TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

# Agent-Based Modeling as Level Design Method for Balanced Gamespaces

Marvin Neske

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

# Agent-Based Modeling as Level Design Method for Balanced Gamespaces

# Agentenbasierte Modellierung als Leveldesignmethode für balanced Gamespaces

| | |
|---|---|
| Author: | Marvin Neske |
| Supervisor: | Prof. Gudrun Klinker, Ph.D. |
| Advisor: | Daniel Dyrda, M.Sc. |
| Submission Date: | 15.08.2022 |

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.08.2022 Marvin Neske

# Acknowledgments

Special thanks to Daniel Dyrda. Thank you for your great guidance and astonishing advice throughout the past four months.

I also want to thank Paul for the countless of hours we spent working parallel in BBB. Although we spent most of the time being muted, it kept me motivated throughout the whole time.

# Abstract

Without balance in gamespaces, players will not like the gamespace and even worse, they will will not like the game. To prevent players from not liking the game, level designer create many iterations of each gamespace. For each iteration of a gamespace, playtest data is required. Rather than using human playtesting to generate the data required, we propose using agent-based modeling (ABM). ABM used in this context swaps out the humans during the playtesting and replaces them with artificial agents. To bring the artificial agents as close as possible to human-like behavior, we use machine learning agents. By training the machine learning agents, human-like behavior was achieved in a small set of scenarios.

The data generated by the trained agents in gamespace can be used by designers to more efficiently create new iterations of their gamespaces.

# Contents

# 1 Introduction

The process of game development is an iterative one [Sch20, p. 99f]. Therefore the process of gamespace development is an iterative one as well [KTy, Chapter: Planning: Iteration & playtesting]. To create a new iteration of a gamespace you first need to know what feels wrong and needs to be changed. To know what needs to be changed it is crucial to gather playtest data. This data can be of certain statistics like the win-rate of the teams in the gamespace. Data like the win-rate can give insights into the balance of a gamespace. Gathering data for balancing gamespaces using conventional methods has clear drawbacks.

To avoid the drawbacks of conventional methods, we propose another approach. The approach uses agent-based modelling (ABM) in combination with machine learning to model a situation similar to normal play-testing by humans. A tool implementing this approach is able to gather the required data to balance a gamespace without the drawbacks of conventional methods.

Before being able to propose the ABM-approach, three papers are presented which aim to improve the balance of shooter gamespaces too (See chapter 2).

An explanation for the general term "gamespace" follows in section 3.1. Since this paper is focused on *arena FPS gamespaces*, an introduction to them is given in section 3.2. Additionally, the last section of this chapter (See section 3.3) describes the conventional methods used to gather the necessary data for balancing out gamespaces while pointing out their drawbacks.

Afterwards, we propose another way to gather the playtest data required for gamespace balancing and explain why the proposed method fits (See section 4.1). This new fitting approach is ABM. After introducing ABM and explaining what components are included in agent-based models in general (See section 4.2), an agent-based model focused on gathering playtest data for gamespaces is conceptualized (See section 4.3). To ensure that a conceptualized model is working as intended, it needs to be verified, validated and replicated. These processes are described in section 4.4.

Replicating a model only works when knowing which tools were used. Therefore chapter 5 presents the tools which are used to implement a showcase prototype in chapter 7.

The showcase prototype is supposed to gather useful data for balancing arena FPS gamespaces. Key elements of a balanced gamespace are discussed in section 6.1.

Knowing the key elements, we identify what kinds of data the agent-based model should gather (See section 6.2). The gathered data can help game designers to more efficiently balance gamespaces.

Finally, chapter 7 describes the showcase implementation in detail. The detailed description allows for easy replication. Besides using replication to confirm the correctness of a model, validation can also be used to confirm the correctness. Thus, in section 7.3, we try to validate the implemented model.

Although many citations within the following work refer to video game *maps* and *levels*, we will refer to the same terms by using the word "gamespace". In cases where citations mention the terms game *maps* or *levels*, we will continue the work using the term "gamespaces" instead. Even tough the term "gamespace" does include all video game *maps* and *levels* within its meaning, not all gamespaces can be described with the terms video game *maps* or *levels*. But the meaning of the term "gamespace" will be explained in section 3.1.

# 2 Related Work

Before we begin to propose the ABM approach for gathering data relevant to gamespace balancing, the following chapter presents work which aims at creating balanced gamespaces.

The first work with the title "Evolving Maps for Match Balancing in First Person Shooters" [LLS14] tackles the problem of "[m]atch balancing [...] in an adversarial multiplayer shooter [...] from a completely different angle" [LLS14, Abstract]. Whereas matchmaking algorithms are the conventional method to generate balanced matches, Lanzi, Loiacono and Stucchi propose to use procedural content generation (PCG) and a genetic algorithm (GA). By using PCG and GA, they aim to create gamespaces which "[...] improve the game balancing for specific combinations of players skills and strategies" [LLS14, Abstract]. They present their work in the open source game *Cube 2: Sauerbraten* [pro].

The generated gamespaces in Cube 2 were evaluated by analyzing statistics of two-player matches in the gamespaces. The analyzed statistics form a *score ratio* for each player. The score ratio for one player refers to "how many points [they] scored with the respect to the overall points scored by both the players during the match" [LLS14, Section: IV, B]. A score ration of 0.5 for both players defines perfect balance for this paper. Therefore, when evolving gamespaces, their GA tries to achieve a final score ratio of 0.5 for both players after the two-player match.

The two-player match is not played by humans though. Instead, "[...] an approximation of the complex behaviors of human players" [LLS14, Section: IV, B] is done by using *bots from Cube 2*. Using these bots is advantageous since it is possible to speed up the simulation of the two-player match and gather results faster. Furthermore, the bots can be set into different difficulty levels allowing to balance for different scenarios [LLS14, Section: IV, B].

In total, they apply their approach to three different scenarios. All three scenarios are match-ups of players with different skill levels. While both players are equipped with the same weapon in the first scenario, the remaining scenarios equip each player with different weapons [LLS14, Section: VI]. After completing experiments on each of the three scenarios, they come to the conclusion that "[their] approach can successfully evolve maps for Cube 2 that improve the match balancing in each of the three scenarios tested" [LLS14, Section: VI].

Another paper regarding the balance of gamespaces tries to identify "[...] how game balance is affected by the level, represented as [a black, top-down] image [only showing objects in color], and each team's weapon parameters" [KLY17, Abstract]. This information is supposed to help "[...] a computational designer in creating a balanced battle environment for a pair of weapons used by opposing teams" [KLY17, Section: 3]. To identify a balanced battle, they "[...] [present] a computational model which can classify a matchup of a team-based shooter game as balanced or as favoring one or the other team" [KLY17, Abstract]. The computational model was trained using more than 50,000 simulated matches [KLY17, Section: 1]. Matches were played by two teams [KLY17, Section: 3]. Each team consisted of three artificial agents [KLY17, Section: 1] from a "[...] *Shooter AI* plugin library" [KLY17, Section: 3.2]. Every artificial agent within one team uses the same weapon type with the goal of scoring more kills than their opponent [KLY17, Section: 1].

The simulated matches were judged in terms of their balance. They measure the balance of one match by using the kill ratio of both teams and argue that it is balanced, if it ends in a tie [KLY17, Section: 3].

Using the playtest data generated by the simulated matches with artificial agents, "[s]everal neural network architectures, topologies, and activation functions were considered and tested" [KLY17, Section: 4.2]. The final results of the tests showed "[convolutional neural networks as] particularly capable of discovering patterns between the level architecture and weapon parameters" [KLY17, Section: 5].

Finally, they come to the conclusion that their trained model is able to increase the speed of automatic playtesting since it predicts the balance of a match by analyzing an image of the gamespace [KLY17, Section: 6]. The analyis of balance is solely based on the score distribution which they admit is not a sufficient measurement. However, they argue to still be able to contribute to level generation and design by "[...] taking into account the effect of players' weapons or inventories" [KLY17, Section: 6].

The last work to be presented is "Multi-Level Evolution of Shooter Levels" by Cachia, Liapis and Yannakakis [CLY15]. Within this work they "[present] a method for representing and evolving levels for first person shooter games which span more than one floors" [CLY15, Section: Conclusion]. The evolution of gamespaces is done via a genetic algorithm (GA). After generating the levels, they are evaluated using both "[...] simulations of artificial agents competing in the level and theory-based heuristics targeting general level design patterns" [CLY15, Abstract].

They also use *Cube 2* as a framework to generate gamespaces. Simulation-based evaluations of gamespaces are done with four artificial agents. The agents, which are included in Cube 2, are set on an average skill level. This skill setting is supposed to "simulat[e] a more human-like aiming skill" [CLY15, Section: Shooter Level Evaluation].

The aim of the simulation-based evaluation is to measure how well a gamespace manages to

1. "[...] have players engage their opponents as much as possible [...]" [CLY15, Section: Shooter Level Evaluation] which is measured as *fighting time* [CLY15, Section: Shooter Level Evaluation]

2. "[...] and maintain a balance in player kills" [CLY15, Section: Shooter Level Evaluation] which is measured as *kill ratio* [CLY15, Section: Shooter Level Evaluation].

While the *fighting time* is supposed to emphasise the generation of gamespaces with good pacing [CLY15, Section: Shooter Level Evaluation], the *kill ratio* " [...] aims to ensure that all players have an equal chance of winning" [CLY15, Section: Shooter Level Evaluation]. An equal chance of winning for all players can be derived from "[a]n equal kill count among artificial agents (of the same skill level) [since] that [means] there is little impact from the initial spawn locations of players" [CLY15, Section: Shooter Level Evaluation].

By combining both these simulation-based balance metrics along with additional metrics based on numerous level design patterns, they were able to conduct experiments to evaluate the mentioned gamespace generator [CLY15, Section: Experiments]. Their experiments and tests yield interesting results as the generated levels " [...] allowed players to engage in combat for longer periods of time [while] an additional floor [...] allows for players to "timeout" from combat" [CLY15, Chapter: Discussion].

Of the three presented papers, the first paper "Evolving Maps for Match Balancing in First Person Shooters" [LLS14] aims to create a balanced experience within shooters. They try "[...] to design the game maps such that the match would result as balanced as possible, independently from the players involved in the match" [LLS14, Section: II, A]. The generated gamespaces were evaluated using artificial agents.

The following paper "Learning the Patterns of Balance in a Multi-Player Shooter Game" [KLY17] does not generate gamespaces but rather aims to "classify a matchup of a team-based shooter game as balanced or as favoring one or the other team [by only using] the level, represented as an image, and each team's weapon parameters" [KLY17, Abstract]. Classifying the match-ups was done using a trained neural network. To train the neural network, they generated playtest data using artificial agents.

The last presented work "Multi-Level Evolution of Shooter Levels" [CLY15] also tries to generate gamespaces using genetic algorithms. For the evolution of gamespaces done by the genetic algorithm, the generated gamespaces were evaluated. Part of the evaluation were artificial agents playing within the gamespace.

All of these papers either tried to immediately generate gamespaces with balance in mind or provide information about the balance of a gamespace. Each of them using some kind of artificial agent as a tool for evaluation. Compared to the presented papers, this work does not try to use artificial agents as a tool for evaluation. Instead, the agents are just used to generate data according to the needs of designers. They are then able to manually improve on their gamespaces.

The next chapter continues with a definition for the term "gamespace" (See chapter 3).

# 3 Gamespaces in Video Games

As this paper aims to generate the data necessary to balance the gamespaces of 3D arena first-person shooters (FPS) we need to first introduce gamespaces (See section 3.1). After introducing gamespaces in general we continue to define how gamespaces for 3D arena FPS look like and why it is important to balance them (See section 3.2). Having a clear idea of what a 3D arena FPS-gamespaces looks like, we discuss how an evaluation of a gamespace is done conventionally while pointing out disadvantages (See section 3.3).

## 3.1 Introduction to Gamespaces

When players play a video game, the video game experience they have always takes place in some kind of *space*.

This space is called the "[...] the 'magic circle' of gameplay" by Schell [Sch20, p. 166]. For Schell the space "[...] defines the various places that can exist in a game [...]" [Sch20, p. 166]. Each place within the game has some kind of relationship to the other spaces in the game. This relationship is also dictated by the *space*.

Totten defines the gamespace, a video game experience takes place in, a bit different. For Totten this space "both embod[ies] gameplay and facilitate[s] the player's journey through it" [Tot19, p. 24]. It "[...] allow[s them] to better experience the game's mechanics [and] do[es] so [...] that players spend more time having fun and less time figuring out how to use the space" [Tot19, p. 24].

From these to definitions, we deduce that gamespaces are a combination of two things:

- the fictional environment the game takes place
- the rules and possible interactions within places of the game

The gamespace can be defined for every game. Schell provides the example of Tic-Tac-Toe and describes its gamespace [Sch20, p. 167ff]. Tic-Tac-Toe can be viewed with two different lenses.

The first lens refers to the functional space of a game. The functional space includes everything that stays when "[we] strip away all visuals, all aesthetics, and simply look at the abstract construction of a game's space" [Sch20, p. 166]. Regarding the abstract construction of Tic-Tac-Toe, all that the game consists of are "[...] nine zero-dimensional cells, connected to each other in a 2D grid" [Sch20, p. 168] (See fig. 3.1). This abstract 2D grid is the functional space of Tic-Tac-Toe. It is still perfectly playable when applying the game rules. However this is not how we usually see Tic-Tac-Toe being played. What

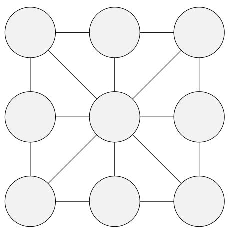


Figure 3.1: Functional space of the game Tic-Tac-Toe

is missing is the aesthetic space of the game.

The aesthetic space is the second lens a game can be viewed from. The aesthetic space mainly describes what players perceive. Players perceive the looks and feel of the game. In the case of Tic-Tac-Toe a game board is commonly drawn with two horizontal and two vertical lines that create a 3x3-grid (See fig. 3.2). Even though you are able

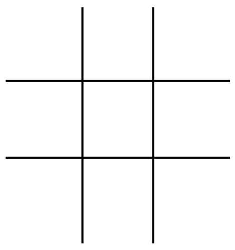


Figure 3.2: Common depiction of the game Tic-Tac-Toe

to place your mark wherever you want within one of the nine grid-cells, it is only important in which grid-cell your mark lies [Sch20, p. 167]. Hence the abstraction from the *continuous* aesthetic space to a *discrete* functional space.

This description of a gamespace can be made for every game. Since this paper deals with data generation for the balancing of 3D arena FPS gamespaces we continue describing them in the next section.

## 3.2 3D arena first-person shooter gamespaces

3D arena FPS games are in general a competition between two teams consisting of one or more players in a specially designed three dimensional continuous playing space [Ada14, Chapter: 3D Shooters: Gameplay Styles]. The core gameplay mechanic is shooting from the first person perspective. Hence the name first-person shooter (FPS). In an *arena-style* FPS the most simplest goal is to win over the enemy in a shooting duel. Those shooting duels are often the basis for gamemodes within the arena-style FPS [Ada14, Chapter: 3D Shooters: Gameplay Styles]. Depending on the mode, an arena FPS is played in different 3D gamespaces. These gamespaces aim to promote a fun, exciting and balanced game experience. Therefore, they are often filled with lots of corridors, covers, open spaces, flanking possibilities and more.

