MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Game Engineering and Simulation Technology

Using Procedural Content Generation via Machine Learning as a Game Mechanic

By: Bernhard Rieder, BSc

Student Number: 1610585006

Supervisors: FH-Prof. Dipl.-Ing. (FH) Alexander Hofmann
Dr. Santiago Ontañón

Wildendürnbach, September 19, 2018

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wildendürnbach, September 19, 2018 Signature

Kurzfassung

Prozedurale Inhaltsgenerierung ist ein maßgebliches Thema in modernen Videospielen, da sie eine erhebliche Rolle bei der Erstellung von Spielinhalten beiträgt. Neue Forschungsergebnisse haben nun mit maschinellem Lernen neue Horizonte für prozedurale Inhaltsgenerierung erschlossen. Dabei wurde festgestellt, dass es noch bestimmte Thematiken zu erforschen gilt, wie etwa die Verwendung von prozeduraler Inhaltsgenerierung durch maschinelles Lernen als Spielmechanik.

Folglich beschäftigt sich diese Arbeit mit der Fragestellung, wie prozedurale Inhaltsgenerierung durch maschinelles Lernen als Spielmechanik verwendet werden kann, was im Folgenden als Spielmechanik bezeichnet wird. Die Arbeit folgt dem Ansatz, einen Leitfaden für Entwickler zu erschaffen, der diese mit theoretischem und praktischem Wissen für eine Entwicklung dieser Spielmechanik vorbereitet. Es wird zuerst Theoretisches behandelt, anschließend näher auf mögliche Spielmechanik-Konzepte eingegangen, woraus eine der Mechaniken als Prototyp entwickelt wird. Die Entwicklung dieses Prototypen wurde dokumentiert, so dass Spielentwickler dieser Umsetzung bei Bedarf folgen können.

Diese Abhandlung zeigt, dass sich die Spielmechaniken für eine große Palette von Spielgenres eignen. Dreizehn unterschiedliche Konzepte von Spielmechaniken werden thematisiert, woraus die Spielmechanik "Wechselnde Waffen" für einen Prototypen gewählt wurde. Das Hauptmerkmal dieser Mechanik ist, dass sie einen Waffen-Generator besitzt, welcher neue, aber ähnliche Waffen – basierend auf Waffen eines bekannten Ego-Shooters – generieren kann. Der Generator verwendet einen in TensorFlow implementierten "Variational Autoencoder", um die verborgene Struktur der zur Verfügung gestellten Waffen abzubilden und ist in der Lage, sinnvolle Waffen Daten zu generieren. Als Beweis, dass der Generator ohne Probleme in häufig verwendeten Spieleentwicklungsumgebungen funktioniert, wurde er in einem Test Spielszenario integriert, welches mit Unreal Engine 4 entwickelt wurde.

Um die Arbeit abzuschließen, wurde ein detaillierter Leistungsbericht erstellt, welcher zeigt, dass die entwickelte Spielmechanik keine nennenswerten Leistungseinbußen mit sich bringt. Das bedeutet, dass es möglich ist diese Spielmechaniken in gewöhnlichen Videospielen zu verwenden. Daher zeigt dieser Nachweis der Machbarkeit, dass für zukünftige Spiele eine neue Ära von Spielmechaniken angebrochen ist.

Schlagworte: Prozedurale Inhaltsgenerierung, Maschinelles Lernen, Spielemechanik, Künstliche Intelligenz, Spieleentwicklung, Neuronale Netze

Abstract

Procedural Content Generation (PCG) is a powerful and essential topic in modern video games which helps developers to create a vast amount of game elements. Recent research now connected PCG with Machine Learning (ML) to enable new horizons of content generation. Nevertheless, the research showed that there is still much to do and left the problem of using PCG via ML (PCGML) as a game mechanic open for further research.

For this reason, this thesis dedicated itself to address this open problem with a theoretical and practical approach and furthermore provides developers with a guideline about the procedure of developing PCGML game mechanics. It first addresses fundamental theoretical issues which help to create awareness for PCGML in the first place. It then addresses possible PCGML game mechanics where one of them was implemented in a game prototype scenario. The entire development process for this prototype was documented so that developers can follow them step by step to implement their PCGML game mechanics.

Now, the research showed that PCGML game mechanics are suitable for a broad range of games without limitation to particular genres. 13 different ideas are described in the thesis and one particular idea called "Changing Weapons" was then implemented in a game prototype scenario. The game mechanics primary feature is a weapon generator which can generate new but similar weapons based on the weapons of a well-known first-person shooter game. In specific, the generator uses a with TensorFlow implemented variational autoencoder to learn the underlying and hidden structure of the provided weapon data and can generate useful weapon data. This generator was then integrated into Unreal Engine 4 to test and prove that a PCGML game mechanic can be used in a typical game engine and showed that this application is possible without any issues.

To conclude the thesis, a performance report was created which showed that the implemented game mechanic does not cause significant performance losses. Thus, it is possible to use PCGML-based game mechanics in regular video games. Therefore, with this proof-of-concept, a new game mechanic area for creating new player experience has opened for future games.

Keywords: Procedural Content Generation, Machine Learning, Game Mechanic, Artificial Intelligence, Game Development, Neural Networks

Acknowledgements

I want to thank the Austrian Marshallplan Foundation which sponsored my stay in the United States of America to write this master thesis. Moreover, many thanks to Dipl.-Ing. (FH) Alexander Hofmann who contributed with his consent, extensive help and support, and the excellent recommendation for the most suitable international University for a study abroad. For this reason, I also want to say thank you to the Drexel University and Dr. Michael Wagner who invited me to work on the thesis in their department. Last but not least, a special thank you to Dr. Santiago Ontañón - my supervisor at the Drexel University - I could not have finished this master thesis without your extraordinary support!

Contents

1 Introduction

Procedural Content Generation (PCG) is an essential and aspiring topic in modern games and is extensively used for decades (Togelius et al., 2011). Therefore, further research on different subjects of PCG is necessary to provide new techniques for games development. Notably, it is primarily a very crucial topic for small independent game development studios due to a low budget, where PCG can generate much content for less effort and human resources (Blatz & Korn, 2017). For example, it can be a difficult task to design and develop a broad range of content in a short amount of time and a small team. With this in mind and according to Moore's Law, more and more storage will be available on a gaming system in the future and thus will offer game developers more space for content. While gamers and players will be getting used to massive amounts of content because of big gaming companies which can establish a broad range of new content without the use of PCG, the small development teams will not keep up as smooth as the market leaders. Here is where Machine Learning (ML) comes into play. PCG is getting much more accessible and compelling with the help of ML which combined form the new technique of Procedural Content Generation via Machine Learning (PCGML) (Summerville et al., 2017).

The critical advantage of PCGML over PCG is that standard PCG techniques need to be finetuned or even explicitly designed for specific generation tasks, while PCGML techniques aim at designing general PCG algorithms that can generate a large class of content by just seeing data via ML. Therefore, a PCGML system opens a lot of new possibilities because it makes use of ML. For example, it can be trained on its own and evolve if they do not generate usable output (Summerville et al., 2017). Furthermore, the system could also be trained by some designers with unique input or by a regular user with their creative input (Summerville et al., 2017). PCGML can be used for many aspects of a game since it can learn from simple examples and existing domains. Most current work on PCGML focuses on creating designed content like unlimited amounts of different levels (Summerville et al., 2017). However, there are some open problems which need to be addressed to utilize the whole power of PCGML, and one of the open problems is about the use of PCGML as a game mechanic (Summerville et al., 2017). In particular, this research problem is very promising for, e.g., evolving the overall player experience in games, which could enhance the future of player experience development.

1.1 Idea

PCGML is a relatively new method and technique for creating different kinds of content in modern video games, and most current work focuses mainly on replicating designed content to provide the player with infinite and unique variations on gameplay (Summerville et al., 2017). In general, Summerville et al. (2017, p. 1) defined PCGML as ”the generation of game content using ML models trained on existing content.”

Another possible innovative use of PCGML is its use as the central mechanic of a game, e.g., presenting the PCGML system as an adversary or toy for the player to engage with (Summerville et al., 2017). However, the promising paradigm of using PCGML as a game mechanic is an open and unexplored research problem (Summerville et al., 2017). On this account, the main idea to pursue in this thesis is to explore the possibilities in the use of PCGML as a game mechanic.

With this in mind, it needs a detailed analysis on how it could be used best in games. For example, a design of mechanics could include enticing the player to generate content that is significantly similar to or different from the corpus the system was trained on, or identify content examples that are outliers or typical examples of the system (Summerville et al., 2017). Alternatively, players could also train PCGML systems to generate examples that possess certain qualities or fulfill specific objective functions, teaching the player to operate a model by feeding it examples that shape its output in one direction or the other (Summerville et al., 2017).

Various design patterns can guide the development of an exemplary PCGML game mechanic system. As illustrated by Treanor et al. (2015), design patterns can follow the concepts of the Artificial Intelligence (AI) as visualized, role-model, trainee, editable, guided, co-creator, adversary, villain or spectacle. Every one of them provides an excellent guiding principle for designing and implementing a PCGML game mechanic.

1.1.1 Advantages

As already mentioned, PCGML can offer an unlimited amount of content when used as a content generation pipeline which is also applicable to game mechanics. In general, PCG mechanics are offering knowingly more replay value than usual game mechanics (Shaker et al., 2016). However, this is going to increase significantly and offers more flexibility with the help of ML by, e.g., behavior learning where a player could replay a game with a different behavior which would lead to different events in the game.

Another advantage offers the use of preference learning for, e.g., a difficulty adaption system in combat between the player and the PCGML system. In that scenario, the system could work against the player’s preference with using, e.g., a specific weapon and counter-attack with a defensive advantage and therefore can increase or decrease its difficulty on demand.

In particular, a player could also experience an emotional connection with, e.g., a trainee-based PCGML system that needs to be taken care of by hatching, raising and training

it throughout the game. This effect of emotional attachment to software is known as the "Tamagotchi-Effect" which has arisen from the famous pet hatching game Tamagotchi (Holzinger et al., 2001). Hence, a system like that can enhance positive and magnificent memories for the players and thus for the game experience and the game itself.

1.1.2 Challenges

One of the leading challenges in creating a PCGML game mechanic is the design of the mechanic which should fulfill some crucial requirements of game design to offer a good player experience. As well, the ML part is going to be a challenging part since it might take much tweaking and data to get a fully working AI algorithm.

1.2 Desired Goals

It is important to note that the main idea of this master thesis is to create game mechanics which rely on the principles of PCGML rather than creating a generic PCGML game mechanic generator.

With this in mind, it is expected to provide a first insight into the use of PCGML as a game mechanic in modern games. The primary goal is to demonstrate the possibilities as well as the development process of game mechanics when it comes to the use of PCGML and also how games should work when using PCGML.

Additionally, there are some further questions which need to be addressed by this thesis. It should impart some theoretical and practical knowledge of PCG, ML, PCGML, and PCGML as a game mechanic. On this account, it is desired to show how an implementation of these concepts can look like and which dependencies are given and needed for a fully working implementation.

Furthermore, it should provide a good overview and function as a primer for developing proprietary PCGML game mechanics in a specific game engine or other environments. In particular, it shall be a focus on implementations in commonly used game engines since most of the independent game developers are using existing free-to-use game engines instead of creating a new engine.

A substantial goal of this work is a fully working game prototype with PCGML as the core game mechanic which acts as a perfect example of what is possible with this kind of functionality. Also, since video games, in general, are performance-heavy applications, it should cover a performance report as a point of reference for future implementations and uses. As an additional point, it should include an outlook of the opportunities of PCGML game mechanics in future games and work, which should also function as motivation for future work in this field of research.

Generally speaking, it should be an overall guideline for bringing PCGML game mechanics into a game.

1.3 Approach

As mentioned before, one goal of this thesis shall be the support of small and independent game developers with an introduction to PCGML game mechanics in a game engine like Unreal Engine or Unity. For doing so, it is going to address all essential topics which are dependent on building PCGML game mechanics and their use in game engines. For this reason, the planned agenda to approach the use of PCGML as a mechanic will be broken down into two parts. The first one is a scientific-informal part about getting to know more about the foundations of PCGML and its use as a game mechanic. Since PCGML is a relatively new subject in game development, it focuses on topics regarding central fields of interest in PCG and ML separately and game mechanics to act as a base for further research on PCGML as a game mechanic and to create awareness for these topics in the beginning. Following topics shall be a part of the informal research:

- Game mechanics and their use in games.
- Necessary and essential theory of PCG and ML which is dependent on PCGML with a constant focus on game mechanics, like types of PCG and some of the most used learning and training models of ML.
- The conceptual use of PCG and ML in a game engine as well as best practices, other approaches, and possible issues.

Afterward, all the beforehand discussed topics shall be combined into PCGML and furthermore into PCGML game mechanics. Therefore, the second part of the research agenda deals with the central scientific problem of this master thesis. It addresses every aspect of PCGML and discusses how to use PCGML as a game mechanic in modern games with a focus on the maximum possible benefit for game developers. This part shall contain the following fields of research:

- The theory of PCGML and its methods in general.
- Research on different PCGML implementations and practical usage possibilities in a game engine.
- Overview of possible game mechanics with PCGML.
- Conceptual implementation of possible PCGML game mechanics in a game engine and subsequent evaluation as well as a detailed comparison.
- Game development insights with one of the best-evaluated PCGML game mechanic as the central game mechanic of the game.
- Proof of concept with a fully working PCGML game mechanic game scenario prototype.

- Research summary with the meaning of PCGML as a game mechanic for the future of games.

However, as necessary as a well-prepared agenda are some methodological questions which need to be raised and answered at both research and implementation time, like:

- Which free-to-use game engine is eligible for PCGML game mechanics?
- What are applicable PCGML game mechanics for games?
- What are appropriate evaluation criteria?

1.4 Thesis Overview

The thesis starts with a short introduction to game mechanics and their possibilities with some considerations to PCGML. This chapter follows an introduction to PCG and ML to create awareness of their development and their use with and without games. Additionally, they mention some best practices and common mistakes during the development. Right after this chapter follows the PCGML chapter which describes the differences to conventional PCG algorithms, lists its most useful use cases, and explains development considerations of previous research and the development for the prototype. A theoretical development example of how PCGML can work in a game closes this chapter.

The next chapter addresses the possible applications of PCGML as a game mechanic and describes 14 different ideas. Each one consists of a short introduction followed by pros and cons as an evaluation for theoretical development. A summary of the game mechanics with a conclusion for the prototype's game mechanic sums up this chapter.

The next step on the agenda is the preparations for the prototype so that developers learn about requirements to implement a PCGML game mechanic. It starts with a simple game design of the prototype scenario, goes straight to the question of which game engine is the most promising one, and explains the hard and software requirements for the development environment and the game. The further focus then moves to the actual data which is used to train the PCGML model. It first shows different sources for the model's training data and then goes over to the actual learning problem which needs to be solved by the PCGML model. This chapter addresses the actual learning problem, shows suitable learning models and lastly introduces the used model so that the reader knows about its unique characteristics.

Now, that there are no further questions about the development, it starts with the proof-of-concept and practical part of the thesis and the development of the prototype PCGML game mechanic. The first chapter addresses the procedure of processing the PCGML model raw data into a usable and convenient class for the prototype. The next and most crucial part of the development is the introduction of the actual PCGML model. It shows the classes, addresses specific topics about the development, and shows empirical observations of the model to prove that it is working. Afterward comes a short introduction to the game scenario and its

implemented features. Following comes the explanation of the workflow and parameters for the most crucial part of the implemented game mechanic. To sum up the prototype development, it shows performance and profiling charts of the used game mechanic in the game engine.

A concluding chapter sums up the thesis, recaps, and analyses the research results in-depth. At last, it addresses possible improvements and future work for the implemented game mechanic and PCGML as a game mechanic.

1.5 Target Audience

Advanced game developers who are interested in using PCGML game mechanics in their game are the core focus of this thesis. The theoretical part assumes knowledge of game design and mechanics, PCG, AI, and ML since it will not be explained everything in detail. In particular, specific topics of PCG and ML which contribute to the use of PCGML as a game mechanic will be discussed and handled in more detail, but it requires an understanding of Neural Networks (NNs).

The practical part concentrates primarily on the design and implementation of a game and game mechanics which makes it necessary for the reader to be familiar with these subjects. Particular algorithms used throughout the chapters will be covered in detail whereas basic algorithm knowledge is assumed. Furthermore, it does not require exceptional game development back-end skills since it addresses the use of the technique in game engines.

2 Game Mechanics

Starting this chapter with a quick insight into the Mechanics-Dynamics-Aesthetics (MDA) framework which was introduced by Hunicke et al. (2004), helps to understand the foundation and the correlation of game mechanics in video games. In general, the MDA framework describes the division of gaming experience emergence into three dependent parts, starting with "Rules" followed by "System" and concluded with "Fun" (Hunicke et al., 2004). On this account, Hunicke et al. (2004) described these fundamental parts of a game by the designs of mechanics, dynamics, and aesthetics. Therefore, mechanics contribute to a significant amount of gaming experience, and it needs adequately thought through mechanics because otherwise, a game will not be fun at all even if it has incredible graphics (Adams & Dormans, 2012). Consequently, game mechanics are acting as one of the most critical roles in game design which is the reason to create awareness for this topic at the beginning of the thesis.

2.1 Definition

As already indicated, a game mechanic is a concept with many underlying sub-concepts like dynamics, aesthetics, rules, systems, processes, procedures or data which all characterize the heart of a game besides story and technology (Adams & Dormans, 2012) (Schell, 2008). It also creates gameplay and the experience of playing a game. However, there is no concrete definition of what a game mechanic is. Nonetheless, there are some essential concepts mentioned by different game designers which contribute to an interpretation of what a game mechanic can, shall be or do:

- Defines how a game is played, their objectives can be achieved or how to lose a game (Schell, 2008). Thus, mechanics are precisely designed, detailed, specified and implemented to fulfill playability (Adams & Dormans, 2012) (Schell, 2008).
- Often used to indicate the most influential and affecting aspect of a game which is also mostly referred to as core mechanic (Adams & Dormans, 2012).
- Enables interaction and control of game objects and elements (Adams & Dormans, 2012).
- Is mostly hidden from the player, media-independent and easy to learn (Adams & Dormans, 2012). For example, players are mostly aware of primary and often explained mechanics like character abilities whereby mechanics like an enemy damage model with its damage points are hidden concepts (Adams & Dormans, 2012).

- Can also be described as a meeting point for a designers question and their provided tools for answering that question by a player (Stout, 2015).

2.2 Types of Mechanics

It is evident that one tries to divide possible mechanics into concrete types based on their various possibilities and shared base ideas. For this purpose, Adams & Dormans (2012) summarized different types of game mechanics which are mainly used in games nowadays. Adams & Dormans first categorized them into the following five types which are listed below with some related mechanics:

- **Physics**: Motion and forces like gravity, shooting, fighting, jumping, moving, driving or any other kind of position change (Adams & Dormans, 2012).
- **Internal Economy**: In general, all game elements which involve transaction like collecting, consuming, harvesting, buying, building, upgrading, risking or customizing of resources like currency, ammunition, portions, power-ups or other kinds of items (Adams & Dormans, 2012). Also the use of energy, health, lives, power, points, popularity or experience and management actions for a team, resources or inventory (Adams & Dormans, 2012).
- **Progression Mechanisms**: Usually the elements or mechanisms which are controlling the players progress in the game world (Adams & Dormans, 2012). For example, quests, missions, competitions, tournaments, races, challenges, levers, switches, locks, keys or unique items which allow a player to defeat an AI (Adams & Dormans, 2012).
- **Tactical Maneuvering**: Is mainly used in strategy games but also in roleplay or simulation games and often deals with the placement of elements on a map like in chess (Adams & Dormans, 2012). Mechanics are for instance internal tactics where a player gains offensive or defensive advantage but also team tactics and management of resources and buildings (Adams & Dormans, 2012).
- **Social Interaction**: Refers to rules that govern play-acting of a player or strategic actions of forming allies to defeat bosses or other allies like in Roleplay Games (RPGs) (Adams & Dormans, 2012). Further mechanics would be, e.g., a reward of giving gifts, inviting new friends to join the game, competition between players or in a co-op game where at least two players are forced to work together to achieve an objective (Adams & Dormans, 2012).

Besides, it is possible to subdivide all prior mentioned mechanics into discrete and continuous mechanics concerning their internal values (Adams & Dormans, 2012). For example, the internal economy is mostly discrete since it is mostly represented by an integer value because, e.g., a player cannot pick up half of a portion — either he or she picks up the portion entirely or not (Adams & Dormans, 2012). In contrast, continuous mechanics make use of high precision

values for accuracy with continuous calculation throughout the game like the movement of a character (Adams & Dormans, 2012).

Furthermore, every type can also be used to categorize game genres based on their average usage in games. Table 1 shows this distinction between mechanics and genres.

	Game Mechanics				
Game Genres	Physics	Economy	Progression	Tactical	Social
Action	x	x	x		
Strategy	x	x	x	x	x
Roleplay	x	x	x	x	x
Sports	x	x	x	x	
Vehicle Simulation	x	x	x		
Management Simulation		x	x	x	x
Adventure		x	x		
Puzzle	x		x		
Social Games		x	x		x

Table 1: Game genres and their related game mechanics (Adams & Dormans, 2012).

However, since the overview of Adams & Dormans (2012) is no universal taxonomy for game mechanics, there is another excellent approach to categorize them as described by Schell (2008). Following somewhat similar types to Adams & Dormans' approach are used which also correlate to some parts described in the MDA framework:

- **Space**: Every game takes place in some game space or spaces (Schell, 2008). Spaces can be continuous or discrete, consists of dimensions and can have bounded areas that may or may not be connected (Schell, 2008). The mechanics of Tic-Tac-Toe are an excellent example of this kind of mechanics which are taking place in a discrete space.

- **Time**: Contains mechanics which are using time, clocks, races or controlled time (Schell, 2008). A well-known example of this kind of mechanics is the game SUPERHOT (SUPERHOT Team, 2016) which tweaks the time to create a unique game experience.

- **Objects, Attributes, States, and Actions**: A comparison between mechanics and the structural elements of a sentence reveals similarities (Schell, 2008). Game objects represent the nouns, attributes and states are their adjectives and actions are the verbs of a game mechanic (Schell, 2008). This paradigm applies to most of the mechanics which enable interaction with game elements (Schell, 2008).

- **Rules**: Combines all spaces, times, objects, actions and their consequences, constraints and the goals to form the behavior of the game (Schell, 2008).

- **Skill**: Shifts the focus to the players and focus on their physical, mental and social skills (Schell, 2008). That means it includes mechanics like dexterity, coordination, memory, observation, puzzle solving, reading or fooling an opponent or coordinating with teammates (Schell, 2008).

2.3 Considerations with Procedural Content Generation and Machine Learning

This chapter shall state some crucial considerations for the next chapters since PCG and ML game mechanics are not visible used in big game titles and therefore need particular attention on their implementation in a game. One of the good things is that there are dozens of possibilities for mechanics which should not create a big problem in coming up with new and novel ideas for new mechanics. With certainty, the focus of implementing such mechanics will lie on the introduction to the player and their ability for interactions because PCG and ML mechanics could confuse some players. Therefore, the implemented mechanics should be kept as transparent as possible if user interaction is needed instead of creating complex but novel and unusual mechanics. A good starting point is to design the mechanics as soon as the central gameplay concept is set and adhere to the design stages of concept, elaboration, and tuning during development (Adams & Dormans, 2012).

It is necessary to list and consequently avoid some possible design flaws since game mechanics shall amaze people instead of frustrate them while playing a game. Besides, much detailed planning is made to come up with new extraordinary mechanics, but plans for their proper introduction are often missing (Pears, 2018). For this reason, it is relevant to address some common mistakes and their possible improvements:

- Do not introduce all mechanics of a game as fast as possible because players need time to learn and get used to them (Pears, 2018). For this reason, it is advisable to introduce just one mechanic at a time.

- Do not introduce mechanics when the player has no time to explore them. They need time at their own pace to explore them otherwise they will not enjoy their new ability (Pears, 2018).

- Use and create feedback loops for game mechanics otherwise players will not know what to do with them (Adams & Dormans, 2012). For example, if someone uses a portion and there is no clear visualization for the use of it, then the player probably does not know what to use it for.

  Introducing the concept of the basic grammar model described by Koster (2013) shows that feedback is one of the most critical elements. This model can be applied to most of the games nowadays, and it loops the concepts of a mental model, intent, input, actual

model and rules, state change and feedback (Koster, 2013). Where the mental model of a player assumes how a game works and what their intentions for the input and the actual input are, and what then really happens with their input regarding applying core mechanics, concluded with feedback for their inputs (Koster, 2013). If there would be a lack of feedback, then the player could never update their mental model and cannot progress through a game. Feedback can appear in a quite simple binary or even complicated way (Koster, 2013).

