of the analysis, namely:

- The Algorithm Performance Report
- The Top Solutions Report
- Balancing Mechanisms explained through Heatmaps
- Fitness Prediction

All analysis was conducted using the statistical R programming language.

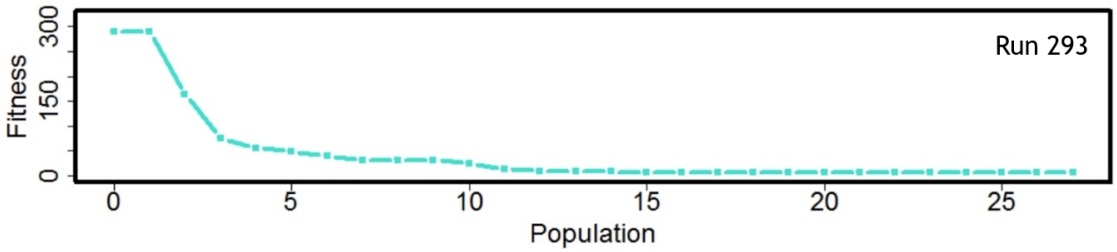
10.1 Algorithm Performance

The *Algorithm Performance Report* provides a comprehensive view on the fulfillment of general requirements during the simulation to be examined. Not only does it include information on time aspects but also overall development of fitness values per population and a list of location parameters per variable in the data set. In addition, it provides a table with information on the settings of the algorithm configuration and general setup of the game. This information is taken from the production schedule and is displayed to ensure all settings have been made as desired.

Figure 37 gives an illustration of what a performance report should look like. In order to know when and how long the simulation was performed, the top left corner of the report delivers the runtime coupled with the start and end time stamp. In the example of figure 37 the algorithm took about four hours. To see if the algorithm converges satisfyingly to a minimum, the graph shows how the mean of the populations evolved over time. For the graph of run 293, the performance graph

Performance Report

Runtime (hours): 4.28
Started: 2016-03-04 19:40:50
Ended: 2016-03-04 23:57:33



Solution	Time2Consume	PU AP	PU AD
Min. : 1.00	Min. :1.000	Min. : 4.000	Min. : 1.00
1st Qu.: 37.00	1st Qu.:1.000	1st Qu.: 7.000	1st Qu.:18.00
Median : 75.00	Median :1.000	Median : 7.000	Median :20.00
Mean : 98.53	Mean :1.269	Mean : 8.084	Mean :18.48
3rd Qu.:170.00	3rd Qu.:1.000	3rd Qu.: 8.000	3rd Qu.:20.00
Max. :230.00	Max. :5.000	Max. :19.000	Max. :20.00

ZU health	ZU AP	Fitness	Goal?
Min. : 90.00	Min. : 2.000	Min. : 6.0	False: 5
1st Qu.: 98.00	1st Qu.: 4.000	1st Qu.: 6.0	True :162
Median :100.00	Median : 4.000	Median : 6.0	
Mean : 97.92	Mean : 5.048	Mean : 35.6	
3rd Qu.:100.00	3rd Qu.: 5.500	3rd Qu.: 47.0	
Max. :100.00	Max. :10.000	Max. :311.0	

P. Alive	Z. Alive	Junk	Food
Min. :0.00	Min. :0.000	Min. : 2.00	Min. : 0.00
1st Qu.:1.50	1st Qu.:0.000	1st Qu.:10.00	1st Qu.: 5.50
Median :2.00	Median :0.000	Median :20.00	Median :10.00
Mean :2.21	Mean :1.042	Mean :19.37	Mean :10.31
3rd Qu.:3.00	3rd Qu.:1.500	3rd Qu.:30.00	3rd Qu.:14.00
Max. :3.00	Max. :5.000	Max. :31.00	Max. :31.00

Algorithm configuration:

Game	Scene	Player AI	Food Limit	Goal	SD	Algorithm	PopSize	Offspring	Max. Gen	Ind#	Config.
ZVG	StB	AggRess	10	SG	1.6	EA2	6	15	27	400	19

Figure 37: Algorithm Performance Report Example.

demonstrates a steep slope at the beginning and convergence to very low fitness values in the long run. This can be interpreted as a fast and at long sight reliable achievement of good solutions. Hence, the algorithm performed well in this case. A summary of all variables given in the data set is listed below the performance graph. This can be exploited to comprehend for instance what value ranges of parameters the algorithm used, how many of the solutions were games the player won or lost, or what minimum fitness value was achieved. In the case of run 293 for example, the algorithm did not exploit the whole ranges of player unit attack power (PU AP) and zombie unit attack power (ZU AP) values. From the algorithm configuration table it can be seen that configuration 19 was used for this run. This was done intentionally since this was delivered as the best configuration so far from the F-RACE algorithm (see chapter 6.5).

To conclude this chapter, sample questions that could be answered using this report are:

- How long did it take to get results?
- Did the algorithm perform reasonably?
- What is the best fitness value achieved?
- What parameter value ranges were checked during the simulation?
- Which algorithm configuration settings were used?

10.2 Top Solutions

One of the core values of this work product lies in the provision of the *Top Solutions Report*. It delivers information on the top solutions the algorithm has found that can be directly applied to the game to be balanced.

The report is split in two parts: A list of the top 20 solutions found during the simulation and a scatterplot visualizing further information regarding them. If a designer for example needs five different but equally good levels, he/she can pick them from the list and then only needs to test if they fulfill the expectations. Furthermore, the scatterplot demonstrates how parameters are related for the top solutions. On the diagonal each of the game parameters and the fitness are shown. The upper right panel and lower left panel depict the respective correlation values or point combinations for each pair combination. The Spearman correlation is shown in the upper panel of the diagram. High absolute values are assigned a larger font size. A

positive correlation means that high values of one parameter lead to high values of the other whereas negative correlation means that high values of one parameter lead to low values of the other. The example of run 293 shown in figure 38 reveals that the PU AP and ZU AP of ZVG are correlated. Especially the zombie parameters have a strong influence on the fitness. The diagonal and the lower panel give a first idea about what good parameter values are. For example, the only value for the time to consume present in the best solutions of run 293 is a value of 1. By comparison, most of the values for the PU AP are at 7, but one could also pick 9 or 8.

Top 20 Solutions Report

Time2Consume	PU AP	PU AD	ZU health	ZU AP	Fitness
1	7	20	99	4	6
1	7	16	100	4	6
1	7	20	98	4	6
1	7	20	100	4	6
1	7	18	100	4	6
1	7	19	100	4	6
1	7	19	98	4	6
1	8	20	95	5	10
1	9	18	100	5	10
1	9	20	100	5	10
1	9	20	99	5	10
1	7	19	100	5	21
1	7	20	100	5	21
1	7	14	100	5	21
1	7	18	98	5	21
1	7	19	98	5	21
1	7	18	100	5	21
1	7	19	97	5	40
1	7	20	96	5	40
1	8	17	94	6	42