For every gamespace, we can follow the schematic of the example Schell gave with the game Tic-Tac-Toe and roughly distinguish the functional and aesthetic gamespace for an arena-style FPS.

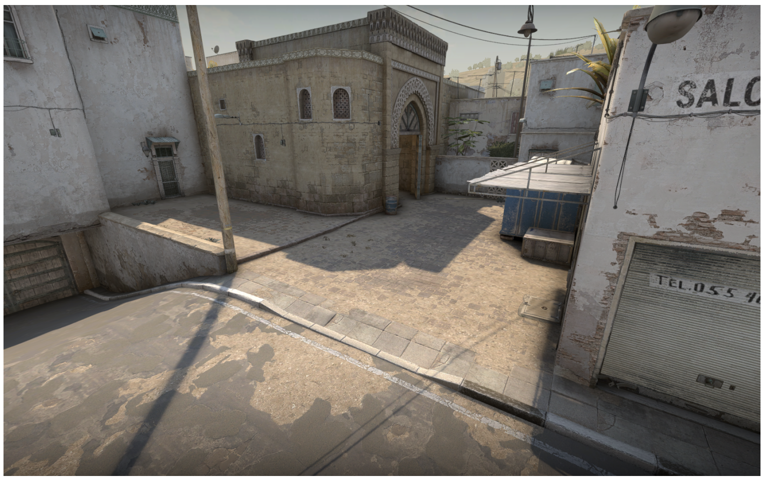


Figure 3.3: Screenshot from playable map "de_dust2" from the game *Counter-Strike: Global Offensive* [12]

When we strip down all the aesthetics from an arena-style FPS gamespace, we lose:

- the sounds like background music, environment sounds like water flowing, etc.
- the textures and basically everything that makes the gamespace look good
- the lighting of the gamespace which can be different depending on the area

Everything that is left after stripping down the aesthetics is the geometry of the gamespace. Figure 3.3 and 3.4 show what a rough stripping down of the aesthetics could look like for a small part of the in *Counter-Strike: Global Offensive* available map "de_dust2".

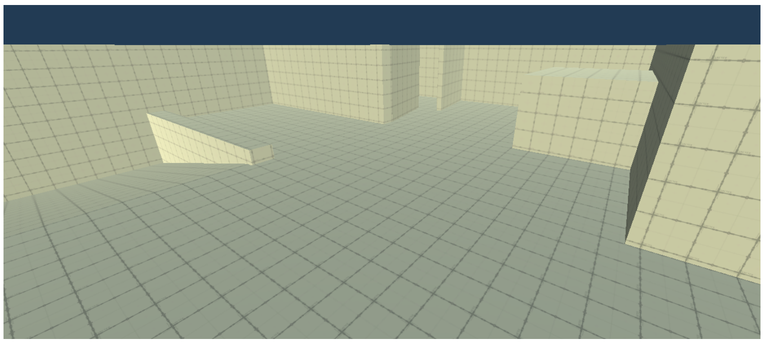


Figure 3.4: Rough breakdown of the functional space seen in Figure 3.3

While the combination of both the functional and aesthetic gamespace combined creates the full experience, the isolated functional space defines the how the game will be played within the space. The way the game is played within that functional gamespace must feel balanced. Otherwise an unbalanced game might feel "[...] monotonous, confusing, and frustrating" [Sch20, p. 212] to players. This would lead to them quickly losing interest and being "[...] terribly disappointed" [Sch20, p. 212]. Disappointing players is obviously not the goal of a game and needs to be prevented. Thus, our gamespaces must be balanced. However the only way to know if a gamespace is balanced, is to evaluate it.

## 3.3 Evaluation of Gamespaces

As balanced gamespaces, for our players to enjoy, are the goal, one must be able to tell that a gamespace is indeed balanced. One method to tell if a gamespace is balanced is to release it and let human players play in it. While human players play the gamespace, it is possible to gather data. If the data points towards an unbalanced gamespace, changes to it after release can fix the balance issues. The developers of the game *Overwatch* [16] have done this with their gamespace "Horizon Lunar Colony". After lots of community feedback for the gamespace, former game director Jeff Kaplan announced in a forum post [1] an upcoming rework for the map which aimed to address balancing issues. However, changing up gamespaces after release is not the best way. Players could

[1]Find the forum post here: https://us.forums.blizzard.com/en/overwatch/t/can-we-talk-lunar-colony-its-a-horrible-map/104996/6 (visited on: 19.07.22)

possibly already have had disappointing experiences within the gamespace, leading to them not coming back to play again.

Looking at what the developers of Overwatch did, it can have an advantage to release a not perfectly balanced gamespace early. The best way to check the balance of a gamespace is "[...] through constant iteration and testing" [KTy, Chapter: Balance] of the gamespace.

At a point of time in which the gamespace has not been released yet, a valid and often used approach is to let the development team or dedicated testers test the gamespace. Playtesting a gamespace with human players costs a lot of money and time. It costs extra money because either developers have to invest their working-time for testing or because dedicated testers have to be hired. Additionally, for every gamespace an extra amount of testing-time has to be accounted for.

If the gamespace gets released early however, the player-base can be much larger. More players automatically means more data about the gamespace, which could save time and cost. The gathered data then can deliver more precise information on how to change the gamespace into a more balanced one. The new gamespace with improved balance can be analyzed and improved the same way, leading to the process of "[...] constant iteration and testing" [KTy, Chapter: Balance].

Constant iteration and testing is the best way to balance a gamespace simply because the other option of building a mathematical model is too complex. To effectively iterate and test however, we need a constant flow of data. Getting this constant flow of data from developers and testers during the development time is not feasible, and getting the data from players after release is not recommendable either. Another way to generate lots of data is via the usage of *Agent-based Modeling*.

# 4 Agent-based Modeling for Gamespace Evaluation

As the conventional evaluation methods for gamespaces are either too expensive or risk bad player experiences, Agent-based modeling (ABM) was proposed as a new approach. This new approach is supposed gather the data required to balance a gamespace without the disadvantages which the conventional methods pose.

Before we explain ABM in more detail in section 4.2, we go over why ABM is useful in the use case of gathering data for balancing gamespaces (See section 4.1). Following the structure described in section 4.2, we propose an agent-based model for our use case (See section 4.3). Every agent-based model needs confirmation of its correctness. Confirmation of the correctness for a model is done using the three techniques presented in the last section (See section 4.4).

## 4.1 Is Agent-based Modeling fitting for Gamespace Evaluation?

Agent-based modeling (ABM) aims to "[model] complex systems composed of interacting, autonomous ′agents′" [MN05, Chapter 1.]. This approach of modeling complex systems can be projected onto our problem for the generation of playtest data required to balance gamespaces. For the generation of playtest data, we require players to play in the gamespaces. ABM seems like a perfect fit to generate the playtest data since players are just "[...] interacting, autonomous ′agents′" [MN05, Chapter 1.]". As agents, they play within a gamespace and thus compose a complex system. This complex system can be modeled via ABM using artificial agents rather than by using human players. Replacing humans by artificial agents solves some of the drawbacks that conventional gamespace evaluation methods come with.

Human developers and specifically hired employees used as tester have the problem of costing extra money while players used as tester might become disappointed due to bad experiences during the tests. Both of these problems are fixed by artificial agents as they do not cost extra money once they are implemented and also do not have feelings. Additionally, the usage of ABM enables a much faster data gathering process as simulations can be sped up and even run over night. Furthermore, simply setting up a simulation and starting it is also much faster compared to issuing a whole playtesting session with human players.

Another reason why ABM fits well for simulating players in a gamespace is because it has "[...] distinct advantages to conventional simulation approaches such as discrete event simulation [...], system dynamics [...] and other quantitative modeling techniques" [MN09, Chapter 5.]. ABM is better than conventional modeling techniques in simulating players in a gamespace because it satisfies six of the eleven proposed criteria by Macal and North. Their proposed criteria are hints for when agent usage can be beneficial. The usage of agents might be beneficial if at least one of the criteria is satisfied. The six fulfilled criteria for our use case are [MN09, Chapter 5.]:

1. *"When the problem has a natural representation as being comprised of agents"*

   - An arena FPS is filled with players as agents.

2. *"When it is important that agents have behaviors that reflect how individuals actually behave (if known)"*

   - The agents we model should reflect how human players would behave within the gamespace. While the actual behavior might not be fully known in a new game, basic strategy and movement patterns can be deduced from the behavior of other similar games.

3. *" When it is important that agents adapt and change their behaviors"*

   - Players in arena FPS adapt and change their behaviors depending on the gamespace and other players. Therefore it is also important that the agents are adaptive.

4. *" When it is important that agents learn and engage in dynamic strategic interactions"*

   - Similar to how the agents should adapt and change their behaviors they should also learn the strategies applicable in the gamespace just like human player do it.

5. *"When it is important that agents have a spatial component to their behaviors and interactions"*

   - In our case, the spacial component is a gamespace. Depending on the gamespace, behaviors and interactions from players change.

6. *"When scaling-up to arbitrary levels is important in terms of the number of agents, agent interactions and agent states"*

   - During the development of the game, different factors might change. Including scaling factors such as the size of each team and the game mechanics which define the interactions. Changes are likely due to the development process being an iterative one [Sch20, p. 100].

Concluding, we see that ABM is applicable for the modeling of players in gamespaces. By modeling players in gamespaces we can generate the data necessary for balancing gamespaces. Compared to other modeling techniques we showed that ABM is superior since it fulfills the majority of criteria by Macal and North. The next step is to look at ABM in general.

## 4.2 A General Agent-based Model

As mentioned already in section 4.1, agent-based modeling aims to "[model] complex systems composed of interacting, autonomous 'agents'" [MN05, Chapter 1.]. While the two main building blocks are the agents themselves and the complex system they are part of, we can further identify and specify the five basic components every agent-based model consists of [WR15]:

1. *The Agents*: They are "[...] self-contained, modular, [...] uniquely identifiable [,] autonomous, [they have a] state that varies over time [and they are] social [due to] having dynamic interactions with other agents" [MN05, Chapter 2.3]. They also may be " [...] adaptive [,] goal-directed [or] heterogeneous [...]" [MN05, Chapter 2.3]. Agents have dynamic or static *attributes* like their name or current position. All possible combinations of their attributes combined define all the possible states of the agent. Additionally, agents have "behaviours that relate information sensed by the agent to its decisions and *actions*" [MN05, Chapter 2.3].

2. *The Environment*: It can be anything from a discrete space to a continuous space or a graph structure. It "[...] may simply be used to provide information on the spatial location of an agent relative to other agents or it may provide a rich set of geographic information [...]" [MN05, Chapter 2.3].

3. *The Interactions*: They specify all possible actions which can take place between the two previous components: agent-self interaction, environment-self interaction, agent-agent interaction, environment-environment interactions, agent-environment interactions [WR15].

4. *The Observer*: It controls the simulation of the model. It sets up all participating agents, starts the simulation and controls everything.

5. *The Scheduler*: It decides in which order parts of the model act. The order of operation can be sequential, synchronous or parallel [WR15].

These are the five components of every agent-based model. We now will identify how a model to generate playtest data should look like by defining each of the five components.

## 4.3 Agent-based Modeling in the Context of Gamespace Evaluation

The five general components of an agent-based model can be projected onto our model of players within a gamespace. Doing so, we receive these five specific components:

1. *The Agents*: In our case the agent should resemble what a human player would do within an arena FPS. Therefore the agent should fulfill these criteria:
   - it must be able to take the *same actions* a human player can take such as shooting and moving
   - it has to have the *same goal* as a human player
   - the agent must have all the *same attributes* a human player character in the game would have as well
   - it must be *adaptive* as human players also adapt to new gamespaces or enemies in the arena FPS
   - it should *behave much like a human player* would do in the gamespace

   In most commercial video games nowadays non-player-characters (NPC) will be implemented into the game to act as possible opponents for players. These NPCs in most cases will at least satisfy some the required criteria. However the in-game NPCs unfortunately won't be a good fit for the agents needed for ABM in this context. The in-game NPCs in commercial video games are designed to deliver a fun experience to the player. They create the fun experience by keeping up a good fight, but ultimately they should not be better than the players and lose. If the NPCs were to be better than the players, they might become frustrated and stop playing. Due to this, conventional NPCs are not suitable for the context of modeling players in a gamespace. Additionally, we want the agents to be adaptive just like human players. For example, they must be able to learn when encountering a new gamespace. Therefore the implementation of the agent should be able to learn.

2. *The Environment*: The agents must be able to navigate through the environment just as human players can do. Therefore the environment must be the same gamespace human players also play in. However the agents do not need to

perceive the aesthetics of the gamespace. Since the agents do not need to perceive the aesthetic space, it is only necessary to use the functional abstraction of the gamespace. This functional abstraction can be created as it is described in section 3.2.

3. *The Interactions*: Regarding the interactions, agents have to be able to interact with the environment, other players or themselves in the same way, human players can. For example when a gamespace comes with an interactable elevator which human players can use, the agents should be able to use it too.

4. *The Observer*: This component has the task of controlling the simulation to run just like a real game with human players. It sets up all the agents, lets it run and ends the game in the same manner after an end-game condition has been met.

5. *The Scheduler*: Looking at an arena FPS, every player can take actions whenever they want. This ability comes from a continuous time flow within the game. Additionally, multiple players can take actions at the same point of time. The scheduler must realise the same possibilities for the agents. An asynchronous scheduler would lead to problems. These problems occur when both agents would like to shoot at each other at the same time. Due to being asynchronous, the scheduler would handle the damage dealt of one agent before the other one. If the dealt damage of the first agent then kills the second agent, the attempt of the second agent to shoot as well would not be considered anymore due to it being dead already. Therefore the scheduler for modeling players in gamespaces must be a synchronous one.

To model players in a gamespace we must implement these five components. They can be implemented during the development phase at a point in which the basic game has already been set up. We would then need to implement these five components into the game. As it is best practice to implement a game with low coupling and high cohesion, it should not be difficult to implement the model into the game. For example, if the game includes a character controller with low coupling and high cohesion, it should not be problematic to let the agent logic control a character. The same principle holds for each of the five components in the model.

Before implementing the model however, one should make sure that the model correct. Checking a model for its correctness can be done by verifying, validating and replicating it.

## 4.4 Verification, Validation and Replication of an Agent-Based Model

When working with and implementing ABM, an important part of the process is to make sure the model is accurate [WR15, p. 311]. Confirming the accuracy can be done by verifying, validating and replicating the model.

### 4.4.1 Verifying an Agent-Based Model

Wilensky and Rand state that "[m]odel verification is the process of determining whether an implemented model corresponds to the target conceptual model" [WR15, p. 311f]. The model described in the last section (See section 4.3) is the conceptual model designed to generate playtest data for balancing gamespaces. An implementation of the model requires a certain degree verification before it is used. To verify an implementation Wilensky and Rand advise to start with a simple model and increment its complexity bit by bit. This approach is done since, "[i]f a model is simple to begin with, it is easier to verify than a complex model" [WR15, p. 312]. When further complexity is then added bit by bit it is also easier to verify the new bits [WR15, p. 312]. However, verifying each bit of the model on its own doesn't verify the whole model due to possible complications in between the bits [WR15, p. 312].

Ultimately when verifying the implementation or just pieces of it, we can use *Verification Testing* [WR15, p. 315]. Verification Testing "[...] is a form of unit testing [in which we create] small tests that check whether individual units are working correctly in the code" [WR15, p. 317]. As an example, for the proposed conceptual model in section 4.3, a number of Verification Tests could be dedicated to check whether the agents have all the same interactions a player has.