- Besides feedback loops, do not forget to provide the player with directions for parts of the mechanic which are or could not be visible to the player (Pears, 2018). Further tutorials should be easily accessible if they are needed because there is nothing more frustrating to a player than being confused (Doan, 2017).
- In particular, essential considerations for core mechanics are to provide clear rules on how to be successful, create a natural interaction but do not forget to challenge the player and provide the possibility for a natural progression of their skills (Doan, 2017). Furthermore, they should adequately guide the player towards completing their in-game objectives with directions and feedback, allow players to progress from objective to objective in a natural way without the necessity of using the core mechanic and provide options besides the core mechanic (Doan, 2017).
- In general, the skill of a player will grow throughout the game which means that the difficulty curve shall match the player's skill during the whole experience (Doan, 2017).
- As described by Zook & Riedl (2014) where their goal was to generate and adapt game mechanics, it is necessary that mechanics fulfill the requirements of playability to create an acceptable experience. For example, a requirement could be that it is necessary that a player can reach the end of a level or win a fight without dying. Overall, it should ensure that a game is playable to a specific given goal with that mechanic (Zook & Riedl, 2014).

3 Procedural Content Generation

PCG is a broad topic in the game industry which has evolved throughout the years, and much research was done and is currently going on to enhance and further explore its possibilities. Accordingly, it is a necessity to introduce general parts of PCG to understand some concepts and therefore be able to understand its further use in PCGML.

3.1 Introduction

Game content creation is an expensive task where often many designers are involved over an extended period (Amato, 2017). That is where the encouragement of PCG comes into play! It aims towards automatic game content generation which is done by different algorithms on their own with or without direct user or designer input to decrease the cost of content creation (Amato, 2017) (Togelius et al., 2011).

However, it would be too easy to come up with a standard definition on what procedurally generated content in a game defines because too many people attempted it with way too many and different approaches (Togelius et al., 2011). With this in mind, procedurally generated content seems to be a concept with fuzzy and unclear boundaries which cannot be defined precisely (Togelius et al., 2011). Nevertheless, for this thesis, content generated by PCG algorithms are seen as content or elements in a game which are affecting the gameplay, for instance, puzzles, quests, rules, dynamics, weapons, stories, terrain, maps and other similar kinds of game elements.

3.1.1 Reasons to Use

There are many reasons why PCG is a significant and rising topic in games. For this reason, Short & Adams (2017) came up with two classifications representing the fundamental motivation behind using and researching PCG techniques.

Utilization

The first argument is utilization which is the principal argument why PCG is popular (Short & Adams, 2017). It can be time-saving because it could produce more content than a human in an hour, for instance, a whole galaxy in No Man's Sky (Hello Games, 2016) (Short & Adams, 2017). Moreover, PCG overcomes technical limitations concerning their use for devices with, e.g., limited space like mobile devices, it is expandable and has reusable code due to modularity

and same field of applications (Short & Adams, 2017). Lastly, it increases replayability by generating many similar but different instances of content (Short & Adams, 2017).

Uniqueness

The second argument why PCG is of particular importance is the uniqueness of their output (Short & Adams, 2017). It offers individual experiences with, i.e., different generated content every time it is played and creates new gameplay and interaction modes for replayability. It is unpredictable which can be a thrilling fact for players but also designers, can create bizarre content no human might think of, and can be an inspiration of infinity because of its possibility for creating infinite various content (Short & Adams, 2017).

3.1.2 Taxonomy

The use cases for using procedurally generated content for different kind of problems with different methods and approaches are almost unlimited. This variety of PCG possibilities made it necessary to find distinctive features and create a taxonomy of PCG to highlight the differences and similarities between approaches (Shaker et al., 2016). In fact, there are two different views for a taxonomy which were described by Hendrikx et al. and Shaker et al.. The first and initial approach was created by Hendrikx et al. whereas Shaker et al. gave the new one derived from Hendrikx et al.'s taxonomy.

As an initial classification, Hendrikx et al. (2013) extracted the following four classes by analyzing all possibilities of PCG which they could think of:

- **Game Bit**: Basic elements of a game that do not affect the player's gameplay. For example, procedurally generated textures, sounds, trees, fire, stones, mountains or clouds. (Hendrikx et al., 2013)
- **Game Space**: Represents game environments and usually consist of different game bits. One can think of, e.g., dungeons maps, whole planets with procedurally generated terrain, lakes, rivers and many more. (Hendrikx et al., 2013)
- **Game System**: Includes all game elements like ecosystems or other relations between game objects like rules or objectives. (Hendrikx et al., 2013)
- **Game Scenario**: Like occurring events in games which could be, e.g., an event in a generated storyboard, the history of a character, a concept of levels or a puzzle. (Hendrikx et al., 2013)

Whereas Shaker et al. (2016) extended their view of possibilities for PCG and came up with the following 13 classes which are more focused on technical issues:

- **Online versus Offline**: Is about runtime generated game elements or content as the player is playing the game versus predefined and or pre-generated content which is created before the start of a game (Shaker et al., 2016). For instance, an interactive maze, generated during runtime, versus a procedurally generated terrain which is used as the uniform environment of a game and does not affect the players playing experience regarding a PCG process during the game.
- **Necessary versus Optional**: Distinguishes content which is necessary or required to reach an objective in a game and content which does not need to fulfill this or other requirements (Shaker et al., 2016). For example, a puzzle could be necessary to finish the game whereas a generated texture is just an optional and cosmetic content.
- **Degree and Dimensions of Control**: Adds control over content generation via adjustable generator parameter or with a specific seed for a Random Number Generator (RNG). In general, content where designers or users and players are in control of the generation space. (Shaker et al., 2016)
- **Generic versus Adaptive**: By means of generic content which does not take the behavior of the player into account whereas adaptive content could adapt on a player's progress in the game and will be generated on top of his or her current progress and skills. (Shaker et al., 2016)
- **Stochastic versus Deterministic**: This paradigm differs content creation in a scientific manner where deterministic algorithms will produce the same content over and over again provided that the same parameters are given whereas stochastic algorithms will create different content every time they are used. (Shaker et al., 2016)
- **Constructive versus Generate-and-Test**: Addresses in one pass generated content versus generated and continuously improved content (Shaker et al., 2016). Latter PCG method generates desired content as a result of continuous testing against requirements in a finite generate and test loop (Shaker et al., 2016). Usually, there is some AI involved in the evaluation of generate-and-test content (Shaker et al., 2016).
- **Automatic Generation versus Mixed Authorship**: By means of fully autonomous content generation provided by an algorithm versus generators where designers and players can change the behavior of the design process due to any input and cooperate with the algorithm (Shaker et al., 2016). For example, the creature creation in the game Spore (Maxis, 2008) with automatically generated and user-created creatures.

3.2 Development

Developing a PCG algorithm can be tough and needs to be well-prepared. On this account, there are given some important design considerations about PCG and also a conceptual imple-

mentation of an algorithm in the next two sections.

3.2.1 Design Considerations

It is a good practice to stick to desirable and required properties when designing and developing algorithms and especially algorithms for generating content. One can quickly lose sight of crucial and influencing factors when developing a PCG algorithm with one of many various options which could lead to lousy player experience. For this reason, Shaker et al. (2016) stated some of the most critical factors of an algorithm which should be satisfied:

- **Speed**: In general, this property depends on the online versus offline class which was described in Chapter 3.1.2. Equally, whether a PCG algorithm produces content during gameplay or generated it before the core game starts, it should never exceed an acceptable amount of time which is needed to generate content because otherwise, it could affect the player's experience (Shaker et al., 2016).

- **Reliability**: Some generators create content from scratch without even knowing what they should produce whereas others are capable of generating and evaluating their content due to given requirements (Shaker et al., 2016). For this reason, it is a very crucial property if someone wants to generate dungeons or puzzles because of their necessity of being a solvable problem which makes it either possible or not to progress throughout a game whereas a tree or flower which looks weird does not break a game (Shaker et al., 2016).

- **Controllability**: Is also one of the most crucial properties of PCG since it is advantageous to be in control in order to specify aspects of the generated content. Mainly, this refers to the classes of degree and dimensions of control, generic, adaptive deterministic as well as mixed authorship which was described in Chapter 3.1.2 and also overlaps with the desired reliability property (Shaker et al., 2016). For example, a designer or a player-adaptive algorithm should have control over parameters to specify a desired outcome (Shaker et al., 2016).

- **Expressivity and Diversity**: This property speaks mostly for itself because the human brain can quickly detect and recognize patterns in various environments (Shaker et al., 2016). That makes it a necessity to develop algorithms which can generate content with a right amount of expressivity and diversity (Shaker et al., 2016).

- **Creativity and Believability**: Following up the last property, it is also necessary that an algorithm produces believable content which cannot be distinguished to human designed content (Shaker et al., 2016). It should not be evident for the players to be able to distinguish between algorithm-generated content and entirely designed content (Shaker et al., 2016).

Notably, a central component which ties expressivity, diversity, and creativity together is their essential use of different RNGs. There are possibilities like using a standard RNGs or the creating of randomness via knowledge presentation, distribution altering or look-up tables (Shaker et al., 2016). For this reason, it is essential to give special considerations to the randomness implementation as well to fulfill the described properties. Some of the most important techniques used with random generations are, e.g., Perlin Noise, Simplex Noise or Fractals (Blatz & Korn, 2017).

Another helpful point is to visualize the PCG for either debugging or gameplay purposes (Shaker et al., 2016). For example, visualization can help to understand the output and distribution of a PCG when used for debugging. Furthermore, if PCG requires interaction with the player, then it is useful to show some visual feedback or visualization so that players can understand the consequences of their actions on the system (Smith et al., 2012).

Lastly, it is recommended to keep PCG algorithms simple and focus them on specific content generation tasks so that a bunch of content generators could be combined afterward (Shaker et al., 2016). Also, players should not be overwhelmed by interactive PCG systems (Blatz & Korn, 2017). For this reason, avoidance is possible with simple sensors which are taking care of the players' experience and furthermore adapt the system, provided that it is an online system (Blatz & Korn, 2017). In general, all discussed points in Chapter 2.3 should be taken in mind when implementing interactive PCG systems.

3.2.2 Possibilities

Using procedurally generated content in a game offers many possibilities as described in the last few sections. One of the most promising facts for using PCG in games is that players can experience a game in a new way each time it is played provided that it uses online systems (Rose, 2012). For this purpose, Liapis et al. (2014) extracted the creative facets of games where PCG can be used to create content in games. Beginning with visuals as the most prominent application where the most successful example is "SpeedTree" (IDV, 2018) which can create 3-Dimensional (3D) models of trees and vegetation (Liapis et al., 2014). However, also textures, every other kind of 3D or 2-Dimensional (2D) models or even whole solar system as in the game "No Man's Sky" (Hello Games, 2016) or visualizations are part of visuals (Liapis et al., 2014). The next classification includes every kind of generated audio and narrative (Liapis et al., 2014). Ludus also offers a vast field of possibilities where the term Ludus refers to activities under a system of rules which defines the outcome of a game (Liapis et al., 2014). Also, level architecture like generated maps or generated gameplay can be found as creative facets of games (Liapis et al., 2014).

In general, the determining term in PCG is "content" which means that one can barely generate everything in a game (Shaker et al., 2016)! There is even a PCG algorithm called "Angelina" which can generate whole games (Cook, 2018).

3.2.3 Conceptual Implementation

PCG algorithms can vary from simple to very complex ones depending on the field of their application. Usually, they are algorithms which are fed with different parameters and then generate some content out of these parameters.

Algorithms, like used for world generation, can consist out of many details and therefore will not be addressed in this section. Instead, the development steps of an algorithm for generating complex rock structures called "Cascades" presented by NVIDIA (2007) will be shown to give a quick insight into how a PCG algorithm can work.

Cascades

The simplification of NVIDIA's approach for a procedurally generated complex rock structure is as follows:

1. 3D texture generation where density values represent either rock or air.
2. Take the texture and make use of the Marching Cubes algorithm to generate the actual 3D model.
3. Use tri-planar texturing to complete the model with textures.

A possible output of this algorithm can be seen in Figure 1 depending on their used RNG and definition of density distribution in the 3D texture.

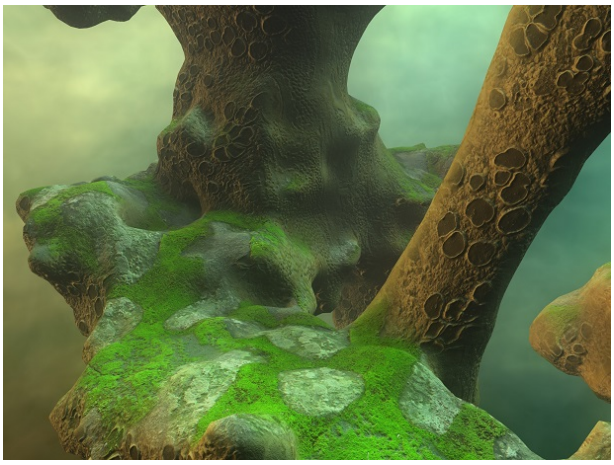

Figure 1: Example of a procedurally generated rock (NVIDIA, 2007).

3.3 Game Mechanics

Now it comes to the most crucial part of this chapter of PCG because of its leading part in PCGML as a game mechanic — the use of PCG as a game mechanic! This chapter provides

an overview of games which are using PCG as a game mechanic and some examples of possible PCG core mechanics.

3.3.1 Current Games

Many games are using procedurally generated content nowadays, but not all of them are using PCG as a game mechanic. The following sections describe three games which are using PCG as a game mechanic.

Galactic Arms Race

Initially, a research project which is a perfect example of PCG-based core mechanics in a game (Evolutionary Games, 2014). Galactic Arms Race is a multiplayer space shooter game in which the player needs to complete tasks and missions to progress through the game (Evolutionary Games, 2014). To complete tasks and missions, players need to fly around in space and try to kill enemies with different weapons. A highlight of the game is the particle weapon system which is used to defeat enemies whereas weapons are entirely generated by a PCG algorithm using evolutionary AI algorithms to evolve and generate new weapons (Hastings et al., 2009*a*). The AI in Galactic Arms Race creates and evolves weapons based upon actions, strategies, and most used weapons of a player (Hastings et al., 2009*a*) (Hastings et al., 2009*b*). Besides, a player can possess three weapons at a time whereby new evolved weapons are continuously spawned in space and dropped by enemies (Hastings et al., 2009*b*). Consequently, players need to decide which weapons they want to use and thus feed the AI with information about their preferences (Hastings et al., 2009*b*). In this case, a player functions as the fitness function of the evolutionary algorithm used by the overall PCG system (Hastings et al., 2009*b*).

In short, a combination of PCG algorithms and evolutionary AI algorithms produces the novel weapon system which represents the core mechanic of Galactic Arms Race.

Inside a Star-Filled Sky

Inside a Star-Filled Sky is an almost entirely procedural generated game (Rohrer, 2011). The objective of a player is to progress through levels and try to reach one of the highest stages of progress to rule the leaderboard (Rohrer, 2011). Players need to fight enemies, collect items, power-ups or weapons to defeat enemies and get over to the next level. Each level is generated and can represent the inside of an enemy, item or another entity in the game which gives the player an infinite choice of possibilities on how to play the game. Moreover, players can move in or out of the recursively nested levels, and every collected item or killed enemy change the random seed for generating the next level (Smith et al., 2012). Players can even move into their character to increase their power and more than 165000 simple weapon combinations are explorable due to PCG (Rohrer, 2011).

Summarized, the core mechanic of Inside a Star-Filled Sky is heavily PCG-based and about exploring and progressing through generated levels with the help of generated weapons and items.

Endless Web

Is an entirely PCG-based game and thus uses PCG as its core mechanic (Smith et al., 2012). It is about fighting the nightmares in human dreams and rescue the trapped ones and thus releasing dreamers from their fears. The central objective of the game is to rescue six dreamers by exploring and make decisions on exploring new areas in the world which affects the parameters of the rescue progress and also the generation of new world parts (Smith et al., 2012). For example, if a player kills an enemy then depending on the configuration, it strengthens or weakens an associated challenge and furthermore changes the world (Smith et al., 2012).

So, Endless Web's core mechanic is about manipulating the generative space where players influence the changes of the world generation with their chosen decisions on how they are playing the game (Smith et al., 2012).

Other Games

Games like Black & White, Diablo, Dwarf Fortress, Elite, Eve Online, Roque, Spelunky or Minecraft are other good examples which made use of PCG and related mechanics as an essential part in their game.

3.3.2 Possible Core Mechanics

There are a few possibilities and use cases for PCG as mechanics in games. As seen, some core mechanics aim at weapon creation and progressing through generated space with the help of generated or altered "helper" mechanisms. Apart from that promising ideas are also other noteworthy ideas:

- One exciting idea is about using PCG for multiplayer games as a multi-instance PCG system given by Smith (2014). A game could use a central PCG system consisting of different and unique systems for each player, and every content generation would affect the PCG systems of other players (Smith, 2014). In that case, the core mechanic of the game would include influencing other players with each others PCG system to work towards a specific goal. For example, a use case would be a collaborative multiplayer game where each content generation causes a new content generated in the other's player space, and they must find a way to communicate how to achieve a mutual objective (Smith, 2014).
- There is much research done in the generation of quests as well. The research of Doran & Parberry (2011) introduced generated quests based on analysis of four existing massively multiplayer online RPGs and their quests. One could create a simulation game where

players need to create random quests by doing interaction with the PCG system in order to provide an AI agent with missions which need to be fulfilled to solve some other problems in the game and thus progress throughout the game.

- Another idea is to adapt existing and successful introduced game mechanics with a PCG-backend. For example, one could create a game like Tetris where the core mechanics of rotating a block generates new blocks. For example, every time the player rotates the block, the next block will be altered with evolutionary algorithms and changes their shapes to increase the difficulty. Moreover, the system could adapt its difficulty to the player's skills regarding the success or failure rate of using the new blocks to maintain an acceptable experience.

4 Machine Learning

ML is such a vast topic so that a detailed explanation of every type, approach, method, and model used in ML would go beyond the scope of this thesis. Therefore, the next sections will describe essential subjects of ML which can be useful to understand its further use in PCGML.

However, let us ask a question first: What is the difference between ML and AI and what is ML about actually? The fact is, there is no difference between ML and AI. In particular, ML developed from fields of research in AI and is thus a subset of AI. It concentrates on using mechanisms to learn from given data where data represents experience for a given problem. For instance, a famous example of ML is an application where the machine can distinguish between apples and pears with the help of a given dataset of features for both apples and pears (Yannakakis & Togelius, 2018).

4.1 Types of Learning Problems

ML consists out of three main types which all address a different kind of problem to solve and goal to achieve. One can classify types of supervised learning, unsupervised learning and Reinforcement Learning (RL) (Bonnin, 2017). Each of them functions as a solver to a specific task or problem where it fits best and creates the desired results.

4.1.1 Supervised Learning

This type of learning is a task-driven approach of ML (Bonnin, 2017). Its primary application is predicting or approximating data based on existing historical or empirical values where the answer to the problem is already known (Yannakakis & Togelius, 2018). The previously described problem of classifying apples and pears is a supervised learning problem which is solved by predicting the specific fruit or also referred to as a class (Bonnin, 2017). In this case, we would provide a sample set of real data with features for both fruits and classify each feature to a specific fruit. With this, the algorithm links the given features to a specific result and can predict those fruits based on a given test set (Bonnin, 2017). A training and test set could exist out of a bunch of pictures or other specific features described with numeric values in a table for each fruit. Hence, the reason why this is called supervised learning is that an ML algorithm is provided with data labels and therefore knows what to learn (Yannakakis & Togelius, 2018).

Typical applications for supervised learning are, for example, image recognition and classification, spam detection, pattern detection, speech recognition, natural language processing, sentiment analysis or forecasting (Bonaccorso, 2017).

Regression

A statistical process is the basis of this supervised learning technique where particular probability distributions of a given training data control the prediction output (Bonnin, 2017). In specific, a regression algorithm is processing independent and dependent variables of a given problem and build relationships between them which are furthermore used to predict the correct answers of a given unknown set (Bonnin, 2017). Independent variables are describing features and dependent variables the meaning or outcome of a regression problem. Usually, regression algorithms are applicable when the output values are constant prediction problems like for example, the predicted time of completion for a game level based on, e.g., the current player position (Yannakakis & Togelius, 2018).

Some use cases where concepts of regression algorithms can be applied are, e.g., for imitation and prediction of a player's behavior or player preference learning (Yannakakis & Togelius, 2018). Some popularly used algorithm for regression are the linear or polynomial regression, Artificial Neural Network (ANN) or Support Vector Machine (SVM). (Yannakakis & Togelius, 2018)

Classification

Addresses problems where classification transition of independent values into specific values is needed (Yannakakis & Togelius, 2018). The famous problem of classifying and distinguishing apples and pears from each other falls into this section.

Like regression algorithms, classification algorithms are applicable for imitation and prediction of a player's behavior, such as prediction of completion time (Yannakakis & Togelius, 2018). However, in this case, the possible outputs are specific values or classes like slow, average or fast instead of continuous values (Yannakakis & Togelius, 2018). Some popularly used algorithm for classification are, e.g., ANN, decision tree, random forests, SVM, K-Nearest Neighbor (KNN) or ensemble learning (Yannakakis & Togelius, 2018).

4.1.2 Unsupervised Learning

In contrast to supervised learning, the base of unsupervised learning is a data-driven approach where the algorithm has no information about the meaning or value of any sample and needs to infer it automatically (Bonnin, 2017). Hence, an unsupervised learning algorithm gets so-called unlabeled data without having a specific relationship to the target output and finds unknown structures and pattern in that set (Yannakakis & Togelius, 2018). Thinking about the apples and pears example, then an algorithm would try to detect that there are two different types or classes in the dataset instead of predicting if it is an apple or a pear.

Typical applications for unsupervised learning are object segmentation, similarity or pattern detection, automatic labeling, pre-training of supervised algorithms or preprocessing data such

as data compression, noise smoothing or outlier detection (Bonaccorso, 2017) (Yannakakis & Togelius, 2018).

Clustering

For example, solves the previous described apples and pears problem by clustering apples and pears out of the given training set with features of both fruits. Based on the trained knowledge due to the training set, it can differentiate new and unknown samples into either apples, pears or an entirely new class (Yannakakis & Togelius, 2018). In detail, clustering algorithms are looking for similarities between given features and values, and by doing this, they are inferring a relationship between them and thus separate specific classes (Bonnin, 2017).

Popularly used algorithms for clustering are, e.g., k-means, NNs or hidden Markov models (Bonnin, 2017).

4.1.3 Reinforcement Learning

In short, RL is a goal-oriented approach where an AI tries to reach a goal with the best strategy (Bonnin, 2017). RL uses so-called agents who are used to get feedback or specific states of an environment which is further used to learn and improve new decisions based on taken decisions (Bonnin, 2017). Inspired by the way humans and animals learn to take decisions, it aims at rewarding the algorithm for good behavior and thus leads it towards the best knowledge and output (Yannakakis & Togelius, 2018).

In general, RL makes use of a bit of supervision in the form of feedback for an action executed by an agent (Dangeti, 2017). Among experts, this feedback exists as the reward for actions in RL algorithms (Dangeti, 2017). The tricky part is that RL usually consists of following decisions and every chosen action — chosen out of a set of actions — by an agent is changing the environment which usually makes it difficult to train a model (Dangeti, 2017). Hence, an agent executes different decisions in a loop and is looking for the highest total reward for its sequence of actions since it will always want to increase its total reward. (Yannakakis & Togelius, 2018) With this strategy, an RL algorithm is getting better and better over time in solving a specific problem until it found the best solution.

During the last years, RL algorithms have been used to learn an AI how to play classical games, find the best strategy to win a game, learn an AI how to walk and many other applications (Bonaccorso, 2017). Algorithms include NNs, deep NNs, Q-Learning or Markov decision process (Dangeti, 2017).

4.1.4 Generative versus Discriminative Models

Another viewpoint to differ ML models is to group them into generative and discriminative models. Both of them have different applications and advantages. As the name of generative models suggests, their application is for data generation problems instead of simple label predictions

problems (Ghotra & Dua, 2017). This fact infers that there is a difference in their underlying learning approach which is the crucial point of the distinction (Ghotra & Dua, 2017). Discriminative models include NNs, decision trees, random forests or SVMs whereas generative models include Generative Adversarial Networks (GANs), Autoencoders (AEs) such as Variational Autoencoders (VAEs), Long Short-Term Memory Networks (LSTMs) or recurrent NNs (Ghotra & Dua, 2017).