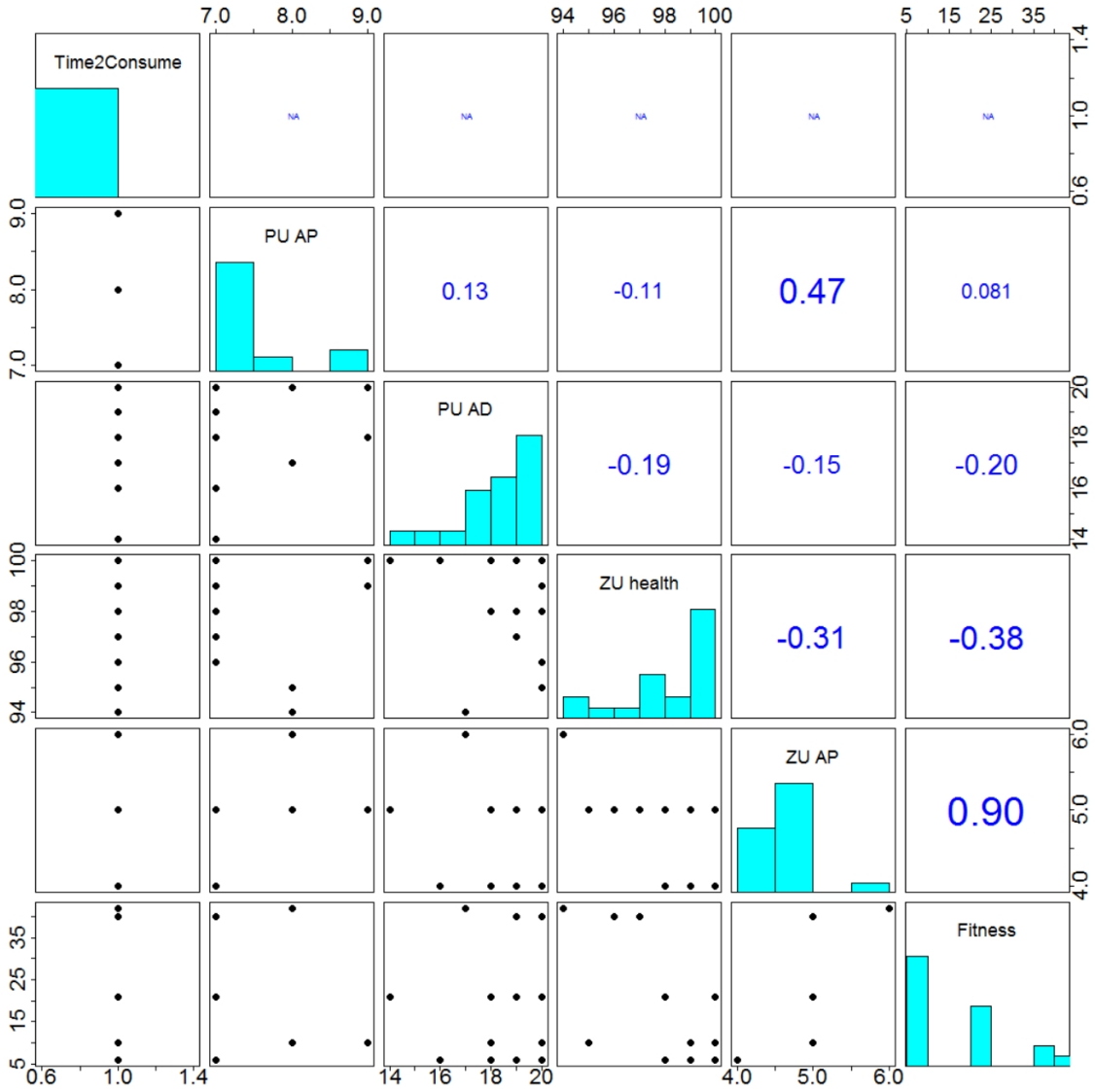


Figure 38: Top Solutions Report Example.

Use cases addressed by the Top Solutions Report would be:

- A certain amount of good, but different levels is needed.
- The influence of parameters on the fitness for good solutions should be examined.
- Parameter correlations need to be understood.
- Parameter values for the best solutions should be shown.
- Good parameter combinations are to be identified.

The described report already is a good starting point to understand how lever relocation of the game affects its result. Even so, a general knowledge on how the game parameters influence its balance cannot be acquired. Nonetheless, this would be of value for a broader understanding of the game and general balancing mechanisms. Thus, another tool was added, which will be delineated in the next section.

10.3 Balancing Mechanisms

To visualize the effect of changes to certain game parameters to the fitness of a game, heatmaps are a good tool to choose. The concept of heatmaps is well-established in statistics and was applied in several scientific areas already, as pointed out for example by [WF12]. As opposed to the widely used cluster heatmaps that display cluster trees on vertical and horizontal margins, the heatmaps generated for this PS follow a more simplistic logic: Each axis represents one parameter and its value range. The space between the axes is then separated into rectangular tiles. Each tile is shaded on a color scale from blue to red to represent the fitness value of the corresponding combination of parameter values.

A separate algorithm class (Two Parameters) in Unity generates a constant, manually defined solution and then simulates the game by iterating through the possible combinations of two parameters. These can be defined by hand prior to running the algorithm. The fitness results achieved are subsequently comparable. That said, it is recommendable to check for good solutions first before the heatmaps are applied. A good solution should be chosen as the constant solution for the algorithm. This can be picked using the Top Solutions Report for example that was explained in section 10.2 before. For the heatmaps presented here, the following constant values were chosen:

- time to consume: 4
- player unit attack power: 7
- power player unit attack distance: 6
- zombie unit health: 90
- zombie unit attack power: 3

A result of heatmaps created for ZVG is depicted in figure 39. Since the ZVG was created in a way s.t. the results of the fitness are deterministic, these heatmaps reliably inform the viewer about good parameter combinations. For the case of ZVG, those tiles were removed that correspond to a solution where the player AI lost the game. Consequently, a very attractive solution would be one with a good fitness value that is also close to one or several unavailable tiles. That is to say that for those combinations the likelihood of the game being challenging for the player is much higher which by logical inference makes the game more interesting and fun.

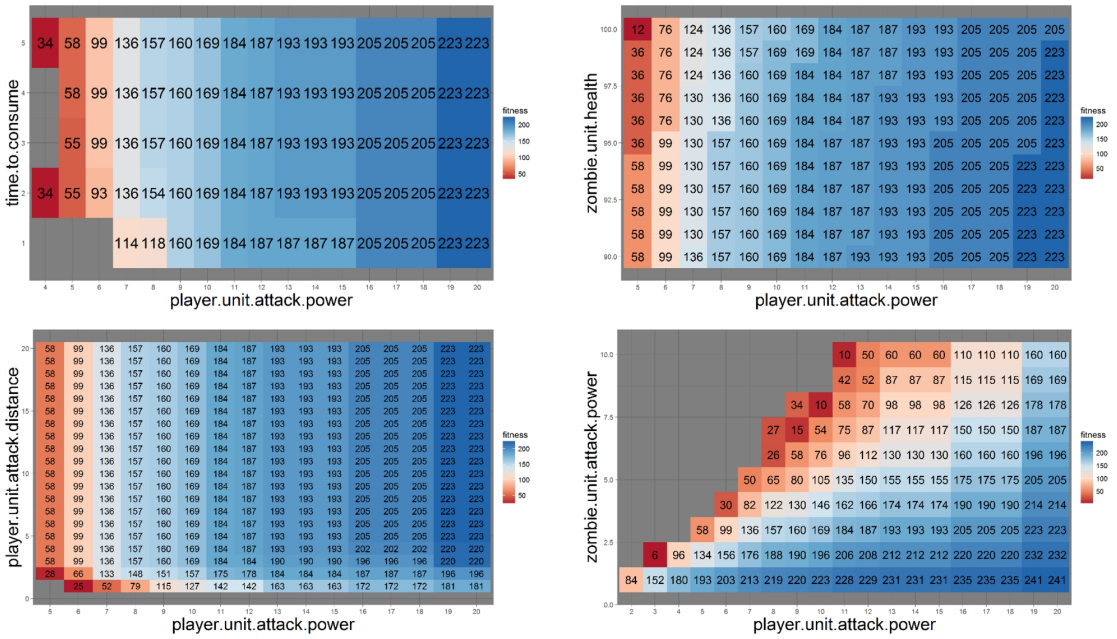
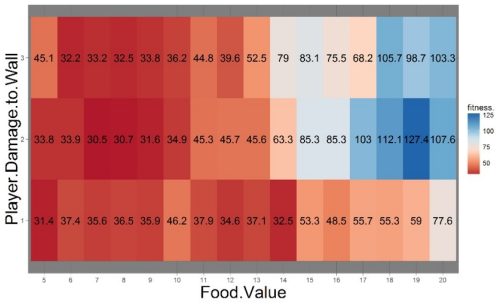


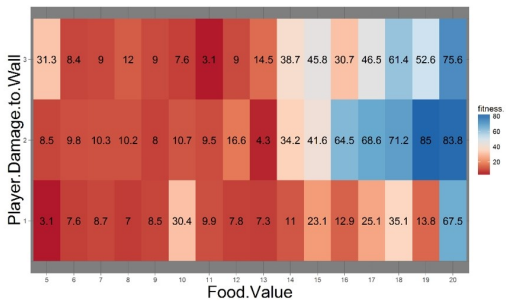
Figure 39: Sample Heatmaps for ZVG.