### 4.4.2 Validating an Agent-Based Model

While through verification it is ensured that implementation fits the conceptual model, via the validation it is "[...] ensur[ed] that there is a correspondence between the implemented model and reality. [...] Models are simplifications of reality; it is impossible for a model to exhibit all of the same characteristics and patterns that exist in reality" [WR15, p. 325]. Therefore it is important to put the focus of the model onto the questions we want it to answer. To ensure a model to be focused around the questions, it must implement the key elements of reality relevant to the questions. Checking a model for this constraint can be done by combining multiple techniques [WR15, p. 325f].

### Microvalidation

The first technique is *Microvalidation*. It is a process "[...] making sure [that] the behaviors and mechanisms encoded into the agents in the model match up with their real world analogs" [WR15, p. 326]. Wilensky and Rand describe the Microvalidation process using a flocking model of birds [WR15, p. 329]. By projecting their approach onto the proposed model for gathering data, it is possible to microvalidate it. During the microvalidation, we first need to keep in mind the question our model tries to answer. The model is supposed to gather playtest data relevant for balancing gamespaces. Therefore some features of the real world analog to our agent, the players, are not necessary for our model. Feature which are not necessary for the agent, and features that are, can be found by either leaving them out or adding them to the model. If that "[...] makes any significant difference to the model results" [WR15, p. 329] we can assume that the features are either necessary or not.

Comparing the agents in the proposed model in section 4.3 to real world players, the agents in theory match up with the players. However, while it is not difficult to match parts of the model such as their attributes or actions, matching the behavior as a whole is a challenge and thus needs special consideration.

### Macrovalidation

While Microvalidation considers the relationship between agents and their real world counterparts, "*Macrovalidation* is the process of ensuring that the aggregate, emergent properties of the model correspond to aggregate properties in the real world" [WR15, p. 326]. Aggregate properties in the real world is the arena FPS game-loop itself as well as the play-styles and common behaviors of real players in a gamespace. A win-oriented player, for example, will in most cases prefer a high-ground over a low-ground in an arena FPS. To macrovalidate the proposed model, it should be possible to point out similar behaviors between the model and players in a real arena FPS.

### Face validation

By using *Face validation* we "ensure that someone who looks at the model " on face " (i.e., without detailed analysis) can easily be convinced that the model contains elements and components that correspond to agents and mechanisms that exist in the real world" [WR15, p. 332]. Hence, an implementation of the proposed model in section 4.3 must be immediately identifiable as an arena FPS. For example, an agent that is constantly spinning would not be face valid because it is not something that someone looking at an arena FPS is expecting. Unexpected behaviors such as constant spinning should not occur in the model as they are unreasonable [WR15, p. 332].

**Empirical validation**

The last technique is *Empirical validation*. Rather than basing comparisons on looks, empirical validation demands that the "[d]ata produced by the model [to] correspond to empirical data derived from the real-world phenomenon" [WR15, p. 332]. In the case of gamespace evaluation we are able to gather different types of data which are related to gamespace balance. For example data such as win-rates or death-locations. By letting both the model and human players play on the same gamespace, we can gather comparable data sets. The more similar the data sets are, the better the model.

### 4.4.3 Replication an Agent-Based Model

Replicating a model means to create a new "[...] implementation by one scientist or group of scientists of a conceptual model ( replicated model ) described and already implemented ( original model ) by a scientist or group of scientists at a previous time" [WR15, p. 337]. As much as replicating a physical scientific experiment is part of the scientific process, replicating is just as much a part of the modeling process [WR15, p. 336].

Replication helps to prove that no mistakes were made during the original modeling process as well as it "[...] increases our confidence in the *model verification* since a new implementation of the conceptual model has yielded the same results as the original" [WR15, p. 336]. Additionally, "[r]eplication can also aid in *model validation* as it requires the model replicators to try to understand the modeling choices that were made and how the original modelers saw the match between the conceptual model and the real world" [WR15, p. 336].

To allow an already implemented *original model* of a conceptual model to be replicated, at least six dimensions have to declared [WR15, p. 337f]. By changing one or more dimensions, a new *replicated model* of a conceptual model is created [WR15, p. 338]. The new replicated model can can have one or more differences across these six dimensions:

1. *Time*: Time is a "[...] dimension of replication that will always be varied" [WR15, p. 338]. If time is the only differing dimension in a replication by the original researcher, the results must be the same. Otherwise it "would indicate that the published specification is inadequate, since even the original researcher could not re-create the model from the original conceptual model" [WR15, p. 338].

2. *Hardware*: Changing either the run-environment to another machine or replicating the model on a different platform "[...] should [not] provide significantly different results" [WR15, p. 338]. In the case of different results, further investigations are necessary [WR15, p. 338].

3. *Language*: Using a different programming language "[...] can cause differences in replicated models. [Therefore for] a model to be widely accepted as part of scientific knowledge, it should be robust to such changes" [WR15, p. 338].

4. *Toolkits*: When replicating models with different toolkits, we often encounter problems which lie within the conceptual model as well as within the toolkits themselves [WR15, p. 339].

5. *Algorithms*: Since there are multiple different algorithms to solve certain problems such as searching, a replication could be implemented using other algorithms than the original had used. An implementation using other algorithms could both output the same or different results [WR15, p.339].

6. *Authors*: If a replication by another researcher is able to generate the same results as the original, it is likely "[...] the model is accurately described and the results are robust to changes in the dimensions of replication that have been altered" [WR15, p.339].

In general, a replication is successful, if "[...] the replicators are able to establish that the replicated model creates outputs sufficiently similar to the outputs of the original" [WR15, p.339].

To allow for a successful replication, we will present tools in the next chapter (See chapter 5) which will later be used for a showcase implementation (See chapter 7).

# 5 Tools used for the implementation

An implementation prototype for the agent-based model proposed in the previous chapter (See section 4.3) can be done via a variety of tools. We chose the tools described in this chapter to demonstrate how an implementation might look like and to build a prototype which will be discussed in detail later on (See chapter 7). First, we propose a game engine to build the model implementation in (See section 5.1). Afterwards we describe a tool for creating the environments for an implementation (See section 5.2). In the end we present a tool that includes two ways of implementing a learning agent within the proposed game engine (See section 5.3).

## 5.1 Unity for the implementation

For the base of the implementation we propose the use of the *Unity game engine software*. The *Unity game engine software* is a tool "[...] to create 2D and 3D games, apps and experiences" [Tec21, Unity User Manual 2020.3 (LTS)]. To create a model of players within a gamespace we require a very basic video games structure. The very simple video game structure is supposed to resemble a real use case within the development of a game. Thus the *Unity game engine software* fits perfectly to build a prototype.

The implemented prototype (See chapter 7 for more detail) was setup in the Unity software version 2020.3.30f1 which uses C# as programming language. Additionally, two unity packages were installed. Packages contain new features. Those features are toolkits that can extend the core line of features the Unity game engine software offers [Tec21, Packages and feature sets].

## 5.2 The Unity ProBuilder toolkit

The first package which additionally got installed for the project is the *Unity ProBuilder* toolkit of version 4.5.2. The Unity ProBuilder toolkit adds features to "[...] build, edit, and texture custom geometry [...]" [Tec21, ProBuilder]. It was used to create different gamespaces in the project which. The different gamespaces were used to gather data and train the machine learning agents.

## 5.3 The Unity ML-Agents toolkit

The second installed package is the Unity *ML-Agents* toolkit of preview version 2.3.0-exp.2. With the Unity ML-Agents toolkit you are able to "[u]se state-of-the-art machine learning to create intelligent character behaviors in any Unity environment" [Tec21, ML Agents]. Created "[...] games [can] serve as environments for training intelligent agents using deep reinforcement learning and imitation learning" [Jul+18, github: About].

### 5.3.1 Reinforcement Learning in Unity

Reinforcement learning [Jul21], in general, uses an action taking *agent* with different *states*. Each state has its own set of actions. Agents decide which actions to take according to their *policy*. Actions taken while the agent is in a certain state can yield a *reward* and/or move the agent into a new state. States have a value depending on how rewarding it is to be in that state. The main goal of an agent is to maximize their total cumulative reward by taking the right actions. The agents themselves are always contained within an *environment* [Jul21, Reinforcement Learning with Bandits].

The Unity ML-Agents toolkit offers the ability to define Agents. These agents can then be trained. They are trained according to

- the observations they make in their training environment
- the actions they can take
- and the rewards they get from the environment

[Tec21, ML Agents]. This Structure was used to implement the machine learning agents for this paper. How exactly the agents are implemented in the project will be described later on (See chapter 7).

### 5.3.2 Imitation Learning in Unity

The ML-Agents package also provides the ability to use "Imitation Learning via Behavioral Cloning" [Jul18]. The difference of Imitation Learning to Reinforcement learning lies within the learning process. Imitation Learning-agents do not solely learn through a reward/penalty system as Reinforcement Learning-agents do. Instead Imitation Learning-agents learn by imitating a demonstration of the intended behavior. Developers can create these demonstrations by playing in position of the agent and recording the inputs and observations.

By using these tools, the proposed agent-based model in section 4.3 can be implemented to generate data. The data is supposed to help designers to balance their gamespaces. The next chapter explains what a balanced gamespace is.

# 6 Balance in Gamespaces

One of the most important parts of game development is the balancing process. Without good balancing, even an astonishing game can end up in a "monotonous, confusing, and frustrating" [Sch20, p. 212] experience. Experience created by a shooter game depend very heavily on their level design [Ada14, Chapter: Summary]. Therefore the level and gamespace design must be in balance too.

To balance the gamespace of an arena FPS, the previous chapter (See chapter 5) presented a number of tools. These tools can be used to implement the agent-based model described in section 4.3. Using an implemented version of the agent-based model it is possible to bypass human playtesting and still gather play data for gamespaces.

The following chapter aims to identify which data might be useful for balancing arena FPS gamespaces. To identify useful data to balance a gamespace, we first go through a few definitions of game balance (See section 6.1). Going through different definitions of game balance allows to identify key elements for a balanced gamespace (See section 6.2). Additionally, different kinds of data are presented that might help a designer to balance their gamespaces.

## 6.1 Different definitions of game balance

To be able to balance gamespaces, one must first know what balance is. However, a central definition for good "game balancing" does not exist [BG20, p. 38]. Multiple definitions of game balance by different author have been studied by Becker and Görlich. The following section will present some of the studied definitions. The presented definitions will be relevant to later on identify key elements for gamespace balance.

In his book *Fundamentals of game Design*, Adams defines a balanced game to be "[...] neither too easy nor too hard" while focusing on player skill as "[...] the most important factor in determining [...] success". A balanced game also has to be "[...] fair to the player (or players) [...]" [Ada13, Chapter 15: What Is a Balanced Game?]. Along other factors like the overall difficulty, Adams defines fairness as an integrated part of a balanced game

When working together with Rollings, Adams again came to the conclusion that the players skill should be the "[...] determining factor for the success [...]" [BG20, p. 28] in a balanced game. Thus in a balanced game "[...] a better player should ordinarily be more successful than a poor one [...]" [BG20, p. 28]. For a better player to ordinarily be more successful, Rollings and Adams include fairness as a "[...] criteria found in well-balanced games" [BG20, p. 28].

Shell's definition also agrees that fairness is essential for balance. He first states that balancing is just the process of "[...] adjusting the elements of the game until they deliver the experience you want" [Sch20, p. 212]. His definition of balancing doesn't mention fairness in the first place but later on he notes fairness as a "[...] quality that players universally seek in games [...]". Therefore including fairness as a factor within the targeted experience.

According to Novak "[a] game is balanced if players perceive that it is consistent, fair, and fun" [BG20, p. 28]. With this statement Novak also includes fairness as a part of game balance.

These definitions clearly define fairness as a part of a balanced game. Marc Brown takes it a step further by saying that "[b]alance is the art of making sure that all options in a multiplayer game are fair [...]" [BG20, p. 30]. Thus for him, a game can't be balanced if it isn't fair.

Besides fairness, many authors such as Adams [BG20, p. 26] also mention dominant strategies as a phenomenon to be avoided. Dominant strategies emerge "[w]hen choices are offered to a player, but one of them is clearly better than the rest" [Sch20, p. 221]. Clearly better choices render the fun out of games since they remove any room for impactful player decisions [Sch20, p. 221].

For Keith Burgun it is important "[...] to keep a player's decision impactful" [BG20, p. 23] and "to prevent a ruination of the game by dominant strategies" [BG20, p. 23]. Achieving these two goals defines game balance for Burgun. To reach game balance, the "[...] the prime objective [lies] in keeping game elements relevant" [BG20, p. 23].

Similar to Burgun's definition, for David Sirlin game balance is about providing players with "[...] meaningful decisions between promising alternatives" [BG20, p. 24].

Dominant strategies lead to a redundant selection process. The selection process becomes redundant as there is no reason to pick a weaker option over the dominant one [BG20, p. 26]. In contrast to having a game with dominant strategies, having perfect equality in a game does not provide a positive experience either [BG20, p. 24]. In a game with "perfect equality [...] there would be no reason left to choose any action over another" [BG20, p. 24]. Therefore Dan Felder and James Portnow suggest having "[...] small power gaps [...]" [BG20, p. 24] or "[...] just a little bit of imbalance [...]" [BG20, p.

27].

Summing up this section, "[s]hooter games depend very heavily on level design for their experience" [Ada14, Chapter: Summary]. To create levels or gamespaces which create a good experience, they need to be in balance. Looking at different definitions of balance, key elements that stand out relevant for arena FPS gamespaces are fairness and the absence of dominant strategies as well as perfect equality

## 6.2 Key Elements of Gamespace Balance

It is possible to project the key elements identified in the previous section (See section 6.1) onto gamespaces. Doing so, gives insight on how to balance gamespaces. At first, the statement from Felder and Portnow regarding the effect perfect quality is discussed in the context of gamespaces. Afterwards, the concept of dominant strategies in terms of gamespace will be introduced (See section 6.2.2). In the end, we try to project the key element of fairness onto gamespaces (See section 6.2.3).

### 6.2.1 Symmetry in Gamespaces

Felder and Portnow state that perfect equality in a game "[...] render[s] a lot of decisions meaningless [...] since since there would be no reason left to choose any action over another" [BG20, p. 24]. A lot of the decision making process also gets lost in perfectly equal gamespaces. A perfectly equal gamespace in an arena FPS would be perfectly symmetrical. Regarding symmetry, Adams additionally states that"[...], video game players generally consider symmetric games rather uninteresting" [Ada13, Chapter 15: Balancing Games with Symmetry]. Therefore designers should avoid creating symmetric gamespaces.

### 6.2.2 Dominant Strategies in Gamespaces

As defined in the last section (See section 6.1), dominant strategies emerge "[w]hen choices are offered to a player, but one of them is clearly better than the rest" [Sch20, p. 221]. To avoid a clearly better option to emerge, Adams suggests to use intransitivity [BG20, p. 26]. For example, Adams brings up the game Rock-Paper-Scissors as an intransitiv system [Ada13, Chapter 15: Intransitive Relationships (Rock-Paper-Scissors)]. In Rock-Paper-Scissors "[...] every game element can be beaten by some other[. This] supports the avoidance of dominant strategies" [BG20, p. 26].

