Michael Steurer

Automatic Gametesting

Bachelor's Thesis
to achieve the university degree of
Bachelor of Science
Bachelor's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor
Dipl.-Ing. Dr.techn. BSc Johanna Pirker

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Graz, August 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present bachelor's thesis.

Date	Signature

Abstract

Game design poses a difficult challenge of adjusting a game in order to get the gameplay feeling right for the end user. The task of optimizing the game experience entails the adjustment of a wide range of parameters to find the optimal game. This can represent a rather impossible task for a game designer to carry out. Therefore, this thesis explores the possibilities of solving this task by developing a model to perform automatic game testing. The motivation behind this automation is to develop a viable replacement for user testing by estimating the difficulty of a parameterized version of the game. Different game configurations are randomly generated and simulated using Monte Carlo simulation with synthetic players. As a side effect of random level generation, automatic game generation is prone to the discovery of new game variants of the game. These new game variants allow game designers to discover new possible ideas and to gather new inspiration.

Contents

List of Figures

1 Introduction

Game designers and developers need to adjust a vast variety of game parameters, which drastically affects the game's difficulty and feel, in order to achieve a desired gameplay experience. It is very hard to get a sense for how a developed game plays and therefore to find the right difficulty to produce a game experience that keeps the user engaged. Game developers become experts of their own game during the development and testing and hence have to guess the player's skill in order to set the parameters accordingly. This lack of knowledge can only be averted by doing extensive user testing of the game. Many, especially smaller game studios or indie developers, do not have the possibility and financial ability to perform these tests and consequently must completely rely on their guessing. This thesis explores the possibility of doing automated game testing by simulating artificial players to reach a sweet spot regarding a game's difficulty.

This work targets to solve the problem of automated difficulty estimation using Monte Carlo simulation of a parameterized version of "Line Runner", which is implemented using Unity. The content of this work does not only include an in-depth description of the parameterized level generation, but also covers the implementation of an Artificial Intelligence designed to play random generated games. In addition, it also highlights the importance of finding unique game variants that can serve as a foundation of new game ideas for designers.

The paper "Measuring the level of difficulty in single player video games" by Aponte, Levieux, and Natkin provides a way to calculate a games difficulty and tries to relate it to a player's ability. Isaksen, Gopstein, and Nealen successfully demonstrate the ability of automated game difficulty estimation of a parameterized version of the game Flappy Bird using Monte Carlo simulation (Isaksen, Gopstein, & Nealen, 2015).

While other papers use a complex approach in modeling a game's difficulty, this thesis tries a more simplified way. The aim is to use a combination of mean and standard deviation of the player's scores from a simulation in order to model the difficulty.

The following description of the chapters shows how this thesis is structured. Chapter 2 covers the theoretical background of game space, Monte Carlo simulation and the pseudo random number generation. Chapter 3 talks about related work in the context of automated gametesting. Chapter 4 describes how the implementation of the automated gametesting is structured. Chapter 5 explains how the Level Generator, the Simulation and the AI is built into the game. Chapter 6 analyzes the outcome of the simulation and highlights unique levels that were found during the random generation. Chapter 7 discusses issues that arose during the development of this work. Chapter 8 gives an outlook to possible future additions. Eventually , chapter 9 summarizes and concludes this thesis by explaining the importance of this work's approach.

2 Background

This chapter focuses on the theoretical background of this work. Hence, it provides an overview about Monte Carlo Simulation as well as the pseudo random generation used in this work.

2.1 Game Space

Games can be partitioned into a set of different parameters that define a fixed gaming experience. Game space as defined by Isaksen, Gopstein, and Nealen represents the whole spectrum of possible games that can be generated by changing the parameters. Each one of these games has its own difficulty value. The difficulty of selecting a parameter vector for the game is to assess an assignment that reflects into a game that is neither too easy nor too hard for people to play.

2.2 Monte Carlo Simulation

Monte Carlo Simulation uses repeated random sampling and statistical analysis to compute a deterministic result. It only needs to sample a mere fraction of the complete value space in order to deduce a valid result. Monte Carlo methods are used to solve difficult or impossible problems (Raychaudhuri, 2008).

Monte Carlo is typically performed by using the following steps (Raychaudhuri, 2008):

1. A deterministic model which closely resembles the real scenario is developed.
2. Identify the underlying distributions which govern the input variables.

3. Generate multiple sets of random numbers from these distributions and collect sets of possible output values.
4. Perform statistical analysis on the output values of the simulation.

2.3 Pseudo Random Number Generator

A pseudo random number generator is a deterministic algorithm which outputs a sequence of numbers which appear to be random. The pseudo random number generator is initialized with a seed vector that determines the number sequence generated by the algorithm. Using the same seed value as initialization, the random number generator is able to always produce the same sequence of numbers (Karimovich, Turakulovich, & Ubaydullayevna, 2017).

2.3.1 Lehmer Pseudo-Random Number Generator

Lehmer's Pseudo-Random Number Generator is a simple and efficient algorithm that satisfies almost any statistical test of randomness and can produce a virtually infinite sequence of numbers (Park & Miller, 1988).