In case a game's fitness outcome is stochastic, the method applied for ZVG would not be sufficient. It is very likely that for many games the assumption of a deterministic fitness outcome does not hold. For instance, the 2DR game logic includes some randomness in the content generation (e.g. positioning of walls and food). Thus, heatmaps would need to account for the underlying uncertainty. To do so, several

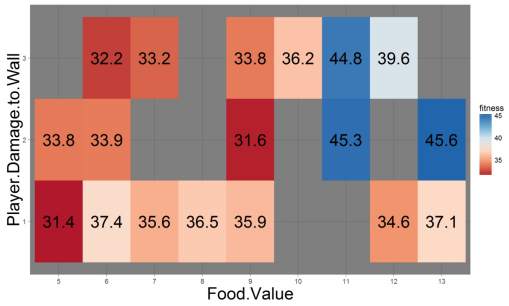
simulation runs for each parameter combination were created and the mean fitness over all results was calculated. The resulting map can however not inform about how far the single values are distributed from the mean. Therefore, an additional heatmap delivers the standard deviation values of each parameter combination. Knowing these, those tiles with the best mean fitness values and lowest standard deviation can now be chosen as solid good solutions by a designer. Sample heatmaps for means and standard deviation are shown in figure 40. To directly infer good solutions, the results from figure 40 a) and b) were merged into one in c), showing only those mean fitness values with a low standard deviation (smaller than ten).



a) 2DR: Mean fitness after five runs



b) 2DR: Fitness standard deviation after five runs



c) 2DR: Mean fitness with low SD after five runs

Figure 40: Heatmaps for non-deterministic fitness outcome in 2DR.

To wrap up the results of this section, it is now possible to analyze games with deterministic as well as non-deterministic outcomes with respect to parameter combinations and their influence on the game's fitness. The heatmaps represent variations of a previously chosen good solution that can be better understood using them. At the current stage, it is necessary to compare parameter combinations side-by-side to learn the best value range for each parameter as it was done in figure 39. A next

step of this could be to aggregate this information to a higher level. Depending on the desired difficulty level of a scene, the heatmaps also give an indication on the parameter combinations that could be chosen.

10.4 Prediction of Fitness and Game Outcome

Even though possibilities exist to shorten the time of simulations, the more complex a game gets, e.g. in terms of the number of game parameters, the more difficult and time consuming it becomes to create a reasonable amount of simulations. One idea to overcome the issue can be the replacement of the real simulation by the accurate enough predictive model. Indeed, since simulation is not a bottleneck anymore meta-optimization process can be done relatively fast, the problem even can be tackled in a brute force way. However, getting one optimal solution is not the central idea here, since any predictive model is not 100% accurate. Game designers can profit from the approach in three ways:

1. most of the near optimal solutions can be identified;
2. complex parameter interdependencies can be inferred;
3. feature importance can be estimated.

The first point can be valuable then interesting, but at the same time very diverse content is required. The last two points are required to better understand the game and predict how parameter changes can influence it.

Predictive analytics techniques can predict the fitness of a game by exploiting a small but large enough data set sample of the search space to be explored. The technique used in this PS is ANNs that were trained by a large ZVG sample (1.5K observations) and then tested in terms of accuracy of the prediction. The implementation of this was supported by the R-packages `neuralnet` (supporting ANN), `dplyr` (facilitate work with data sets), and `MASS` (support the boxplot transformation). The code can be found in file `RCode/Model.R` of the PS gitlab. Since the results of the ANNs models fitting and cross validation are non-deterministic, reproducibility is guaranteed using a seed value at the beginning. Then, data are prepared, such that if the goal is not achieved (player AI has lost the game) a penalty of 100 is added to the fitness value. Scaling usually is recommended, thus the data were transformed using min-max scaling. Two models are used to predict the outcome and fitness of

the simulation, both models utilize ANN with two inner layers (8 and 3 neurons).

Accuracy of the models is validated by 20-fold cross validation. For the 20-fold cross validation every observation is assigned to a fold, for the model to be fitted on all except one fold and tested on the unseen fold. After cross validation and splitting the data into training and testing set, two neural nets were fitted. Both were needed to check for two different outcomes: goal achievement and fitness value. To calculate the errors, the min-max transformation is reverted and the Mean Squared Error (MSE) for the fitness and misclassification rate for the goal prediction are calculated and visualized. The results can be seen in figure 41. Concluding from the boxplots, the prediction of the goal achievement is quite reliable with most of the values being roughly between one and three percent and a maximum value of five percent. The mean squared error is shown in the lower boxplot of Figure 41, the mean error of 21 seems to quite reliable taking into account the general range of fitness from 0 to 500. Whereas the model and code provided proof that the general concept of fitness prediction is very promising; for example, the model can be used to enhance optimization algorithm pruning weak solutions.

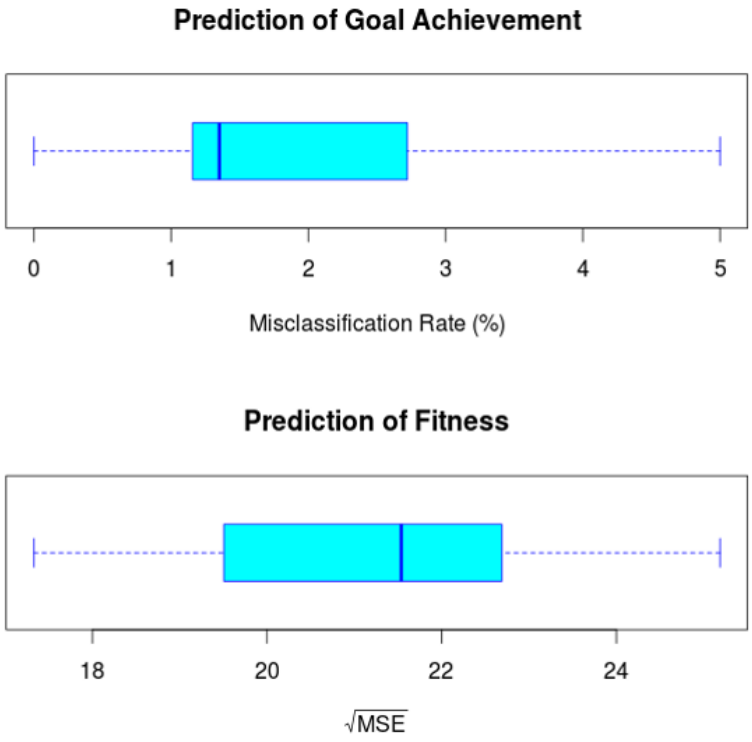


Figure 41: Prediction of the outcome of the ZVG simulation and fitness value. Errors based on 20-fold cross validation.