Creating gamespaces with intransitivity in mind will help to end up with a more balanced design from the get go. For example, it is most likely not advisable to have an advantageous position available for a team which has clear vision over a strategic position this team has to defend but in return is not reachable or very much visible for the opponents. As long as the position is not reachable or very much visible for the opponents, players of the defending team have a dominant strategy in just waiting at the advantageous position and eliminate every opponent coming to the strategic position.

A solution to this situation can be found with intransitivity. By adding a flanking route leading to the advantageous position, it becomes beatable and is not a dominant strategy anymore.

Even if designers create gamespaces with intransitivity in mind, their first attempt will never be as good as a later iteration that has been improved using playtest data. There are many different kinds of playtest data which can help expose dominant strategies.

**Data to expose Dominant Strategies in Gamespace**

While doing playtests, one can gather different kinds of data depending on the subject to test. When it comes to exposing dominant strategies in gamespaces a few types of data might help.

For example a heatmap showing where players are being eliminated. Knowing where the most common dangerous spots are, designers might choose to place a cover there for the next playtest iteration.

Another possibly useful kind of heatmap shows where players eliminate opponents from. When most of the eliminations are done from a single highground, it might be time to reiterate on that highground.

Besides avoiding dominant strategies, fairness has been identified as a key element to balanced gamespaces.

### 6.2.3 Fairness in Gamepaces

Similar to how the aforementioned authors define "game balance" differently, they also have different definitions for the term "fairness". These definitions will now be reviewed to finally come up with a measurable unit for fairness in gamespaces. The measurable unit then can be used by game designers to balance the gamespace.

One great distinction has to be made before looking at different definitions of fairness. This is because fairness can be defined for two different types of games. The first type being fairness for player vs. player games (PvP-games) and the second type being player vs. environment games (PvE-games). PvE-games are games in which players do not directly play against each other. This paper is focused on arena FPS-games and thus definitions of fairness for PvE-games will not be further discussed. We will continue with definitions referencing either the general concept of fairness or fairness-definitions directly aimed at competitive PvP-games.

Adams considers PvP-games in general to be considered fair by players if [Ada13, Chapter 15: Making PvP Games Fair]:

- the rules give each player an equal chance of winning when play begins
- the rules do not give advantage or disadvantage to players unequally during the game in ways that they cannot influence or prevent apart from the operation of chance (in moderation)

Thus creating a concept of fairness in which all players have an equal chance of winning in the beginning and none are privileged during the game.

Together with Rollings, Adams adds another point to their definition of fairness. They state that a "[...] player perceives fairness by always being able to win, even after early mistakes" [BG20, p. 28]. Thus a dire game situation should always be able to be turned into a favourable one by players.

Schell describes a feeling of a game being unfair when opposing forces have an advantage which makes it feel like they are impossible to defeat [Sch20, p. 213]. Avoiding the feeling of an unwinnable situation also implies to always offer a possibility to win just like Rollings and Adams proposed.

As mentioned earlier Novak believes that a balanced game isn't complete if it isn't fair. Regarding fairness he states that "a better player should be more successful in general [...] than a less-skilled-player, unless the game is based purely on luck instead of skill" [BG20, p. 27]. This agrees with previous statements as it demands a game in general to not give out unfair disadvantages to more skilled players which would make it impossible for them to win. Or on the other hand give out advantages big enough to less skilled players leading to a safe victory.

Agreeing with Adams, Sirlin demands a fair game to offer "[...] an equal chance of winning [to each player]" [Sir02, Part 1: Definitions]. This condition must stand "[...] even though [players] might start the game with different options" [Sir02, Part 1: Definitions].

Different options per player make another point of distinction. Besides the distinction of a game being either PvP- or PvE-centric, games also lie on a symmetry-spectrum [Sir02, Part 1: Definitions]. Sirlin calls games symmetric when they offer the same starting options to all players. They mention the game Chess as an almost perfectly symmetric game. Only *almost perfectly symmetric* though, since white moves first. Thus white has different starting options compared to black which makes Chess a slightly asymmetric game. Similar to how Chess isn't perfectly symmetric other games can be more or less symmetric by offering varying options to each player. The more asymmetric a game becomes the more difficult it becomes to make it fair for all players. While asymmetric games aren't fair from the ground up, Becker and Görlich have concluded "[...] that fairness is inherent to symmetrical games [...]" [BG20, p. 38].

Concluding the reviewed definitions, we can distinguish a few points that must hold for a game to be fair:

- No player should receive an advantage big enough to win purely because of it unless the opposing force receives it too.
- Perfectly symmetrical games are the ground truth for fairness.
- All players should have roughly the same probability of winning at the beginning of a game
- In general, when a more skilled player competes with a less skilled player, the more skilled player should win more often. This makes skill the determining factor for success.

Following these guidelines we can deduce a simple measurement for fairness. Assuming we have a game fulfilling the guidelines. The game is played by two teams directly competing against one another in a PvP-style. The starting positions of our game are equally strong and give no advantage to one of the teams by either being fully symmetric or properly setup. During the game player-skill is still the main determining factor for success. Thus if one of the teams was significantly more skilled than their opponents, they would win more often over a series of games. However if

- the games starting positions are fair
- no game-changing advantages are handed out by the game to only one team
- the teams are equally strong

we would expect both teams to win roughly half of their games *if the gamespace if fair*. Winning half of their games or rather a roughly 50% win-rate in a gamespace thus should be a sufficient measurement for fairness.

If a roughly 50% win-rate is the result of the playtests made for a gamespace, we can assume that it is more fair by balancing. Thus, keeping track of the win-rates during playtests is key to making a gamespace more balanced. Knowing in which direction a gamespace might be favored can be useful for designers as they might want to achieve a certain favor within a gamespace.

Knowing how to intentionally create a favored direction within a gamespace based of playtest data can be especially useful. Some games such as *Counter-Strike: Global Offensive* [12] feature a defending/attacking theme. In games with a defending/attacking theme designers might want a whole map or level to be fair and for both teams to roughly have a 50% win-rate. However, a whole map or level can be broken down into smaller gamespaces which all have their own balance. The designer-intended balance within a smaller gamespace might not even be fair. Since "[...] many games are actually made far more engaging by just a little bit of imbalance, multiplayer games especially" [BG20, p. 27].

By combining both the data from playtests for avoiding dominant strategies and for fairness, game designer are able to make better decisions on how to change a gamespace in order to achieve a better balance.

The next chapter takes the proposed agent-based model from chapter 4 and uses the tools presented in chapter 5 to implement a showcase prototype for gathering the data described in this chapter.

# 7 Description of the Implementation

The previous chapters have described all the theories and concepts that are required to implement and use the ABM approach used to gather the playtest data for balancing.
In this chapter we want to demonstrate the usage of the ABM approach. However, the approach would be used during the development of a real game. Since we do not have the chance of using the ABM approach on a real game in development, we demonstrate the approach by implementing an agent-based model for an imaginary arena FPS.

This imaginary arena FPS will be very simple. It's about two opposing teams within a gamespace as the arena. Within the gamespace, all players can move around freely after the game starts. Both teams start in a random location within predetermined start boxes. Goal of one round in the game is to eliminate all players of the opposing team. Eliminating is done by shooting opponents and reducing their health attribute to zero. Thus the players have:

- an actions for shooting their opponents
- an action or actions which give them the ability to move freely within the gamespace and rotate to look around
- a health attribute which resets every round

Possible outcomes of the game are wins for either team or a tie, if both teams are fully eliminated at the same time.

For the presented imaginary game, we implement an agent-based model to gather playtest data. At first we describe the model using the five component structure of every agent-based model (See section 7.1). Afterwards, section 7.2 explains the creation of the desired agent behavior using the aforementioned Unity ML-Agents toolkit (See section 5.3 for more detail). The last section (See section 7.3) validates the model, to confirm its correctness.

## 7.1 Implementation of the Agent-Based Model in Unity

As described in chapter 5, the prototype has been built using the Unity game engine. Within the Unity game engine a simple framework for an arena FPS has been developed. This framework only implements the five components explained in section 4.3. Each of these five components will described in more detail now.

In addition to describing the five components for our implementation, we will describe how each component could be implemented during the development of a real game. For the real game, we assume it to use the principles of *low coupling & high cohesion*.

By implementing the five components into a real game, we are able to use the ABM approach and gather data for balancing gamespaces in that real game.

### 7.1.1 Implementation of the Observer

The first component to explain is the observer controlling the main simulation loop. In general, one loop consists of one game round. It will already be possible to loop game rounds in a real game. Therefore, we are able to use the already implemented main game loop when using the ABM approach for a real game.

For the imaginary game however, a completely new main simulation loop has to be implemented. The main simulation loop always looks at one arena match. Within this match two teams play against each other in an environment until one of them wins. The win-condition in this case triggers, if one team has eliminated all the members of the opposing team. When both teams eliminate each other at the same time, we experience a tie. Additionally to the win-condition, every match has a maximum running time. If either the time for a match runs out, or an elimination-based end-condition is met, the observer stops the match and announces the result.

Afterwards, a new match is started by the observer. At the start of a new match, everything reset into its original state.

This concludes the main loop simulating players within the gamespace. Next we describe how the main loop is scheduled to be synchronous.

### 7.1.2 Implementation of the Scheduler

The scheduler must be synchronous due to the reasons that have already been described in section 4.3. Actions by agents must be handled at the same time and equally. Otherwise one agent might have an advantage over the other one. We assume that a commercial arena FPS game already handle actions synchronously. If it does, the ABM implementation for a real game can use the already implemented synchronous action handler.

Since we use the Unity game engine for the prototype, we are bound its script lifecycle [Tec21, Order of execution for event functions]. The Unity script lifecycle is not synchronous. Due to the lifecylce not being synchronous, implemented scripts in the prototype will be executed one after another. This could lead to the asynchronous scheduler problem described in section 4.3. We avoid the problem within the prototype by using the functionalities of the *MonoBehaviour.LateUpdate()* [Tec22, LateUpdate] and *MonoBehaviour.Update()* [Tec22, Update]. As the names suggest, the *LateUpdate*-function is executed after all *Update*-functions have been executed. We use the order of execution of the *LateUpdate*- and *Update*-functions to implement a synchronous execution of agent actions.

Instead of letting the agents themselves take actions while they are *updating*, they only *register actions* in the scheduler. The scheduler will collect all registered actions during the update-phase. After the update-phase ends and all agent-intended actions have been registered, the scheduler will execute the registered actions in the LateUpdate-function in a synchronous manner. For example, to synchronize the action of shooting, the scheduler will handle all registered shooting-actions following this scheme:

1. Go through all registered shooting-actions
2. For each shooting-actions, deal the registered damage of the action to the health attribute of the target-agent
3. Execute the shooting-action regardless of the attacking agents health. Thus the shooting will be executed even though the agent might have been killed by another agent at the same time
4. Only after every shooting-actions has been handled, check for agents that have been eliminated

This system implements a synchronous scheduler as it is required to correctly model human players in a gamespace. How the human players have been implemented as agents will be shown next.

### 7.1.3 Implementation of the Agent

While the observer is handling the main loop and the scheduler manages a synchronous update model, the agents acting on schedule during the main loop are implemented in the following way. The criteria listed in section 4.3 state that our agents require the same possibilities a human player would have in an arena FPS. Since we are creating a prototype, the agents will implement only basic features of an arena FPS. Therefore they must have at least these points implemented:

- systems allowing the agents to shoot at each other
- systems allowing the agents to navigate through a gamespace and rotate
- all the attributes required for the player character of the imaginary game such as movement, rotation speed, field of view radius and health

In the case of implementing the ABM approach for a real game, the required systems will be implemented already as they are necessary for players to even play the game. As the real game is assumed to use the principles of low coupling and high cohesion, it should be easy to give the control over a player character to an artificial agent.

Due to having no real game for the showcase implementation, the systems have to be implemented from the ground up.

The first system which allows agents to shoot at each other will be explained in the section for interactions (Section: 7.1.4). The shooting action is part of the sections for interactions because it is implemented to be a direct interaction between two agents rather than being an action agents can take.

Therefore the explanation of the shooter system is skipped for now and the first explanation is for the navigation system. Afterwards, a listing of all the agent attributes is presented.

**Implementation of the Navigation System**

Agent navigation through a gamespace was implemented into the prototype via the *NavMesh navigation system* [Tec21, Building a Navmesh] of the Unity game engine. NavMeshs in unity are an easy way to " [...] create characters which can navigate the game world" [Tec21, Navigation System in Unity]. Instead of using a system in which the agents navigate by giving directional inputs like "Move right", "Move back", they can just choose a goal location within the gamespace. After choosing a goal location, the Unity NavMesh system will automatically move the agent towards it without them getting stuck somewhere.

For the NavMesh system to work, we need to bake a NavMesh for a gamespace beforehand. The baking step calculates "[...] an approximation of the walkable surface" [Tec21, Building a Navmesh] while considering agent specifics like its radius, height, maximum steepness of ramps and maximum dropping distance. After baking the

NavMesh for a gamespace, the approximated walkable area looks like shown in figure 7.1. Everything that is shown in blue is walkable for the agents. Agents are also able to drop down from the ramp on the right hand side of figure 7.1. The ability to drop down is shown by the arrows directing from the top of the ramp down to the floor.

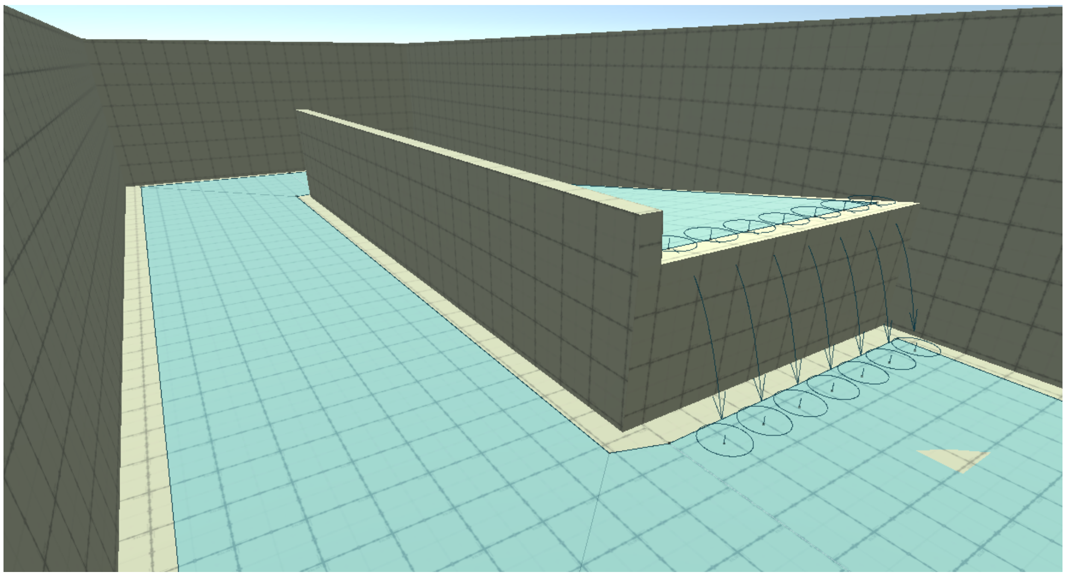


Figure 7.1: Depiction of the baked NavMesh in a gamespace.

Via the usage of the Unity NavMesh we were able to implement a sufficient and easy to use navigation system. Everything developers have to do, is to bake a NavMesh for the gamespace before they use the ABM model.

**Implementation of Agent Attributes**

In section 4.3 it was made mandatory for the agents to have the same attributes as human players. Therefore, all required attributes will be presented now.