In particular, the primary distinguishing feature between the models is their approach on how to learn the relationships of the training attributes to either the labels or all the other attributes as a whole. With this said, discriminative models try to learn the conditional relationship between the features and the labels, denoted as $P(Y|X)$, whereas generative models try to learn the joint probability distribution of the whole data and then infer the conditional relationship, denoted as $P(X, Y)$ (Ghotra & Dua, 2017).

4.2 Development

Developing an easy to use AI needs to be a well-structured and planned process. Especially, when using ML as a focus of interaction, it is even more important to be well-prepared. For this reason, this chapter addresses some design considerations about developing user-friendly AI systems in games, as well as standard pitfalls to keep in mind, standard procedures for data preprocessing and possibilities of how to use ML algorithms in two commonly used game engines.

4.2.1 Design Considerations

Someone cannot just develop a new novel AI system and expect to create a whole new experience — it needs to have a thoughtful plan how to achieve new experience (Eladhari et al., 2011). Firstly, it is useful to think about the MDA framework, which was described in Chapter 2, because an AI-based game is often tightly integrated into its game mechanics and therefore it is necessary to make no mistakes in the first place to fulfill playability and emerging experience (Eladhari et al., 2011).

Here are some critical considerations for AI-based games described by Eladhari et al. (2011) which are also worth noting for implementing an ML-based AI system:

- First, develop a rough design of the game system and then model the AI system upon that game system or vice versa (Eladhari et al., 2011). In particular, for AI-based games, the AI system should support the designed core experience of the game (Eladhari et al., 2011).
- An AI should provide possible exploration and allow a player to experiment with it which means it should be robust enough and not lead to game crashes or misbehavior (Eladhari et al., 2011).

- Avoid a too mechanical and unnatural player experience with an unpredictable system (Eladhari et al., 2011).
- The environment of an AI-based game should be observable for an AI system which means it should be able to access states of different game entities at any time (Eladhari et al., 2011). Consequently, game states should be described in a way so that an AI system can easily access and read it (Yannakakis & Togelius, 2018).
- A system should be as accessible and transparent as possible for a player so that a player will not be overwhelmed by the system and its possibilities (Eladhari et al., 2011). Notably, an interactive ML system as an AI system in the game could be a confusing thing for a player if it, e.g., exposes too much information (Eladhari et al., 2011). That is of particular importance when using RL as the backbone AI agent algorithm.
- Provide ways for emergent gameplay with the help of the AI system with, e.g., possible strategies which can be applied to the system (Eladhari et al., 2011).

Lastly, the discussed issues and considerations in Chapter 2.3 also apply for the implementation of an ML-based AI agent.

4.2.2 Common Pitfalls

Following pitfalls are commonly occurring problems when implementing and training ML models for a specific problem.

Under- and Overfitting

ML models are used to approximate unknown output based on the given training data (Bonaccorso, 2017). When talking about fitting, then it is referred to fit a model to a given training set and its features which are used to train the model. This set can consist of many independent variables and different entries which can create some issues:

- **Underfitting**: Happens when the training set for the model consists of too less information or independent variables (Bonaccorso, 2017). In that case, the model is not able to capture the dynamics of the values (Bonaccorso, 2017).
- **Overfitting**: Is exactly the opposite of underfitting. When a model is over-fitted, then it is fed with too much information and variables so that it is not able to generalize the dynamic relationship of variables during the training (Bonaccorso, 2017).

Therefore, a right amount of information for the training of a ML model should be provided to achieve an excellent fitted model. A method to check and prevent the model from being over or under-fitted is the technique of validation such as cross-validation which helps to detect those problems (Bonaccorso, 2017).

Curse of Dimensionality

This problem often occurs when the training set is smaller than the number of feature variables, or also called the dimensions of a set, which are used to train a model (Bonaccorso, 2017). In this case, if the number of features increases then the performance of the model gets dramatically reduced (Bonaccorso, 2017). Possible ways to prevent and solve this problem are, e.g., a decrease of dimensions or providing more training data (Bonaccorso, 2017).

4.2.3 Data Preprocessing

A standard procedure in ML is the technique of data preprocessing so that data values are consistent with each other. Table 2 shows a simple example data table which could be usable for the systems training. The table's data can contain any discrete or continuous values which could originate from any decoded non-numerical data.

$\mathbf{V_1}$	$\mathbf{V_2}$	...	$\mathbf{V_n}$
2.1	A	...	100
5	B	...	64
...	...	...	...
-0.958	A	...	986

Table 2: ML training data example.

Encoding

For example, if the dataset provides categorical data like the values for $\mathbf{V_2}$, as shown in Table 2, then this data needs to be encoded so that it does not differ from the other data values. This procedure is a standard procedure in the data preprocessing for training an ML model and can be done with so-called "One Hot Encoding." Table 3 shows an example of what an one hot encoder does with categorical data as provided by $\mathbf{V_2}$. The new values generated by the one

$\mathbf{V_2}$		$\mathbf{V_2{}^1}$	$\mathbf{V_2{}^2}$	$\mathbf{V_2{}^3}$
A	$\longrightarrow$	1	0	0
B	$\longrightarrow$	0	1	0
C	$\longrightarrow$	0	0	1

Table 3: One hot encoder example.

hot encoder replace the old categorizes values, and the dataset is ready to go. It is essential to keep in mind that this encoding can cause the curse of dimension if there are too many categories in the data.

V_1	$V_2{}^1$	$V_2{}^2$	$V_2{}^3$	...	V_n
2.1	1	0	0	...	100
5	0	1	0	...	64
...	...	...	...	...	...
-0.958	0	0	1	...	986

Table 4: Encoded training data example.

Feature Scaling

Feature scaling is a crucial process in data preprocessing. It helps to make unbalanced datasets more manageable for the training and especially during backpropagation or error correction (Bonnin, 2017). The occurring problem is that most optimizer functions in ML, for example in KNN, are using Euclidean distance to measure the cost of, e.g., a prediction which is then used to calculate the delta for backpropagation. Now, the main problem with taking the Euclidean distance is that it is getting easily biased with, e.g., significant values in the dataset which furthermore dominate all the other ones. For example, the last attribute in column $\mathbf{V_n}$ in Table 4 is an immense value which would create a more significant delta for all other attributes during backpropagation. Now, there are two commonly used methods to avoid this problem:

- **Standardization**: Aims to balance the dataset so that its distribution is closer to a normal distribution (Bonnin, 2017). The formula for standardization is: $z = \frac{x - \mu}{\sigma}$.
- **Normalization**: Aims to transform the distribution of the dataset into a range between 0 and 1 (Bonnin, 2017). The formula for normalization is: $x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$.

Applying normalization to the example dataset in Table 4 results in a dataset shown in Table 5.

V_1	$V_2{}^1$	$V_2{}^2$	$V_2{}^3$	...	V_n
0.513	1	0	0	...	0.039
1	0	1	0	...	0
...	...	...	...	...	...
0	0	0	1	...	1

Table 5: Normalized training data example.

4.2.4 Game Engine Plugins

Indeed, all the different ML models or the one which applies to the game prototype of this thesis could be self-implemented. However, there are other people out there who have already implemented and improved their implementations which are a good starting point when implementing

a PCGML game mechanic. As discussed at the beginning of the thesis, it is going to focus on an implementation in commonly used game engines which is the reason why to envisage the two most used free to use game engines nowadays.

Unity

Unity made an incredible effort in its research and use of ML in their engine in the last year (Unity Technologies, 2018*a*). They recently released a beta version of an open-source ML agents plugin that enables games to train an AI with RL, imitation learning, neuroevolution or other ML methods (Unity Technologies, 2018*b*). The base of the plugin is Google's open-source ML framework TensorFlow (TF) (Google, 2018) which is accessible via a simple-to-use Python Application Programming Interface (API) (Unity Technologies, 2018*b*). They are also providing essential instructions and documentary for the usage of their plugin which is a promising, definite and significant advantage when implementing the PCGML game mechanic.

Unreal Engine 4

At the time of writing this section, there is no available officially announced ongoing plugin or feature development for ML in Unreal Engine 4 (UE4). However, there are others who were concerned about the missing feature of ML in UE4. For this reason, Kaniewski (2018) created an open-source TF plugin for UE4 which implements a Python API accessible interface based on a fork of a UE python plugin. Indeed, there are also some other plugins available via UE4's marketplace such as a Q-Learning plugin, but there do not look that promising as Kaniewski's TF plugin. Also, provided documentation and guidance on how to use the plugin leads it to a tolerable option besides to Unity's ML agent plugin.

4.3 Game Mechanics

It is unclear which games used specific methods of ML for a game mechanic because there are not any research paper or published articles about, i.e., game companies who talked about using ML in their mechanics. Nevertheless, there is a research focus on AI-based games, including game mechanics which can be adapted with ML and is therefore used to come up with ideas for ML game mechanics.

4.3.1 Current Games

Two games which use AI as a core mechanic were already described in Chapter 3.3.1 and are the games Endless Web and Galactic Arms Race. Both games make use of a PCG and AI mixture mechanic where the AI generates new world segments based on game world states (Smith et al., 2012) or new weapons based on player preferences (Hastings et al., 2009*a*). These

games are going to be first reference points and inspiration when implementing a PCGML game mechanic. Other games which used ML techniques as a mechanic were, e.g., Black and White or Creatures (Champandard, 2007).

Black and White

The player is in control and owns a creature who can be trained to do things for the player. They used techniques such as decision trees and NNs to train and learn it the prediction of a player's action (Dalmau, 2005) (Champandard, 2007). In particular, the belief-desire-intention approach was used to implement the creature's behavior (Champandard, 2007). The creature's training happens due to a player's reaction upon the creature's executed actions which therefore follows the rules of RL where the creature would like to achieve the best total reward for its behavior (Dalmau, 2005).

Creatures

Creatures is a game where players need to hatch animals and try to teach them how to behave and survive in their world (Champandard, 2007). As well as Black and White, Creatures also used NNs to teach and learn animals how to behave based on a player's input (Champandard, 2007).

4.3.2 Possible Mechanics

Treanor et al. (2015) introduced nine design patterns, summarized in Table 6 which illustrates ways to develop game mechanics based on AI techniques and furthermore to create AI-based games. Following sections describe some examples of possible mechanics or core mechanics with ML, based on Treanor et al.'s design patterns.

Artificial Intelligence as a Role-Model

For example, a game in which a player needs to follow an AI to solve quests or missions. Based on the players progress in each quest and imitation level, the AI adjusts its solution of solving the current or a new problem and provides the player with either a less competitive or harder solution and reward. Hence, player experience is adapting due to player behavior learning and predicting an appropriate difficulty level. For instance, a quest could be following or remembering the path of the AI to a treasure whereas built-in traps are crossing the way. Hypothetically, if the player cannot remember and succeed in the path after, e.g., three times, then the AI shows a new and less challenging path to a less rewarding treasure.

Pattern Name	Description	Role of Player	Role of AI
Visualized	Visualization of AI system states so that gameplay revolves around state manipulation	Observation and manipulation of an AI system	Shows states and gives information
Role-Model	Player needs to imitate AI agents	AI system imitation	Shows actions and behavior, e.g., as a puzzle
Trainee	AI system needs to be trained to perform gameplay tasks	Teacher for AI system	Learns desired behavior
Editable	Player needs to change elements of an AI agent which are related to gameplay	Observation and manipulation of the AI	Handles changes and shows a new behavior
Guided	Player partly assists or guides an AI agent who is threatened by a problem	Guidance and management of the AI	Acts on its own but mostly does what the player wants
Co-Creator	Player and AI work together as equal partners	Learns how to play the game with AI assistance	Acts as co-creator and assistance for the player
Adversary	Player needs to defeat an AI system	Adapts to an AI and defeats it	Tries to defeat the player
Villain	Player needs to defeat an AI system whereas AI system does not want to defeat the player	Adapts to an AI and try to defeat it	Acts as villain and mob the player
Spectacle	AI which implements a complex system and the player needs to observe, interact or overcome it	Observation, interaction or manipulation of the system	Spectacles and simulates the system

Table 6: AI-based game mechanics design pattern (Treanor et al., 2015).

Artificial Intelligence as a Trainee

Is one of the most promising design pattern since it revolves around teaching an AI different things whereby it fits perfectly for using ML methods. As introduced before, the games Black and White and Creatures are using precisely this kind of pattern for implementing their creature behavior. A game could use a mechanic the same way and introduce new mechanics on how to interact and use the AI agent by teaching it how to behave in specific situations. A simple idea would be a teaching game in which the player needs to educate fighters to defeat enemies. Alternatively, a tower defense game in which the player uses defense entities to teach an ML agent first target preferences.

Artificial Intelligence as a Co-Creator

An exciting base idea of a co-creator AI game would be a split-screen game in which the player plays on one side and the AI agent on the other side. The gameplay would be symmetrical, and everything a player does, the AI will try to imitate. In doing so, the player needs to do things to complete challenges whereas done actions affect the AI agents space and vice versa. For example, a challenge could be pushing a button which only exists in the space of the AI

but opens, e.g., a door in the player's space. Therefore, the player would need to behave in a specific manner to lead the AI towards the button to progress through the game.

5 Procedural Content Generation via Machine Learning

A purpose for research in AI-based PCG methods like PCGML was to invent algorithms which can create and generate content of the same type and style based on existing content with or without human involvement (Yannakakis & Togelius, 2018) (Summerville et al., 2017). In general, Summerville et al. (2017) defined PCGML as ”the generation of game content using ML models trained on existing content.”

On this account, different research was and is carried out to generate music, sound effects, images, textures, worlds or levels (Yannakakis & Togelius, 2018). Research showed that ML-backed PCG methods are working well for music and images but are facing some issues when it comes to world or level generation because of its necessity of playability to create an adequate experience for the player (Yannakakis & Togelius, 2018). Thus, the generation of gameplay content raised challenges, demanding further research for solving such problems (Yannakakis & Togelius, 2018). Therefore, PCGML approaches were not used in games yet. Another reason is that there is often not enough game content available to train a PCGML model for a specific content generation problem (Yannakakis & Togelius, 2018).

The next chapters introduce the difference to usual PCG, use cases of PCGML, some developmental features used in other research and explains how PCG and ML work together to generate new content in an example. Overall, this chapter is not going into many details since all essential cornerstones of PCGML where discussed and introduced in the previous chapters.

5.1 Difference to Procedural Content Generation

Usual PCG and PCGML have in common their hand-made algorithms, parameters, and constraints made by developers (Summerville et al., 2017). Typically, designers would use existing game data and content as a foundation for new ideas and develop generators for creating new material on top of that inspiration, whereas PCGML itself can get utilize existing data to develop new content and thus assist the designer (Summerville et al., 2017). So, the basic idea is to train the PCGML model on various but similar game content to produce new one (Summerville et al., 2017).

In contrast, different PCG algorithms might use ML for evaluation, but their content generation process entirely relies on their specific domain space rather than a trained model space (Summerville et al., 2017). For example, experience-driven PCG relies on models of player experience whereas PCGML models can use existing balanced domain-specific content to create

new one (Summerville et al., 2017).

5.2 Use Cases

The focus of PCGML is to generate new content based on training with existing content. For this reason, PCGML can be used for adaption, reparation, evaluation, critique or analysis of new content but also for general PCG tasks like autonomous generation, co-creative, and mixed design or data compression (Summerville et al., 2017).

- **Autonomous Generation**: Is one of the primary application for PCGML because it can generate game content without human input which is especially useful when online content generation is needed (Summerville et al., 2017). Also, a designer could create, e.g., a set of levels and for a model's training to create new and novel but similar levels with the help of the PCGML system (Summerville et al., 2017).
- **Co-creative and Mixed-initiative Design**: This use case focuses on the collaboration between designers and the PCGML algorithm (Summerville et al., 2017). For instance, the algorithm could generate new content based on provided examples as a draft for the designer who could polish and finish the drafted content afterward.
- **Data Compression**: PCGML offers significant potential for data compression with the help of ML. Extracting specific dimensions of game content can save data size and would allow more efficient storage (Summerville et al., 2017).
- **Recognition**: This is where PCGML stand out from the conventional PCG algorithms. Its recognition capabilities make it possible to analyze, evaluate, adapt, critique or repair either existing, new designed, player created or algorithm generated game content (Summerville et al., 2017).
- **Repair**: In particular, reparation of a designed or generated content can be a useful tool. For example, during generation or learning time, the algorithm could check for, e.g., playability of a level and repair it immediately or suggest a fix (Summerville et al., 2017).

5.3 Development Considerations

There are some unique development considerations to take when someone wants to make use of PCG via ML for game elements. In general, every design consideration, possible issue, pitfall and best practices described and discussed in Chapters 3.2 and 4.2 will apply for implementing PCGML algorithms. Furthermore, an initial examination of the data representation for the training set during the ML model's training could be helpful. On this account, the research of Summerville et al. (2017) showed that specific datasets with significant difference share possible training methods to obtain excellent and practical generation results. Summerville et al.

(2017) further organized their used datasets into sequence-, grid- and graph-based data. They made that distinction because it fitted best with their main problem addressed in the paper which was primarily about generating levels.

5.3.1 Machine Learning Models

Because PCGML generates new content based on an existing one, it seems reasonable that the models which fit best for most of the PCGML tasks are generative ML models.

This is confirmed by the fact that the primarily used training models in Summerville et al.'s research of using PCGML for, e.g., level generation, were based on the paradigm of NN and are models such as convolutional NNs, LSTMs, Markov models like N-grams or multi-dimensional Markov chains, Bayes models, AEs, clustering or Matrix factorization. Nevertheless, not all of them are pure generative models which means that discriminative models can be a better fit sometimes.

5.4 Development Example

There is no right or wrong proceeding in the generation process for PCGML. PCGML is a very new paradigm in the content generation community, and its research results drive the right or wrong approaches. With this in mind, the PCGML generation process can take shape in any form, and someone should do what fits best for a given problem.

This section explains a development example of a generation process for data with depending attributes realized with discriminative models as ML backend. However, Chapter 7 provides the detailed development walk-through of a PCGML system. For this example, a data's structure example of a PCGML system provides the Figure 2. As it is a sequence of data, there is a connection between all variables. In the figure, the node "S" represents a start value, and all other nodes represent independent attributes or features. Figure 3 shows dashed rectangles

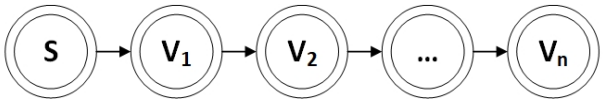

Figure 2: Basic training data for a PCGML system.

around the attributes which indicates separately trained ML models for each attribute pairs. It is

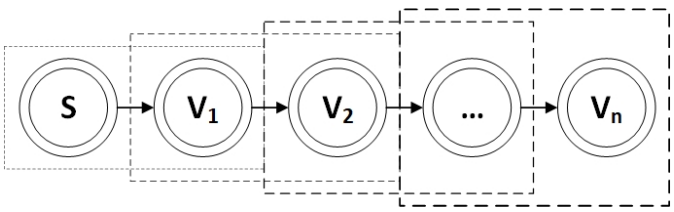

Figure 3: Basic training data for a PCGML system with shown ML models.

worth noting that values do not need just one predecessor but can have multiple which would change the dashed rectangles, ML models and their associated variables shown in Figure 3.

5.4.1 Data

First of all, it is necessary to come up with efficient and sufficient data for the system's training. Depending on the variables, the model requires more or fewer data. For example, a data structure with around ten features would probably need at least 100 data rows to train the ML model properly. In particular, there is an unofficial rule of thumb that the dataset should be ten time times bigger than the provided number of parameters. As soon as the data is available, preprocessing for the training should be applied, following the steps described in Chapter 4.2.3.

5.4.2 Training

After provision and preprocessing of the dataset follows the training. Training can happen with different training models as described in Chapter 5.3. For example, the use of an ANN could be a suitable training model for a given example dataset. In the scenario of previously described training data, the NNs would be trained for each pair of values to output the next value in the sequence. Figure 5 shows an example of a simple feed-forward NN where the start value and the first variable are used to predict the second variable. The neurons in all NNs shown in this thesis will follow the color scheme introduced in Figure 4.

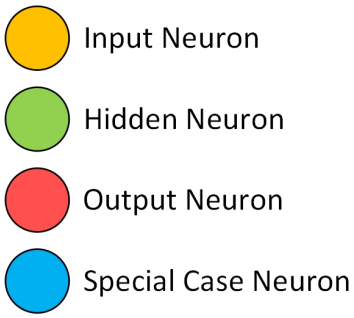

Figure 4: Neuron legend for all NNs in this thesis.

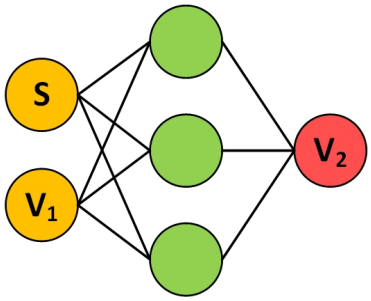

Figure 5: Basic feed-forward NN to output a variable based on given input variables.

In specific, the training of several NNs for every following variable of the data sequence is necessary to output and generate a complete result. Another possibility to use NNs would be to make use of the so-called "Softmax" function which outputs the categorical distribution or probability distribution of a variable. With this probability, it can generate and train the following variables. Figure 6 shows this kind of NN with an example probability distribution in the output layer.

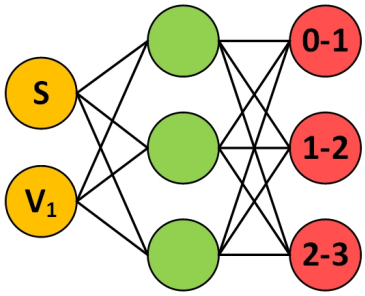

Figure 6: NN with a probability distribution example in the output layer.

5.4.3 Generation

The next step in developing a PCGML system after the correct training of the model is the generation process. This process can differ depending on the used training model, given data and data representation. For example, the system could generate the first variable and get the other ones from the model. Alternatively, it could generate any variable in the sequence and retrieve the others from that one. Alternatively, it could make use of the probability distribution output by the Softmax function which leads the variables and generates a new piece of game element. As mentioned at the beginning of this chapter, there are no standards in the PCGML generation process, and someone can make use of every possibility which is provided by the system.

6 Possible Game Mechanics

The initial ideas on how to use PCGML as a game mechanic were created by Summerville et al. (2017). They came up with the following seven different ideas based on the AI-based game design pattern described in Chapter 6:

- **Role-Model**: "A PCGML system replicates content which is generated by players of various levels of skill or generates content suitable for players of certain skill levels. New players are trained by replicating the content or by playing the generated content in the form of generative tutorial." (Summerville et al., 2017, p. 13)
- **Trainee**: "The player trains a PCGML system to generate a piece of necessary content (e.g., part of a puzzle or level geometry)." (Summerville et al., 2017, p. 13)
- **Editable**: "Rather than training the AI to generate the missing puzzle piece via examples, the player changes the internal model's values until acceptable content is generated." (Summerville et al., 2017, p. 13)
- **Guided**: "The player corrects the PCG system's output to fulfill increasingly difficult requirements. The AI, in turn, learns from the player's corrections, following the player's guidance." (Summerville et al., 2017, p. 13)
- **Co-Creator**: "The player and a PCGML system take turns in creating content, moving towards some external requirements. The PCGML system learns from the player's examples." (Summerville et al., 2017, p. 13)
- **Adversary**: "The player produces content that the PCGML system must replicate by generation to survive or vice versa in a "call and response" battle." (Summerville et al., 2017, p. 13)
- **Spectacle**: "The PCGML system is trained to replicate patterns that are sensorically impressive or cognitively interesting." (Summerville et al., 2017, p. 13)

These ideas were used to come up with more ideas on how to use PCGML as a game mechanic in modern video games.

6.1 Concepts and Development Evaluation

Each of the following chapters describes an idea of a PCGML mechanic followed directly with a development related evaluation. It is to say that in general, the central issue in the implementa-

tion of the PCGML system will be the training data for the ML model rather than the PCG output itself. Proper training for the ML model requires the right amount of training data to function as intended. The size of the right dataset can differ from application to application based on the given values as dependencies for an output.

Moreover, each idea has the advantage of PCGML that it can generate similar elements as used during training time, the possibility of changing the training set and thus the output. Also, there is no need for finetuning to generate a broad range of elements in contrast to standard PCG algorithms. For this reason, as it is the implication of using PCGML, these described advantages are not mentioned in the evaluation. Also, the provision of playability is a necessary criterion which contributes equally to each game mechanic.