It is outlined already that this approach can be relatively useful for a game designer to create diverse content consisting of different good solutions. Thus, developed predictive models were queried by all the possible combinations of game parameters (220K). A solution was considered good enough if the predicted outcome is a win of a player and fitness is below 50. Figure 42 shows a sample of 10 out of 560 identified solutions. It can be seen from the figure that most of the solutions are the trade-offs between a pair of parameters, exemplary trade-offs are marked by red arrows at the figure. Some of the identified solutions were manually tested and the general impression was that the approach worked quite well, predictions were really close to the real values.

	time.to.consume	player.unit.attack.power	player.unit.attack.distance	zombie.unit.health	zombie.unit.attack.power
1	1	8	20	99	5
2	1	7	20	94	5
3	1	7	19	94	5
4	1	6	20	95	4
5	1	7	20	95	5
6	1	8	19	99	5
7	1	7	18	94	5
8	1	7	19	95	5
9	1	6	19	95	4
10	1	7	17	94	5

Figure 42: Good Solutions.

In order to understand underlying parameter interdependencies the identified 560 good solutions were scrutinized. Indeed, since trade-offs are clearly seen in the obtained sample, it would be reasonable to identify functional dependencies between some parameters. This analysis can help the game designer to better understand possible correlations and predict the outcomes of possible changes in the game. Figure 43 clearly shows the linear functional dependency between player and zombie attack powers. Interestingly, this figure perfectly matches with Figure 39 (bottom right plot) that is based on the actual simulations, in spite of the fact that all the solutions were found querying predictive models.

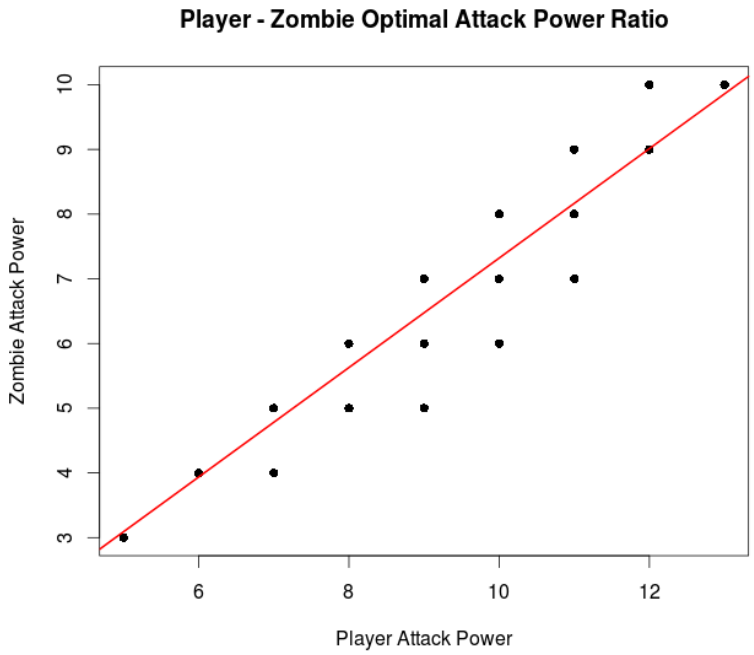


Figure 43: Optimal Attack Power ratio.

Another important application of the approach is feature relevance, indeed game

designer can either underestimate or overestimate the influence of the parameter on a game. Such misunderstanding can dramatically affect the game after a "small" update. Thus, in order to give an impression about parameter relevance predictive models can be used again. In order to estimate parameter relevance, each parameter was shuffled within the sample and then the increase of error rate was measured. This approach voids the affect of the parameter, but at the same time considers it as relevant. After evaluating the error increases they were scaled by the biggest increase. Thus, the parameter that gives the biggest increase of error rate has 100% of importance. Figure 44 shows the feature importance related to goal achievement and fitness. It can be seen from the figure that the results match with the logic and zombie and player power attack both dominate other features in terms of importance.

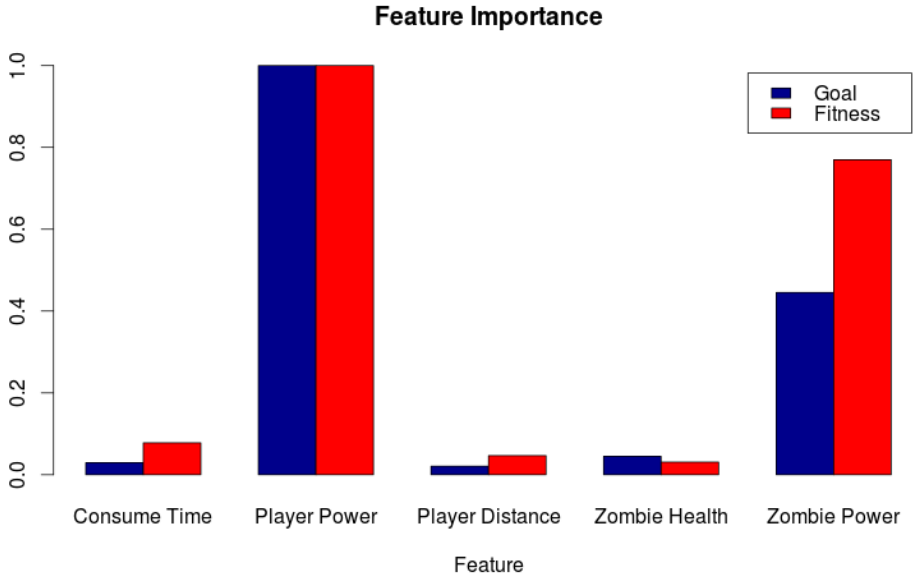


Figure 44: Feature Importance.

The predictive approach seems to be a rather promising idea, it both can be used for exploring good solutions and understanding game parameters. However, it has a crucial limitation - the bigger the number of parameters is, the bigger sample should be obtained to train the model. At some point it can become infeasible to obtain such a big sample. In the above example a sample of 1.5K observations was used and the model captured the underlying relationships quite well. However, considering that an optimization process normally requires 400 simulations one can say that such an approach is an unaffordable luxury in game design.

11 Balancing Process Model

11.1 Modeling Approach: BPMN

Our model aims to provide a shared understanding of the mechanisms in Game Balancing and connects the design and implementation layer. We modeled our concept processes in the Business Process Model and Notation (BPMN) language, an established standard in Business Process Management utilizing flowchart displays. BPMN features three main flow objects (*Events*, *Activities* and *Gateways*) which are linked by connector objects and grouped in *Swim Lanes*. Furthermore, artifacts (e. g. data objects/reports) can be attached to activities. Activities also can be specified forms (e. g. user/service).

11.2 Main Process: Balancing in Games

Our main process presents an overview of the flow of the five subprocesses and their associated data objects (cf. Figure 45).

11.3 Subprocess 1: Assess Context

To begin the process of game balancing, some preliminary steps are necessary. The first step in our main process is the process of assessing the context. As a pre-evaluation a game is selected first. The game is described in regards to its content, genre and mechanism in the *Game Description*. This description is used as a reference point to define the goals of the balancing process. The idea of how a game should be in the final state can be very different as can be seen e.g. the twelve described balancing goals by [Sch14]. The selection of goals for the balancing process has to be in line with the intended game as defined in the *Game Description* . The *Goal Description* includes a detailed description of the chosen goals. The Stakeholder Report includes information about the target group of the game as well as the parties involved in the game development process. Once the balancing goals are set, the pre-evaluation is completed and one can continue with the next subprocess *Set Environment*(cf. Figure 46).

Input

- No specific input.