The first attribute resembles the agents health. In an arena FPS with the goal is to eliminate all opponents, a health system is obviously required. A health system can be used to implement damage done after getting shot. It is a simple integer value which every agent has. The integer value gets reduced by the damage taken when the agent is shot. If the health attribute of an agent reaches zero, it dies. In the case of all agents of one team dying (or both teams in a tie), the game round is ended. When the observer resets the game to begin a new round, the health is reset to its initial value as well. A similar system will exist in every real arena FPS. Thus when using the agent-based approach during the development of a real game, one can just use the already existing health system.

Our agents must navigate through the gamespaces in the same way human players can. When developing a real game, a navigation system for player will be available to use for the agents too. Since no real game with an already existing navigation system is available, the agents use the Unity NavMesh for navigations. When using the Unity NavMesh system, the agents require the Unity NavMeshAgent component [Tec22, NavMeshAgent]. The Unity NavMeshAgent component offers lots of navigation specifics such as acceleration or speed. These navigation specifics were adjusted to emulate the navigation a human player has in a real arena FPS.

The last necessary attribute in an arena FPS is the field of view (FOV) value. The FOV defines the viewing angle of players. In real game development this is usually either fixed for all players or variable for players to choose. If the FOV is fixed for all players, the agents must have the same FOV to accurately gather play data. If the FOV is variable and players can choose it, the agents should use the greatest FOV option. A greater FOV value results in a bigger viewing angle which yields an advantage compared to using an smaller FOV. Therefore agents should use the greatest FOV option also available to players. Otherwise they might miss opportunities which arise by using the greates FOV option.

### 7.1.4 Implementation of the Interactions

As stated in section 4.3, our interactions need to resemble the interactions available in the game and gamespace we try to gather data for. In the section about agent-based models in general, we listed all the possible points of interactions. Since the implemented prototype is very simple and only supposed to show off the potential of the ABM approach, we do not implement any agent-self, environment-self, environment-environment or agent-environment interactions. However, we do implement an agent-agent interaction. The agent-agent interaction implemented, is a work-around to how shooting would be done within a conventional game. In conventional games, players can take the *action of shooting*. Rather than implementing shooting as an action agents can take, we implement it as a direct *interaction* between two agents.

#### Implementation of the Shooting System

The shooting system was implemented in a very simplified way compared to a shooting system in an arena FPS. "Shooter games [and therefore also arena FPS] present the majority of challenges as tests of the player's physical skills at hitting targets with projectiles [...]" [Ada14, Chapter: What Are Shooter Games?]. We disregard the " [...]

physical skills [of] hitting targets with projectiles [...]" [Ada14, Chapter: What Are Shooter Games?] for the implementation of our agents.

As soon as an agent has an opposing agent within their field of view, Unity *Ray*'s [Tec22, Ray] are sent towards the opposing agent using Unity *Physics.Raycast* [Tec22, Physics]. The rays origin is at the head of the shooting-agent, while the rays destinations varies. Destinations vary because the amount of rays is variable. The first ray always goes to the head of opposing agent. The head of the opposing agent is located at the horizontal middle and the highest vertical point of its body. All rays after the first one have the same horizontal position but are distributed equally along the vertical axis of the agents body. For example, if the total amount of rays is two, the second ray goes straight towards the vertical middle of the opposing agents body. Figure 7.2 demonstrates the situation of a blue attacking agent shooting the red opponent with a total number of three (red) rays. For three rays, the opponents body will be divided into three. The first ray hits the head, the second ray hits the body at 66% of the body's height and the last one hits the body at 33% of the body's height.

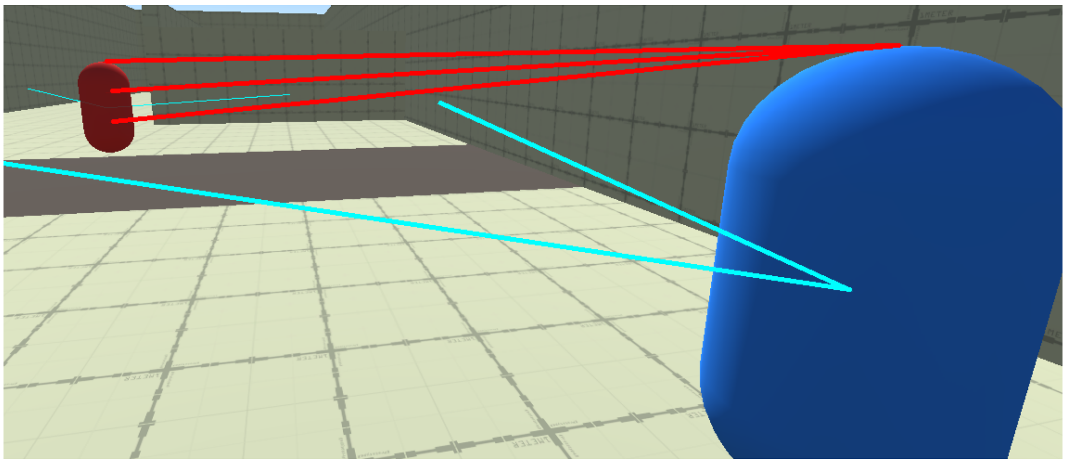


Figure 7.2: Demonstration of the vision system with three rays. The blue agent is shooting at the red agent depicted by the red lines. The light blue lines depict the field of view.

Figure 7.2 additionally depicts lines in light blue. These lines are the borders of the agents field of view. While being in the field of view, opposing agents count as in line of sight. Since the opposing agent counts as being in line of sight, rays will be sent. If the rays do not get interrupted by a line of sight breaking object, it means they hit their destination on the opposing agent. Depending on how many rays hit their destinations, the damage dealt by the shooting action changes. For each ray hitting, one damage is dealt to the opponent. Thus, in figure 7.2, the blue agent would deal three damage to the red agent. The damage is dealt every update cycle for as long as the red agent stays within the field of view of the blue agent.

Other types of interactions are not modeled as we implement a very simple prototype.

### 7.1.5 Implementation of the Environment

The last missing component is the environment. In section 4.3 we defined the environment to be the functional space of a gamespace within the game we try to model and gather data for.

During the development of a real game, we would want to evaluate the environments created by the level designers. After an evaluation the level designers then would be able to interpret the gathered data and iterate on their level design.

Level designers thus create lots of environment prototypes through iteration. These prototypes are most commonly just the functional space in the beginning. It is not worth it to go further than just the functional space and make the prototypes aesthetic. Making prototypes aesthetic is just extra work since it would have to be done for every iteration until finishing with a fun and balanced gamespace. To find out if a gamespace is fun and balanced, all we need is the functional space. Since the functional space is everything the designer create anyway, no extra work is necessary when using the ABM approach on a real game.

However, for the implementation of the prototype we do not have a real game in development. Therefore we have to create the gamespaces ourselves. To create gamespaces we use the tool *Unity ProBuilder* which was described in section 5.2. After creation, we can then bake the Unity NavMesh to make agent navigation possible as described earlier in section 7.1.3.

Another part of the environment are the start box positions for both teams which need to be set too. Luckily when level designers built a prototype, they determine the start positions as well. Otherwise the built gamespace would not be playable. In our scenario however, we need to define the start box locations ourselves too as we also had to create the gamespaces ourselves. Figure 7.3 shows an example for a gamespace. Within the gamespace two starting boxes in red and yellow are placed for the two teams.

These five components build up a framework for the proposed agent-based model. To complete the framework, the following section will explain the basics for creating the agent behavior.

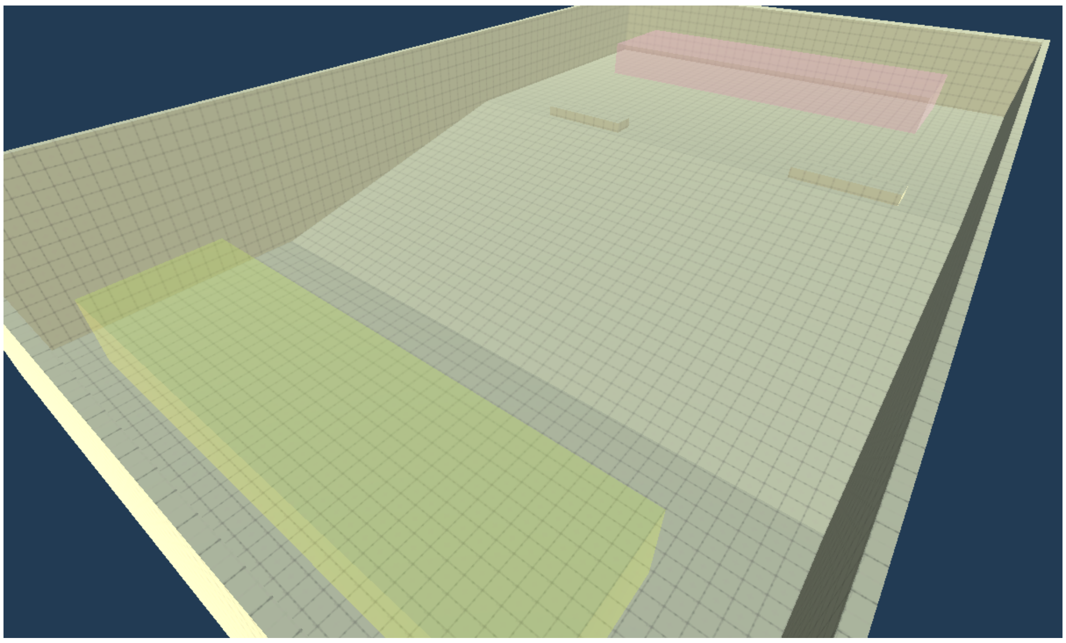


Figure 7.3: Depiction of the starting boxes for both teams. Team red starts somewhere in the red box while team blue starts somewhere in the yellow box.

## 7.2 Creation of the Agent Behavior using the Unity ML-Agents Toolkit

In section 4.3, a list of criteria was presented. The list of criteria includes everything an agent has to satisfy for the ABM approach. In the last section (See section 7.1), the base of the ABM approach was explained. However, the base of the implementation does not satisfy all the criteria for an agent. Within the base explained until now, agents satisfy the criteria of having the same attributes and actions as human players.

Missing from the list of criteria are these:

- the agents have to have the same goal as human players
- the agents must be adaptive as human players also adapt to new gamespaces or enemies in the arena FPS
- the agent should behave much like a human player would do in a gamespace

We seek to solve these three remaining criteria using machine learning. By using the machine learning toolkit *Unity ML-Agents* [Jul+18] (See section 5.3 for more detail), the desired behavior for the agents is created.

### 7.2.1 Usage of the Unity ML-Agents framework

To achieve a human-like, adaptive and goal driven behavior in the agents, machine learning was used. Since the Unity game engine was used for the showcase project, machine learning could be implemented via the Unity ML-Agent toolkit. The toolkit provides the possibility to "[u]se state-of-the-art machine learning to create intelligent character behaviors in any Unity environment" [Tec21, Unity Manual: Verified packages: ML Agents]. Therefore, the base for the agent-based model was first implemented in the Unity game engine as described in the last section (Section: 7.1. Afterwards, the desired agent behavior was implemented using the framework of the Unity ML-Agent toolkit. The framework offers a number of different functions and features to use. How the most crucial functions were used, will be explained now.

#### Implementing a custom Unity ML-Agent

The Unity ML-Agents toolkit comes with comes with a public class called "agent". This class is the base for a custom Unity ML-Agent. To create a custom Unity ML-Agent, a child class is created by inheriting from the agent class. In the child class we implement custom behavior by overriding certain functions provided by the agent class. By overwriting the following functions, a arena FPS agent was created:

- *OnEpisodeBegin*: The first function to be overwritten is the OnEpisdodeBegin-function. An *Episode* is one round of training for an agent. The training of an agent is done by completing many episodes one after another. An Episode begins with OnEpisodeBegin and ends with a call of the function *EndEpisode*. For teams consisting of more than one agent, the episode is ended with *EndGroupEpisode*. For the arena FPS agent, one episodes resembles one round of the imaginary arena FPS. One round of the imaginary arena FPS starts with the players (or agents) in their start boxes. If one or both teams are eliminated, the round is ended by the obsverver. The observer then officially ends the episode with either EndEpisode or EndGroupEpisode. Both EndEpisode and EndGroupEpisode make calls to OnEpisodeBegin for every agent participating. Therefore the observer keeps the training loop running by repeatedly restarting episodes with calls to end an episode which then again call OnEpisodeBegin.

  The OnEpisodeBegin function itself is responsible for resetting an agent. Therefore if the function is called for an agent, all changing attributes are reset. An example for a changing attribute is the agents health.

  Two more changing attribute are the position and rotation of the agent. The rotation around the agents y-axis is randomized to a value in the range of [0;360] degrees while the position is changed to a random position within the agents start box.

- *CollectObservations*: The next important function to be overwritten for implementing custom behavior is *CollectObservations*. It is called by the Unity ML-Agents toolkit itself and as the name suggests, the function exists to provide the agent with the observations required for the desired behavior. An example for a desired behavior could be an agent that is supposed to move from its current position to a goal location in a straight line. For such a desired behavior, the only observations required, are the position of the goal and the agents own position. In general, more complex observations are required to achieve more complex behaviors. Extending the former example by adding a hole to avoid for the agent, the former list of observations is not sufficient anymore. It is not possible for the agent to avoid the hole, if the hole position is not known. To achieve the more complex behavior of dodging a hole, additional observations are required.

  To model the complex behaviors that human players exhibit in gamespaces, these observations are given to the arena FPS agent:

  - *Information about the agent itself*: This includes its xyz-position as a Vector of length three and the rotation around its y-axis.
  - *Information about the opponents of the agent*: For every opponent which is still alive in the round, the agent is given these observations:
    - The first opponent-observation is an integer value resembling *how visible the opponent is for the agent*. If the opponent is not visible at all, the observation is zero. Otherwise it is the count of vision rays that hit the opponent (See section 7.1.4).
    - Afterwards, the same calculations are done again but in reversed directions. Doing the calculations reversed provides the agent with an observation that indicates *how visible the agent is for the opponent*.

      Both the observations of how visible the agent is for the opponent, and how visible the opponent is for the agent, are required to identify advantageous positions. In general, a position is advantageous for a player, if they are less visible to their opponent than the opponent is visible to them. Or in other words, if a player is more visible, they are more vulnerable.

This also translates to the implemented shooting system. In the shooting system, a position in which the agent always hits three rays while the opponent only hits two rays is advantageous to the agent. Three hits deal three damage while two hits only deal two damage (See section 7.1.4). Therefore the opponent would be eliminated first, as long as it can not turn the situation in its favor.

∗ As advantageous positions often come from having a higher position than the opponent, the agent also observes the *signed difference in height to its opponents*.

∗ The last observation is the *angle difference towards an opponent*. It describes the amount of degrees an agent has to turn, to look straight at the opponent. The angle difference is within in the range [-179; 180]. If the agent looks straight at an opponent, the angle difference to that opponent is 0 degrees. Meanwhile, when looking in the exact opposite direction, it is 180 degrees. The angle difference is signed to be able to distinguish between looking to the left hand side or right hand side. To calculate the angle difference, the Unity *Vector3.SignedAngle(Vector3 from, Vector3 to, Vector3 axis)* function is used. [Tec22, Vector3]. As inputs we used the agents own Unity *Transform.forward* vector [Tec22, Transform] as the from-input, a direction vector from the agent to its opponent as the to-input and a Untiy *Vector3.Up* [Tec22, Vector3] vector as axis-input.