6.1.1 Rules and Behavior

A possible idea is a PCGML system which generates rules for a strategy game. A generic and modular strategy game offers the perfect foundation for generating its internal rules and behavior based on its internal elements. In particular, a strategy game mostly exists out of different sequences of actions emerging from new elements in the game which furthermore lead the game in a specific direction. All of those available actions have a different effect in the game whereas the effects can be changed with other appropriate effects causing different changes during the game. More particular, the design of a strategy game can introduce more game elements than used in the game itself. This kind of game element preselection often arises in cardboard games where players need to choose the available cards in advance which affects the flow of the game. For example, if a game provides the players with 25 different elements and just ten elements are used in the game, then it provides, regarding the mathematical theorem of combinatorics, 3268760 possible ways to play a game. This formula only applies if the introduction of each element happens after a specific time with no reuse of elements. Moreover, specific conditions for, e.g., the specific timeslot of an element's revelation can also be a part of the generation process which emerges a more dynamic game.

In this case, working configurations of the game provide the training set of the PCGML system with which it learns the different dependencies between elements and effects to each other.

Pros and Cons

- A proper learned model can be saved to generate the same sessions again.
- Providing more designed content in the training set offers more possibilities for the gameplay.
- It needs a sophisticated game design with balanced dependencies between the available elements.

- Possible complications in the training of the ML model because elements can consist of many values which create complex dependencies.
- The game design must include a broad range of game elements with harmonization between each other and proper design and balancing.

6.1.2 Changing Weapons

Game mechanics often share a connection with some interactable objects. In the case of a First Person Shooter (FPS) game, different weapons like guns, rifles or shotguns represent these interactable objects. Moreover, different weapons often result in different mechanics such as particular tactics. With this in mind, another idea of using PCGML as a game mechanic is its use as a weapon generator. In specific, an FPS game with generated weapons where providing different weapons at the training time results in different ways to play the game. A virtually created weapon can have different properties, such as weight, recoil, firing rate, ammunition and others which form the base to generate new weapons procedurally. Furthermore, as weapons need ammunition, the generation of new bullets and projectiles can also be part of the PCGML system.

For example, a game could be an FPS game with generated weapons from a pre-trained PCGML system whereby the model will be fed with the new generated weapons during runtime and furthermore specializes in a specific direction. After a bunch of time, the system only generates a specific weapon, and that is where the game or session would end.

Pros and Cons

- Open source FPS projects and FPS communities are offering a good base of training data to build on.
- Properties of weapons and ammunition are not quite complex to train the ML model.
- It does not matter if the generated weapon works in every environment. Different weapons offer different advantages in different environments.
- There is no need for sophisticated game design.
- The diversity of weapons can be minimal if there is no training set with enough weapons because of the non-complex structure.

6.1.3 Changing Powers

Often, the central mechanic provided in most RPGs is fighting with a sword and other weapons or by applying magic spells. Moreover, the weapons often possess a magic power as an advantage or disadvantage. These powers and the magic spells themselves can be part of a

generation process powered by a PCGML system. The system could generate new weapons or spells affecting the duration, spell category, continuing effect on the enemy or player, energy cost and other similar and often used properties. Providing useful weapons with advantages or disadvantages or some magic attacks as an example would serve as the training of the system.

Pros and Cons

- Magic attacks and spells do not need to be generally effective against enemies because their tactical use forms their effectiveness in the game.
- There are no general rules of what and how spells and magic attack appear in games, thus they can be anything.
- Often, they do not consist of complex structures which should make ML model training easy.
- It needs proper designs of balanced and tweaked spells.
- Visuals should also be part of the generation process because there is no universal usable appearance for magic attacks.

6.1.4 Solver Weapon

The solver weapon is an idea based on the co-creator pattern described in Chapter 6. It is about a game mechanic of using a generation weapon to progress through the game. The weapon's purpose is to help the player in particular conditions such as solving puzzles with a missing piece in it. The highlight of the game mechanic is the required drawing interaction to generate a missing piece. A player needs to draw the missing piece on a given "drawing panel" connected with the weapon which tries to replicate the figured piece. Drawings could consist of big solid circles indicating corners and lines representing edges. The detection of the drawing involves object detection via ML which forwards detected chunks to the PCGML system. For example, introductive gameplay could be to find or generate the missing key for a door whereas the player is aware of the lock's shape.

This idea could work as follows:

1. Different input with different drawings of puzzle elements or geometric shapes depending on the style train the ML model. The system is then able to detect essential points on the drawing and suggests what the drawing would be.
2. The PCGML system detects the drawings with any object detection algorithm. As the system has obtained the chunks from the detection algorithm, it uses each chunk data to generate a fragment of geometry and finally combines all of them into a combined shape.

Furthermore, an extension of this idea is able with the trainee and guided pattern where the player would provide the system with feedback if the PCGML system outputs the wrong object. In this case, it would learn the specific drawing conventions of a player and adjust itself to the player.

Pros and Cons

- Training set for each model can be easily created by hand.
- The concept is straightforward and easy to learn.
- Virtual reality offers an easy-to-use drawing mechanic and fits perfectly.
- It needs two trained ML models. One for the object detection and another for the generation of puzzle objects.
- It needs the design of proper puzzles where the mechanic of generating objects is applicable.

6.1.5 Defeat of the Enemy

The PCGML system tries to defeat the player, and one or more players try to defeat the enemy. The adversary design pattern described in Chapter 6 where the role of the AI is to defeat the player forms the base of this idea. In this case, the system is trained on simple player behavior such as movement paths or attack preferences in a specific environment and uses this knowledge to overcome the player. Moreover, it would also be able to find a new behavior pattern during the fight. In short, the output of the system consists of new tactics, behaviors or attack sequences. Furthermore, the PCGML system could categorize used behavior regarding their success or failure and learn from these outcomes to increase its accuracy.

An example scenario would be that an AI has the intention of applying a critical hit to the player. In doing so, the PCGML system generates, e.g., a possible path to fulfill the desire of hitting the player based on, e.g., both the players and AI positions and the pre-trained knowledge.

Pros and Cons

- The training input for the ML model can be real data of fights with a standard AI enemy, extended with randomized values.
- Possible improvement of the ML model during runtime with system feedback.
- PCG generation and especially an interactive modification of terrain can be complicated.
- It needs a well-balanced training set for not overwhelming the player with a too tricky enemy.

6.1.6 Caught in a Thunderstorm

A game which revolves around a player which is in control of a thundercloud and has the objective to destroy cities or enemies with the power of lightning. In this scenario, a PCGML system would generate a pattern for the diffusion of lightning and impact points based on different weather conditions, cloud height, enemy or building position, physical materials and similar properties. This idea offers its potential as a thunderstorm simulation game supported by empirical or real weather data as the training set for the PCGML system.

Pros and Cons

- Pattern generation is easy to implement with, e.g., the L-system.
- Training of the ML model is possible with statistical, empirical and real data.
- It needs a proper game design with a proper sense of using the mechanic.
- The proper simulation of weather conditions and changes can be too complicated to implement.

6.1.7 Train to Progress

A player is forced to play with an empty and untrained PCGML system. The system can generate an essential part of the gameplay but needs to be trained first to activate its full potential of creating necessary or missing elements. In this case, a player's primary objective is to train the PCGML system to progress towards a given goal. For instance, a puzzle game where the central puzzle is the system's training to generate, e.g., some missing piece. A level could contain hidden objects which need to be found first to train the system correctly.

For example, if the goal would be to unlock a door with a generated key, then the player needs to find sample keys first to train the system with those which are similar to the lock's key. If there isn't any success in generating the right key, then the player can reset the internal model of the system and try it again with another pair of sample keys. This principle also offers its application with other kinds of objects and is mainly based on the trainee design pattern.

Pros and Cons

- The ML model does need pretraining because it is done in the game by the player.
- The creation of puzzles does not need to consider the PCGML system because the puzzle forms the base of the system's implementation.
- It takes time to train the model by the player correctly, and it can be frustrating if the model does not learn properly.
- Each puzzle element needs its design and its PCGML system.

6.1.8 Building with Assistance

The mechanic revolves around a PCGML system, trained to generate incomplete game elements for a specific goal. For example, a building game where the PCGML system never outputs the correct element which is currently needed. On the other hand, the player can improve and overhaul the generated elements. All these changes made by the player will change the internal model to guide it to a desired generation output direction. Thus, the guide design pattern with the focus on guidance is the base of this idea. The training of the PCGML system would consist of examples of incomplete but usable and modifiable game elements.

Pros and Cons

- The PCG is easy to implement since it just needs to generate objects proper to the game corpus.
- There is a margin for generation errors because it revolves around guidance of the PCGML system.
- The generation of usable and extendable elements need to be secured.
- The model takes a while to get well guided, and it can be frustrating if the system does not generate anything usable at the beginning of the game.
- It needs an interaction system with generated objects.

6.1.9 The exploring Co-Worker

Another idea based on the co-creator pattern could be, e.g., a strategy game where the player's focus is to mine, build and create strategic plans to overcome his or her opponent. Moreover, to evolve in the game, the player possesses a PCGML system which is used to explore and conquer new sections of a map. This system would work entirely on its own and functions as a helper and teammate for the player. The exploration process of a new section consists mainly of creating new terrain by the system. In this case, the system is trained to generate new chunks or section of unexplored terrain mainly based on the player's progress, mined resources, and similar elements.

Pros and Cons

- The PCGML implementation focuses only on the exploring task.
- The terrain generation can be based on the game corpus and does not apply to specific rules. Thus, the training of the ML model with sample terrains should be easy.
- PCG generation and especially an interactive modification of terrain can be complicated.
- The terrain needs to be interactable with the player's mechanics to build and mine.

6.1.10 Observe and Learn

A PCGML system which is trained to generate and modify terrains including a particular path to reach a specific goal. The highlight is that the system does not always generate possible ways, demanding the player to change the properties of the system influencing the terrain, the level, the map or the world. In doing so, the player is forced to observe the interactive and smooth changes in the system to reach the current goal. The visualization and editable design pattern form the base for this idea.

Pros and Cons

- Terrain generation can be based on the game corpus and does not apply to specific rules. Thus, training of the ML model with sample terrains should be easy.
- There is a margin for errors since it is the system conception to work faulty and the player's task to interact with the system to reach the goal.
- PCG generation and especially interactive modification of terrain can be complicated.
- The necessity of "incorrect operation" for challenging terrain generation needs to be secured.

6.1.11 Express Yourself

An application as a game mechanic in a point and click game could be a PCGML system, trained to generate new storylines, events, choices and outcomes based on already made decisions. The system is trained on designed stories and learns the dependencies to create new and novel storylines with individual possible choices and outcomes. Like in the idea described in 6.1.1, the designer can provide more game elements than used in the game which provides the system with a variety of possibilities on how to compose the game. The spectacle pattern is the base of this idea of a PCGML generated story.

Pros and Cons

- As described in the first idea about "Rules and Behavior, the same type of considerations apply to this idea about generating whole storylines with interaction.

6.1.12 Big Boss Helper

This mechanic is based on the villain pattern but here the AI rather mobs players than tries to defeat them. This kind of design pattern fits almost perfectly in, e.g., an RPG big boss fight with more than one Non-playable Character (NPC) to defeat. For example, two NPC's must be defeated to win the fight and obtain a unique item. One of the NPC's applies mainly damage,

and the second one protects the first one. This helper NPC is represented by the PCGML system which is trained on player behavior such as attack preferences or movement behavior in a specific level or environment to generate or adapt the terrain or level, keeping the player away from the main enemy.

Pros and Cons

- The same pro and cons described for the mechanic "Defeat of the Enemy" apply to this concept since their PCGML systems are operating in the same manner.

6.1.13 Figure it Out

A possible application in a multiplayer game could be a mechanic in which each player possesses a PCGML system for generation purposes. However, the challenge in the game is that each generation of the system changes the other's systems output, and they need to work together towards a specific goal. For instance, a puzzle game where players need to figure out the connection between their system. The PCGML system would be trained with a specific game corpus and enables interaction based on the editable pattern. One player changes parameters which subsequently influences the other player, all while they need to generate an object combined from each system's output. Parameter adjustment can either be done directly or indirectly by moving stones around in a room and find their right position. Furthermore, this idea is also applicable in a single player puzzle game without the need of another player.

Pros and Cons

- Creation of puzzles does not need to consider the PCGML system because the system revolves around the puzzle.
- The training of the ML models should be easy since the system is modeled based on a puzzle.
- Each PCGML system focuses on its piece of the puzzle element.
- It needs training for each ML model and puzzle.
- Each puzzle element needs own design and own PCGML system.
- The connection between PCGML systems will be complicated; therefore it needs a proper design of the system's connection.

6.1.14 Novel Vehicles

Another possible application for a PCGML system can be the generation of cars or vehicles as they also consist of many properties and parameters. The finetuning process of these proper-

ties is usually a lengthy process which is a perfect field of application for an ML trained PCG system.

Pros and Cons

- The generated vehicles can decide the kind of game; therefore it does not matter what the system's output is as long as it secures playability.
- Vehicles are not limited to cars; therefore it can be any wheeled vehicle.
- The training set needs the designs of many vehicles so that the system can learn the complex dependencies of a vehicle.
- The design of vehicles often involves a long process of balancing.

6.2 Summary

The ideas how someone can make use of PCGML as a game mechanic show that there are many possibilities which differ in their implementation complexity. Appendix A provides a summary of each PCGML game mechanic training data and output.

Many of the ideas are about generating items used as a game mechanic, but PCGML also offers the possibility to generate any item such as quests, missions or challenges. Furthermore, the implementation of any progression mechanics is also applicable via PCGML. It just needs the right idea and a training set with an appropriate data representation to properly train the ML system.

6.2.1 Game Mechanic for the Prototype

As already mentioned, all the introduced ideas differ in their development complexity. Most of the ideas require a sophisticated game design where the PCGML game mechanic can work and create an adequate experience for the players. For instance, ideas like "Train to Progress" or "Figure it Out" offer an exciting approach for creating a unique player experience but require a very sophisticated game design and much balancing and polishing. For this reason, the most straightforward and most applicable ideas for showing the potential of PCGML as a game mechanic in this thesis offer "Changing Weapons" and "Changing Powers." Availability of open source FPS projects, real weapon data, and communities of big FPS titles lead the decision for the prototype game mechanic to the idea of "Changing Weapons." Therefore, the rest of this thesis deals with the implementation of a PCGML game mechanic prototype which revolves around PCGML generated weapons.

7 Prototype Preparations

The dedication of this chapter is the development preparations for a game prototype using the PCGML game mechanic chosen in the last chapter. Every section in this chapter is in direct relation with the picked game mechanic which does not mean that the process is not applicable to other PCGML game mechanics. Most of the described processes are transferable and follow a fundamental principle suitable for every other kind of PCGML game mechanic.

The next sections contain every conceivable and necessary topic for implementing the PCGML mechanic "Changing Weapons" in a game prototype. From an initial game idea to the most useful game engine to the training model. Note that this idea functions as the foundation of a prototype for implementing PCGML game mechanic rather than being a sophisticated designed game. This chapter and the next chapter functions as a proof-of-concept and shows how a PCGML game mechanic can be implemented and used in a game.

7.1 Test Scenario

A one player arena-like 3D player versus environment game scenario in which the player is in control of a PCGML system which produces weapons such as pistols, machine guns and other kinds of guns to interact with. The weapon generator is of vital importance, and the game forces the player consistently to make use of it to win the game. Specific datasets of weapons can bias the system which leads the kind of game and its flow.

7.1.1 Environment and Objective

The environment is a mountain-forest-like environment. The player spawns anywhere on the map and needs to find the arena and the last AI opponent which then starts "the battle of the changing guns" which also represents the last fight of the scenario. The player's primary objective is to defeat the end game boss AI in a battle at the end of the scenario. The player either wins or loses the battle but can retry after a loss.

Training Session

However, first of all, the player needs to fight against NPC enemies on the way towards the arena to specialize and get used to the weapons. Even though this is a training session, the player has a specific amount of lives and need to take care of them. If the player loses all of the

lives during this session, then the game is lost, and it ends. Specific spawn points provide the player with health, ammunition, and armor packages.

Battle of the Changing Guns

Initialization of the battle happens as soon as the player is in a specific range to the AI in the arena. As soon as the player initiated the battle, the training session ends and recovers the player's lives for an epic final battle. The player can now move everywhere on the map and needs to defeat the AI with his weapons to win the game. During the battle, ammunition, health, and armor packages are randomly spawned in the player's range, forcing her or him to move around on the map to give the battle more energy. Furthermore, the end boss is powerful which forces the player to use the weapon generator very often.

7.1.2 Weapon and Ammunition

The player can hold a maximum of two generated weapons but can only use one at a time. Generated weapons come with a specific amount of ammunition with the possibility to be refilled during the game. The player can either use a weapon as long as ammunition is available or dismantle it at any time to generate a new one.

Every generated weapon is fed into the existing PCGML model to specialize the system's model in a specific direction. Newly generated weapons are biased based on the kill count and time of usage of the dismantled weapon. A Head-up Display (HUD) shows statistics about the generated weapon to provide the player with information about it.

7.1.3 Player and Enemies

The player is a third-person character with a specific amount of health, armor, and lives with no customization options. An initial synchronization between these mentioned properties and the properties of the generated weapons is necessary to maintain a balanced game.

A simple AI controls all enemies to act as a training enemy for the battle. Enemies come in three different types of ascending power and health. They spawn in fixed locations throughout the training path. The enemies desire is to eliminate the player in a straightforward way rather than in a complex and sophisticated way.

The design of the boss AI enemy defines specific base health, armor and damage. The base values could increase with the time of the training session and regenerate during the battle.

7.2 Which Game Engine?

The decision of the most applicable game engine is not only based on how quickly the implementation for a FPS game could be done but also about their support for ML. The decision

envisages the two most used free to use game engines nowadays but is not necessarily limited to them. Another reason to further examine these two particular engines is the writer's expertise for them.

As described in Chapter 4.2.4, both Unity and UE4 have available ML plugins for their engine. On the one hand is Unity's ML plugin which is boosted by Unity's developers themselves whereas on the other hand is UE4's community member created open source ML plugin. Nevertheless, both are using the same concepts of including ML functionality into the engine. In specific, they are using a Python interface or plugin to make use of the open source ML framework TensorFlow (TF). With this in mind, the next chapters concentrate on the workflow in each engine to find the most suitable game engine for implementing the chosen PCGML game mechanic.

7.2.1 Unity

Unity's ML plugin is actively in development, is mainly focused on the development of ML-agents or so-called NPCs and supports multiple environments and Operating Systems (OSs) (Unity Technologies, 2018*b*). They are focusing on the development of learning methods like RL, imitation learning and curriculum learning but also enable the application of other methods (Unity Technologies, 2018*b*). Moreover, their GitHub page offers an extensive introduction to ML and their plugin, some example projects on how to use the plugin and is a perfect starting point to dive in.

With this said, in the beginning, and before any functionality of the plugin is usable, it needs an installation of a Python and TF environment on the Personal Computer (PC). This installation is a different procedure and can sometimes be tricky, depending on the OS and preferences for TF.

Once the installation process of all prerequisites is complete, nothing stands between the developer and a working ML environment. The first thing to do is a setup of a training environment to train a new model to make use of it. This learning environment is a typical Unity scene and consists of an academy, brain and the ML-agents itself, created and programmable in C#:

- **Academy**: Is mainly used to communicate with the brain, agents and Python API (Unity Technologies, 2018*b*). It is also involved in environment observations and the decision-making process (Unity Technologies, 2018*b*).
- **Brain**: Encapsulates the logic for decision-making process for the agent (Unity Technologies, 2018*b*). For example, the brain holds the policy for the agent and the given rewards for received observations for an RL-based brain (Unity Technologies, 2018*b*).
- **Agent**: Is just a Unity GameObject which generates observations and processes provided actions by the brain (Unity Technologies, 2018*b*).

As soon as the training environment is set up, the real training can begin with a build of just the training environment. Only the external Python environment is involved in the training of the

model. This environment contains the logic of the ML algorithm based on the TF framework. The Python environment itself can start the training environment build and communicates with Unity's academy with the help of JavaScript Object Notation (JSON) data sent over an open socket connection. After the training's completion, it is possible to save the trained model to use it in the game. A trained model is saveable in a ".byte" data format which can then be used instead of the previously created brain.

7.2.2 Unreal Engine 4

Other than Unity's ML plugin, the open source and community member developed ML plugin for UE4 does not focus on any specific ML algorithm and offers only an API to develop projects with TF (Kaniewski, 2018). The plugin is under continuous development by just one developer, works with one of the newest versions of TF and currently only supports Windows. As a starter help, Kaniewski (2018) provides different releases for different engine versions and hardware, documentation about the installation process and the API, and an example project in which the plugin is used to predict hand-written numbers.

The installation process of the plugin is pretty straightforward. One downloads a release build from the GitHub page and copies the plugin folders into a UE4 project. The plugin itself is dependent on two other plugins. One for a Python environment and the other one for open socket connections, called "SocketIO," directly integrated into UE4. After importing the plugins and restart of the project, the Python plugin triggers the installation process to obtain all necessary dependencies for a working ML Python environment. The ML environment is now fully working and usable for development and does not need external installations.

Setting up an ML environment in UE4 is rather simple and is done in two steps:

1. **TensorflowComponent**: Add a so-called TensorflowComponent to a UE4 Actor which can communicate and execute a so-called TFPluginAPI Python script. The component itself is a Python scripted UE4 ActorComponent and can only be attached to a UE4 Actor. One might provide some functionality within the component for inputting or outputting data to or from the Python API. The communication uses JSON data, sent via an open socket connection to the Python plugin.

2. **TFPluginAPI**: Create a Python script which derives from the TFPluginAPI which is a base class script for handling all TensorflowComponent calls. This script can override functions of the TFPluginAPI for setup, input via JSON data, training start and end, and provides functionality to send custom events. ML code goes in the training functions and will be called either by the TensorflowComponent or if deactivated, by a user.

Other than Unity's procedure, the training process for this plugin does not need a separated environment. The training can happen directly during the game or separately in advance to use an external trained model. Another possibility is to train the model during the game and save it for use in another game session. Every functionality provided by a usual Python API and the

TF framework can be used to develop an ML application within UE4. Furthermore, save and loading functionality of trained models make use of TF's specific data format.

7.2.3 Conclusion

Unity's ML plugin is very focused on implementing ML-agents and provides much code which is more or less useless for the implementation of the PCGML game mechanic. All of their code revolves around implementing an agent with an ML brain, and it would need an overall inspection of their code to set up an own environment to implement the chosen game mechanic. Furthermore, they are making use of an external Python environment which means that every user would need to set up an own ML environment on their PCs. This fact limits the possible application of a PCGML system implementation to an offline system where the training of the brain happens separately, and the build then includes the pre-trained model.

In contrast to Unity's plugin, the plugin for UE4 is just focused on providing a simple interface to create projects with TF and has no specific code in its API. The plugin provides and inherits all documentation needed to implement a working project since it uses mainly the TF framework. Furthermore, there is no necessity of setting up an external environment which makes it possible to include every involved plugin into a fully functional build of a game. Therefore, it is possible to use this plugin for online and offline PCGML systems.

These reasons of a basic TF framework to implement ML algorithms on top of it and that it does not need to set up an external Python and TF environment drives the decision for the game engine to UE4 with its open source community member developed TF plugin. Therefore, the rest of this thesis addresses the implementation of a PCGML game mechanic in UE4.

7.3 Hardware and Software Requirements

UE4 itself does not require particular hardware to run. They support many different OS and hardware. This support leads the requirements examination of this chapter on the UE4 TF plugin.

7.3.1 TensorFlow Plugin

The UE4 TF plugin currently only supports the Windows OS (Kaniewski, 2018). To get started with the plugin, it needs the installation of TF on the executing system or in this case in the UE project. TF supports 64-bit and x86 desktops or laptops with Windows 7 or later (Google, 2018). There are two installation options for enabling different types of TF on the PC:

- **Central Processing Unit (CPU)**: TF with CPU support is the standard alternative which works for most of the standard PCs in use nowadays. The only significant drawback of using the CPU variant of TF are longer training computation times.

- **Graphics Processing Unit (GPU)**: The GPU alternative of TF offers faster computation of ML algorithms and significantly reduces training time. The only requirement for using the GPU variant is that the user needs to use a GPU which is eligible for NVIDIA's CUDA toolkit (Corporation, 2018) to run the GPU variant of TF. Nevertheless, modern PCs with an integrated NVIDIA graphic card are usually eligible to support the CUDA toolkit.

As already mentioned before, the only fundamental difference between the installations is their result in different computation times. Notably, this is getting an important role when the game uses online training, trains a large model and requires suitable computation times for the model's training. For this reason and to avoid initial delays, it is advisable to use a pre-trained model and apply changes to a copy of the model during runtime so that players are not concerned about the computation time.