Formula 2.1 shows how Lehmer's Generator can be used to calculate a pseudo-random number, where the modulus m is a prime number and K is a primitive root for the modulus. Lehmer suggested $K = 14^{29}$, $m = 2^{31} - 1$ and $0 < X_0 < 2^{31} - 1$ (Payne, Rabung, & Bogyo, 1969).

$$X_{n+1} = K \cdot X_n \mod m \tag{2.1}$$

3 Related Work

3.1 Exploring Game Space Using Survival Analysis

"Exploring Game Space Using Survival Analysis." by Isaksen, Gopstein, and Nealen describes how to use automated play testing to calculate the difficulty of variants of a parameterized version of the game Flappy Bird. The difficulty of a game is calculated using a Monte Carlo simulation and a player model based on human motor skills. After running the simulation, the distribution of scores is analyzed using exponential survival analysis (Isaksen, Gopstein, & Nealen, 2015). The paper "Comparing player skill, game variants, and learning rates using survival analysis" by Isaksen and Nealen gives a mathematical overview of survival analysis and analyzes the impact of learning effects on the score.

The papers (Isaksen, Gopstein, & Nealen, 2015; Isaksen, Gopstein, Togelius, & Nealen, 2018) further explore techniques to efficiently find levels with a certain difficulty using an optimization algorithm.

Exploratory computational creativity is the process of finding playable game variants that are most different from already discovered versions. This is helpful for designers to find inspiration, new ideas, and interesting game variants, as is covered in (Isaksen, Gopstein, Togelius, & Nealen, 2015). A method to find games of varying difficulty and help designers explore game space is presented (Isaksen, Gopstein, & Nealen, 2015).

Furthermore, a user study is done to compare the predicted difficulties to human play testers, showing the model is effective at predicting game difficulties (Isaksen, Gopstein, & Nealen, 2015).

3.2 Measuring the Level of Difficulty in Single Player Video Games

"Measuring the level of difficulty in single player video games" by Aponte et al. discusses the interest and the need to evaluate the difficulty of games. The paper shows that by using a simple experiment it is possible to obtain a difficulty curve for a game whose difficulty can be tuned accordingly to a parameter. In their experiment a Pacman-like game was used and the difficulty was calculated using the number of eaten pellets. They used the speed parameter to analyze the difficulty.

The paper argues that there is a lack of a precise definition of difficulty in games and further game designers lack of a methodology as well as tools to measure it. A first step to fill this gap is performed by proposing a measurable definition of difficulty for challenges and providing a method to explore the relation between what the player learns and the probability he has of overcoming a particular challenge (Aponte et al., 2011).

3.3 Procedural Personas for Playtesting

The paper (Holmgard, Green, Liapis, & Togelius, 2018) demonstrates automated playtesting of the game "MiniDungeons 2" using artificially intelligent gameplaying agents. The personas are controlled using a variant of Monte Carlo Tree Search (MCTS). Experiments show that the personas can help map out the gamespace afforded by game levels while designing it (Holmgard et al., 2018).

3.4 Exploring Automated Game Testing

"Exploring Automated Game Testing" by Larch builds on top of this thesis and covers various optimizations to enhance the amount of playable levels that are generated. The implementation of this work is ported to WebGL and various webpages to allow easier level tweaking and analysis are created. Additionally, it conducts a user study to verify the results of the difficulty calculations done in "Exploring Automated Game Testing." Larch finds an accordance between the difficulty calculations and the results of his user

tests.
Larch further proposes an approach to difficulty prediction of levels by using regression and demonstrates its capability by validating the predicted values. By analyzing the relation between parameters and the game's difficulty Larch is able to develop two different approaches into achieving dynamic difficulty adjustment in Line Runner (Larch, 2018).

4 Design

The design of the implementation of this thesis orients itself on the previous work of Isaksen, Gopstein, and Nealen, 2015. The goal of this work is to analyse a different game and to apply a simpler statistical difficulty calculation in order to classify all randomly generated games.

4.1 Game Description

The game developed and analysed in this work is a simple 2D platformer with constant side-scrolling that is called Line Runner. The challenge of the game is to jump over obstacles and holes in order to get as far as possible without dying. A screenshot of the final game can be seen in Figure 4.1

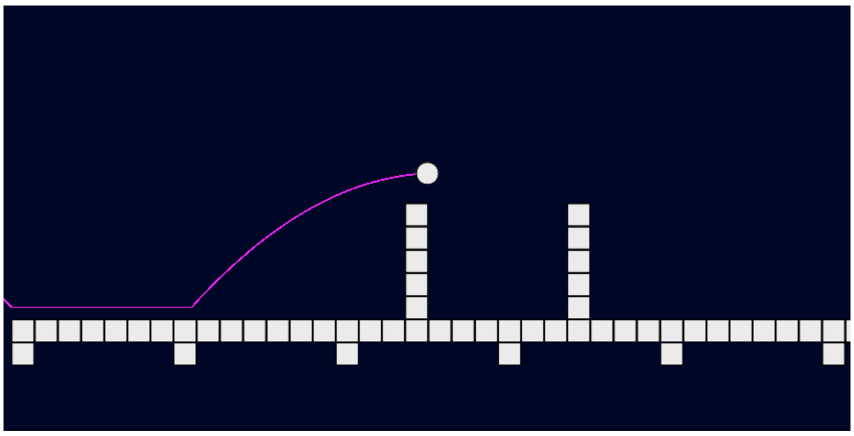


Figure 4.1: A screenshot of Line Runner.

Each level of the game consists of a sequence of the three different block types `BlockType.Flat`, `BlockType.Obstacle` and `BlockType.Hole` shown in Figure 4.2. If the user crashes into an obstacle block or falls into a hole block, the game ends.

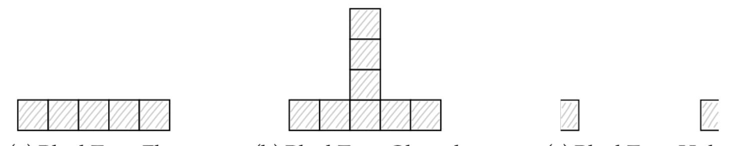


Figure 4.2: Different block types that can be found in the game.