While it is possible to add more observations regarding the opponents, the environment or the agent itself, we ultimately chose to restrain us from adding more observations. Adding more observations would in return add more complexity to the agent. A more complex agent is more difficult to train which we could not achieve within this paper. However, we experimented with other observations that might be useful to achieve more complex and human-like behaviors:

– A *cover score* system which evaluates the current location of the agent. Locations could be evaluated in terms of how visible the agent is in them. Using this information, agents could be able to find advantageous positions within the gamespace.

– *Predetermined locations of advantageous positions* such as cover or highgrounds.

  Compared to using the cover score system, the addition of predetermined locations to the observations might decrease the required amount of time for an agent to find advantageous positions.

– *More information in general* about the agent and their opponents. This general information could include their health attributes or the angle difference an opponent has towards the agent. In some situations in which the agent is at a health disadvantage, it might useful to know whether the opponent is looking in the agents direction or not. If the opponent is not looking in the direction of the agent, the agent might decide to not engage in combat but rather wait for a more optimal situation.

  - In the case of a more complex game with special features such as health / weapon pick-ups or jump-pads, it might be useful to include the *locations of these features* into the list of observations as well. When knowing the locations of such features, agents might be able to form strategies around these key locations. As an example, it might be useful to know the location of the nearest health pick-up, in case the agent or its opponent are on a low health level.
  - *Information about previous deaths* of the agent such as its position, own rotation or angle difference towards the killing opponent. By using death information, agents might be able to prepare and prevent the same death from happening again.

  It is also crucial to reduce complexity whenever it is possible as "the presence of irrelevant attributes should considerably slow the rate of learning" [BL97, Section 2]. For example, if a gamespace has no differences in height and is practically a flat space, it is possible to omit the agents own height from the position observation. The agents own height does not change in a flat space and is therefore irrelevant. Omitting the height observation, would reduce the position observation from a xyz-vector of length three to a xz-vector of length two. Thus, we decrease the number of observations which might increase the training speed.

- *OnActionReceived*: The last function to overwrite is the *OnActionReceived* function. Like the CollectObservations-function, the OnActionReceived function is called by the Unity ML-Agent toolkit itself.

  The input parameter for the CollectObservations-function is a Unity ActionBuffer [Tec22, ActionBuffers]. By using the given ActionBuffer, it is possible to read the actions that the machine learning agent wants to take. Actions an agent might take are either discrete or continuous. How many continuous and discrete actions are taken by the agent is part of the agent creation process in the Unity game engine. While agents always choose a value in the range [-1;1] for a continuous action, for discrete actions it is required to additionally set a range in which the agent chooses the action.

  For example, a discrete move action could have a range of three. By choosing a number from the range [0;2], the agent then takes an action. The action the agent chose then can be read from the ActionBuffer in the OnActionReceived-function. As the action is just a integer number, a mapping is required. For the discrete move action, a mapping of number to action could look like this:

– 0 -> don't move
– 1 -> move left
– 2 -> move right

To ensure recognizable patterns for the agents to learn and use, the mappings of chosen numbers to actions must be consistent during the learning process. However, the initial mapping from numbers to actions does not matter.

For the agent-based model of the imaginary game, the agent implements only three actions. Since shooting is done automatically and based on line of sight (See section 7.1.4), it is not implemented as an action agents can actively take. The actions agents can actively take are

– a continuous move action.
– and a second discrete action for rotation.

The move action consists of two separate continuous actions that form a vector of length two. The formed vector is used as destination for the Unity NavMesh of the gamespace.

The formed vector is not usable for the destination input of the Unity NavMesh system from the get go. NavMeshs in Unity require a vector of length three consisting of the world coordinates on the Unity NavMesh of the gamespace [Tec22, NavMeshAgent.SetDestination]. Meanwhile, the formed vector is of length two and consists of two values in the range [-1;1]. Therefore the vector provided by the agent is mapped onto a world position within the gamespace. Figure 7.4 shows the mapping for a completely flat gamespace. To reduce the complexity of the agent, the height in the move action was omitted. Instead, a projection of the 2D vector onto the real world position is done disregarding the height in the gamespace. Disregarding the height is sufficient as long as no second floor exists in the gamespace.

To look around, the agent needs to be able to rotate around its y-axis. The action implementing this rotation is discrete. The discrete action has a range of three. One input maps to no rotation at all while the other two inputs map to rotating either left or right. Rotations always happen with the same rotation-speed.

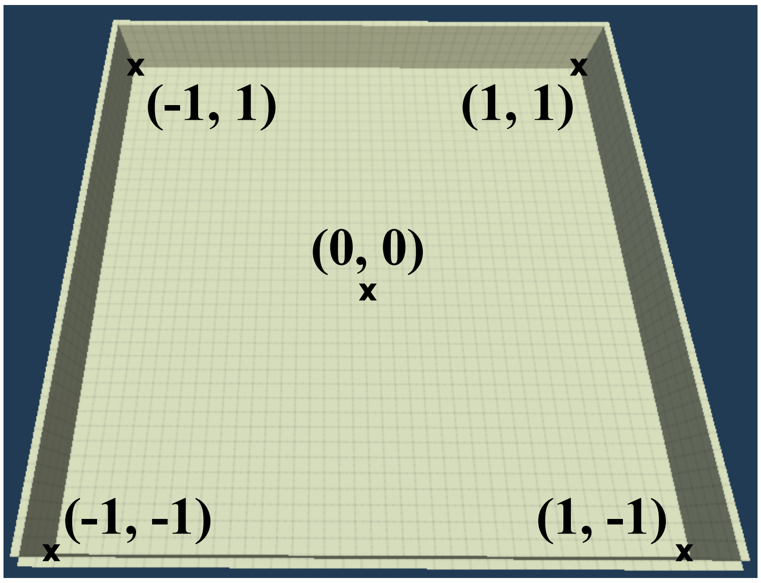


Figure 7.4: Depiction of the mapping from the 2D, [-1,1], space to a 3D gamespace

These overwritten functions implement the custom behavior specifically of our agent. To implement an agent for an arbitrary real game, these functions must be overwritten according to the game. When a round within the real game ends, both players and agents should be reset equally. Observations and Actions of both players and agents should also be the same or at least work in an equal manner. Additionally, the observations should be normalized to ensure a more effective training.

### 7.2.2 Normalization process for the Unity ML-Agents

The implementation uses the Unity ML-Agents toolkit to create machine learning agents based on neural networks. When working with neural networks "[...] it is considered a best practice [to normalize observations]" [Jul+18, github: Learning-Environment-Design-Agents.md]. Normalizing observations can help the neural network to "[...] converge to a solution faster" [Jul+18, github: Learning-Environment-Design-Agents.md].

The process of normalization is different for every model and thus also for every implementation of the proposed ABM approach. In the last section (See section 7.2.1), the observations of the ABM implementation for the imaginary game (See chapter 7) were described. How the observations are normalized will be presented now.

### Normalization of the Agent Rotation

The first observations include the agents xyz-position and the rotation around its y-axis. Regarding the rotation, it is read as a float value in the range of [0;360]. To normalize it into a range of [0;1], we simply divide the value by the float value of 360.

### Normalization of the Agent Position

Normalizing the xyz-position is more difficult. In section 7.2.1 it was required to transform the agent move action from a 2D-[-1;1] range into 3D world coordinates applicable for the Unity NavMesh. To normalize the position we now have to do the opposite by transforming a 3D world coordinate to a 3D-vector with each component normalized.

Beginning with the y-component resembling the agents height within the gamespace. The y-component is normalized into a [0;1] range. To transform an arbitrary value into a [0;1] range, we use the following formula [Jul+18, github: Learning-Environment-Design-Agents.md]:

$$normalizedValue = (currentValue - minValue)/(maxValue - minValue) \tag{7.1}$$

The *currentValue* is the float value of the y-component. We relate the y-component to the gamespace in which the agent is permanently contained in. Therefore we can use the upper and lower bounds of the gamespace as *max- and minValues*. We get them by using the Unity *Collider.bounds.max.y* and *Collider.bounds.min.y* [Tec22, Collider] [Tec22, Bounds] which are from the Unity *Mesh Collider component reference* [Tec21, Mesh Collider component reference].

The remaining x- and z-component of the 3D world position are normalized into a [-1;1] range. Figure 7.4 depicts the mapping from 2D coordinates in [-1;1] range to a 3D gamespace. For the normalization the reverse mapping is done. It maps from a position in the 3D gamespace to a 2D coordinate in [-1;1] range by first calculating the agent position relative to the gamespace center:

$$relativePosition = agentWorldPosition - gamespaceWorldCenter \tag{7.2}$$

in which all variables are vectors of length three and the *gamespaceWorldCenter* is retrieved via *Collider.bounds.center* [Tec22, Bounds] of the same bounds used for the y-component.

Having the relativePosition, it is possible to calculate the normalized x- and z-components via:

$$normalizedX = relativePosition.x/gamespace.extents.x \tag{7.3}$$

and

$$normalizedZ = relativePosition.z/gamespace.extents.z \tag{7.4}$$

in which the *gamespace.extents* are retrieved via *Collider.bounds.extents* [Tec22, Bounds]. The extents have a value of half the size of the gamespace in the corresponding component direction.

By combining all three components we receive the normalized agent position in relation to its gamespace.

### Normalization of the Angle Difference towards Opponents

The next observation to be normalized is the angle difference towards the opponents. As the angle difference is in the range [-179;180], we only need to divide the calculated angle difference by 180 to normalize it into the range of [-1;1].

### Normalization of the Height Difference towards Opponents

To normalize the height difference an agent has to its opponent, we first take the normal difference of their height and afterwards divide it by the maximum distance in height they can be apart of within the gamespace:

$$normalizedHeightDifference = \frac{(agentPosition.y - enemyPosition.y)}{(gamesspaceMaxHeight - gamespaceMinHeight)} \tag{7.5}$$

in which the *gamespaceMax- and MinHeight* are the same *Collider.bounds.max.y* and *Collider.bounds.min.y* used before (See section 7.2.2). The result is a normalization into the range of [-1; 1].

### Normalization of the Vision Counts

The remaining observations are the vision counts both from the opponent towards the agent and vice versa. Both vision counts depend on the number of rays which hit. While the maximum amount of rays that can hit is just the total amount of rays we shoot, the smallest amount of rays that can hit is when all miss, hence zero rays. Using the same normalization equation used in section 7.1, the following equation results:

$$normalizedVisionCount = amountOfHitRay/amountOfRaysShot \tag{7.6}$$

Doing the normalization with this equation produces a normalized vision count in the range of [0;1].

These normalizations help to train the model more effective. However, effective training of a false model is really not effective anymore. Therefore the next chapter tries to validate the implemented model.

## 7.3 Validation of the Agent-Based Model

*"Validation is the process of ensuring that there is a correspondence between the implemented model and reality. Validation, by its nature, is complex, multilevel, and relative. Models are simplifications of reality; it is impossible for a model to exhibit all of the same characteristics and patterns that exist in reality. When creating a model we want to incorporate the aspects of reality that are pertinent to our questions. Thus, when undertaking the validation process, it is important to keep the conceptual model questions in mind and validate aspects of the model that relate to these questions."* [WR15, p. 325f]

Keeping the goal in mind of gathering data for the practical task of balancing gamespaces, it might not be necessary to have an implementation which is completely accurate. An implementation can still provide useful and meaningful data even though it might not perfectly represent the real world analog

In general, the real world analog would be a real game. However, this implementation is built for an imaginary game. Because of this, when validating, we instead use common implementation techniques for certain problems as real world analogs for our implementation.

Although the implementation does not perfectly represent its real world analog, the following section does try to validate most of its components using the four techniques for validation presented in section 4.4.2.

### 7.3.1 Microvalidation of the Agent-Based Model

The first of the four techniques is microvalidation. To recapitulate section 4.4.2, microvalidation tries to show matches between the agents and their real world analogs in terms of mechanics and behaviors [WR15, p. 326].

#### Matching the agent mechanics

Concerning the mechanics, it is easy to find matches between the implementation and its real world analog. One way of finding matches is by examining relevant properties of the real world analog and to look for similarities afterwards [WR15, p. 329]. The process of examining the relevant properties to look out for similarities might be more complicated when using ABM for real world phenomena like the flocking of birds. However, as the ABM approach is used during the development phase, direct access is available to all properties of players within the game. Therefore the process of matching properties of players and agents is simple. The properties of a player in the game are the possible actions and attributes.

Matching the attributes of an agent and a player within a game has already been described in section 7.1.3. In a best case scenario both the agents and real players just use the same system for their attributes. If that is not the case, it is a simple act to implement the same attributes for an agent, a player has. In either case, the attributes are microvalid.

Our showcase implements the agent navigation using the Unity NavMesh sytem [Tec21, Building a NavMesh] (See section 7.1.3 for further detail). The NavMesh system allows agents to move freely within a gamespace. Comparing the free navigation of the agents in the showcase implementation to common navigation methods in shooter games like Counter-Strike: Global Offensive [12] or Overwatch [16], a clear match is obvious. Both common navigation implementations in commercial FPS and the showcase implementation navigation allow for free movement within the gamespace. Therefore, the implemented navigation system is microvalid.

The second action to match is the shooting action. As section 7.1.4 describes, the shooting action is implemented as a line of sight interaction rather than a test of the aiming skill. For the line of sight shooting interaction to be microvalid, it must match shooting with aim. However, as we mentioned earlier in the section, it is sufficient to have a system which provides results that give insight about the balance of the gamespace. The following section tries to prove that line of sight is sufficient for this paper's cause.

#### Matching the shooting system

Instead of implementing the shooting *action* of an agent as the " [...] physical skills [of] hitting targets with projectiles [...]" [Ada14, Chapter: What Are Shooter Games?], the agents implement it as a mere *interaction* between two agents (See section 7.1.4 for more details). Within the interaction we only test for line of sight to the opposing agent. When having an opposing agent within line of sight, agents automatically shoot, hit and damage the opponent rather than taking an *action of shooting* and trying to hit. The simplified line of sight shooting system is sufficient for generating data about the balance of a gamespace. Human players within a gamespace will sometimes lose duels against other human players even if the gamespace gives them a severe advantage. Advantages given by the gamespace sometimes do not matter in human vs. human duels due to shooting being a physical skill. Therefore during duels of human players there is always a little bit of chance involved. This chance tarnishes data regarding the actual balance of a gamespace. To omit this chance, we simplify the act of shooting to a line of sight based shooting. Line of sight based shooting simulates a "perfect" player in terms of aiming capabilities. Having a "perfect" player provides better insight into the balance of the gamespace as there is no chance or skill involved. Without chance or skill involvement, a better position should always provide a better result.