7.4 Data Acquisition

The most important part of the prototype development focuses on the data for the ML model's training. To get a well-trained model, it requires to have a representative sample of the generation space. For the sake of simplicity in the data search, this dataset size is defined to contain at least 100 entries. However, the right amount of data entries depends on the available features to train the model, and there is no simple formula for this problem as described in Chapter 4.

The search for appropriate data starts with the sample and template projects provided in UE4 since its the engine used for the prototype. The templates can offer a significant advantage because they could provide usable data and implementations for the prototype. Unfortunately, these template projects are not eligible, and the search ends with the conclusion of either using real-world or games data. The next chapters are going into the detail of all found and thus evaluated data sources.

7.4.1 Unreal Tournament and Template Project

Epic Games provide the source code for Unreal Tournament to registered users on their GitHub account (Epic Games, 2018*b*). After following their installation instructions, the project is ready to go and ready for inspection.

It shows that Unreal Tournament does not use as many weapons as needed for the training. The implementation contains only 16 different guns with ten common and usable parameters to train the model. Nevertheless, the source code of their weapons and used parameters are the perfect reference or template for implementing an own weapon class.

The other starting point for finding appropriate data is the so-called "Shooter Game" project which provides an introduction for implementing FPS games (Epic Games, 2018*a*). Unfortunately, this project does not provide any usable data. It only provides the developer with basic

implementation for two different kinds of weapons with an example for each one. The variables used in their implementation are similar to Unreal Tournament's weapon parameters.

7.4.2 Counter-Strike: Global Offensive

The next point of an initial reference is the favorite eSports game Counter-Strike: Global Offensive (CS:GO). Websites with community-driven analysis on the game's central mechanic are the perfect base for finding appropriate data to train an ML model and use the data for weapon generation. Unfortunately, CS:GO does not use many weapons and specializes in 32 weapons for the game (FANDOM Games Community, 2018*b*). Nevertheless, the available analysis data provides an excellent base to work with and to extend the available and balanced game data with, e.g., real-world data.

7.4.3 Real-World Data

Plenty of websites about weapons are findable on the world wide web. However, most of them do not provide much data about the weapons themselves. The search leads to, e.g., army and manufacturer websites to find the appropriate weapon data for specific weapons. Nevertheless, the procedure for finding and obtaining real-world weapon data is a lengthy process. Besides, it shows that weapon manufacturers do not always provide all the data, and it would take real-world tests to obtain specific weapon data. In particular, all found weapon data have about seven to eight similar variables which are usable for training the model. These parameters are the type of weapon, average weight, overall length, barrel length, bullet caliber, magazine size, muzzle velocity, and the rate of fire.

For sure, this data is suitable to train a model and generate weapons even if this takes more time to implement and balance. It would be easier to use already balanced game data to train the model and generate new weapons. For this reason, the search leads to the most valuable data source which is usable for the implementation.

7.4.4 Battlefield 1

The investigation showed that the game Battlefield 1 (BF1), released in 2016, uses more than 100 different weapons constellations in the game. The setting of the game takes place during World War One (WW1) and therefore uses weapons of this period. Fortunately, gamers gathered all possible weapon statistics for Battlefield games on the community website "http://symthic.com" (Symthic, 2018). They provide a list of 148 weapons constellations used in the game with very detailed description and statistics (Symthic, 2018). In fact, they provide more than 70 parameters, describing every weapon (Symthic, 2018).

Further evaluation specializes in these parameters to 18 most suitable and meaningful variables for a weapon implementation. Nevertheless, one can use even more, less or other param-

eters to generate weapons and train the model - but not all of them are always useful. Table 7 shows these most useful parameters for the future weapon class and model training. However, the number of the variables is a subject to change during the implementation.

Name	**Description**
Damage Points	Damage points per hit.
Firerate	Fired bullets per minute.
Num of Pellets	Number of pellets in one shot.
Muzzle Velocity	Bullet velocity when exiting the muzzle.
Bullet Drop	Gravitational bullet drop.
Magazine Size	Number of bullets in a magazine.
Projectile	Type of projectile used.
Horizontal Dispersion	Horizontal pellet dispersion.
Vertical Dispersion	Vertical pellet dispersion.
Reload Empty	Reload time with an empty magazine.
Reload Ammo	Reload time with a magazine with ammo left.
Reload Single Bullet	Reload time for a single bullet.
Recoil Up	Recoil in the up direction.
Recoil Left	Recoil in the left direction.
Recoil Right	Recoil in the right direction.
Recoil Decrease	Recoil decrease after a specific time.
Spread Increase	Spread increase per shot.
Spread Decrease	Spread decrease after a specific time.

Table 7: Eligible weapon statistic for the prototype provided by Symthic (2018).

7.5 The Learning Problem

A quick recap of the game mechanic: It is about generating new weapons based on existing weapons or more specifically weapons from the game BF1. That means that there is a training dataset X and it is desired to generate data which is similar to X. In more specific, this fact let the problem seems to be an unsupervised learning and a clustering problem, earlier described in Chapter 4.1.2. Therefore, the model should be able to learn relations among the attributes in the data. This procedure is also known as probability density estimation which attempts to learn the underlying probability distribution of all the data features (Doersch, 2016). In other words: The given weapons X share a real and unknown distribution $p(X)$ and the model should ap-

proximate this distribution with a new distribution $\hat{p}(X)$ by learning its underlying relationships. This process is also known as density estimation or maximum likelihood estimation.

However, there are also other solutions to this problem of generating similar features of a weapon than the identified way. Therefore, the next section describes all possible methods and approaches which are possible to solve the problem of generating new weapons.

7.5.1 Suitable Models

Now, the PCGML system infers that the model should be able to generate content which leads the logical answer of suitable models to generative models. The model should be able to cluster all the weapons to furthermore generate new ones out of the existing ones. Although it is a generative model problem, discriminative models might also be a possible solution to the problem.

Notably, as seen throughout the previous chapters, there are no distinctive right or wrong methods to train a PCGML model. PCGML is a relatively new approach, and many different methods are suitable to reach a particular goal. For example, the same proceeding as presented in the PCGML development example in Chapter 5.4 is usable for this problem but is most likely not the most useful and fastest. For this reason, this section focuses on generative models instead of discriminative models. Nevertheless, it would be interesting for future work to see if that model would work better than a generative model.

Besides, as the research of (Summerville et al., 2017) shows, methods like NN, LSTM, AE, deep convolutional networks, Markov models or Markov chains are applicable for a model's training to generate, e.g., levels of 2D games. Nevertheless, because TF is a framework for developing deep NNs, the chosen model for this problem and the PCGML system implementation are NNs. Following sections describe two top-rated generative models for deep NNs (Doersch, 2016). However, it is to keep in mind that this does not mean that they are the only ones possible for the generation problem.

Generative Adversarial Network

GAN is a network which fundamentally finds its primary use case in generating high-quality pictures (Bonnin, 2017). The uniqueness of GAN is its composition out of two different types of networks, securing a high-quality. It is composed of a generative network for generating new data and a discriminative network for verification (Bonnin, 2017). This synergy of both networks secures that the model is trained to output high-quality data.

In particular, the generative part is the so-called "Generator" whereas the discriminative model is called the "Discriminator" (Bonnin, 2017). Its workflow is as follows:

1. The generator takes a random sample from a random normal distribution and produces data which could be from the same distribution as the input data (Bonnin, 2017).

2. The discriminator then takes the real input data, compares it to the generated data, tries to identify if it is from the real or a fake distribution and rejects it if it identified as fake (Bonnin, 2017).

The repeating of steps 1 and 2 stops as soon as the generator network can "beat" the discriminator network with fake data (Bonnin, 2017). This repeating counterplay is the reason why GAN got its adversarial part in the name. Specific models for the generator and discriminator can differ, but a possible model for the generator would be a modified VAE (Ghotra & Dua, 2017).

Variational Autoencoder

An VAE is an advancement of an AE with the same principal strategy (Doersch, 2016). AEs aim to reduce and compress the feature dimension of input data which enables the possibility of decompression and reconstruction of all features afterward (Ghotra & Dua, 2017). This method means that, for example, an image can be compressed into a minimal number of features and decompressed to the same image afterward with a small amount of reconstruction error. Although, a crucial difference to GANs is that AEs and VAEs do not reproduce with the same quality because they compress and decompress the data and often lose some detail as a result (Ghotra & Dua, 2017).

Now, the distinctive characteristics about VAEs are that they make use of the compressed state to transform the reduced features onto a normal distribution (Doersch, 2016). It achieves this by additionally sampling from a normal distribution right after the feature reduction which further enables the ability to process random input and reconstruct new data in the same distribution as the training data distribution (Doersch, 2016).

7.5.2 Model Conclusion

Both of the described models are promising and eligible for the PCGML model implementation. Nevertheless, the training time of a GAN might take longer than a VAE due to the generator and discriminator interplay. Furthermore, it would overall require a bigger network and increase the computation times during training. However, this would be a reasonable network if the mechanic would not require retraining.

For this reason, the decision leads to the model of VAEs. They are possibly easier and faster to train than GAN and still can generate data which is similar to the trained data. Therefore, the rest of this thesis addresses a weapon generator game mechanic which uses VAE as the ML backbone for the generation task.

7.6 Used Model Introduction

As already described in the previous chapter, AEs aim to reduce the features of the input to create a compressed feature vector. Therefore, they are also used for encoding and decoding the same structure, feature extraction and data compression (Ghotra & Dua, 2017). Figure 7 shows a schematic representation of an AE which gets a pistol as input and outputs a pistol with a little bit of inaccuracy. The left side of the network is called the encoder which gets the input and encodes all features into the so-called "latent variables" or "latent space" whereas the right site is called the decoder and decodes the latent variables back into the input data (Ghotra & Dua, 2017).

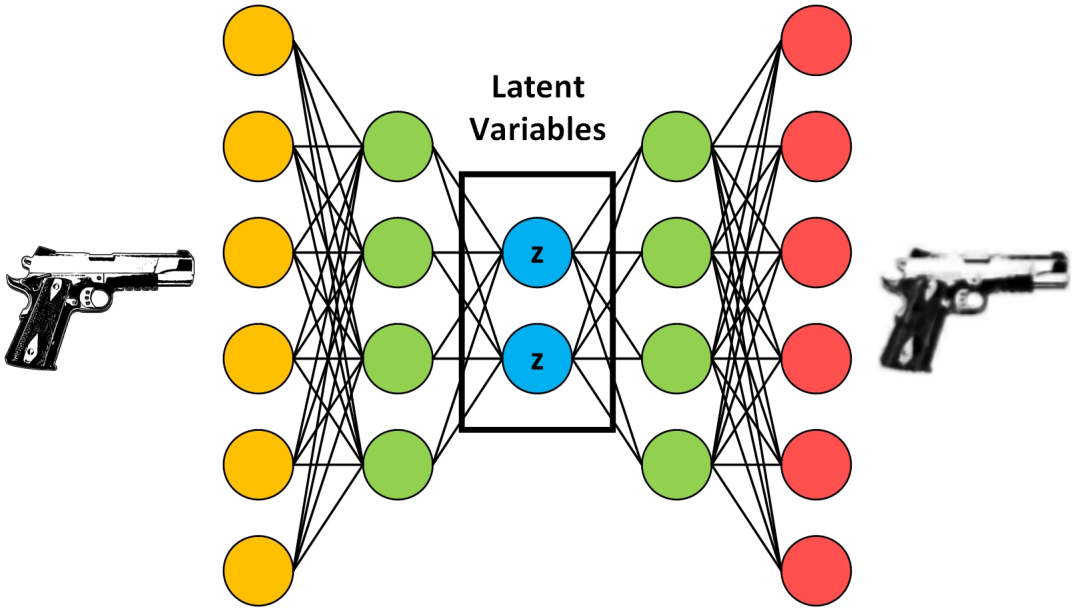

Figure 7: Schematic representation of an AE NN with a blurry output.

The latent space got its name because someone does not necessarily know which features led to it (Doersch, 2016). It represents a transformed representation of the most valuable input data features. For example, someone could think of handwritten numbers and identify a latent variable in the number eight as the upper circle of the number. With this, the decoder would always reproduce this circle during decoding since it learned it as it would be a shared feature among all features. This assumption would make sense if there are only the same numbers in the input data but not if there are many different. Furthermore, if the latent space has the same dimension as the input or hidden space, then it could easily happen that all the information is fed straight forward through the network without any encoding, decoding and latent vector creation. This network constellation is the worst case and is not intended.

Now, the VAE makes use of the same technique and resembles a traditional AE (Doersch, 2016). The primary difference is that it alters the latent space with new variables and a sampling process to map the input data latent space onto a normal distribution like a Gaussian distribution instead to a fixed vector (Doersch, 2016). Figure 8 shows a schematic representation of a VAE

network and the difference to regular AEs.

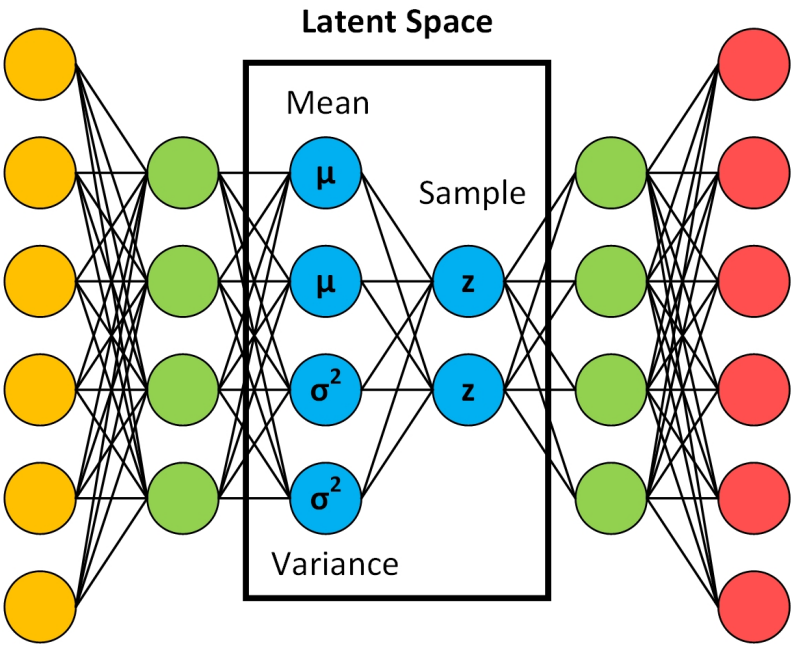

Figure 8: Schematic representation of a VAE NN.

It is worth noting that all of the mentioned details of VAEs are excerpts from the excellent tutorial on VAEs from Doersch (2016) which shall serve as a further reading source if the reader is interested in the detailed mathematics of VAEs.

Now, as seen in Figure 8, the VAE adds two new variables to the latent space. The variables μ and σ^2 represent the mean and variance of a normal distribution and are trained and learned via backpropagation and the so-called "reparameterization trick." The variable z is the actual latent vector which is a combination of μ, σ^2 and a random number sample ε from a normal distribution. Now, the reparameterization trick introduces the random sample variable ε which enables the possibility of backpropagation and helps to map the estimated distribution onto a normal distribution. In specific, the sampling process is applied to alter the latent vector and furthermore the output of the decoder network. However, the actual normal distribution mapping happens with the optimization part of the network which works as follows:

1. Calculate the reconstruction loss of the decoded data compared to the real data.
2. Calculate the so-called "Kullback-Leibler divergence" or short "KL-divergence" which makes sure that the latent vector z stays close to a normal distribution with a mean of zero and a standard deviation of one.
3. Combine the reconstruction loss and the KL-divergence to form the actual cost of the current training batch.
4. Take the calculated overall cost and use backpropagation to optimize the model and minimize the cost.

It is worth noting that only the variables μ and σ^2 are affected and adjusted by the backpropagation algorithm. The latent vector z is omitted and acts as a helper vector for the sampling process. Otherwise, if z would not exist, then it would create a barrier for the backpropagation algorithm since there would be a stochastic unit in the network (Doersch, 2016). For this reason, the random sample ε acts as a new input layer and stochastic unit in the network which is applied to z and not affected by backpropagation (Doersch, 2016). That means z gets indirectly adjusted and optimized because it is a combination of μ, σ^2, and the sample ε.

Furthermore, as the latent vector is always trained with a random normal distribution sample and mapped to a normal distribution, it enables the decoder to decode from noise data (Doersch, 2016). Figure 9 shows an example of this behavior where the decoder decodes noise data back to the learned data.

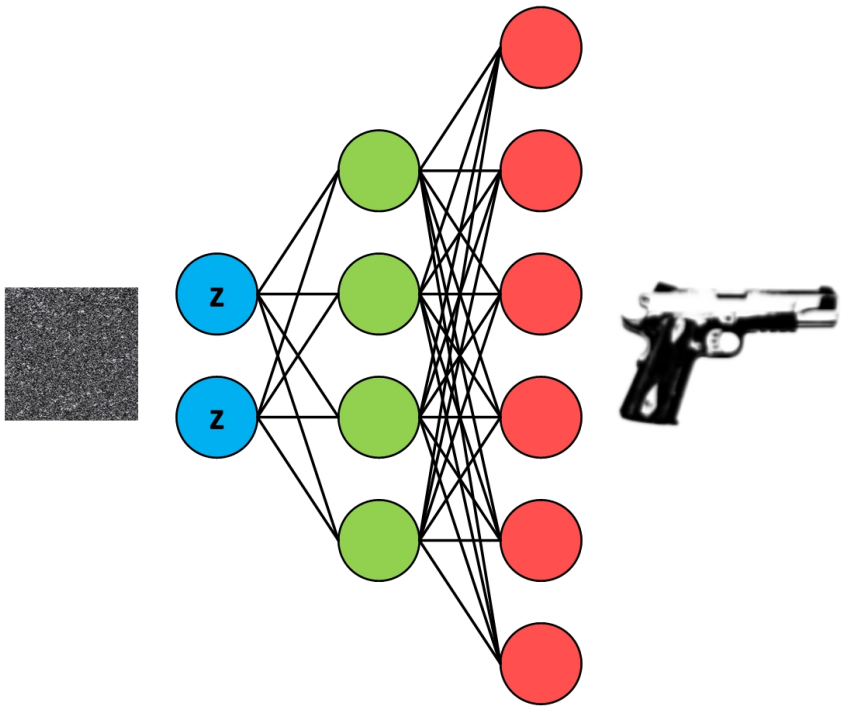

Figure 9: VAE latent space noise input decoding.

This possibility offers an excellent advantage for generating weapons from noise of a random normal distribution. In specific, the explanation for why this is even possible is easy to understand with the words of Doersch (2016, p. 6): "Any distribution in d dimensions can be generated by taking a set of d variables that are normally distributed and mapping them through a sufficiently complicated function."

However, decoding from noise input is not the only advantage. Another advantage is that this also enables the use of noisy input on the encoder side which means that one can add noise to the parameters of the input and the network will still generate a new weapon based on the data.

8 Prototype Development

This chapter addresses every step of the development process of the weapon generator game mechanic but does not cover UE4 specific game development topics. All previously mentioned considerations apply to this implementation. It shows the development from the beginning to the end and sums it up with additional profiling plots for performance concerned readers. The ordering of all following chapters reflects more or less the actual workflow taken during the development. The prototype was implemented with Python version 3.6.4 and TF 1.8.0 in Jupyter Notebook and then integrated into UE4 4.19.2; the source code developed for the prototype is available on GitHub[1].

8.1 Data Preprocessing

The first step of the development is the gathering and preprocessing of the BF1 community weapon analytic raw data. All of the weapon data is accessible to everyone via the community website of Symthic (2018). The community of Symthic kindly provides a database dump on request which was extracted with a script by a community administrator. This database dump contains every parameter of the weapons in a readable and manageable way.

8.1.1 Dump Extraction

More specifically, the dump contains 148 weapons with 90 parameters each. Some of the parameters can contain a list of further parameters. For example, the damage parameter consists of a collection of different damage values which are listed related to distances. For this reason, the damage and distance parameters were split into the first and last values to simplify the extraction and the weapon implementation. The full list of available parameters and their explanations is available in Appendix B.

Now, it would be more than impracticable to manually extract the useful data out of the dump data file. The file has 13764 lines, and it would probably cause dozens of errors and wrong extractions. Therefore, the development of a simple Python script to extract all the data into a ".csv" file was the simplest way to achieve a consistent data extraction.

The script reads a ".txt" file with the specified headers to extract in it as input and then extracts these specified header names from the dump file. Right after completing the data gathering

[1] www.github.com/bernhardrieder/PCGML-Game-Mechanics

process, it creates a new .csv file with a timestamp and adds the extracted data in an organized way so that someone can further process it by hand.

8.1.2 Manual Dimension Reduction, Addenda and Unification

One of the most significant issues in the extracted data was the ammo which was listed as it would be a different one for some weapons even if it is the same. For this reason, a manual examination of all ammo types was necessary to unify the range of all ammo types.

Weapon Type and Fire Modes

It turned out that the dump does not include the specific types and fire modes of the weapons. Hence, it was necessary to examine that from the different game communities Symthic (2018), FANDOM Games Community (2018*a*), and IGN Entertainment, Inc. (2018). This proceeding added the unified weapon types of pistols, rifles, sniper rifles, shotguns, light machine guns, and sub-machine guns as well as the unified fire modes of single-action, semi-automatic and automatic. Some of the weapons in BF1 have more fire modes available which were further simplified by the highest available fire mode. For instance, if a weapon can shoot with semi-automatic and automatic, then automatic was chosen. Besides, the game specifies more fire modes than the ones used. For example, they are using bolt-action, pump-action, single-barrel fire, and lever-action which were summarized as single-action weapons since they are using the same mechanic. Additionally, the weapons of BF1 have the possibility of double-action and double-barrel fire which is now a semi-automatic weapon in the dataset. Charts about the distribution of these two categories in the dataset are available in Appendix C.

Reload Times

Another problem was caused by the reload times of the weapons because BF1 uses distinct reload mechanics for some weapons. In particular, the weapons of BF1 are from WW1 and weapons of this time do not always have the same magazines as used in modern weapons. Some of them do have special strip clip magazines instead of closed magazines which introduced a particular reload mechanic. However, the community of Symthic (2018) provides an explanation to this reload mechanic which made a manual calculation and unification of the reload times for all magazine types possible so that there is no need of a distinction in the prototype implementation.

Omitted Parameters and Final Set

Table 7 shows some parameters which are not very useful for the training of the VAE which makes it necessary to omit some of these variables further. For example, the following parameters were omitted for specific reasons:

- The bullet drop is almost the same for each weapon and would not have an significant impact during the training.
- The ammo types would need encoding which could probably cause the curse of dimension since there are about 30 different used ammo types in the dataset.
- The reload time for ammo if some bullets are available in the magazine is more comfortable to handle if it is directly calculated from the reload time for an empty magazine.
- The last omitted parameter is the recoil in the left direction which showed that it is always the same value as the recoil in the right direction.

This refinement led the final set of parameters to the parameters shown in Table 8. Note that the names of the parameters, except type and fire mode, are the same names as in the dump file provided by the Symthic community. Moreover, the damages and distances are listed as two parameters but are four parameters, as already mentioned in Chapter 8.1.1.

Parameter	**Explanation**
Type	The type of the weapon.
FireMode	The fire mode of the weapon.
Damages	Damage points applied, listed based on distance.
Dmg_distances	The distances in correlation to each damage entry in the "Damages" parameter.
HIPRecoilDec	Weapon recoil decrease if the character is not aiming down sight.
HIPRecoilRight	Weapon recoil upper bound of random recoil if the character is not aiming down sight.
HIPRecoilUp	Weapon recoil upwards if the character is not aiming down sight.
HIPStandBaseSpreadDec	The decrease of shot spread if the character is standing, not moving, and not aiming down sight.
HIPStandBaseSpreadInc	The increase of shot spread if the character is standing, not moving, and not aiming down sight.
InitialSpeed	Muzzle velocity.
MagSize	Size of one magazine.
ReloadEmpty	The time it takes if the magazine is empty.
RoF	The rate of fire.
ShotsPerShell	The number of pellets of one shot.

Table 8: Used weapon parameters for the training of the VAE.

8.1.3 Encoding and Feature Scaling

The next part of the data preprocessing is the encoding and feature scaling task so that the VAE can efficiently process and train the network.

Encoding

The first part is to encode the two only categorical parameters in the dataset. On this account, TF offers encoding mechanisms which help to encode the fire modes and weapon types. The encoding procedure is the same as demonstrated in Chapter 4.2.3 and encodes the two categorical parameters into nine parameters. Therefore, the total number of features in the dataset after encoding is 23.

Feature Scaling

Now, the last part is to normalize or standardize all features. In particular, this is a necessary procedure for this dataset because there are many different value ranges. For example, the rate of fire and muzzle velocity are large numbers whereas the values of the spread increase or decrease are small. If there is no normalization or standardization, then this would cause significant extra expenses, as explained in Chapter 4.2.3.