The user interaction of the game is kept very simple as the user is only able to perform one action to interact with the game. By pressing the 'Space Bar' key on the keyboard the user can apply a vertical force to the player's rigid body. This allows the user to leap the player over holes and obstacles in order to avoid a game over.

4.2 Game Parameters

Liner Runner is a game that is defined by certain rules and parameters that define a certain game behaviour, whereas each one of the parameters can be adjusted independently. Table 4.1 lists all the parameters that define the Liner Runner's behaviour of the implementation in this thesis.

Parameter	Symbols	Description
P(flat)	p_{flat}	Probability of `BlockType.Flat` blocks
P(hole)	p_{hole}	Probability of `BlockType.Hole` blocks
P(obstacle)	$p_{obstacle}$	Probability of `BlockType.Obstacle` blocks
speed	v	Horizontal speed of the player
force	f_{jump}	Vertical force applied to the players when jumping
gravity	f_g	Gravity constant of the simulation (in g)
obstacle height	h_{obst}	Height of the obstacle blocks in the game
block length	l_{block}	Length of the blocks of the game
seed	$seed$	Initialisation vector for the random generator used in the game. Describes a unique sequence of blocks per seed.
max players	$n_{players}$	Amount of synthetic players used for testing.

Table 4.1: Modifiable parameters of Liner Runner

All parameters are chosen randomly before the game starts and are kept constant during each game session. The game's definition doesn't allow an alteration of the parameters during simulation, due to the fact that it could lead to a positive or negative influence of the game. It can be used for dynamic difficulty adjustment (DDA) of the game (Larch, 2018).

4.2.1 Force Parameters

The parameters f_{jump}, f_g together define the force the player is able to move in vertical direction and is therefore used to determine the player's capacity to jump.
Only the parameter v defines the player's velocity in horizontal direction, which is kept constant throughout the simulation of a level.

4.2.2 Block Type Probabilities

P_{flat}, $P_{obstacle}$ and P_{hole} are used by the level generator to determine the distribution of the different block types. The sum of these three probabilities must be exactly 1, otherwise the level generator would fill up the blocks not covered by any probability with type `BlockType.Flat`. Depending on the block placement of this distribution the game can result in a different difficulty.

4.2.3 Seed

The seed allows to regenerate the exact same sequence of block types. It initializes a separate random generator, which allows to change seed and keep the same parameters. In addition, this gives the opportunity to change other parameters (except block type probabilities) but retain the seed and therefore the block sequence.

4.3 Pseudo Random Generator

At the beginning of the simulation process, a pseudo random number generator randomizes the block types. The pseudo random number generator generates block sequences that only appear random. Its "pseudo" randomness is sufficient enough for the usage in this thesis. The Lehmer Pseudo Random Number Generator is used in the implementation. It allows to have better control over the number generation as well as the reconstruction of the same number sequence by initializing the generator with a seed value. This trait becomes useful since it allows to reconstruct and replay the exact same game, which can be used for a more thorough analysis.

4.4 Player Model

For game testing to get the best results the game must be tested by real human players. This process poses a challenge for small game development studios, since it is highly cost and time-consuming to realize. The following section of this thesis explores the possibility of automating this process, by replacing human players with synthetic ones. Using such a virtual player model to repliacte human behaviour, the results of human testers can be approximated. In order to develop this player model a reaction time test and a hits-per-second test were carried.

4.4.1 Reaction Time Test

The first test conducted for this thesis is the reaction time test. For this purpose five people between 18 and 35 years were selected. They had to perform the task of interacting with a python script to measure their reaction time. The listing below shows an outline how the test program, that was used, works. The test's measurements resulted in a average reaction time of $t_{react} = 327$ milliseconds.

- State 1: „Ready?“ text gets shown to the user
- State 2: When the user hits the return key of the keyboard, the scripts waits between 1-2 seconds and then outputs the text „Hit return!“
- State 3: The user hits the return key again and the program moves back to the first state. When this process has been repeated for 10 times the program prints the average reaction time and exits.

Person	Reaction time [s]
Testperson 1	0.327
Testperson 2	0.328
Testperson 3	0.332
Testperson 4	0.336
Testperson 5	0.311
Average	0.327

Table 4.2: The average reaction time of each user summarized in the overall average reaction time.

4.4.2 Hits Per Second Test

A further advantage machines have over humans is their ability to trigger an action repeatedly in a fraction of a second. Humans are not able to press a button at a steady interval at a high speed. This disadvantage is subject to another user study in order to measure the exact hits humans are able perform per second. The test users were asked to continuously press the return key of the keyboard as fast as they could for 10 seconds. The test persons were able to perform an average number of $n_{hits} = 7.257$ hits per second. This value does not include an estimation of human fatigue, since it does not influence the outcome of a Line Runner game.

Person	Hits per second
Testperson 1	8.007
Testperson 2	7.005
Testperson 3	7.734
Testperson 4	6.657
Testperson 5	6.884
Average	7.257

Table 4.3: The hits-per-second scores for each user and the average score

4.5 Simulation

A Monte Carlo simulation paired with an actor system is used for player simulation. During each update of the simulation each player calculates its next ideal jump position. This ideal jump position will be offset by a randomly chosen value between $-l_{react}/2$ and $l_{react}/2$, where l_{react} denotes the distance traveled by the player in the reaction time t_{react}. This approach allows the program to simulate human players and calculate the difficulty of the game in the game space.