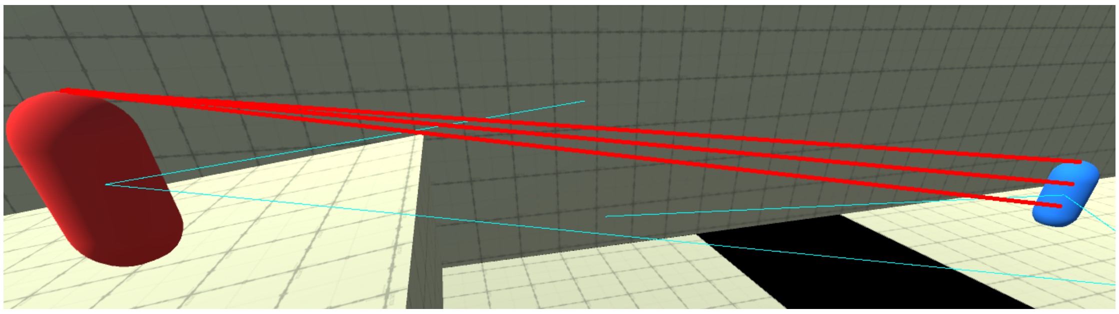
Section 7.1.4 also describes the damage mechanic. For every ray hit by the line of sight shooting, the targeted opponent receives one damage. This approach implements a basic rule of: "The more visible you are, the more vulnerable you are" which is the key rule for winning a duel outside of player skill. The rule allows the agent-based model prototype to simulate the effectiveness of positional advantages in the gamespace. One example for a positional advantage is depicted in figure 7.5. It shows the red agent in a higher position. A higher position covers up the lower body of the red agent. Therefore only two rays from the blue agent hit while in total three rays were sent. The red agent however, hits all rays. Thus it will deal more damage to the blue agent than the blue agent deals to the red one. If both agents do not move anymore, the red agent will ultimately win the duel purely because of the advantage given by the gamespace. The explanation should prove that an aim based shooting system is not mandatory to gather data about the balance of a gamespace.

However, just matching the implemented mechanics to the real world analog is not enough as they also have to be used just like in their real world analog.

**Matching the agent behaviors**

The real world analog to the agent behavior would be perfect human behavior within a gamespace. To come as close as possible to this, the implementation uses machine learning. By using machine learning, we managed to train the agents to behave like humans on the flat gamespace shown in figure 7.4. Withing the gamespace as environment, the agents were trained using the reinforcement learning made available by the Unity *ML-Agents* toolkit [Jul+18] (See section 5.3 for more detail).

The training itself was conducted in two stages. Goal of the first stage was to only train the "aiming" of the agents while the second stage combined "aiming" with movement. Since shooting is based of line of sight, the "aiming" task for the agent


(a) Red agent shooting blue agent

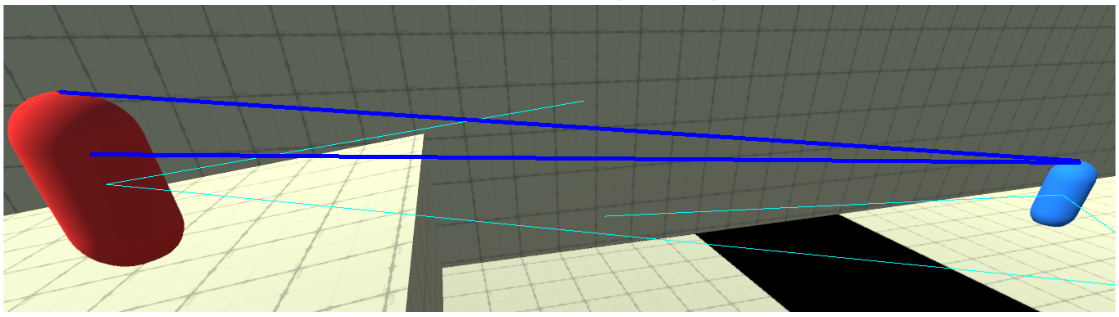

(b) Blue agent shooting red agent

Figure 7.5: Demonstration of positional advantage of the red agent.

consists of rotating towards their opponent. To teach the agents to rotate towards their opponent, they started each episode in a random location within the flat gamespace. At that random location they were randomly rotated as well. Since the goal was to only teach the rotation, they could only rotate and were not allowed to move yet.

Reinforcement learning works by giving out rewards to the agents. The agents then try to maximize their cumulative rewards. Therefore, during training, two positive rewards were given to the agents. While the first reward was given for just keeping an opponent within their line of sight, the second, bigger reward was given for eliminating an opponent. On the other hand, negative rewards or penalties were given for receiving damage and getting eliminated. Agents receive the maximum cumulative reward, if they manage to eliminate their opponent without getting hit themselves.

Additionally, an *existential penalty* was given to each agent every time they updated themselves. A small existential penalty reduces the cumulative reward for the agents over time.

This reward and penalty distribution encouraged the agents to gain rewards by rotating towards each other. Moreover, due to the existential penalty, they were encouraged to rotate towards each other as fast as possible. They tried to go as fast as possible because a maximum cumulative reward exists. The maximum cumulative reward diminishes for every update cycle they took. Thus, to minimize the loss and maximize the cumulative reward, they start to rotate towards each other as soon as an episode begins.

Rotating towards each other translates to "shooting" at each other in the line of sight shooting system. As soon as they optimized the shooting, the second stage was started.

Goal of the second stage was to teach the agents shooting while moving. Just like in stage one, they started in random locations within the flat gamespace with random rotation. The reward system did not change either. The only difference to stage one was that they now were allowed to move.

The two stage training yielded an agent behavior much like you would expect from humans players in the same gamespace. While no obvious movement patterns emerged, they learned to shoot each other properly. Figure 7.6 depicts how the trained agents choose to rotate. They always choose the rotation direction that is the shortest as they have been trained to aim fast.

The trained aiming of our agents behavior matches the expected aiming behavior from human players in a flat gamespace. As the winning condition for two human players within a flat gamespace is to eliminate their opponent before being eliminated themselves, they too try to shoot the opponent as fast as possible. Within this scenario

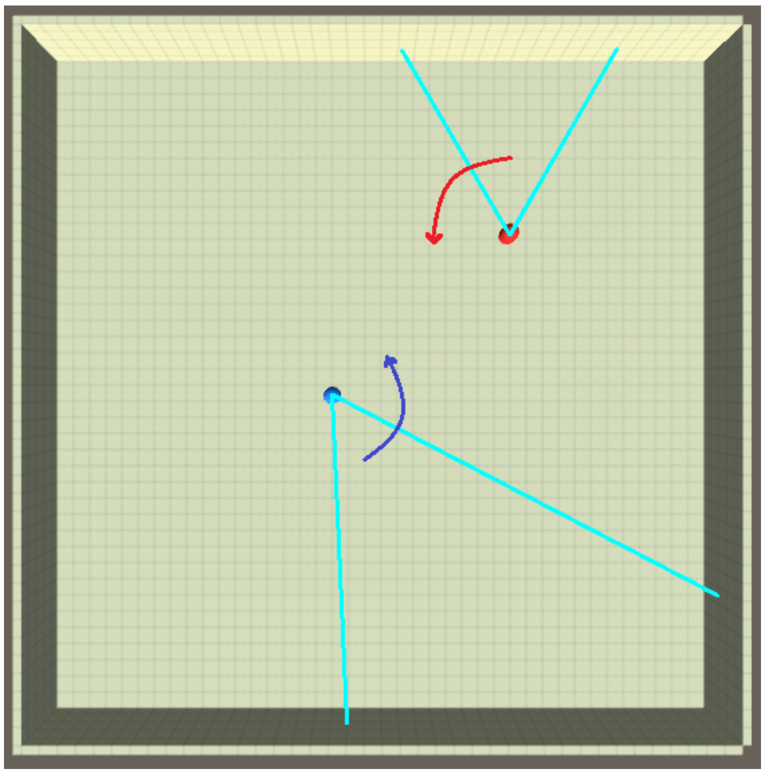


Figure 7.6: Depicts in which direction the trained agents choose to rotate with the colored arrows. The rotation direction is the shorter rotation towards their opponent.

our trained agents behavior is therefore microvalid.

Besides the basic aiming behavior, we managed to train an agent to match another human like behavior. Figure 7.7 depicts the gamespace for the training. Goal of the training was to teach an agent how to use a ramp as advantageous position. The training was conducted in two stages again.

Stage one of the two only aimed to teach the agent to use the ramp as a highground. To train this behavior, the blue agent was replaced by a stationary bot which did not rotate but had the same amount of health as the red agent. In the beginning of every training episode, the red agent started inside the area marked in blue in figure 7.7. We kept the same reward system that was used in the flat-space-aim-training and also initialized the red agent with the knowledge gained in the training. This way, the red

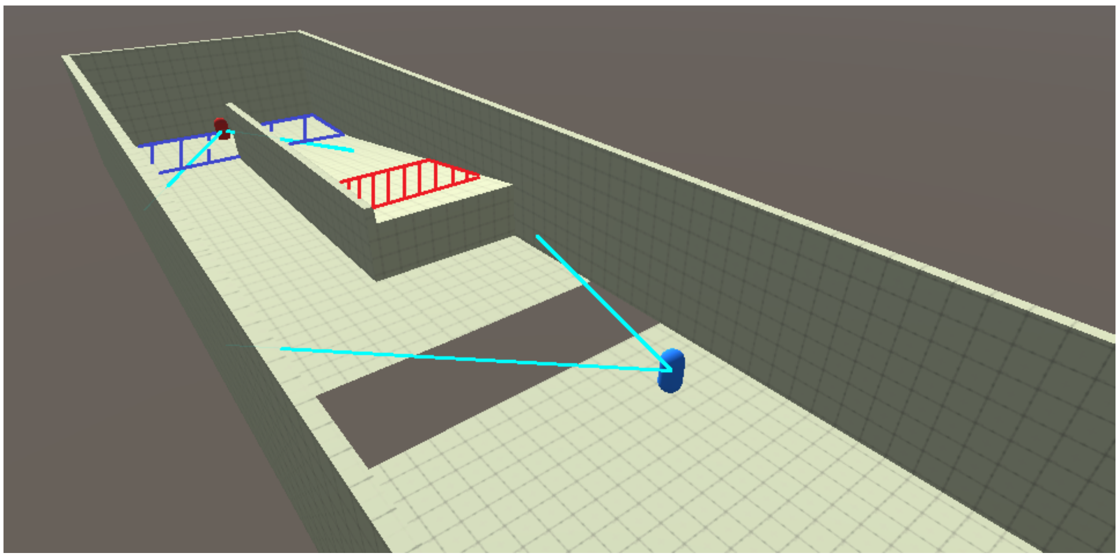


Figure 7.7: Depiction of the gamespace designed to train an agent to choose the high-ground over the lowground. The area marked in red roughly shows the area providing an advantageous position. The blue area marks the starting area for the red agent

agent knew how to aim right from the beginning

During the training, the agent quickly learned that walking up the ramp provides an advantage. Figure 7.5 shows why using the ramp is advantageous. The advantage comes from the small area roughly shown in figure 7.7. Within the small area, the red agent has more vision on the blue agent than the blue agent has on red. Due to the red agent having more vision, it deals more damage and ultimately wins the duel. It wins the duel because it builds up an health advantage towards the blue agent while being in the small area.

However, the agent did not use the ramp as one would expect from a human. Since both the blue and red agent had the same amount of health, it was sufficient for the red agent to not perfectly use the ramp as an advantageous position. Instead of using the advantageous position roughly marked in figure 7.7, it kept on moving towards the blue agent and eventually dropped of the ramp (See figure 7.1 for the drop-off point). This kind of behavior does not match the real world analog.

Stage two tried to deal with this mismatching behavior. To teach the agent how to properly use the ramp as an advantageous behavior, we gradually increased the health of its blue opponent. Gradually increasing the opponents health removed the health advantage of the red agent. This made it mandatory for the red agent to use the ramp as advantageous position. Otherwise the blue agent would win the duel due to having a health advantage.

After the training, our agent always walked up the ramp and roughly stayed in the area marked red in figure 7.7. This is the expected behavior from human players in this situation as well. As using the ramp as cover, gives them a higher chance to win a duel. Therefore this taught behavior is also microvalid.

### 7.3.2 Macrovalidation of the Agent-Based Model

The second validation technique is macrovalidation. Recapulating section 4.4.2, "[m]acrovalidation is the process of ensuring that the aggregate, emergent properties of the model correspond to aggregate properties in the real world" [WR15, p. 326].

This means that the implemented showcase model corresponds to what we see in commercial arena FPS. The following section presents a few scenarios and compares them to what we would expect in commercial arena FPS. If more of the presented scenarios correspond to their expected result, it further macrovalidate our model implementation.

#### Scenario 1: Ties in the model

At first, we present a scenario to validate the implemented scheduler (See section 7.1.2 for more detail). Section 4.3 demands a synchronous scheduler in which no agent has an advantage due to being updated first. We expect the same to be the case in commercial arena FPS as they would otherwise be unfair.

Figure 7.8 depicts the setup trying to macrovalidate the scheduler. In the scenario two opposing agents are fixed in position and rotation so that they immediately shoot each other. In a commercial arena FPS this scenario is to be expected to end in a tie. If it does not end in a tie, the game would be unfair.

The result of this scenario in the model implementation always ends in a tie and therefore macrovalidates this scenario.

#### Scenario 2: Duels in flat gamespace

Scenario 2 tries to macrovalidate a situation in which two opposing players play in a flat gamespace. Their goal is to eliminate the opponent first. Starting locations as well as rotations are random for the agents. Additionally, they have the same amount of health and use the same neural network which makes them equally skilled.

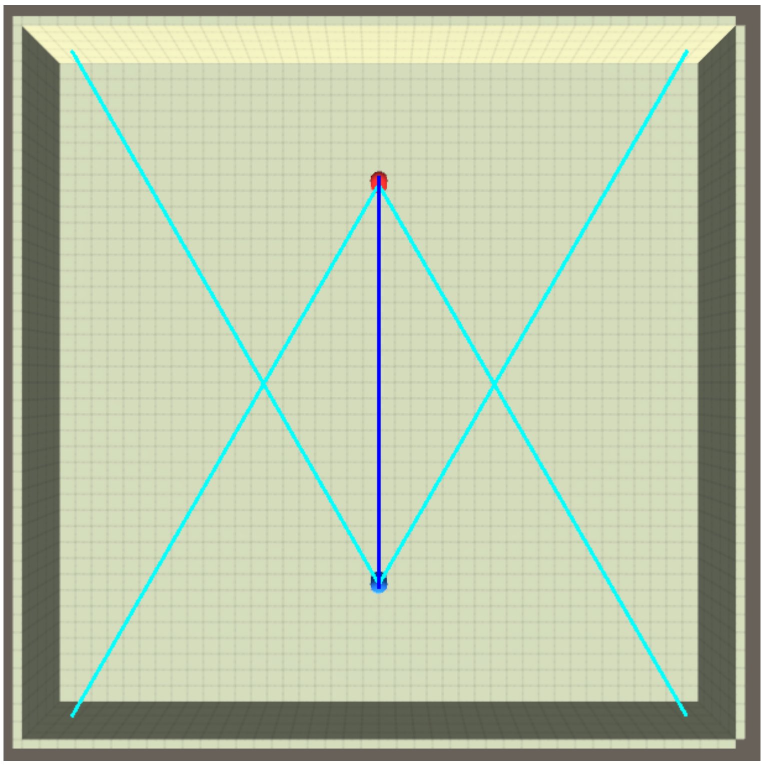


Figure 7.8: Depiction of scenario 1 (from 7.3.2) in which a tie is forced

How the agents were trained for this scenario is described in section 7.3.1. Having human players *of equal skill* in such a scenario in a commercial arena FPS, we would expect them both to roughly have a 50% win-rate.

The 50% win-rate is also achieved by the model implementation. During 90,489 episodes conducted in total, 4,499 episodes ended in a tie, 42,913 episodes were won by the red agent and the remaining 43,077 episodes were won by the blue agent. Both agents come close to a 50% win-rate. The red agent has a win-rate of 47.42%, while the blue agent has a win-rate of 47.60%.