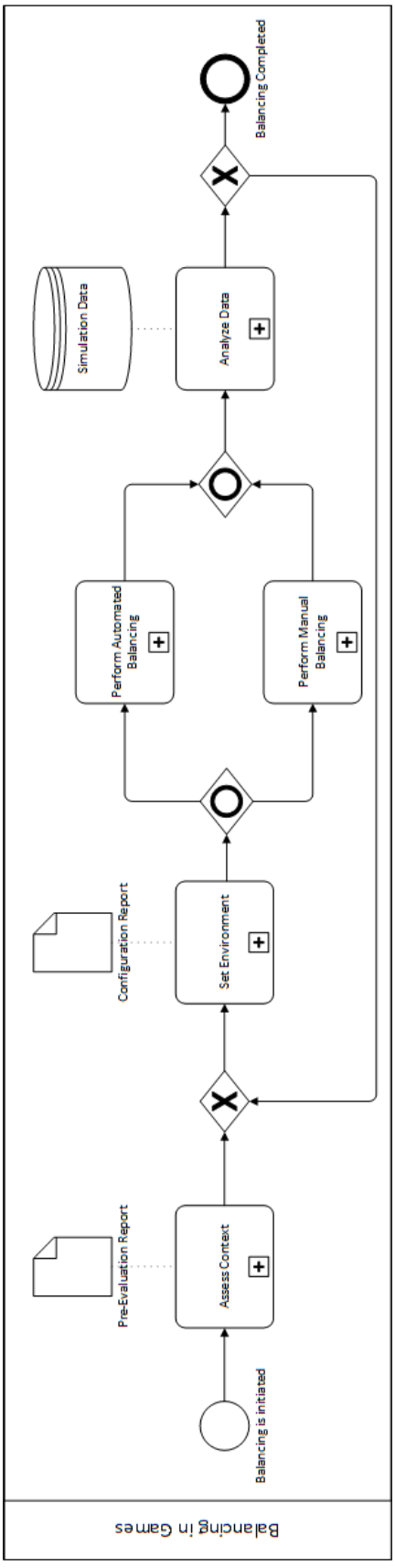


Figure 45: Balancing in Games Main Model.

Output

- Game Description
- Stakeholder Report
- Goals Description
- Short description of the Subprocess and Outcome
- Objectives
- Requirements

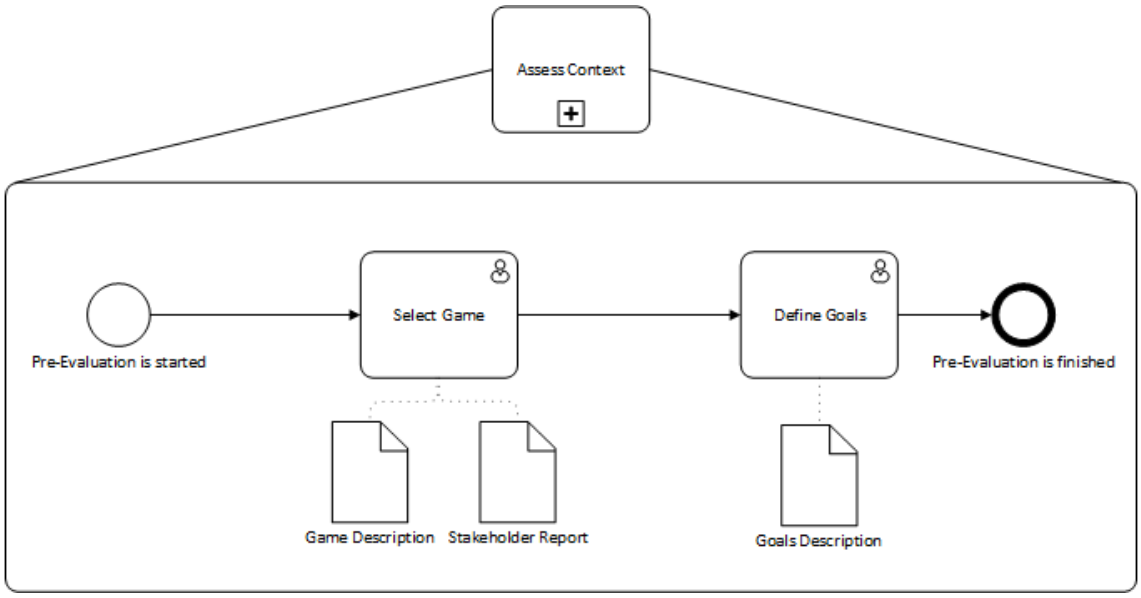


Figure 46: Subprocess: Assess Context.

11.4 Subprocess 2: Set Environment

Setting up the environment follows the logic of chapter 5 (cf. Figure 47).

- **Choose Set of Scenes**: At first, it should be chosen which scene or set of scenes want to be used for balancing. The selection of a scene can have large implications on the balancing outcome. If one chooses a scene which is a comprehensive representation of the full game version including all its potential complexity, then this will have dependencies on the choice and availability of the other components, like the player AI or winning condition (goal). At the

current state of research, it is recommendable to apply the BE to a dedicated selection of the final game version to balance a certain subset of the game.

- **Choose Parameters**: After the choice of the scene follows the parameter selection. This is the set of game parameters that are subject to change. They are kept as the only real decision variable within the BE for finding an optimum. It needs to clarified which parameters exactly should be optimized in terms of balancing. The amount of parameters generally has an impact on the time complexity of the environment as well as the value ranges. Hence, additionally the boundaries need to be defined, i.e. within which value ranges is the BE allowed to generate-and-test candidate solutions. Importantly, all of these parameters are should be different to the ones measured by the fitness function. For example, if the fitness function measures the distance to the remaining player health, the game parameter player health itself should ideally not be included to keep the search space as static as possible.

- **Choose Fitness Function**: The fitness function is the objective function towards which the game parameters are optimized. This should be an accurate representation of the intended goal of the game, e.g. challenge.

- **Choose Fitting Optimization Algorithm**: Depending on the hardness and type of the problem, either a deterministic or non-deterministic optimization method should be chosen. For a small set of parameters with small value ranges can be sufficient to apply a brute force or even random search. For a bigger problem, a greedy hill climber may suffice, while for cases with many more parameters, e.g. above 30 with discrete ranges between 50-100 for each, it becomes meaningful to apply a non-deterministic algorithm such as an EA.

- Based on the selection above, one can begin developing the BE, i.e. relating the necessary classes and methods with each other in a way described by the pseudocode in chapter 5.1.

- Lastly, the parameters of the optimization algorithm should be optimized with a proper methodology, e.g. F-RACE. This step is especially useful, if it is known that the current setup of the BE will be used many times.