The results of the simulation are dependent on the amount of players used. If this number is too low, the result of the simulation will be highly variable and therefore the results would not be accurate. On the other hand the exaggeration of the amount of players also would not deliver any better results, because the simulation would start to lag and consequently become unusable. Hence, the amount of players should not be too low while at the same time also not too high that the simulation machine is still able to maintain the calculations and the framerate. For this thesis a 2014 MacBook Pro was used as test device. The best results on this testing device were achieved with 100 players, which is high enough to eliminate any inaccuracies of the calculated game difficulty and the simulation still runs smoothly.

4.6 Game Difficulty

In order to describe how challenging a game is the variables d, σ_d S and $n_{survivors}$ shown in Table 4.4 are calculated accordingly during simulation time. The two variables difficulty d and difficulty standard deviation σ_d are used to describe the difficulty of a game and stand for the average death position of all players.
The difficulty d is computed by calculating the mean of the difficulty for each player. The difficulty for a player d_{player} is calculated using the player's death position S_{player} and the maximum reachable score S_{max}, as illustrated in Formula 4.1.

$$d_{player} = 1 - \frac{S_{player}}{S_{max}} \tag{4.1}$$

The difficulty d shows how long players were able to survive on an average basis, with $d = 0$ being an easy game where every player made it to the end position S_{max} and $d = 1$ being an impossible game, where every player died instantly at the beginning. In contrast, the difficulty standard deviation shows the distribution of the positions of each player's death. In a game with $\sigma_d = 0$, all users finished the game at the same position, which also indicates that the level is only playable until that position. Games with $\sigma_d > 0$ have a wider distribution of player's death position and are therefore more likely to be playable.

Name	Symbol	Description
difficulty	d	The calculated difficulty of the game.
difficulty_standard_deviation	σ_d	Standard deviation of difficulties per player
Score	S	The score of the best player (time in seconds)
Survivors	$n_{survivors}$	How many players made it to the end position S_{max}

Table 4.4: Variables describing the difficulty of each level.

4.7 Survival Diagram

The individual scores of the players are used to create the survival diagram. The diagram shows how many players were alive at each point in time of the game simulation. It allows for a detailed analysis of the player's survival rate and provides a quick overview of a game's playability. An example of a survival diagram for one level is shown in Figure 4.3.

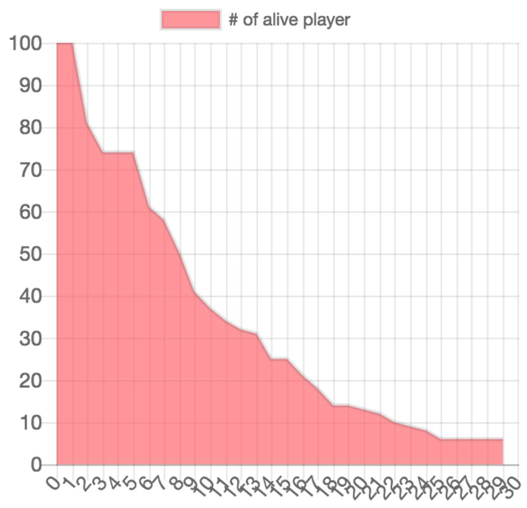
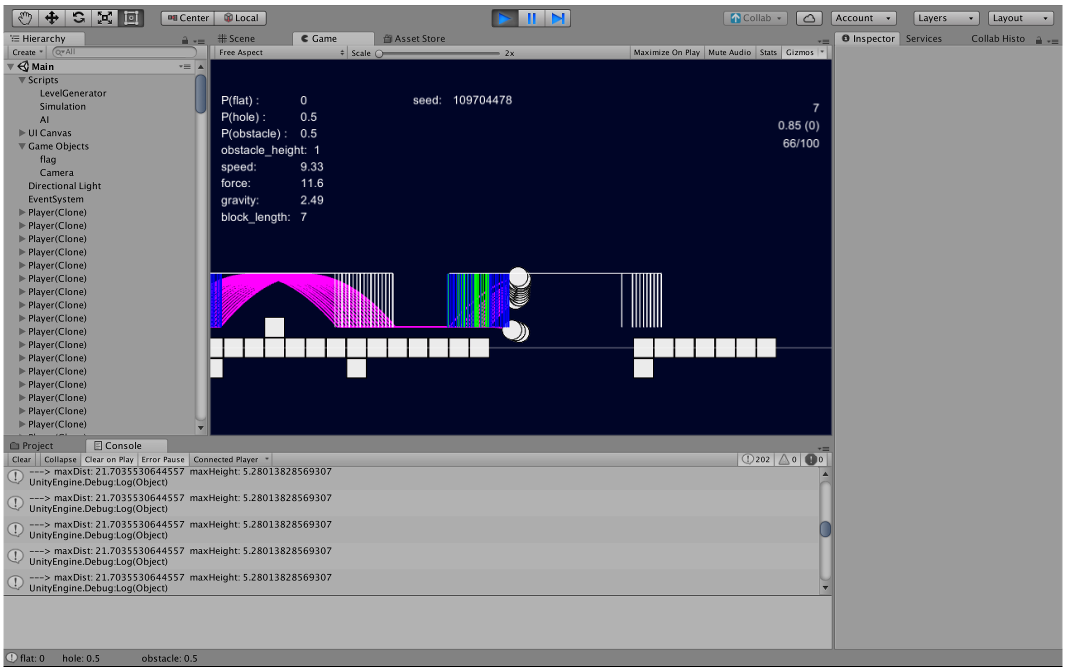


Figure 4.3: The survival diagram shows the number of alive players for each point in time during a game.