Our model implementation comes close to the expected 50% win-rate for both agents which further macrovalidates it.

**Scenario 3: Highground advantage**

The last scenario is supposed to validate that having a highground is actually an advantage for the implemented agents. Therefore we set up an unfair gamespace which is depicted in figure 7.9. In this scenario, both agents have the same amount of

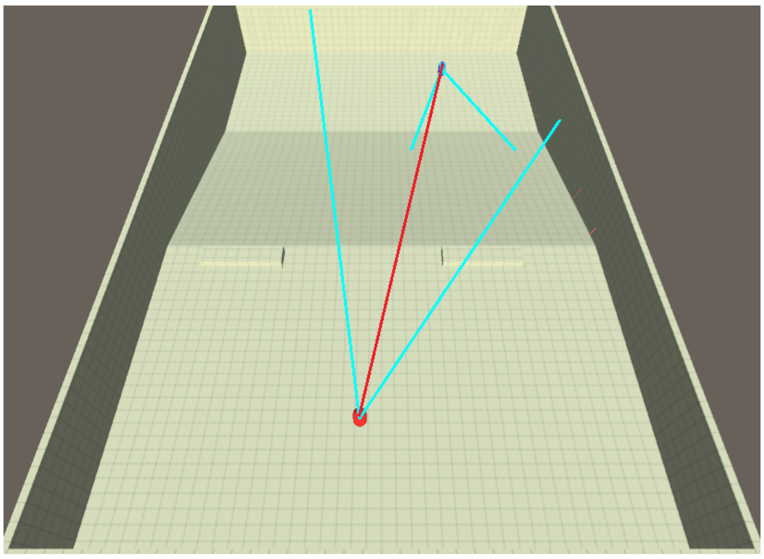


Figure 7.9: Depiction of scenario 3 (from 7.3.2) in which a unfair highground scenario is forced

health and use the same neural network. However, the blue agent always starts on the bottom, while the red agent starts on the highground. Having a highround gives you an advantageous position in general. Therefore it is to be expected that a human player in the situation of the red agent wins more often against another human player (of equal skill) in the situation of the blue agent. This expected outcome is also resembled by the model implementation.

Out of 92,463 total games played

- one game was not decided due to a time out,
- 1,203 games ended in a tie,
- 15,144 games were won by the blue agent (16.38% win-rate)
- and the remaining 76,115 games were won by the red agent (82.32% win-rate).

The results match the expected result of the same scenario with human players which further macrovalidates the implemented model.

### 7.3.3 Face validation of the Agent-Based Model

Via face validation we "ensure that someone who looks at the model “ on face ” (i.e., without detailed analysis) can easily be convinced that the model contains elements and components that correspond to agents and mechanisms that exist in the real world" [WR15, p. 332].

When looking at the figures in this paper, the model implementation is easily recognizable as a arena FPS. The model implementation has an arena (the gamespace) and two opponents fighting each other within the arena. A game is won by eliminating the opponent. Those are all features of an arena FPS.

Moreover, face validation rules out any unreasonable behaviors. As the trained behaviors are microvalid and match with the real world, they are also face valid.

### 7.3.4 Empirical Validation of the Agent-Based Model

The last technique is Empirical validation. Empirical validation expects the implemented model to generate similar data to the real world. The section for macrovalidation (See section 7.3.2) has already shown that the implemented model generates similar data in terms of the win-rate. Therefore in terms of win-rate, the implemented model is empirically validated.

This concludes the validation of the implemented model. We have shown that in many points our model comes close to a real arena FPS played by human players. However, the following chapter (See chapter 8) will discuss certain topics regarding the validation and other points of this paper's work.

# 8 Discussion

This paper aimed to introduced agent-based modeling to use it as a tool for gathering playtest data regarding the balance of gamespaces. Additionally, we implemented a showcase implementation and presented it.

In the following chapter we will discuss the proposed method and implementation.

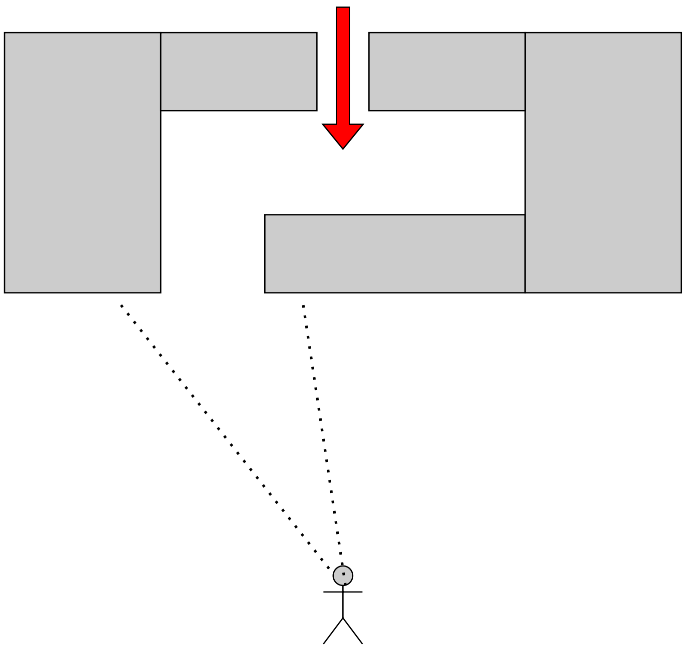
#### Discussion of the Differences between Model Implementation and its Real World Analogs

In section 4.3 the required environment component for the agent-based model was described. We argued that it is only necessary to use the functional space of any gamespace to gather data regarding the balance. However, some cases exist in which knowledge of the aesthetic space is necessary. For example, dark corners of a gamespace are not translated into the functional space. Dark corners can lead to human players overseeing an opponent hiding in them. Situations in which human players might oversee opponents are completely disregarded by the proposed model. Therefore it might generate incomplete data.

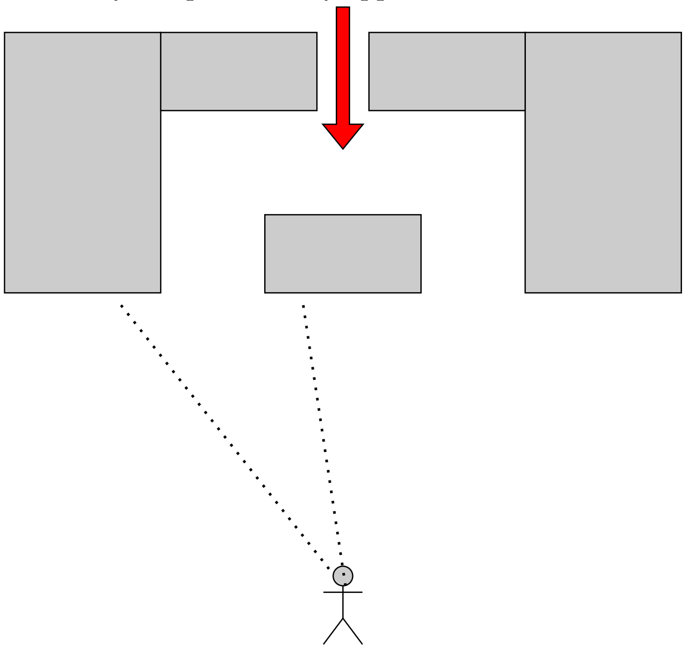
Further gameplay elements which get lost in the functional space are sounds. Sounds might add or even take from the information available to human players. The agents in the proposed model do not depend on sound which creates a discrepancy between the real world and the model.

Additional information like (stepping-)sound can give human players insight about the location of opponents. The location of opponents or rather the direction to them is always observed by the implemented agent (See section 7.2.1 for more detail). Not knowing the opponents position is an important factor as it can be used to balance gamespaces. An example for using an unknown opponent-position as a tool for balance is demonstrated in figure 8.1. The red arrow depicts an incoming opponent and the dotted line depicts the field of view of the player.

While in figure 8.1a an opponent only has one possible exit, in figure 8.1b they can either exit from the left-hand side or the right-hand side. Therefore, a waiting player has to choose which exit they guard. Forcing players to choose which exit they want to defend, makes it overall harder to defend both exits. This balances the gamespace in



(a) Only one possible way opponents can come from



(b) Two possible way opponents can come from

Figure 8.1: Demonstration of how knowing the opponents position might change balance.

favor of the attacking force because defenders cannot predict the attackers decision.

However, as the agents always know where the opponent will come from, such balancing options get lost in the implemented model.

Another discrepancy lies in the implementation of the rotation. The implemented agents rotate with a rate of one degree per update cycle. One degree per update cycle is very slow compared to the possible turning speeds in commercial video games. However, the turning speed is a minor discrepancy. It is only minor because the implemented agents always know the direction in which their opponent is. This allows them to prepare their rotation. By preparing the rotation, they decrease the negative effect of the slow rotation.

In section 7.3.1 the training and its results were presented. We managed to teach the agent human-like behavior in a flat gamespace and to use a ramp as advantageous position. While the taught behaviors can already provide data, they are not very close to actual strategic, human-like behaviors in arbitrary gamespaces. Especially since the behavior was only trained within small, simple and contained gamespaces and not in big gamespaces which are common in commercial videogames.

### 8.0.1 Discussion of the Validation, Verification and Replication

While section 7.3 does try to validate the implemented model, "[...] it is impossible for a model to exhibit all of the same characteristics and patterns that exist in reality" [WR15, p. 325]. However, more validation is necessary in particular, when more complex human-like behavior is modeled.

In addition to validation, section 4.4 presents two more techniques which make sure a model is correct.

The first technique is verification. Verification aims to get rid of "bugs" in the implemented program. An implemented model however, is not described by a binary verified/unverified status but it rather lies on a verification spectrum because more ways of testing and checking for correctness always arise. Therefore Wilensky and Rand suggest, that it is the authors responsibility to decide whether the current state of verification is enough. [WR15, p. 325]

Although every implemented system was tested thoroughly, no verification tests were implemented. The lack of verification tests is the reason why more verification is required for the model.

The second technique to test for correctness of the model is replication. Replicating the implemented model has not been done in this paper but all the required dimensions are provided and a replication should be possible.

### 8.0.2 Discussion of the Usage

Assuming the modeled ABM approach would work and simulate human players perfectly, it still only generates data. While the data will most likely improve the overall quality of created gamespaces, the model misses an important point of human playtests. Human playtests allow the designers to survey the players and get feedback regarding fun, feeling and overall game experience.

To not miss out on this crucial information, human playtest should still be conducted. However, the tool can reduce the amount of human playtests needed to reach a desired quality level in terms of balance.

# 9 Conclusion and Outlook

The process of creating a gamespace is done via constant iteration. To improve on the previous iteration, game designers require lots of playtest data. To generate the required data, conventional methods use human players. However, using human players has its drawbacks. To reduce the amount of human playtests required, we use machine learning agents and let them play instead of humans. Letting machine learning agents play, is part of our proposed method. The proposed method is supposed to use ABM for generating the required data to balance gamespaces.

While the machine learning agents do not resemble human players very well yet, they showed promising results in simple scenarios. Within these simple scenarios, the expected result of human players matched with the generated data from our agents.

Future work includes fixing the discrepancies between the implemented model and real arena FPS. Afterwards, a more complex and human-like behavior for the agents should be the main goal, since the generated data improves with the quality of the agent. Especially an agent that is able to work in arbitrary gamespaces would allow for modeling any arena FPS.

Parallel to further improvements in terms of agent behavior, further verification, validation and also replication are necessary to confirm the correctness of the model. Using real playtest data in different scenarios to further macrovalidate and empirically validate is a recommended first step.

# List of Figures

# Bibliography

[12] *Counter-Strike: Global Offensive*. Version: 1.38.2.1. 2012.

[16] *Overwatch*. Version: 1.72.1.0 - 99843. 2016.

[Ada13] E. Adams. *Fundamentals of game design, ThirdEdition*. New Riders, 2013.

[Ada14] E. Adams. *Fundamentals of Shooter Game Design*. New Riders, 2014.

[BG20] A. Becker and D. Görlich. "What is game balancing?-an examination of concepts." In: *ParadigmPlus* 1.1 (2020), pp. 22–41.

[BL97] A. L. Blum and P. Langley. "Selection of relevant features and examples in machine learning." In: *Artificial intelligence* 97.1-2 (1997), pp. 245–271.

[CLY15] W. Cachia, A. Liapis, and G. Yannakakis. "Multi-level evolution of shooter levels." In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 11. 1. 2015, pp. 115–121.

[Jul+18] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, et al. "Unity: A general platform for intelligent agents." In: *arXiv preprint arXiv:1809.02627* (2018). Corresponding github page: `https://github.com/Unity-Technologies/ml-agents`.

[Jul18] A. Juliani. *ML-Agents Toolkit v0.3 Beta released: Imitation Learning, feedback-driven features, and more*. 2018. URL: `https://blog.unity.com/technology/ml-agents-v0-3-beta-released-imitation-learning-feedback-driven-features-and-more` (visited on 07/17/2022).

[Jul21] A. Juliani. *Unity AI-themed Blog Entries*. 2021. URL: `https://blog.unity.com/technology/unity-ai-themed-blog-entries` (visited on 07/17/2022).

[KLY17] D. Karavolos, A. Liapis, and G. Yannakakis. "Learning the patterns of balance in a multi-player shooter game." In: *Proceedings of the 12th international conference on the foundations of digital games*. 2017, pp. 1–10.

[KTy] A. K."TychoBold". *Level Design - In Pursuit of Better Levels*. URL: `https://docs.google.com/document/d/1fAlf2MwEFTwePwzbP3try1H0aYa9kpVBHPBkyIq-caY/edit#` (visited on 07/19/2022).

[LLS14] P. L. Lanzi, D. Loiacono, and R. Stucchi. "Evolving maps for match balancing in first person shooters." In: *2014 IEEE Conference on Computational Intelligence and Games*. IEEE. 2014, pp. 1–8.

[MN05] C. M. Macal and M. J. North. "Tutorial on agent-based modeling and simulation." In: *Proceedings of the Winter Simulation Conference, 2005.* IEEE. 2005, 14–pp.

[MN09] C. M. Macal and M. J. North. "Agent-based modeling and simulation." In: *Proceedings of the 2009 winter simulation conference (WSC)*. IEEE. 2009, pp. 86–98.

[pro] open source project: *Cube 2: Sauerbraten*. http://sauerbraten.org/ (visited on 08/03/2022).

[Sch20] J. Schell. *The Art of Game Design: A book of lenses (Third Edition)*. CRC Press LLC, 2020.

[Sir02] D. Sirlin. *"Balancing multiplayer games"*. 2002. URL: `https://www.sirlin.net/articles/balancing-multiplayer-games-part-1-definitions` (visited on 07/06/2022).

[Tec21] U. Technologies. *Unity Manual*. 2021. URL: `https://docs.unity3d.com/2020.3/Documentation/Manual/UnityManual.html` (visited on 07/17/2022).

[Tec22] U. Technologies. *Unity Scripting API*. 2022. URL: `https://docs.unity3d.com/2020.3/Documentation/ScriptReference/index.html` (visited on 07/21/2022).

[Tot19] C. W. Totten. *Architectural Approach to Level Design*. CRC Press, 2019.

[WR15] U. Wilensky and W. Rand. *An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo*. Mit Press, 2015.