Input

- Game Description

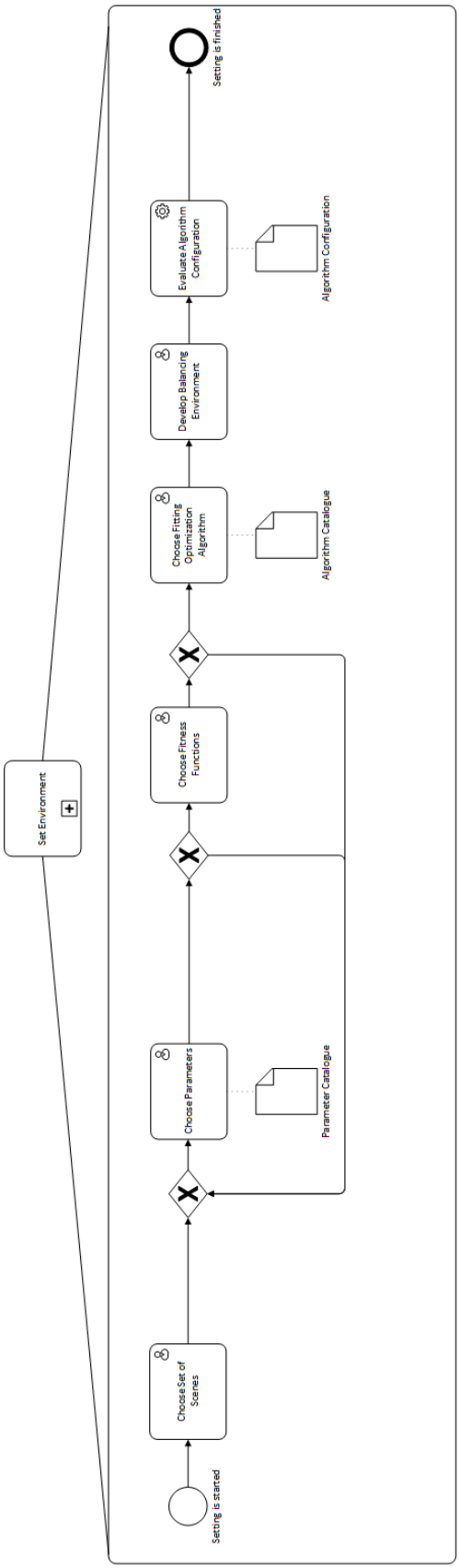


Figure 47: Subprocess: Set Environment.

- Balancing Goals

Output

- Parameter Catalogue
- Algorithm Catalogue
- Algorithm Configuration

11.5 Subprocess 3: Perform Automated Balancing

After setting the environment for the balancing process automated balancing can be performed. Since no human testers can be used for the automated approach, a AI has to be configured that is able to play the game as similar as possible to the way a human plays it. With the configured AI and the set *Algorithm Configuration* the game can be simulated. The simulated parameters and their corresponding results are documented in two separate files, the *Population Log* and the *Solution Log*. As a concluding step the found good solutions have to validated before the automated balancing process can be finished (cf. Figure 48).

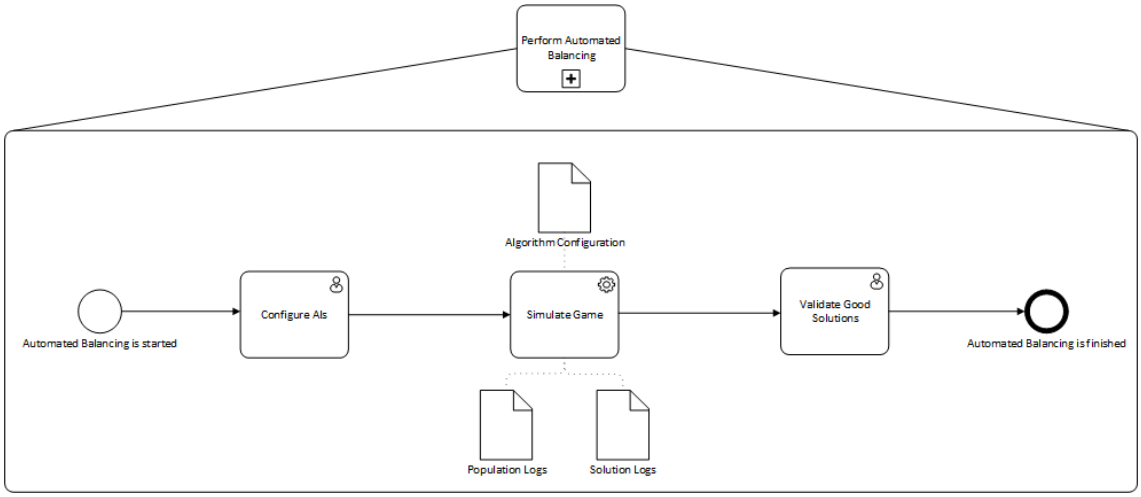


Figure 48: Subprocess: Perform Automated Balancing.

Input

- Algorithm Configuration

Output

- Population Logs
- Solution Logs

11.6 Subprocess 4: Perform Manual Balancing

As an additional part of the balancing process a manual balancing can be performed to on the one hand try to find solutions the automated balancing process does not find or on the other hand test solutions found by the automated balancing process.

Before starting with the actual manual balancing, a spreadsheet has to be created. This spreadsheet is used as a supportive tool to document the balancing setup itself, the results of the performer of manual balancing and keep track of his parameter changes. A description of the used spreadsheets for the ZVG can be found under 8.1.4 as well as for the 2DR game under 8.2.4.

With such a spreadsheet the actual manual balancing process can begin. The performer of the balancing starts by adjusting the standard parameters with the help of methods like doubling or halving. He then conducts a playtest with the changed parameters. After that the adjustments and results of this playtest are documented in the created spreadsheet. These three steps are repeated until the performer finds a satisfying solution. The finding of such a solution ends the manual balancing process (cf. Figure 49).

Input

- Balancing Goals

Output

- Spreadsheet
- Documentation of results

11.7 Subprocess 5: Analyze Data

As a last step the generated data from the automated as well as from the optional manual balancing process has to be analyzed (cf. Figure 50). The analyzed data is used to:

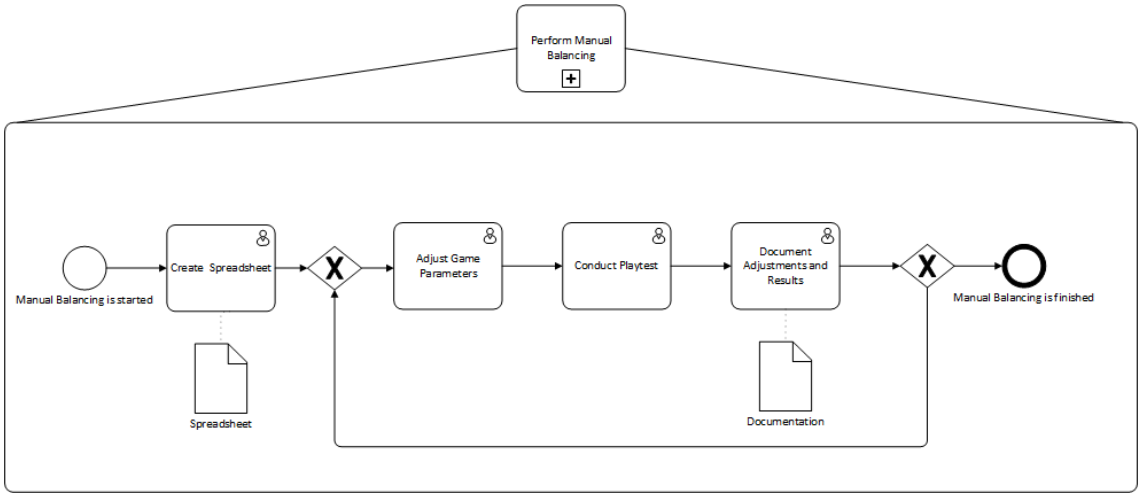


Figure 49: Subprocess: Perform Manual Balancing.