5 Implementation

The Line Runner is implemented using the Unity Game Engine (version 2017.2.0f3) as it already includes a physics engine that can be used for the development of the game. Unity also allows to export the game for different platforms such as Android, iOS, WebGL etc. and thus, allows to broaden the target group of players the game can be developed for.

Figure 5.1 shows a screenshot of the Unity editor which was used for developing and debugging the game. It shows labels displaying the current level parameters, the score, the current calculated difficulty and debug rays. The rays were used for debugging the calculated jump positions and estimating the landing positions.



```
P(flat) :          0                seed:   109704478
P(hole) :          0.5
P(obstacle) :      0.5
obstacle_height:   1
speed:             9.33
force:             11.6
gravity:           2.49
block_length:      7
```

```
7
0.85 (0)
66/100
```

```
---> maxDist: 21.7035530644557  maxHeight: 5.28013828569307
UnityEngine.Debug:Log(Object)
---> maxDist: 21.7035530644557  maxHeight: 5.28013828569307
UnityEngine.Debug:Log(Object)
---> maxDist: 21.7035530644557  maxHeight: 5.28013828569307
UnityEngine.Debug:Log(Object)
---> maxDist: 21.7035530644557  maxHeight: 5.28013828569307
UnityEngine.Debug:Log(Object)
---> maxDist: 21.7035530644557  maxHeight: 5.28013828569307
UnityEngine.Debug:Log(Object)
```

```
flat: 0      hole: 0.5          obstacle: 0.5
```

Figure 5.1: A screenshot of the game inside the Unity Editor.

This implementation of the game Line Runner can be configured by various parameters. A modification to one of these parameters can alter the game's experience and lead to a complete different level, yet consisting of the same rules. In order to prevent useless configurations, each one of these parameters is clamped between a minimum and a maximum value as shown in Table 5.1.

Parameter	minimum	maximum
P(flat)	0	0.8
P(hole)	0	1
P(obstacle)	0	1
speed	5	20
force	4	20
gravity	0.6	3.6
obstacle height	1	10
block length	2	40
max players	1	100

Table 5.1: The parameters along with their correlating minimum and maximum values, that are used to prevent useless level generation.

5.1 Structure of the Implementation

The implementation of the game is split into the three parts Simulation, Level Generation, and the Player's AI. In the following sections this thesis will go more into detail on how each one of them is implemented.

5.2 Simulation

The Simulation, as illustrated in Figure 5.2, subsists of four subsequent steps. It starts by randomizing the parameters of the Level Generator and the players. The Level Generator then generates the game based on these parameters and places the different blocks into a specific order. Consequently, the simulation adds the previously defined amount of players to the scene and finally starts the actual game simulation.

The game ends when all players are either game over or reached the maximum score S_{max}. Then the difficulty value is calculated and the simulation is restarted.

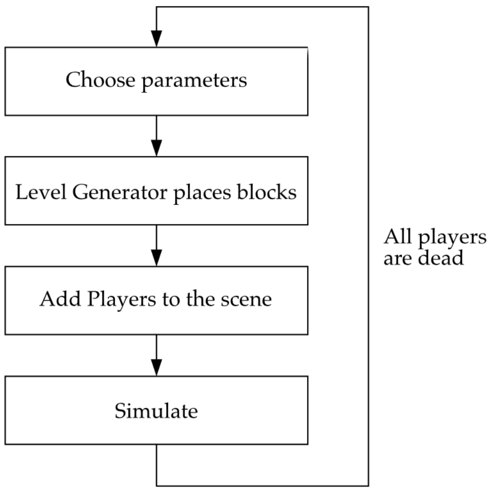


Figure 5.2: A flow diagram of the functionality of the simulation

5.3 Level Generation

The simulation passes the randomly generated parameters into the Level Generator. Afterwards the Level Generator initiates its work by starting to place a sequence of chunks in the scene. Chunks are the outermost abstraction layer used internally by the Level Generator. Each chunk consists of a sequence of 40 blocks. Every block is l_{block} sprites long and has either `BlockType.Flat`, `BlockType.Obstacle` or `BlockType.Hole` as type. The smallest component in the Level Generator's hierarchy is a sprite. The hierarchy used by the Level Generator is visualized in Figure 5.3.
Level generation chunks are generated on the fly, which means that the chunk generation is dependent on the player's position. As soon as a player reaches the middle of the newest previously generated chunk, a new chunk is generated. This on demand generation allows the game to gain performance and minimize the loading overhead, while still enabling an endless game experience.

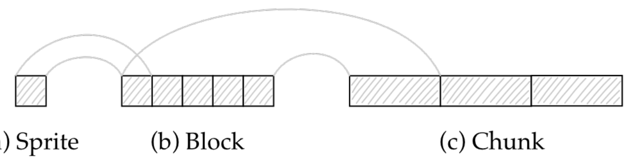


Figure 5.3: The hierarchy used by the Level Generator. Chunks consist of blocks and blocks consist of sprites.

A `BlockType.Flat` block consists of a number of l_{block} horizontally arranged sprites where the player can move along. The `BlockType.Hole` block does not contain any sprites, which results in the game over whenever the player falls through the gap. A `BlockType.Obstacle` block consists of a `BlockType.Flat` block and $h_{obstacle}$ sprites vertically placed in the center of the block. If the player crashes into the obstacle at the center of the block, he goes game over.