The approach used for the dataset is the standardization process since it transforms the data in a way so that it has the properties of a standard distribution which may be useful during the VAE's training.

8.1.4 Training and Test Dataset

The total dataset consists of 148 weapon constellations. A typical split of the data would be 80% training data and 20% test data. For this reason, the dataset split is a training set of 127 weapons and a test set of 21 entries which are about 16%. The selection of the weapons for the test set was done randomly and took into account to include at least as many weapons as different fire modes, and weapon types are available, as well as different weapons with values in low, middle and high range for damage, rate of fire, muzzle velocity, magazine size and reload time.

8.1.5 Convenience Class Overview

TF offers many tutorials on how to use and develop with TF. A standard dataset used in the tutorials is the MNIST dataset which encapsulates its data into a convenience class. This class served as a base and inspiration for an own convenience class which encapsulates the weapon data. Following Listing 1 presents the most valuable functions and accessible members of this convenience class:

```
global functions:
   get training and test data

class DataSet:
   properties:
      standardized and encoded data
      number of features in data
      number of data examples
      standardized minimum values
      standardized maximum values
   functions:
      get next batch for training either shuffled or not
      add new weapons to the dataset
      re-standardize the whole data
      encode and standardize features
      decode a processed tensor from TensorFlow
      encode decoded tensor from TensorFlow
```

Code 1: Essential functions and properties of the weapon dataset convenience class.

8.2 Variational Autoencoder

The implementation of the VAE was driven and inspired by the explanation of Metzen (2015). Metzen used a high-level approach in his design which is similar to high-level ML APIs. These high-level APIs are easily modifiable and extensible which was the main reason for following this approach. An easy modification is necessary to test different network hyperparameters without significant changes in the code and thus to find the most suitable network parameters. In particular, Chapter 8.2.3 explains the entire proceeding of the hyperparameters selection process. Now, the next chapters show an overview of the VAE class, how the model accuracy was measured to obtain an adequately trained model, address the already mentioned network parameter selection, show two weapon generation examples and list development issues.

8.2.1 Class Overview

Listing 2 shows a VAE class overview with most of the available and necessary functions for the generation task. The actual code contains some more functions which are not essential to explain. As already introduced in the weapon convenience class, this class also has some global convenience functions to obtain different stages of a VAE.

Two of the most essential functions of the VAE class for the weapon generation tasks are the functions "decode from latent space" and "encoding and decoding" since they provide the functionality to generate data similar to the training data. It is to mention that the VAE initializes itself on construction where it calls all the initialization functions to maintain a working network.

The last most important function to mention is the loss calculation function since it is necessary to secure a working generation during generation time so that the weapon is actually from the distribution of the network and that the network was able to recreate a weapon from that distribution.

```
global functions:
   get an untrained variational autoencoder
   get a fully trained variational autoencoder

class VariationalAutoencoder:
   public functions:
      train the model with a batch
      decode a batch from latent space
      encode and decode a batch
      load the trained model from file
      save the trained model to file
      calculate the loss from a batch
   private functions:
      create the encoder network
      create the z sampling operation
      create the decoder network
      create a hidden layer
      create the weights and biases
      create the loss optimizer
      calculate the reconstruction loss
      calculate the Kullback-Leibler divergence
```

Code 2: Essential functions of the VAE class.

8.2.2 Model Accuracy Measurement

One of the most important parts of training the VAE is to secure a high model accuracy. Two different measurements will secure this accuracy:

- The value of the test set reconstruction loss.
- The value of a random input reconstruction loss.

Now, as already mentioned in Chapter 8.1, the data was split into 127 weapons as training data and 21 weapons as test data. Both of the inputs are feed into the encoder side of the network which outputs the manually examined test results. Following steps were taken to ensure a high model accuracy:

1. Measure the model reconstruction accuracy by calculating the average loss caused by the test set.
2. Generate a random input in the range of the normalized values of the test set and size of the test set and measure the caused average loss.

3. Make sure that the average reconstruction loss of the test set is as small as possible and the loss of the random input is as massive as possible.

Now, if the random input loss is very high, then it means that the model has learned the internal structure and dependencies of the training data because it cannot classify the input data regarding the training data. In other words, a significant loss with random input data means that the model has no idea which data is that supposed to be and cannot reconstruct anything useful. On the other hand, if the reconstruction loss of the test set would be very high, then it would not know what to reconstruct either. Therefore, it is crucial to secure a low loss with the test set and a high loss of the random input.

8.2.3 Network Hyperparameter Selection

Finding the right network hyperparameters for a NN seems to be one of the trickiest parts in designing a NN. Different constellations of learning rate, optimizer function, hidden layer size, latent space size, number of training data features, activation functions after the hidden layers, batch size and training epochs can have similar, better or worse performance since the weights and biases in the NN are randomly initialized, dynamically trained and can emerge in either the right or wrong way. For this reason, the development of a simple hyperparameter test script was the easiest way to tackle that problem and detect the most promising parameters. This test script iterates over each possible constellation and measures the average loss of an entire test set. However, before any of the test iterations were in progress, the assumptions for a properly working network were evaluated as follows:

- **Learning rate**: Between 0.01 and 0.001 because the weapon variables are standardized and bigger rates would not make sense.
- **Size of the hidden layers**: Bigger than the latent space but smaller than the input layer size to prevent the straightforward passing on of the variables without any training of the weights.
- **Size of the latent space**: Not smaller than two and not bigger than the hidden layer because of the same reason as mentioned in the hidden layer.
- **Activation function after hidden layers**: The commonly used hyperbolic tangent activation function "tanh" which outputs values between -1 and 1.
- **Optimizer function**: The common optimizer function "Adam" (Ruder, 2016).
- **Batch size**: Not larger than ten because of the small amount of training data.
- **The number of training epochs**: Inestimable but probably not more than 500.

However, after some test iterations, the values to all the hyperparameters were set empirically, as those that achieved the best performance in the application domain.

- **Learning rate**: 0.01.
- **Size of the hidden layers**: 26 neurons in the first hidden layer and 12 in the second hidden layer.
- **Size of the latent space**: 2 neurons.
- **Activation function after hidden layers**: The exponential linear unit activation function "elu" which leads to higher classification accuracies (Clevert et al., 2015).
- **Optimizer function**: The optimizer function "RMSProp" (Ruder, 2016).
- **Batch size**: 4.
- **The number of training epochs**: A number of 400 shows the lowest costs.

It was interesting to see that the exponential linear unit activation function was able to detect the underlying structure of the data much better than the standard hyperbolic tangent activation function. This detection was then further boosted with a collaboration with the RMSProp optimizer. Why these particular activation function and optimizer works so good would be interesting to know but exceed the scope of this thesis and therefore is not addressed.

Latent Space Visualization

A lucky coincidence of these hyperparameters is that the latent space has a dimension of two neurons which means that it is easy to illustrate the values in the latent space. For this reason, Figure 10 shows the parameter μ of an untrained latent space and Figure 11 shows the parameter μ of a trained latent space after 70 epochs. The left side of the figure shows one dot per weapon whereas the right side shows the same as the left side but with a density per dot.

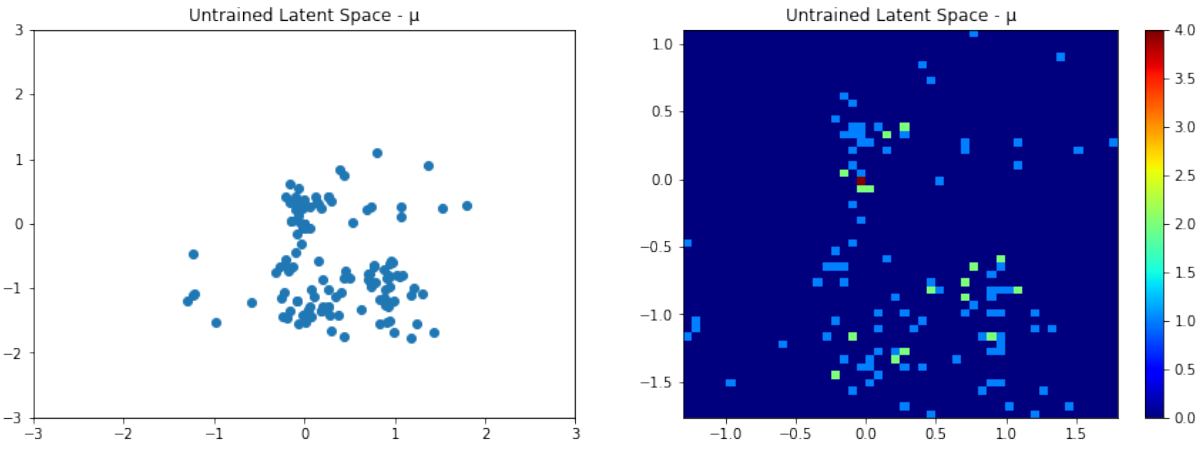

Figure 10: Visualization of the untrained latent space.

Looking at Figure 11, one can easily see that the model was able to learn an underlying structure of the provided weapon data. Furthermore, someone can easily assume and connect the six visible clusters in the plot to the available weapon types in the training data.

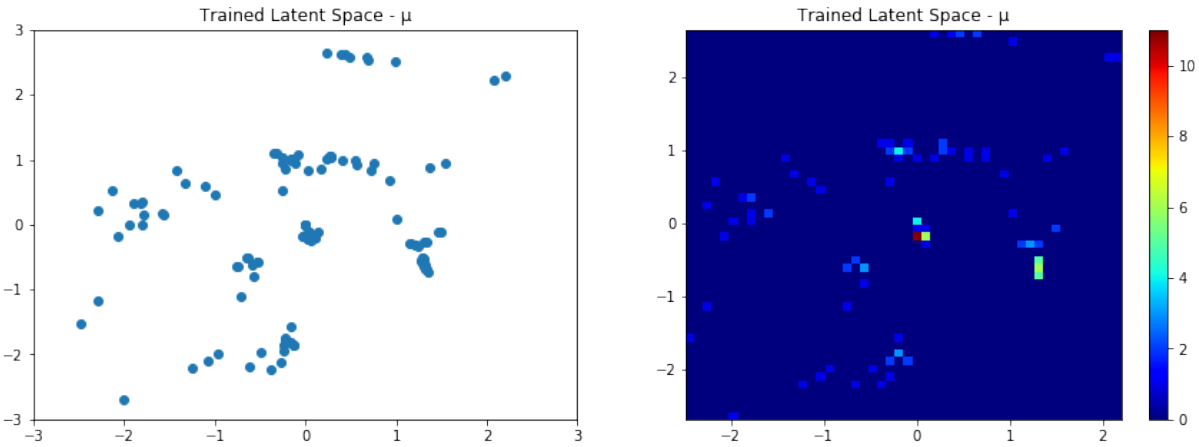

Figure 11: The trained latent space after 70 training epochs.

Cost per Training Epoch

Another chart which helps to determine the right amount of training epochs shows Figure 12. This chart shows that there is a minimum average cost at around 400 to 1000 training epochs which will slightly rise with more and more epochs. This chart contains the average costs of 50 observations with 20 percent outliers included.

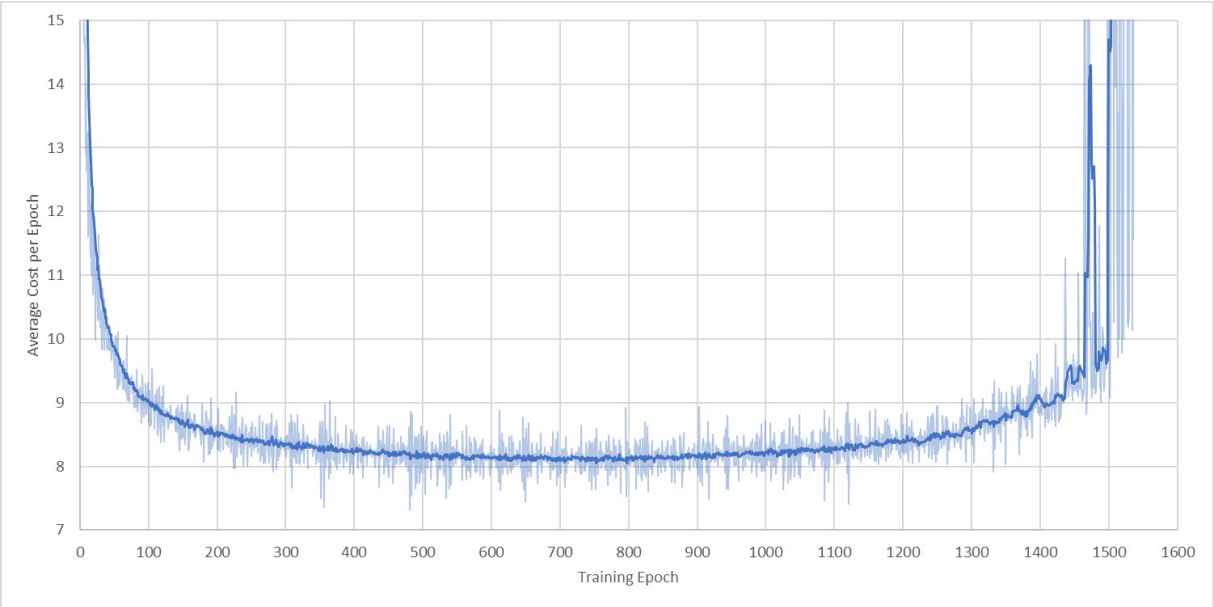

Figure 12: The average cost per training epoch of 50 observations.

Test Data Sample versus Random Sample Input

Another benchmark to determine the right amount of training epochs is to compare the test results of the network with the different input of random sample and test data sample input.

Figure 13 shows this comparison in a chart with separate lines for each input. Each observation point in the chart was calculated with the average of 25 observations and includes 20 percent of outliers. Calculated cost values over 1000 or the ones not identified as a number were clamped to 1000. Furthermore, please take note, that these samples are inputted into the encoder and that a small average cost indicates a successful recreation of the input. Therefore, Figure 13 perfectly demonstrates that the network does not know what to do with random sample input.

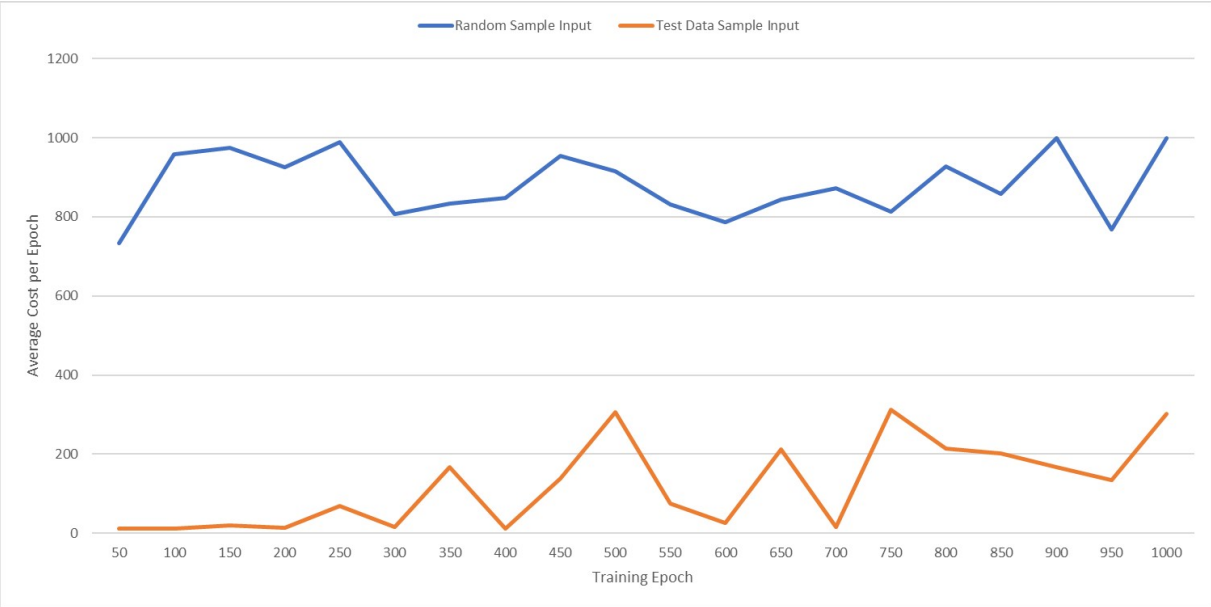

Figure 13: A comparison of the average costs of the random sample versus test data sample input.

Now, an examination of the comparison chart shows that the average costs or rather losses for the test dataset starts to fluctuate with more and more training epochs whereas the average cost for a random input sample remains high. Hence, the decision for the training epochs in the weapon generation was set to 70 epochs in the beginning and will increase with every newly added weapon to the dataset.

8.2.4 Random Input Generation Example

Table 9 shows a comparison of a training weapon and a weapon which was generated with random noise input to prove that the trained VAE can generate useful weapon data. One can see that the VAE has issues with predicting and generating the correct category for a weapon. This issue is still an unsolved problem but is easily solvable by checking the highest value of all categories. In this case, it is most likely a sniper rifle with semi-automatic fire mode. Please note that the listed parameters are the same parameters which are outputted by the VAE.

8.2.5 Development Issues

Two particular issues occurred during the development with TF:

Output Parameter	Training Weapon	Random Generated Weapon
damages_first	26.5	42.767
damages_last	6.25	30.294
distances_first	14.0	13.988
distances_last	21.0	60.002
firemode_Automatic	0.0	0.302
firemode_Semi	0.0	0.333
firemode_Single	0.0	0.234
hiprecoildec	6.0	5.183
hiprecoilright	1.2	0.554
hiprecoilup	11.0	2.436
hipstandbasespreaddec	4.5	6.511
hipstandbasespreadinc	0.3	0.323
initialspeed	333	559.643
magsize	2	20.258
reloadempty	5.333	3.702
rof	299	298.027
shotspershell	12	1.527
type_Pistol	0.0	0.167
type_Rifle	0.0	0.189
type_Shotgun	1.0	0.067
type_Sniper	0.0	0.189
type_SMG	0.0	0.088
type_MG	0.0	0.095

Table 9: Comparison of a training weapon and a generated weapon with random noise as input.

1. It needs so-called sessions in TF to run TF operations. With the creation of this session is a so-called graph included which contains all network nodes and there will be no graph cleanup at session termination. In case of the necessary retraining for this network, this missing cleanup created massive pollution of the graph which slowed the execution bit by bit. The solution to this problem is a simple reset of the graph after the session termination and right before the new trained model update procedure.

2. TF works with nodes which have a specific shape. If the node has no specific shape at

creation time, then it will get one during the training, and this shape will consist in the trained and saved model. For this reason, all of the network nodes will have a similar shape as the batch size because it propagates itself through every node in the network during the training. That means that encoding and decoding, as well as simple decoding from latent space, is just possible with an input which has the same size as the training batch. Of course, this is not a big problem but narrows the possibilities of using the VAE. This problem was solved with a simple replication of the input if it is not the same size as the batch size and mean calculation of the output. For example, if the batch size is four, then it will generate four different weapons of the replicated input and returns the mean values of the generated weapons.

8.3 Game Scenario

The prototype scenario was implemented in UE4 version 4.19.2 and includes a simple map with all requested elements described in Chapter 7. Figure 14 shows the created map of the scenario from a top-down view, the possible spawn points of the player in green and the location of the end boss AI in red. All 3D models and assets are either reused from UE4 template projects or were freely downloaded from the internet and did not violate any copyrights.

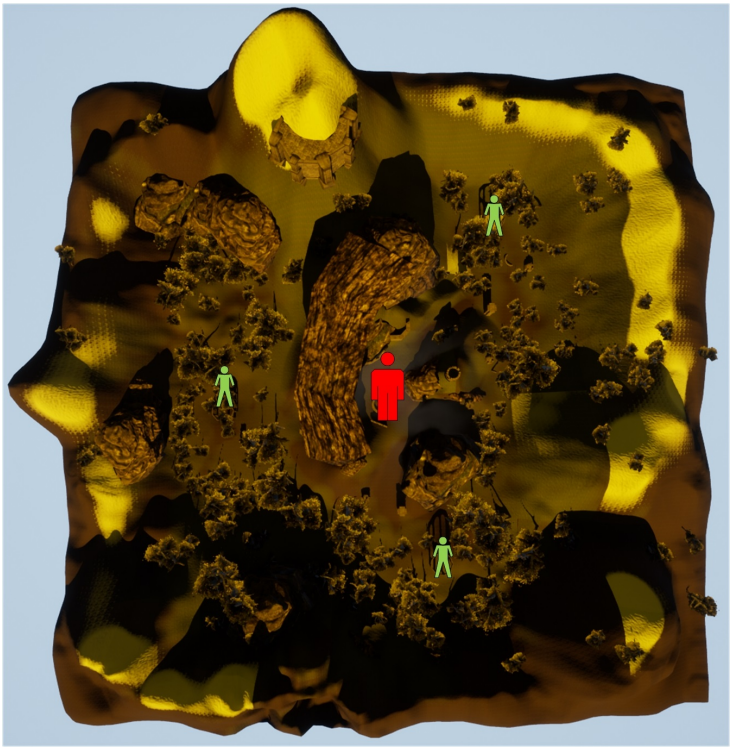

Figure 14: Prototype scenario map with player spawn points in green and boss in red.

Furthermore, Figure 15 shows an in-game screenshot from the player's perspective with the HUD, an enemy AI with a health bar over the head and the waiting end boss AI.

Figure 15: Prototype scenario in-game screenshot.

8.3.1 Head-up Display

As designed in Chapter 7, the HUD should provide the player with feedback to the mechanic and also the game. For this reason, it shows the most critical parameters of the currently equipped weapon on the top left, as visible in Figure 15. Furthermore, it shows the current status of the weapon generator which ranges from "Starting" to "Waiting for Input" to "Generating." The status of the available ammo, current health and armor, and lives are also shown in the HUD so that the player knows about the vitals.

8.3.2 Player

The player can do everything as in common FPS games and additionally has the possibility of using the weapon generator to generate a new weapon. If the player decides to dismantle the equipped weapon, then the currently equipped weapon will be removed from the player and the player can only use the remaining weapon until the generation is complete. Specific movement types increase or decrease weapon shooting behavior. Besides, the player can gather power-ups which are placed randomly on the map and increase or refill movement speed, health, armor, and ammo.

8.3.3 Bots

The implementation of the bots comes in three different types. The easiest one is a simple ball bot which moves towards the player and explodes some time after reaching the player. The two

other bots are from the same type which is visible in Figure 15, they move towards the player and shoot him or her with different weapons in a straightforward way. The easy bot does not cause much damage but tries to heal itself if the health is low whereas the difficult bot causes more damage, has more health and does not heal itself.

The end boss bot is even harder than the difficult bot and has an own weapon which has a higher rate of fire than any other weapons of the bots. Furthermore, the end boss has many health points so that the player needs to use the weapon generator to win against the boss.

8.3.4 Weapons

The weapon class contains almost every feature shown in Table 8. The only omitted parameter is the initial speed or also known as muzzle velocity. For generator reasons, this parameter needs to be saved in the weapon class but does not affect the weapon since they are all working as ray-tracing weapons with instant hits. For further simplicity in the prototype, one class functions as a superclass for all different weapon types. Furthermore, to have a visible representation of the weapon, every weapon type has its weapon model so that the player can distinguish between the weapons inside the game.

Moreover, each weapon has specific modifier which affects either the weapons or the player. For example, machine guns are heavier than pistols and therefore lower the movement speed of a character. Alternatively, if a player aims down sight, then the shooting behavior regarding recoil and spread is improved. These improvements can be adjusted for each weapon class and propagate to the generated weapons of the same weapon type.

8.4 Weapon Generator

As already mentioned at the beginning of this chapter, the weapon generator was implemented outside UE4 and imported as soon as it worked. Listing 3 shows the essential functions of the weapon generator API class which inherits the functions from the TF plugin API class, described in Chapter 7.2.

```
class WeaponGeneratorAPI:
   functions:
      on setup:
         initialize the network
      on begin training:
         train the model
      on json input:
         check if a new trained model is available
         add a dismantled weapon to the dataset and retrain the network if
             necessary
         process the input and return a new generated weapon
```

Code 3: Essential functions of the TF plugin weapon generator API class.

8.4.1 Workflow

The beginning of the game scenario directly triggers the initialization of the weapon generator as well as the setup and training of the VAE model. This training runs on an own background thread so that the player does not notice the training at all at runtime. That is the moment in which the HUD displays the status "Starting" as feedback of the generator for the player. Figure 16 shows the workflow of the weapon generator initialization.