1. Analyze the performance
2. Analyze the solutions
3. Analyze the balancing dependencies

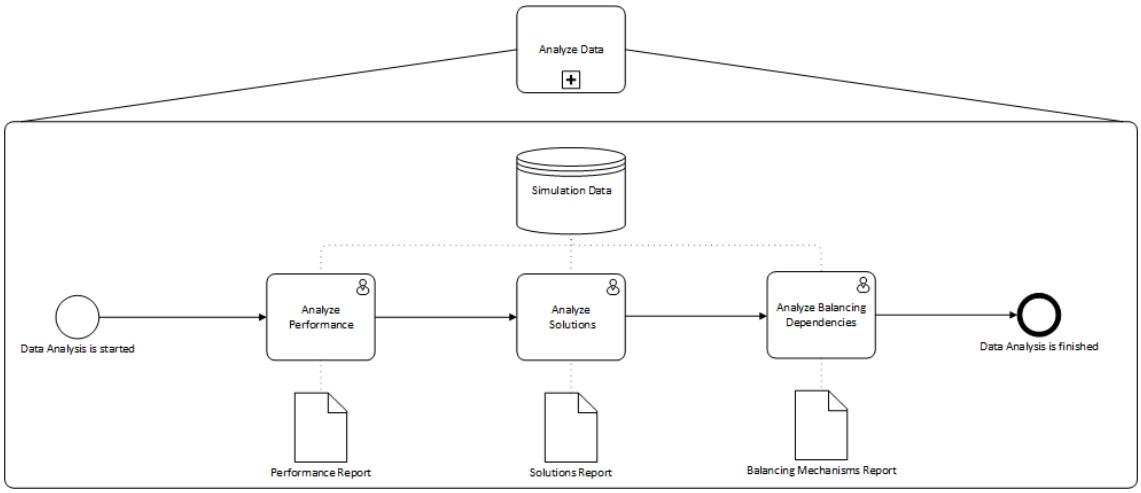


Figure 50: Subprocess: Analyze Data.

Input

- Simulation Data

Output

- Performance Report
- Solutions Report
- Balancing Mechanisms Report

12 Conclusion

12.1 Practical Use of Automated Balancing Tools

Limitations After the final presentation at the Blue Byte GmbH office several developers and designers discussed the re-usability and application areas of the project seminar results in practice. Most agreed that the research field of automated balancing has to be uncovered further and tested by smaller companies before being used in a large project. As the outcome yet has to prove its profitability, new approaches like automation result in too much risk for the project. Further problems indicated are high initial (setup of the BE) and maintenance costs (updating the code before, during and after every project) and the time pressure video game projects usually have to resolve.

Opportunities On the other hand, some designers liked the idea of having an educational tool for game designer – new designers especially could benefit from the support and insights the balancing environment could provide. The professional field of balancing mostly relies on experience and having the "right feeling" for parameters and concepts. Furthermore, designers have a single point of view which can be expanded by the results provided by the BE that might not have been considered otherwise. The tool may aid designers by turning the experience (tacit knowledge) to explicit knowledge which can be understood by other designers, companies or students. The participants agreed that currently a lack of teaching methods for new designers exists that could be filled with the results of the balancing tool. The possibility to re-use the BE for games that are part of a series, e.g. *Anno* or *Assassin's Creed*, featuring similar game elements and code was discussed as well. However, since a new entry in a series of games may introduce new core concepts or omit older concepts and may even use a different game engine, there is no obvious answer as to whether or not game series are an area where some of the limitations mentioned earlier do not apply - the approach should be evaluated case-to-case.

Procedural Stages for practice usability Finally, the lead designer proposed three potential procedural stages that the tool will have to pass until fully utilized in practice:

1. Embedded Support Tool for Designer
2. Highly Embedded Support Tool for Designer

3. Embedded Automated Design Tool

12.2 Discussion

The result of the PS has shown that it is feasible to apply a non-manual or semi-automatic method for the purpose of game balancing, which was achieved through algorithmic means.

From the practitioner's perspective, the process of manual game balancing can be costly with regard to time and money. Alternatively, an automated or semi-automated approach has the potential to reduce these costs. The application of an optimization algorithm with a suitable fitness function can then be leveraged to find promising candidates for a balanced game scenario that suits the intended target group and supports reaching the intended goal of the game. On the other hand, it may be required to ex ante invest additional conceptual work as well as developing activity into this alternative working method. The inputs shown in the working concept and process model need to be crafted in a fashion to optimally represent the target player to whom the game shall be marketed. This is specifically true for the player AI that acts as an agent to simulate the human player. Moreover, in addition to higher setup costs, the approach bears the risk of not being an accurate enough representation in the end. Additionally, modern games are complex and offer large open worlds. Therefore, at this stage in research, any complex game is best divided into sub-parts or sub-scenes with sub-goals that represent certain aspects of a game. These sub-scenes can be integrated more easily and with higher flexibility into a solution like the BE, reducing the risks and increasing the potential of delivering reliable results.

From a researcher's perspective, it was shown that there exists a research gap in applying CI techniques dedicated to the purpose of game balancing. The research at hand has shown that the methodology can in principle be applied. However, this does most certainly not mean that this research gap has been filled substantially, rather it has been opened up to spur further investigation. The approach in this PS was carried out within a very simplified game environment. On the other hand, it was applied in two different games from two different genres reinforcing the finding of the other. In both cases, the technique delivered valid results in a reliable fashion, which were play-tested manually and rated differently. Conclusively, good solutions were

among the population of all candidate solutions. To arrive at this point of knowledge about good value ranges of game parameters manually consumes significantly more time.

12.3 Outlook

Within the PS it could be demonstrated that the intended methodology of balancing game parameters through algorithmic means is feasible. This comes not without saying that there naturally exist limitations that offer suitable candidates for further research.

- The AI that acts as an agent to represent the human player behavior is an important lever in simulating the game and algorithmically finding balanced scenarios. Increasing the quality of the player AI will also increase the quality of the results from the optimization runs. Developing an accurate model of a human player is a research stream of its own [YT14]. Hence, it affords to combine these two research streams similar to the computational discovery of new game variants [IGT$^+$].

- EAs are known to deliver robust results to very hard problems. In this sense, the amount of parameters for balancing could potentially be increased significantly. On the other hand, for a smaller number of game parameter configurations, it may be more efficient to apply greedier or faster optimization algorithms and arrive at the same results, but faster. Hence, further comparison studies of different algorithms may be of interest to identify the threshold for reasonability in applying heavier population-based methods.

- It is unknown whether in the industry an automated or semi-automated approach for game balancing exists. The industry is spurred by a growing number of game releases. Moreover, the market situation has developed towards a more centralized distribution of games via online platforms. This drives competition and indirectly the need for higher cost-efficiency in game developing. In contrast of these circumstances it may be of high interest whether industry is already applying such methods or if there exists a demand for it. This could be quantified through a representative survey combined with a more qualitative approach of interviewing decision-makers, designers and developers.

- The process model outlined above was derived from the procedure applied and experience gained within this PS. Before putting this model into practice, it needs to be confirmed, rejected, or reformulated by another group of researchers or students dedicated to a similar scenario.
- The BE was developed within the Unity game engine directly due to technical constraints. An engine-independent approach would bring several advantages such as increased simulation speed as well as more possibilities of integrating other methodologies, e.g. different player AI.
- The conducted play-testing of the algorithmic solutions acted as a purely subjective benchmark. Empirical quantifying the conducted manual play-testing could validate player fun.
- Applying this concept to further games of potentially higher complexity.

Literature References

[ATA+11] Phillipa Avery et al. "Computational intelligence and tower defence games". In: *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE. 2011, pp. 1084–1091.

[BCP+10] Thomas Bartz-Beielstein et al. *Experimental methods for the analysis of optimization algorithms*. Springer, 2010.

[BDF+99] Andrew J Booker et al. "A rigorous framework for optimization of expensive functions by surrogates". In: *Structural optimization* 17.1 (1999), pp. 1–13.

[BPW+12] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 4.1 (2012), pp. 1–43.

[CBS+08] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *AIIDE*. 2008.

[CCP99] Valeria Cardellini, Michele Colajanni, and S Yu Philip. "Dynamic load balancing on web-server systems". In: *IEEE Internet computing* 3.3 (1999), p. 28.

[CR15] Alejandro Cannizzo and Esmitt Ramirez. "Towards Procedural Map and Character Generation for the MOBA Game Genre". In: *Ingenieria y Ciencia* 11.22 (2015), pp. 95–119.