To prevent the generation of unplayable games and to give human players time to become familiar with the parameters, the first four blocks of the first chunk placed always include the type `BlockType.Flat`. The types of all other blocks are chosen by the pseudo-random number generator using the defined block probabilities P_{flat}, P_{hole} and $P_{obstacle}$.

An issue that can lead to the generation of impossible games are consecutive `BlockType.Hole` blocks. Therefore, the generation is prohibited to generate a `BlockType.Hole` block followed by another `BlockType.Hole` block. In such a case the random number generator generates a new block type in a loop until a block type different than `BlockType.Hole` is found. To prevent the uncertain possibility of having an endless loop, after *chunk_length* iterations, `BlockType.Flat` is chosen automatically. In the rare case of generating a game that has $p_{hole} = 1$, this will result in alternating `BlockType.Hole` and `BlockType.Flat` blocks.

5.4 AI

During the simulation each player is controlled by an AI, which imitates an ideal player limited by human motor functions. In every FixedUpdate the AI checks whether an obstacle appears on the screen, in which case the next jump position of the player is calculated.

There are four particular cases that need to be taken into account by the AI when calculating the next jump position.

- When a player lands between the next calculated jump position and its obstacle he would jump as soon as he touches the ground. A human player isn't able to do this and therefore the jumping is delayed by a random value between 0 and t_{react}.
- When approaching a `BlockType.Hole` block, the player must always jump on a block prior to the hole or he would fall into them. For obstacles this is not the case, as the player can also move onto the obstacle block and jump over the barrier at the center of the block.
- If the player already passed the center of an obstacle block and lands on the same block, he suppresses the jump and continues by calculating the next jump position.
- If a player lands on an obstacle, he is allowed to jump.

5.4.1 Obstacle Observation Range

To simulate the limited amount of obstacles human players are able to see on screen, the AI is also only able to see a specific amount of blocks ahead. This amount of blocks is dependent on l_{block}, which means the more sprites per block the fewer blocks are visible on the screen at the same time. This value is calculated using formula 5.1. The value 35 is chosen for the usage on a 13″ MacBook Pro and it means that 35 sprites are able to fit on the screen simultaneously.

$$blocks\_visible = max(\left\lfloor \frac{35}{l_{block}} \right\rfloor, 2) \tag{5.1}$$

5.4.2 Jumping Range

When the AI sees an obstacle it calculates the perfect jump position P_{jump} for each player in order to pass the obstacle. Since human players are not able to constantly meet the perfect jumping position, the AI calculates the theoretical prefect jumping position and adds an offset that is randomly selected in the range between $-l_{jump}/2$ and $l_{jump}/2$. l_{jump} which is the distance traveled by the player in the reaction time t_{react}. It is calculated by multiplying the speed of the player in horizontal direction by the human reaction time t_{react} (Formula 5.2).

$$l_{jump} = v * t_{react} \tag{5.2}$$

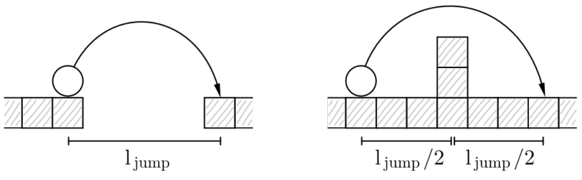
By changing the reaction time t_{react}, l_{jump} adjusts in a directly proportional manner, which allows the AI to simulate an better or a worse player respectively.

5.4.3 Perfect Jump Position

An ideal player would always jump at the perfect position in order to safely jump over an obstacle. Perfect position means, that it aligns the highest point of the jump trajectory at the center of the obstacle block in order to maximize the probability to make it past (Figure 5.4). Formula 5.3 and 5.4 can be used to calculate the perfect position for the current parameter assignment. v_0 denotes the initial speed at the jump origin and α the launch angle of the player. Formula 5.4 calculates the center of the obstacle block by adding $l_{block}/2$ to $obstacle\_position_x$, the origin of the block, and subtracts half the jumping range calculated in 5.3.

$$l_{jump} = \frac{v_0^2 * sin(2\alpha)}{g} \tag{5.3}$$

$$P_x = obstacle\_position_x + \frac{l_{block} * l_{sprite}}{2} - \frac{l_{jump}}{2} \tag{5.4}$$



(a) BlockType.Hole (b) BlockType.Obstacle

Figure 5.4: The perfect jump position to make it past a hole block (a) and an obstacle block (b)

5.4.4 End Position

To force the AI from playing trivial levels endless, the game contains a winning position S_{max}. Once the players surpasses that position, the level is automatically won. While a too small end position comes with a higher loss of accuracy, a high end position implies longer simulation times and therefore an unnecessary additional effort.
To find S_{max}, 20 randomly selected levels were examined on their impact to the overall game difficulty. Figure 5.5 shows that the derivation of the difficulty standard deviation σ_d comes very close to 0 at an end position S_{max} of 30. This position perfectly balances the ratio between simulation time and accuracy of the final result of the simulation.