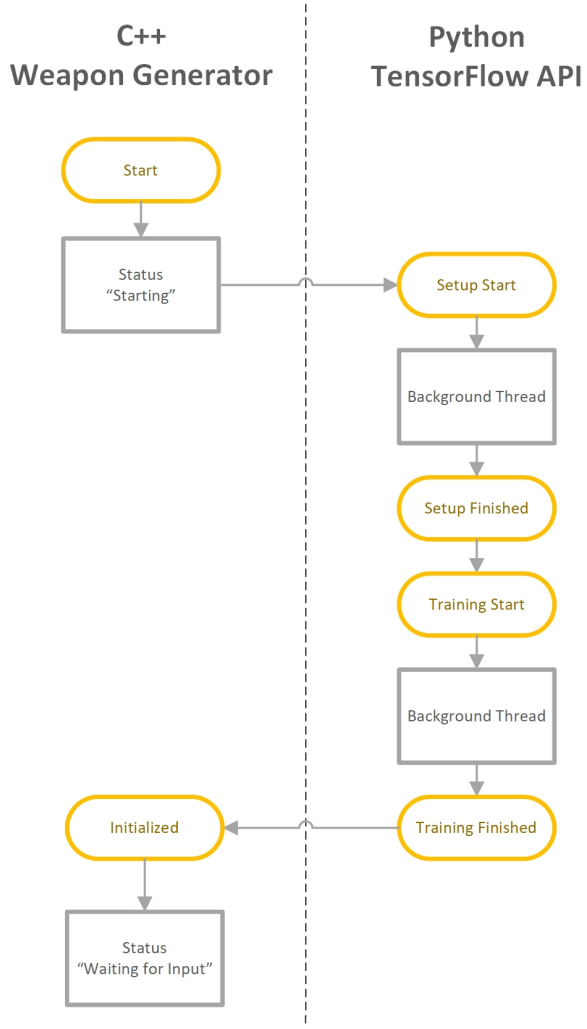

Figure 16: Weapon generator setup and training workflow.

As soon as the generator has changed the status to "Waiting for Input," it is ready to use for the player. The player can now dismantle the equipped weapon which sends it to the weapon generator. Subsequently, the generator extracts all parameters of the weapon, applies some modification based on statistics, creates a JSON formatted string and sends it to the Python API. The applied statistics are the kills and used time of the weapon which changes the random change range of the parameters. The strength of each parameter and the starting range of the random change rate are adjustable parameters. A detailed description of how this operates

is described in the next chapter. Figure 17 shows the workflow of the dismantling process for generating a new weapon. Note that the generation process in the API runs on a background thread so that the game does not freeze during that proceeding.

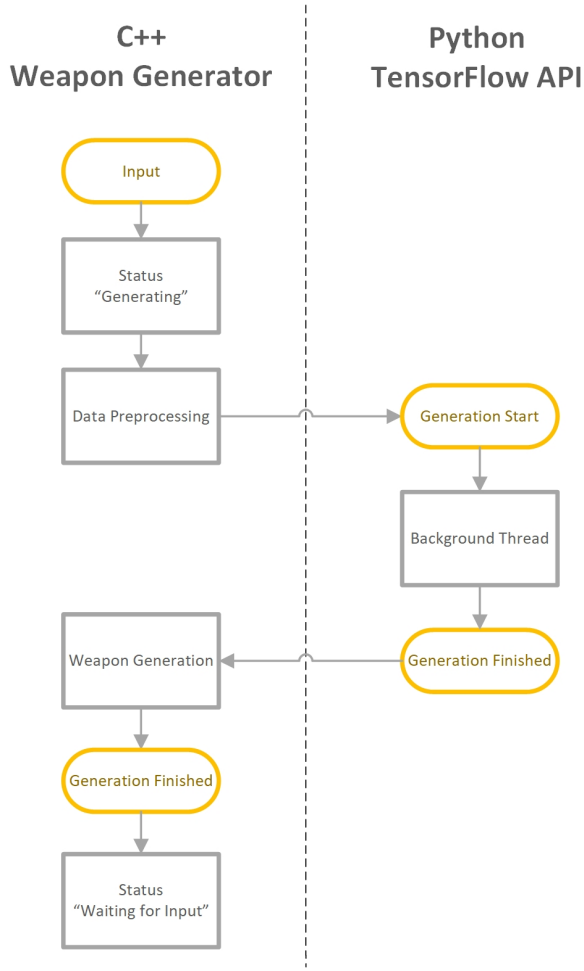

Figure 17: Weapon generator generation workflow.

The generator changes the status to "Generating" during the processing of the dismantled weapon and the generation of the new weapon. Sometimes, this can take more time because of the retraining of the model, triggered after a specific amount of dismantled weapons. The generator creates either a weapon based on the statistics of the dismantled weapon or a random weapon if the average reconstruction costs exceed a specific threshold or a not unprocessable number.

As soon as the VAE generator API is ready and has sent the result back to the weapon generator, it creates a new weapon and notifies the player about the completion. Subsequently, the generator changes the status to "Waiting for Input" and can receive a new dismantled weapon. During the creation process of the new weapon, the generator checks and makes sure, that all of the parameters do not exceed critical values which could cause bugs in the game.

8.4.2 Parameter Modification and Adjustable Parameters

As already mentioned, the parameters of the dismantled weapon are modified before sent to the generator API which includes some adjustable parameter. The parameters of weapons are either increased or decreased depending on whether it would help the player against the boss. For example, the damage is increased whereas reload time is decreased and so forth. Now, the change of parameters results according to a random change rate which can be offset with kills or using time. The starting range and the strength of the kills and time are adjustable in the weapon generator.

Other adjustable parameters are tolerance values which can add a random element to the weapon type and fire mode selection. The problem tackled with this proceeding was mentioned in Chapter 8.2.4. The VAE sometimes generates small numbers which are almost similar to each other for every categorical parameter. The solution to this problem is first to obtain the highest value among the received values and then check which of the values lie in the specified tolerance range, based on the highest found value. Therefore, this tolerance causes a random selection among almost similar weapon types or fire modes.

8.4.3 TensorFlow Plugin Changes

The default TF version of the plugin is version 1.6.0. Nonetheless, the prototype development used version 1.8.0. For this reason, the version of TF in the plugin was changed to 1.8.0 which could be done without any issues since the plugin is not dependent on any TF specific variables.

The other significant change in the plugin was a change in the API class which does not use a background thread for the input processing by default. This missing feature is causing a game freeze during the weapon generation and is not tolerable. This freeze will also happen if multithreading is disabled in the plugin which is not advisable. Therefore, the API code was changed to run the input processing on a background thread and notify the API on completion so that it can propagate the result to the weapon generator.

8.5 Performance and Profiling

Profiling was done in the UE4 Editor in development configuration with the following test device specifications:

- **Product**: MSI GS60 2PE Ghost Pro
- **OS**: Windows 10 64-bit
- **CPU**: Intel Core i7-4710HQ CPU @ 2.50GHz
- **GPU**: NVIDIA GeForce GTX 870M
- **Random access memory**: 16 gigabyte

All of the following profiling charts were obtained with the same testing scenario. The scenario map was loaded, the weapon generator either initializes or not, depending on its availability, and the game was then actively played for more than 30 seconds. Hence, the weapon generator was constantly used if available in the specific testing environment.

Figure 18 shows a profiling chart of the scenario map without any weapon generation mechanic enabled. The TF plugin, its dependent Python plugin, and the SocketIO plugin were fully deactivated and did not affect the engine's performance.

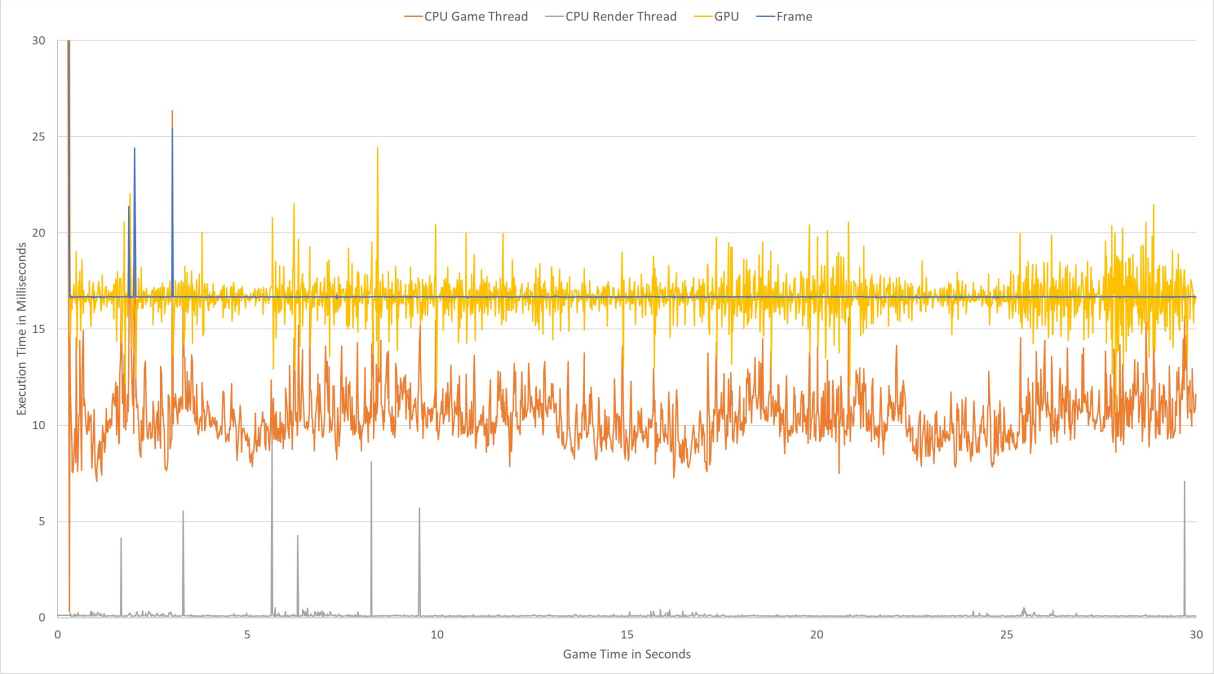

Figure 18: Profiling chart with the deactivated TF Plugin.

Figure 19 shows a profiling chart with an enabled TF plugin but which was not actively used in the game. Therefore, the weapon generator was not available for the player.

Figure 20 shows a profiling plot with an actively used TF plugin and weapon generator in the game. The TF plugin in this scenario was using the CPU for computation.

Figure 21 shows a profiling plot with an actively used TF plugin and weapon generator in the game. The TF plugin in this scenario was using the GPU for computation.

8.5.1 Conclusion

The profiling charts show that there is a performance loss of about four milliseconds in the CPU render thread when using the TF plugin actively in the game. Moreover, it seems that the CPU variant of the plugin works better than the GPU variant in this case. This advantage might be because the TF plugin uses CPU multi-threading for the setup, training, and input processing tasks whereas the GPU variant interrupts the GPU from time to time because it needs to obtain the calculated tensors from TF - but that is just an assumption and cannot be evidenced by

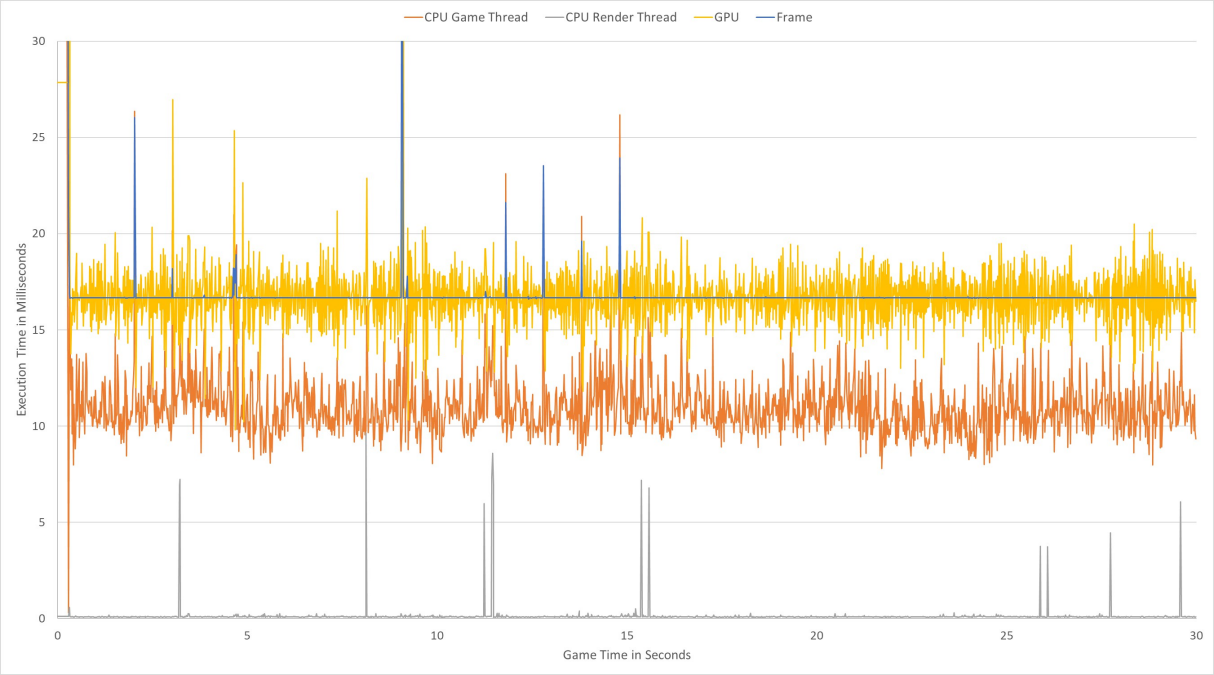

Figure 19: Profiling chart with the activated but not actively used TF Plugin for CPU.

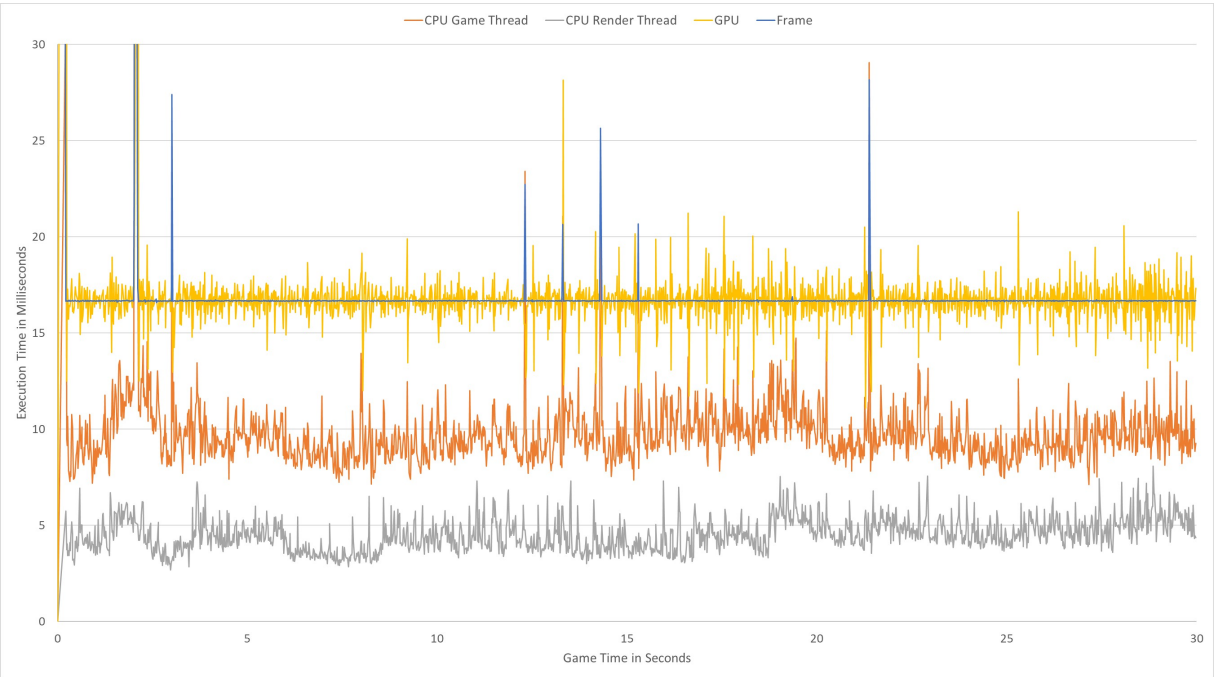

Figure 20: Profiling chart with the actively used TF Plugin for CPU.

facts. Nevertheless, it should be noted that this profiling charts reflect only this specific use case and could have other performances with other use cases.

However, this shows that it is not necessary that someone trains the model in advance and loads a trained model on the first start. This procedure would only be necessary if the training

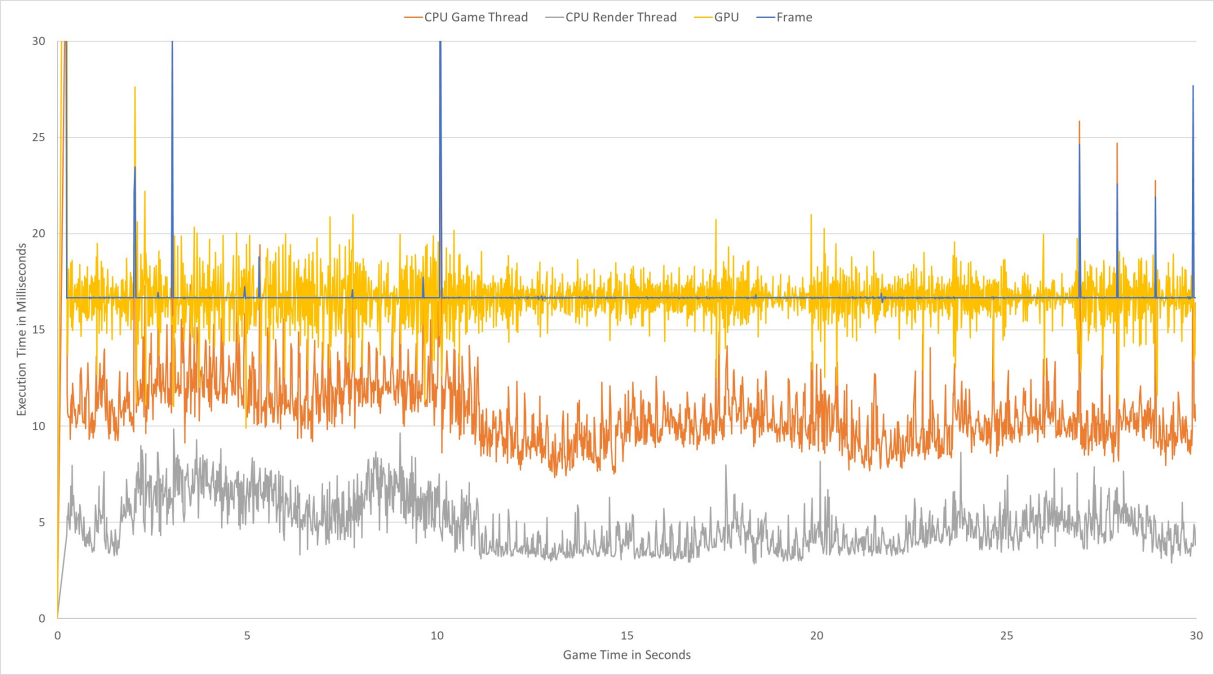

Figure 21: Profiling chart with the actively used TF Plugin for GPU.

would include a high number of training epochs and would prevent the player from using the mechanic. However, this is not the case for this mechanic and therefore is not necessary.

9 Conclusion

The idea of this master thesis was to explore the possibilities of PCGML as a game mechanic with a focus on their applications in video games. Its primary goal was to provide a demonstration of the possibilities as well as to address every aspect of developing a PCGML game mechanic in common free-to-use game engines. Overall, it shall serve as a concise overview and primer for developing PCGML game mechanics with a prototype development example workflow.

In particular, the thesis fulfills all of the demanded requirements from the listing of unique game mechanic ideas for PCGML up to an in-depth development of a prototype in a game engine. The development of the "Changing Weapons" mechanic shows the potential for PCGML as a game mechanic in future work and helps curious developers to quickly dive into every knowledge necessary to start developing new games with PCGML game mechanics. It shows the space for improvement and lists essential development considerations for implementing a PCGML game mechanic so that there are no unexpected mistakes. In summary, this thesis establishes and reflects an introduction and overall guideline for bringing PCGML game mechanics into a game.

9.1 Research Result

The focus of this thesis was to provide theoretical possibilities for PCGML game mechanics, a practical development example and a showcase that PCGML game mechanics are suitable for a broad range of games with no limitation to particular genres. Chapter 6 provides these possibilities and furthermore shows some pros and cons of a theoretical development. The game mechanic "Changing Weapons" was then implemented and introduced the development workflow of a PCGML game mechanic. Overall, the thesis fulfilled all expectations of using PCGML as a game mechanic and shows valuable research results.

9.1.1 Theoretical Introduction

The primary focus of "Changing Weapons" is a weapon generator which helps the player to obtain its objective of defeating the end boss AI of a game. Hence, the development of this generator shows developers necessary preparations such as the decision between possible game engines, the examination of hardware and software requirements, ML data acquisition, data preprocessing knowledge, and at last, addresses the specific learning problem which needs to

be tackled by the ML model. Moreover, it introduces some suitable models which could be used for the generator model and focuses on the most promising model.

9.1.2 Practical Examination

The practical part of the development showed the necessary steps for data preprocessing, an overview of a weapon data convenience class, the actual ML model implementation and some empirical observations of the working model used in the game, as well as the generator API and the weapon generator in the game itself.

The prototype showed that the intention of emerging the model in a specific direction works but is as not fixed as expected because of how the network initialization works. Therefore, weapons during the game emerge in different directions for every new game due to the initialization of the weights and biases which means the model always trains in another way. The solution to that problem would be the pretraining and loading of the model at the beginning of the game. Also, the weapons in the training dataset bias the model in a specific direction, for example, shotguns are limited in the dataset and therefore are not often generated. Appendix C shows the distribution of the weapons which is the reason for higher generation numbers for specific weapons.

Research showed that there are dozens of possible network configurations for the ML model which means that it has still room for possible improvements. Nevertheless, the developed ML model works as intended and outputs the desired data. However, the model sometimes produces unusable data during the generation process which was avoidable with omitting the faulty data. Hence, this does not break the actual game since the generator API does not propagate the faulty data into the game. This problem is due to the small training dataset and is solvable with more data to train. Another open problem is that the model cannot always reproduce and predict categorical features very well for which a workaround was introduced by using the category with the highest value.

Last but not least, the profiling of the game shows that the used development configuration does not affect performance too much. That means that it is possible to develop much more detailed and sophisticated game mechanics with PCGML without significant performance losses. Nevertheless, the next chapter proposes a possible minimization of the performance loss.

9.2 Future Work

Now, there is plenty of space for improvement and future work when implementing PCGML game mechanics. If speaking of this particular use case of implementing a weapon generator then there are following improvements possible:

- First of all, it would be possible to get rid of the TF plugin in UE4 with a full C++ UE4 ML plugin. This C++ plugin would probably save much performance because it would

not rely on a Python plugin, the SocketIO plugin or TF itself. Of course, this would work even better with an engine integration of ML. Therefore, the only possible engine with that advantage in the near future could be Unity with their current ML plugin development since UE4 does not officially seem to show interest in an ML integration.

- Another improvement of the game mechanic would be an increase or replacement of the weapons in the training dataset to refine the trained model. More weapons enable training of the model with much more training features, and therefore it can learn the underlying structure even better.

  Nevertheless, this would currently cause a necessary refactoring of all the code to integrate the new features.

- It would be a valuable improvement if the current classes would be extended to work as generic types of classes and ML models for more use cases than the prototype game. This generic implementation would enable designers to change features of a traditional FPS game weapon with, e.g., RPG like features of causing fire damage and burning on the hit enemy.

- The generation of different and new ammo types would also be a possible improvement since the current prototype only uses ray-tracing weapons instead of projectile weapons. This additional generation would enable a much more diverse game mechanic and experience for the player. For example, the new ammo types could follow a specific movement pattern as used for the weapons in the game "Galactic Arms Race," described in Chapter 3.3.1.

- The weapon generator itself offers even more improvement. Currently, the generator does not limit any values which means that because of the retraining functionality of the model, the weapon can get extraordinary high damage and other values. These high values are still okay if its intended but most likely they should be trimmed before sending them to the generator API or creating the new weapons for the player.

- The gameplay concept for using the weapon generator also offers space for improvement. In particular, it is to see how the weapon generator can be integrated into the gameplay in a more natural way than pressing a button and get a new weapon. For example, a possible concept would be for the player to call for reinforcements, and they will send new random weapons. In this scenario, the player would need to hold on to his current weapon until the reinforcement has arrived.