[DST+08] Holger Danielsiek et al. "Intelligent moving of groups in real-time strategy games." In: *CIG*. 2008, pp. 71–78.

[ES08] AE Eiben and JE Smith. "Introduction to evolutionary computing (natural computing series)". In: (2008).

[GHM12] James E Gentle, Wolfgang Karl Härdle, and Yuichi Mori. *Handbook of computational statistics: concepts and methods*. Springer Science & Business Media, 2012.

[Gri10] Jaime Griesemer. "Design in Detail: Changing the Time Between Shots for the Sniper Rifle from 0.5 to 0.7 Seconds for Halo 3". In: *Game Developer's Conference*. 2010.

[Hag12] Johan Hagelbäck. "Potential-field based navigation in starcraft". In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE. 2012, pp. 388–393.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pp. 100–107.

[Hun05] Robin Hunicke. "The case for dynamic difficulty adjustment in games". In: *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*. ACM. 2005, pp. 429–433.

[IGT+] Aaron Isaksen et al. "Discovering Unique Game Variants". In: ().

[Jaf13] Alexander Jaffe. "Understanding Game Balance with Quantitative Methods". In: (2013).

[LBC10] Chong-U Lim, Robin Baumgarten, and Simon Colton. "Evolving behaviour trees for the commercial game DEFCON". In: *Applications of evolutionary computation*. Springer, 2010, pp. 100–110.

[Luc08] Simon M Lucas. "Computational intelligence and games: Challenges and opportunities". In: *International Journal of Automation and Computing* 5.1 (2008), pp. 45–57.

[Mad60] Albert Madansky. "Inequalities for stochastic linear programming problems". In: *Management science* 6.2 (1960), pp. 197–204.

[MAK+07] Sheri Markose et al. "A smart market for passenger road transport (SMPRT) congestion: An application of computational mechanism design". In: *Journal of Economic Dynamics and Control* 31.6 (2007), pp. 2001–2032.

[Mar06] Dietmar G Maringer. *Portfolio management with heuristic optimization*. Vol. 8. Springer Science & Business Media, 2006.

[MF09] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

[MR11] Rafael Marti and Gerhard Reinelt. *The linear ordering problem: exact and heuristic methods in combinatorial optimization*. Vol. 175. Springer Science & Business Media, 2011.

[MS14] Juliane Mueller and Christine A Shoemaker. "Influence of ensemble surrogate models and sampling strategy on the solution quality of algorithms for computationally expensive black-box global optimization problems". In: *Journal of Global Optimization* 60.2 (2014), pp. 123–144.

[PKH+13] Mike Preuss et al. “Reactive strategy choice in StarCraft by means of Fuzzy Control”. In: *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE. 2013, pp. 1–8.

[Pre15] Mike Preuss. *Multimodal Optimization by Means of Evolutionary Algorithms*. 2015.

[RDJ+06] Alex Rogers et al. “Computational mechanism design for information fusion within sensor networks”. In: *Information Fusion, 2006 9th International Conference on*. IEEE. 2006, pp. 1–7.

[RNI95] Stuart Russell, Peter Norvig, and Artificial Intelligence. “A modern approach”. In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), p. 27.

[Rum11] Rummell. “Adaptive AI to play tower defense game”. In: *Computer Games (CGAMES), 2011 16th International Conference on*. IEEE. 2011, pp. 38–40.

[Sch14] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC Press, 2014.

[Sir09] David Sirlin. “Balancing multiplayer competitive games”. In: *Game Developer’s Conference*. 2009.

[Tal09] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. Vol. 74. John Wiley & Sons, 2009.

[Tur50] Alan M Turing. “Computing machinery and intelligence”. In: *Mind* 59.236 (1950), pp. 433–460.

[TYS+10] Julian Togelius et al. “Search-based procedural content generation”. In: *Applications of Evolutionary Computation*. Springer, 2010, pp. 141–150.

[WF12] Leland Wilkinson and Michael Friendly. “The history of the cluster heat map”. In: *The American Statistician* (2012).

[WM97] David H Wolpert and William G Macready. “No free lunch theorems for optimization”. In: *Evolutionary Computation, IEEE Transactions on* 1.1 (1997), pp. 67–82.

[YT14] Georgios N Yannakakis and Julian Togelius. “A panorama of artificial and computational intelligence in games”. In: (2014).

Web References

[Blia] Blizzard Entertainment. *Diablo III Gameplay Guide*. Last accessed: 2015-11-05. URL: https://us.battle.net/d3/en/game/guide/gameplay/playing-with-friends#party-gameplay.

[Blib] Blizzard Entertainment. *Diablo III Season 3 EU Leaderboards*. Last accessed: 2015-11-05. URL: https://eu.battle.net/d3/en/rankings/season/3/rift-team-4.

[Blic] Blizzard Entertainment. *Diablo III Season 3 US Leaderboards*. Last accessed: 2015-11-05. URL: https://us.battle.net/d3/en/rankings/season/3/rift-team-4.

[Blid] Blizzard Entertainment. *Warhound Liquipedia Entry*. Last accessed: 2015-11-05. URL: http://wiki.teamliquid.net/starcraft2/Warhound.

[Blie] Blizzard Entertainment. *What is StarCraft II?* Last accessed: 2015-11-05. URL: http://us.battle.net/sc2/en/game/guide/whats-sc2.

[Dre] Dreamhack. *Dreamhack Winter 2014 Quarter Final: Fnatic vs. LDLC*. Last accessed: 2015-11-05. URL: https://youtu.be/mVGgT1lyp3A.

[Fna] Fnatic. *Fnatic Statement to Dreamhack Winter 2014*. Last accessed: 2015-11-05. URL: https://www.facebook.com/fnatic/photos/a.107427857589.94423.5985827589/10152908993022590/?type=1.

[Pre16] Mike Preuss. *Modern Game Artificial Intelligence Methods [pdf slides]*. Last accessed: 2016-03-14. 2016. URL: https://sso.uni-muenster.de/LearnWeb/learnweb2/pluginfile.php/780928/mod_resource/content/1/cig-ai-game-overview.pdf.

[Tec] Technopedia.com. *What is game balance?* URL: https://www.techopedia.com/definition/27041/game-balance.

[Vala] Valve Corporation. *CS:GO Release Notes 2014-12-10*. Last accessed: 2015-11-05. URL: http://blog.counter-strike.net/index.php/2014/12/11053/.

[Valb] Valve Corporation. *CS:GO Release Notes 2015-01-08*. Last accessed: 2015-11-05. URL: http://blog.counter-strike.net/index.php/2015/01/11119/.

[Wik] Wikipedia.org. *Balance (game design)*. Last accessed: 2016-03-11. URL: https://en.wikipedia.org/wiki/Balance_(game_design).

[WWUa] WWU Muenster. *Module compendium*. Last accessed: 2016-03-17. URL: https://www.wi.uni-muenster.de/prospective-students/master/curriculum.

[WWUb] WWU Muenster2. *Profile Information Systems and Statistics*. Last accessed: 2016-03-17. URL: https://www.wi.uni-muenster.de/department/groups/statistik/profile.

[Yan] John Yang. *Diablo III Patch 2.3.0 Developer Q&A*. Last accessed: 2015-11-05. URL: https://us.battle.net/d3/en/blog/19889980/patch-230-developer-qa-transcript-9-8-2015.

Declaration of Authorship

We hereby declare that, to the best of our knowledge and belief, this project seminar documentation titled "Documentation for the Balancing in Games Project Seminar Winter Term 2015/2016" is our own work. We confirm that each significant contribution to and quotation in this documentation that originates from the work or works of others is indicated by proper use of citation and references.

Münster, March 18, 2016.