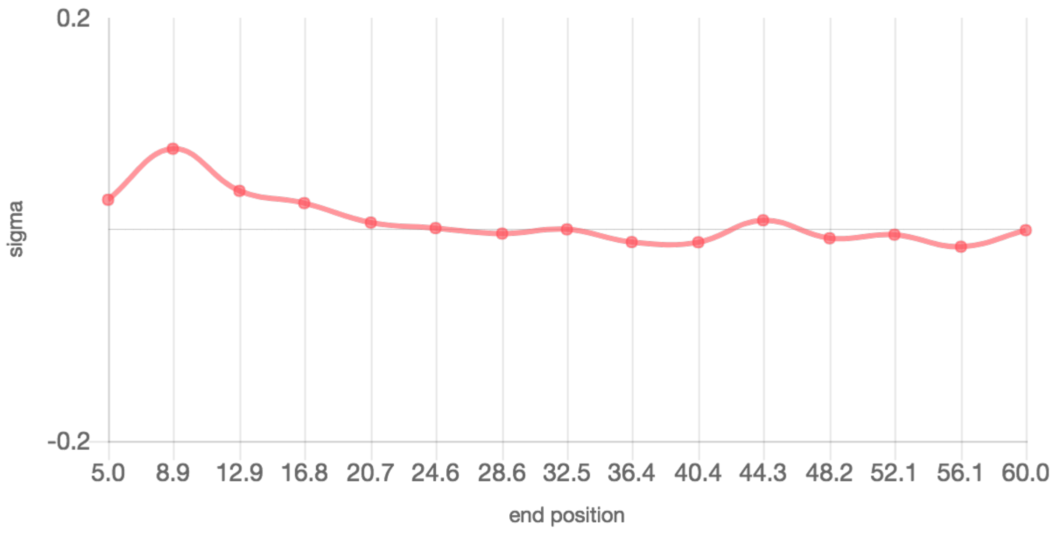


Figure 5.5: The derivation of σ_d when sampling S_{max}.

6 Analysis

The Analysis chapter of this thesis analyses the data of 500 randomly generated levels. It explores the correlations between different parameters in the 2D parameter space of the game. Furthermore, it discusses the difficulty distribution of the levels generated by the Level Generator and why some levels are unplayable. Section 6.3 "Unique Levels" discusses certain outliers that were generated during the random level generation and levels that were created by changing parameters over their limits. The last section compares different games with the same difficulty and analyses them.

6.1 Game Parameter Analysis

This section discusses how different game parameters are affecting the resulting game difficulty.

The graph shown in Figure 6.1 compares the parameters l_{block} and v to d. The data clearly shows that a channel can be drawn across the graph, reaching from bottom left to top right. This graph can be used to partition the set of levels into a set of too easy playable games as well as too hard to play. It also shows that the combination of l_{block} and v can be used to adjust the game difficulty. Most levels with a small l_{block} have a high difficulty and the influence of speed is very small. When evaluating games with a higher l_{block}, one can see that the speed clearly influences a game's difficulty. Furthermore, games with a higher l_{block} value, can be transformed into a more difficult ones by raising the speed parameter.

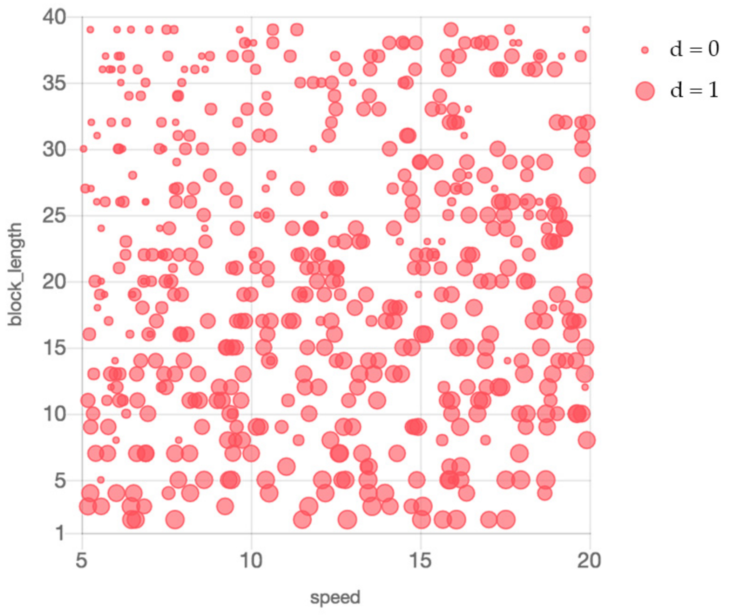


Figure 6.1: This graph shows the impact of the parameters l_{block} and v on the difficulty d which is visualized using the radius of the points.

A similar result emerges when comparing l_{block} and v to σ_d. Most of the games with a $\sigma_d > 0$ are in the lower right section of the graph in Figure 6.2. Meaning the higher the speed v, the bigger l_{block} can be without creating a game with $\sigma_d = 0$. This graph perfectly shows that l_{block} and v have to be balanced in order to create a playable and challenging game with $\sigma_d > 0$.

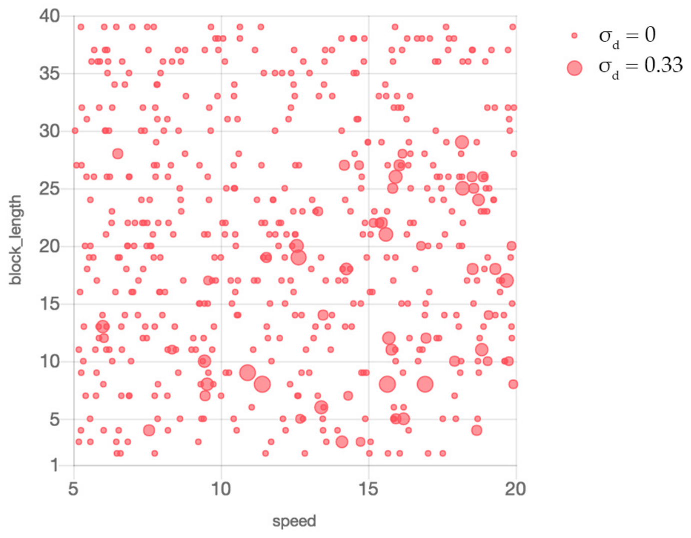


Figure 6.2: This graph shows the impact of the parameters l_{block} and v on the difficulty standard deviation σ_d which is visualized using the radius of the points.