As someone can see, there are many possibilities with PCGML game mechanics. This thesis only shows an example game mechanic which was rather easy to implement but offers much more sophisticated game mechanics which enable even more innovative player experience. It is now a further research question to find out whether the designs of the in Chapter 6 introduced game mechanics will work out as intended or not.

Bibliography

Adams, E. & Dormans, J., 2012. *Game Mechanics: Advanced Game Design*. 1st edition. Thousand Oaks, CA, USA: New Riders Publishing.

Amato, A., 2017. Procedural Content Generation in the Game Industry. In: *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, pp.15–25.

Blatz, M. & Korn, O., 2017. A Very Short History of Dynamic and Procedural Content Generation. In: *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, pp.1–13.

Bonaccorso, G., 2017. *Machine Learning Algorithms: A Reference Guide to Popular Algorithms for Data Science and Machine Learning*: Packt Publishing.

Bonnin, R., 2017. *Machine Learning for Developers*: Packt Publishing.

Champandard, A. J., 2007. *Top 10 Most Influential AI Games*: AiGameDev.com. Available at: <http://aigamedev.com/open/highlights/top-ai-games/> [Accessed 30.03.2018].

Clevert, D.-A., Unterthiner, T. & Hochreiter, S., 2015. Fast and accurate deep network learning by exponential linear units (elus).. *CoRR*, abs/1511.07289. Available at: <http://dblp.uni-trier.de/db/journals/corr/corr1511.html#ClevertUH15> [Accessed 19.06.2018].

Cook, M., 2018. *Games By Angelina - Developing an AI that can automatically design videogames*: Michael Cook. Available at: <http://www.gamesbyangelina.org/> [Accessed 21.03.2018].

Corporation, N., 2018. *CUDA Toolkit | NVIDIA Developer*: NVIDIA Corporation. Available at: <https://developer.nvidia.com/cuda-toolkit> [Accessed 16.05.2018].

Dalmau, D. S.-C., 2005. *Postcard from GDC 2005: Tutorial - Machine Learning*: Gamasutra. Available at: <https://www.gamasutra.com/view/feature/130633/postcard_from_gdc_2005_tutorial__.php> [Accessed 30.03.2018].

Dangeti, P., 2017. *Statistics for Machine Learning*: Packt Publishing.

Doan, D., 2017. *GameDev Protips: How To Design More Meaningful And Engaging Game Mechanics*: Gamasutra. Available at: <https://www.gamasutra.com/blogs/DanielDoan/20170322/294224/GameDev_Protips_How_To_Design_More_Meaningful_And_Engaging_Game_Mechanics.php> [Accessed 12.03.2018].

Doersch, C., 2016. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, .

Doran, J. & Parberry, I., 2011. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. Bordeaux, France: ACM, pp.1:1–1:8.

Eladhari, M. P., Sullivan, A., Smith, G. & McCoy, J., 2011. *AI-Based Game Design: Enabling New Playable Experiences*. Available at: <https://www.soe.ucsc.edu/research/technical-reports/UCSC-SOE-11-27> [Accessed 19.03.2018].

Epic Games, I., 2018*a*. *Shooter Game*: GitHub. Available at: <https://docs.unrealengine.com/en-us/Resources/SampleGames/ShooterGame> [Accessed 24.05.2018].

Epic Games, I., 2018*b*. *UnrealTournament - Help us build the next Unreal Tournament game!*: GitHub. Available at: <https://github.com/EpicGames/UnrealTournament> [Accessed 24.05.2018].

Evolutionary Games, 2014. *Galactic Arms Race*: Evolutionary Games. Available at: <http://gar.eecs.ucf.edu/> [Accessed 22.03.2018].

FANDOM Games Community, 2018*a*. *Battlefield Wiki | FANDOM powered by Wikia*: FANDOM Games Community. Available at: <http://battlefield.wikia.com> [Accessed 18.06.2018].

FANDOM Games Community, 2018*b*. *Category:Weapons | Counter-Strike Wiki | FANDOM powered by Wikia*: FANDOM Games Community. Available at: <http://counterstrike.wikia.com/wiki/Category:Weapons> [Accessed 24.05.2018].

Ghotra, M. S. & Dua, R., 2017. *Neural Network Programming with Tensorflow*: Packt Publishing.

Google, 2018. *TensorFlow*: Google. Available at: <https://www.tensorflow.org/> [Accessed 29.03.2018].

Hastings, E. J., Guha, R. K. & Stanley, K. O., 2009*a*. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4), pp.245–263.

Hastings, E. J., Guha, R. K. & Stanley, K. O., 2009*b*. Evolving content in the galactic arms race video game. In: *2009 IEEE Symposium on Computational Intelligence and Games*, pp.241–248.

Hello Games, 2016. *No Man's Sky*: Hello Games. Available at: <https://www.nomanssky.com/> [Accessed 21.03.2018].

Hendrikx, M., Meijer, S., Van Der Velden, J. & Iosup, A., 2013. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1), pp.1:1–1:22.

Holzinger, A., Pichler, A., Almer, W. & Maurer, H., 2001. Triangle: A multi-media test-bed for examining incidental learning, motivation and the tamagotchi-effect within a game-show like computer based learning module. In: *Proceedings of EdMedia: World Conference on Educational Media and Technology 2001*. : Association for the Advancement of Computing in Education (AACE), pp.766–771.

Hunicke, R., Leblanc, M. & Zubek, R., 2004. MDA: A Formal Approach to Game Design and Game Research. , 1. Available at: <http://www.cs.northwestern.edu/~hunicke/pubs/MDA.pdf> [Accessed 12.03.2018].

IDV, 2018. *SpeedTree Vegetation Modeling*: IDV. Available at: <https://store.speedtree.com/> [Accessed 21.03.2018].

IGN Entertainment, Inc., 2018. *Weapons - Battlefield 1*: IGN Entertainment, Inc. Available at: <http://www.ign.com/wikis/battlefield-1/Weapons> [Accessed 18.06.2018].

Kaniewski, J., 2018. *getnamo/tensorflow-ue4: TensorFlow plugin for Unreal Engine 4*: GitHub. Available at: <https://github.com/getnamo/tensorflow-ue4> [Accessed 29.03.2018].

Koster, R., 2013. *Theory of Fun for Game Design*. 2nd edition: O'Reilly Media, Inc.

Liapis, A., Yannakakis, G. N. & Togelius, J., 2014. Computational game creativity. In: *ICCC*.

Maxis, 2008. *Spore*: Electronic Arts Inc. Available at: <http://www.spore.com/> [Accessed 22.03.2018].

Metzen, J. H., 2015. *Variational Autoencoder in TensorFlow*: Jan Hendrik Metzen. Available at: <https://jmetzen.github.io/2015-11-27/vae.html> [Accessed 19.06.2018].

NVIDIA, 2007. *Cascades*: NVIDIA. Available at: <http://www.nvidia.co.uk/coolstuff/demos#!/cascades> [Accessed 21.03.2018].

Pears, M., 2018. *Game Design: Introducing Mechanics*: Gamasutra. Available at: <https://www.gamasutra.com/blogs/MaxPears/20180307/315172/Game_Design_Introducing_Mechanics.php> [Accessed 12.03.2018].

Rohrer, J., 2011. *Inside a Star-filled Sky*: Jason Rohrer. Available at: <http://insideastarfilledsky.net/> [Accessed 22.03.2018].

Rose, M., 2012. *5 tips for using procedurally-generated content in your game*: Gamasutra. Available at: <https://www.gamasutra.com/view/news/181853/5_tips_for_using_proceduralllygenerated_content_in_your_game.php> [Accessed 21.03.2018].

Ruder, S., 2016. *An overview of gradient descent optimization algorithms*. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam. Available at: <http://arxiv.org/abs/1609.04747> [Accessed 19.06.2018].

Schell, J., 2008. *The Art of Game Design: A Book of Lenses*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Shaker, N., Togelius, J. & Nelson, M. J., 2016. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*: Springer.

Short, T. & Adams, T., 2017. *Procedural Generation in Game Design*: CRC Press.

Smith, G., 2014. *The Future of Procedural Content Generation in Games*. Available at: <https://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/9078> [Accessed 16.03.2018].

Smith, G., Othenin-Girard, A., Whitehead, J. & Wardrip-Fruin, N., 2012. PCG-Based Game Design: Creating Endless Web. *Foundations of Digital Games 2012 (FDG '12)*, .

Stout, M., 2015. *Trinity, Part 3: Game Mechanics*: Gamasutra. Available at: <https://www.gamasutra.com/blogs/MikeStout/20150622/246683/Trinity_Part_3_Game_Mechanics.php> [Accessed 12.03.2018].

Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A. & Togelius, J., 2017. Procedural Content Generation via Machine Learning (PCGML). *CoRR*, abs/1702.00539. Available at: <http://arxiv.org/abs/1702.00539> [Accessed 12.03.2018].

SUPERHOT Team, 2016. *SUPERHOT - The FPS where time moves only when you move*: SUPERHOT Team. Available at: <https://superhotgame.com/> [Accessed 04.04.2018].

Symthic, 2018. *BF1 Weapon Stats: Damage, Accuracy etc. | Symthic*: Symthic. Available at: <http://symthic.com/bf1-stats> [Accessed 24.05.2018].

Togelius, J., Kastbjerg, E., Schedl, D. & Yannakakis, G. N., 2011. What is Procedural Content Generation?: Mario on the Borderline. In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. Bordeaux, France: ACM, pp.3:1–3:6.

Treanor, M., Zook, A., Eladhari, M. P., Togelius, J., Smith, G., Cook, M., Thompson, T., Magerko, B., Levine, J. & Smith, A., 2015. *AI-based game design patterns*. Santa Cruz, CA: Society for the Advancement of Digital Games. Available at: <https://strathprints.strath.ac.uk/57219/> [Accessed 12.03.2018].

Unity Technologies, 2018*a*. *Unity - Unity Machine Learning tools and resources*: Unity Technologies. Available at: <https://unity3d.com/machine-learning> [Accessed 29.03.2018].

Unity Technologies, 2018*b*. *Unity-Technologies/ml-agents: Unity Machine Learning Agents*: GitHub. Available at: <https://github.com/Unity-Technologies/ml-agents> [Accessed 29.03.2018].

Yannakakis, G. N. & Togelius, J., 2018. *Artificial Intelligence and Games*: Springer. http://gameaibook.org.

Zook, A. & Riedl, M. O., 2014. *Generating and Adapting Game Mechanics*. Available at: <http://www.fdg2014.org/workshops/pcg2014_paper_03.pdf> [Accessed 19.03.2018].

List of Figures

List of Tables

List of Code

List of Abbreviations

2D	2-Dimensional
3D	3-Dimensional
AE	Autoencoder
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BF1	Battlefield 1
CPU	Central Processing Unit
CS:GO	Counter-Strike: Global Offensive
FPS	First Person Shooter
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
HUD	Head-up Display
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbor
LSTM	Long Short-Term Memory Network
MDA	Mechanics-Dynamics-Aesthetics
ML	Machine Learning
NN	Neural Network
NPC	Non-playable Character
OS	Operating System
PC	Personal Computer

PCG Procedural Content Generation

PCGML Procedural Content Generation via Machine Learning

RPG Roleplay Game

RL Reinforcement Learning

RNG Random Number Generator

SVM Support Vector Machine

TF TensorFlow

UE4 Unreal Engine 4

VAE Variational Autoencoder

WW1 World War One

A Procedural Content Generation via Machine Learning Game Mechanics Summary

	Output	**Training**
Rules and Behavior	Configuration of a game with all its game elements.	Working configuration of a game.
Changing Weapons	New and novel weapons.	Existing balanced and unbalanced weapons.
Changing Powers	New and novel powers.	Existing useful powers like magic spells.
Novel Vehicles	New and novel vehicles.	Existing balanced and useful vehicles for a specific environment.
Solver Weapon	E.g., elements of a puzzle.	Object detection and example drawings.
Defeat of the Enemy	New and successful attack patterns.	Possible AI behavior such as attacks or movement for a specific environment.
Caught in a Thunderstorm	Lightning diffusion pattern.	Example diffusion pattern based on real weather data.
Train to Progress	E.g., elements of a puzzle such as a key for a lock.	No need for training due to training with player.
Building with Assistance	Incomplete elements which allow modification.	Samples of incomplete elements for the game.
The exploring Co-Worker	Terrain generation or manipulation.	Example terrains based on player progress.
Observe and Learn	Terrain generation or manipulation.	Example terrains.
Express Yourself	Storyline with events and choices connected to it.	Configuration and story for possible games.
Big Boss Helper	Terrain generation and modification.	Example terrains with obstacles based on a player behavior in a specific environment.

	Output	Training
Figure it Out	E.g., elements of a puzzle.	Example puzzle elements to progress in a game.

Table 10: PCGML game mechanic training and output summary.

B Battlefield 1 Community Weapon Analytic Dump Parameters

Parameter	Explanation
ADSCrouchBaseMax	Maximum shot spread if the character is crouching, not moving, and aiming down sight.
ADSCrouchBaseMin	Minimum shot spread if the character is crouching, not moving, and aiming down sight.
ADSCrouchBaseSpreadDec	The decrease of shot spread if the character is crouching, not moving, and aiming down sight.
ADSCrouchBaseSpreadInc	The increase of shot spread if the character is crouching, not moving, and aiming down sight.
ADSCrouchMoveMax	Maximum shot spread if the character is crouching, moving, and aiming down sight.
ADSCrouchMoveMin	Minimum shot spread if the character is crouching, moving, and aiming down sight.
ADSCrouchMoveSpreadDec	The decrease of shot spread if the character is crouching, moving, and aiming down sight.
ADSCrouchMoveSpreadInc	The increase of shot spread if the character is crouching, moving, and aiming down sight.
ADSProneBaseMax	Maximum shot spread if the character is lying, not moving, and aiming down sight.
ADSProneBaseMin	Minimum shot spread if the character is lying, not moving, and aiming down sight.
ADSProneBaseSpreadDec	The decrease of shot spread if the character is lying, not moving, and aiming down sight.

Parameter	Explanation
ADSProneBaseSpreadInc	The increase of shot spread if the character is lying, not moving, and aiming down sight.
ADSProneMoveMax	Maximum shot spread if the character is lying, moving, and aiming down sight.
ADSProneMoveMin	Minimum shot spread if the character is lying, moving, and aiming down sight.
ADSProneMoveSpreadDec	The decrease of shot spread if the character is lying, moving, and aiming down sight.
ADSProneMoveSpreadInc	The increase of shot spread if the character is lying, moving, and aiming down sight.
ADSRecoilDec	Weapon recoil decrease if the character is aiming down sight.
ADSRecoilLeft	Weapon recoil to the left if the character is aiming down sight.
ADSRecoilRight	Weapon recoil to the right if the character is aiming down sight.
ADSRecoilUp	Weapon recoil upwards if the character is aiming down sight.
ADSStandBaseMax	Maximum shot spread if the character is standing, not moving, and aiming down sight.
ADSStandBaseMin	Minimum shot spread if the character is standing, not moving, and aiming down sight.
ADSStandBaseSpreadDec	The decrease of shot spread if the character is standing, not moving, and aiming down sight.
ADSStandBaseSpreadInc	The increase of shot spread if the character is standing, not moving, and aiming down sight.
ADSStandMoveMax	Maximum shot spread if the character is standing, moving, and aiming down sight.
ADSStandMoveMin	Minimum shot spread if the character is standing, moving, and aiming down sight.
ADSStandMoveSpreadDec	The decrease of shot spread if the character is standing, moving, and aiming down sight.
ADSStandMoveSpreadInc	The increase of shot spread if the character is standing, moving, and aiming down sight.
AltDeployTime	Remnant parameter gathered by the script.
Ammo	Type of ammunition used in the weapon.
Bdrop	A bullet's drop due to gravity.
BRoF	The rate of fire in burst mode.

Parameter	Explanation
BridgeDelay	Delay added to the first reloaded single bullet.
Class	Defines which character class in BF1 uses this weapon.
Damages	Damage points applied, listed based on distance.
DeployTime	Is the time it takes to weapon be able to fire after switching to said weapon.
Dmg_distances	The distances in correlation to each damage entry in the "Damages" parameter.
Drag	A bullet's drag.
Edmg	Least applied damage points.
FirstShotADSSpreadMul	Spread multiplier applied on the first shot or the final shot in burst mode if the character is aiming down sight.
FirstShotHIPSpreadMul	Spread multiplier applied on the first shot or the final shot in burst mode if the character is not aiming down sight.
FirstShotRecoilMul	Recoil multiplier applied on the first shot or the final shot in burst mode.
FirstSingleBulletTime	The time it takes to reload the first bullet if single bullet reloading is available.
HIPCrouchBaseMax	Maximum shot spread if the character is crouching, not moving, and not aiming down sight.
HIPCrouchBaseMin	Minimum shot spread if the character is crouching, not moving, and not aiming down sight.
HIPCrouchBaseSpreadDec	The decrease of shot spread if the character is crouching, not moving, and not aiming down sight.
HIPCrouchBaseSpreadInc	The increase of shot spread if the character is crouching, not moving, and not aiming down sight.
HIPCrouchMoveMax	Maximum shot spread if the character is crouching, moving, and not aiming down sight.
HIPCrouchMoveMin	Minimum shot spread if the character is crouching, moving, and not aiming down sight.
HIPCrouchMoveSpreadDec	The decrease of shot spread if the character is crouching, moving, and not aiming down sight.
HIPCrouchMoveSpreadInc	The increase of shot spread if the character is crouching, moving, and not aiming down sight.

Parameter	Explanation
HIPProneBaseMax	Maximum shot spread if the character is lying, not moving, and not aiming down sight.
HIPProneBaseMin	Minimum shot spread if the character is lying, not moving, and not aiming down sight.
HIPProneBaseSpreadDec	The decrease of shot spread if the character is lying, not moving, and not aiming down sight.
HIPProneBaseSpreadInc	The increase of shot spread if the character is lying, not moving, and not aiming down sight.
HIPProneMoveMax	Maximum shot spread if the character is lying, moving, and not aiming down sight.
HIPProneMoveMin	Minimum shot spread if the character is lying, moving, and not aiming down sight.
HIPProneMoveSpreadDec	The decrease of shot spread if the character is lying, moving, and not aiming down sight.
HIPProneMoveSpreadInc	The increase of shot spread if the character is lying, moving, and not aiming down sight.
HIPRecoilDec	Weapon recoil decrease if the character is not aiming down sight.
HIPRecoilLeft	Weapon recoil lower bound of random recoil if the character is not aiming down sight.
HIPRecoilRight	Weapon recoil upper bound of random recoil if the character is not aiming down sight.
HIPRecoilUp	Weapon recoil upwards if the character is not aiming down sight.
HIPStandBaseMax	Maximum shot spread if the character is standing, not moving, and not aiming down sight.
HIPStandBaseMin	Minimum shot spread if the character is standing, not moving, and not aiming down sight.
HIPStandBaseSpreadDec	The decrease of shot spread if the character is standing, not moving, and not aiming down sight.
HIPStandBaseSpreadInc	The increase of shot spread if the character is standing, not moving, and not aiming down sight.
HIPStandMoveMax	Maximum shot spread if the character is standing, moving, and not aiming down sight.

Parameter	Explanation
HIPStandMoveMin	Minimum shot spread if the character is standing, moving, and not aiming down sight.
HIPStandMoveSpreadDec	The decrease of shot spread if the character is standing, moving, and not aiming down sight.
HIPStandMoveSpreadInc	The increase of shot spread if the character is standing, moving, and not aiming down sight.
HorDispersion	Horizontal pellet dispersion (for shotguns).
InitialSpeed	Muzzle velocity.
MagSize	Size of one magazine.
NumBulletsReloaded	The Number of bullets reloaded when using magazines or strip clips.
PostReloadDelay	Post-reload delay after reloading mechanic ends.
ReloadDelay	Pre-reload delay before actual reloading mechanic begins. (Not strip clip reload or single bullet reload)
ReloadEmpty	The time it takes if the magazine is empty.
ReloadLeft	The time it takes if the magazine there is still some ammo left.
ReloadThrs	The fraction of the reload time before someone can swap weapons and still receives fully reloaded ammo.
RoF	The rate of fire.
SDmg	Maximum damage points without taking distance into account.
ShotsPerBurst	The number of pellets of one shot in burst mode.
ShotsPerShell	The number of pellets of one shot.
SingleBulletReloadTime	The time it takes to reload single bullets after "FirstSingleBulletTime" was applied. (Semi-automatic and bolt-action weapons)
StripClipSize	The number of bullets reloaded by "strip" clips.
StripReloadTime	The time it takes to reload the stripper clip.
TimeToLive	Seconds how long the bullet lives before despawning.
VerDispersion	Vertical pellet dispersion (for shotguns).
__CustomReload__	This parameter only applies in super individual cases.

Table 11: BF1 community weapon analytic dump data parameter.

C Battlefield 1 Weapon Distribution Charts

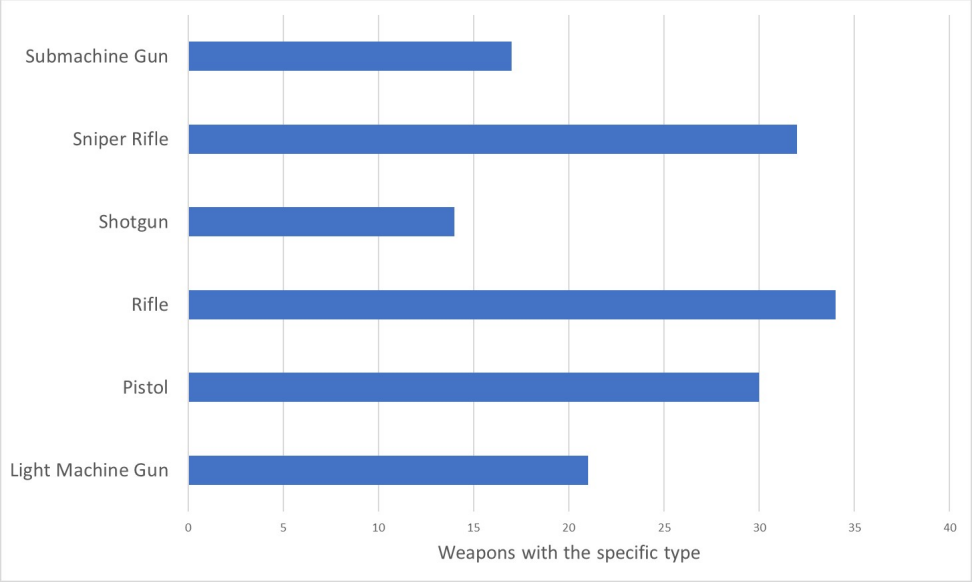

Figure 22: Distribution of weapon types in the BF1 weapon dataset.

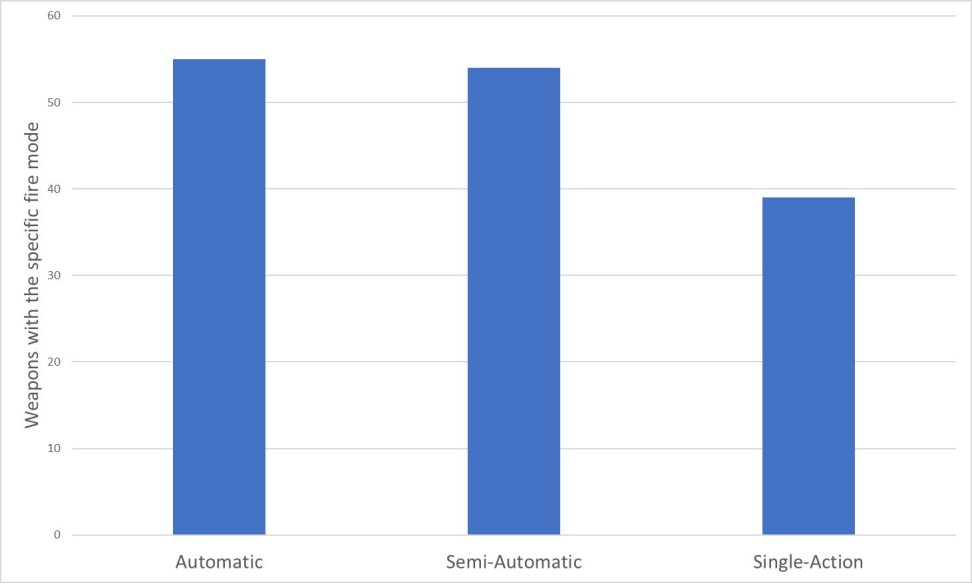

Figure 23: Distribution of weapon fire modes in the BF1 weapon dataset.