Figure 6.3 shows that at a higher speed v most of the games that are generated result in a higher difficulty d. In contrast low speed leads to a more uniform distribution of the difficulties amongst the generated levels.

The graph in Figure 6.4 shows that games with a lower p_{flat} mostly have a higher difficulty while games with a higher P_{flat} have a more uniform distribution of the difficulties and tend to be easier.

Of the 135 games with $P_{flat} = 0$, only 11.11% have $d = 0$. Of the 187 games with $P_{flat} = 0.8$ on the other hand, 30.48% have $d = 0$.

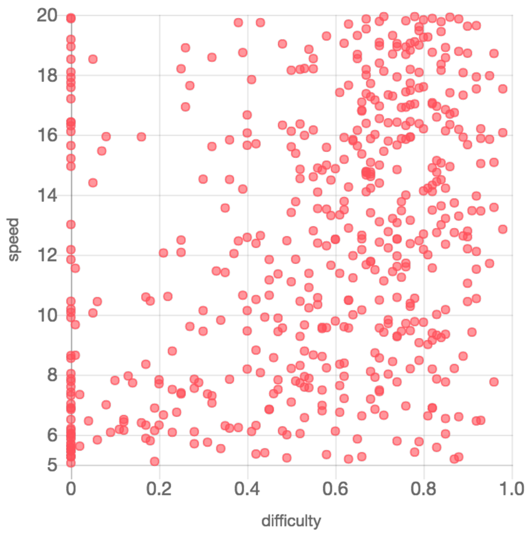


Figure 6.3: This graph shows the impact of v on the difficulty d

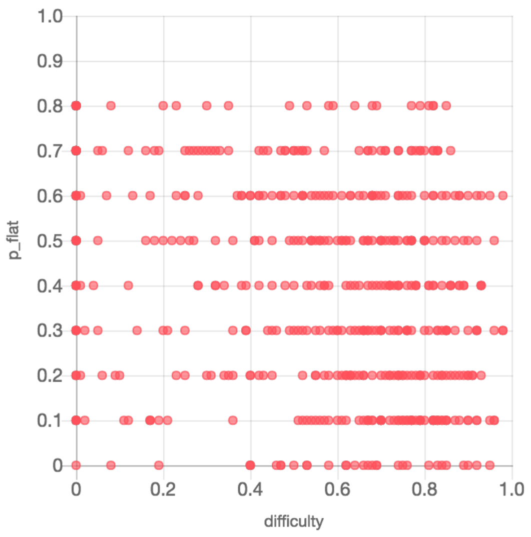


Figure 6.4: This graph shows the impact of p_flat on the difficulty d

6.2 Game Difficulty Probability

The game difficulty probability distribution allows to measure the efficiency of the level generation. The graphs of this section show the distribution of d

and σ_d among the games generated in the random level generation.

The game difficulty has a peak at $d = 0$ with 10.5% as can be seen in Figure 6.5, these are levels where every player made it to the end position S_{max}. This means that 78.8% of all generated games are either impossible or playable.

The graph in Figure 6.6 shows that no level has reached a difficulty value of $d = 1$. Upon closer inspection of the generated data, a upper difficulty boundary of $d = 0.98$ becomes visible, where no game has reached a difficulty above this boundary. This can be explained due to the way the levels are generated. A difficulty of $d = 1$ would only occur in the case of all players dying immediately at the starting position. This is not possible because the first four blocks of a game are predefined to always be of type `BlockType.Flat` and therefore the players need time $t > 0$ to reach the first possible death position.

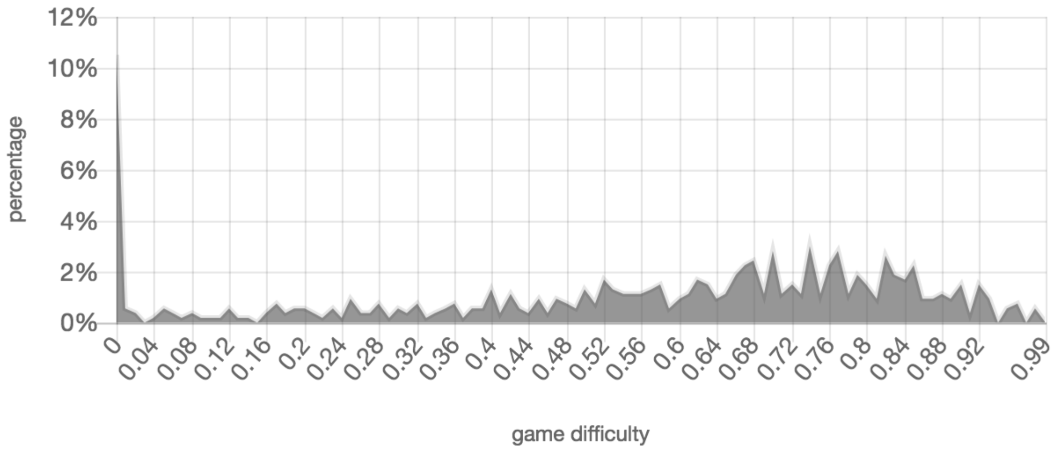


Figure 6.5: This graph shows the distribution of game difficulties generated by the level generator.

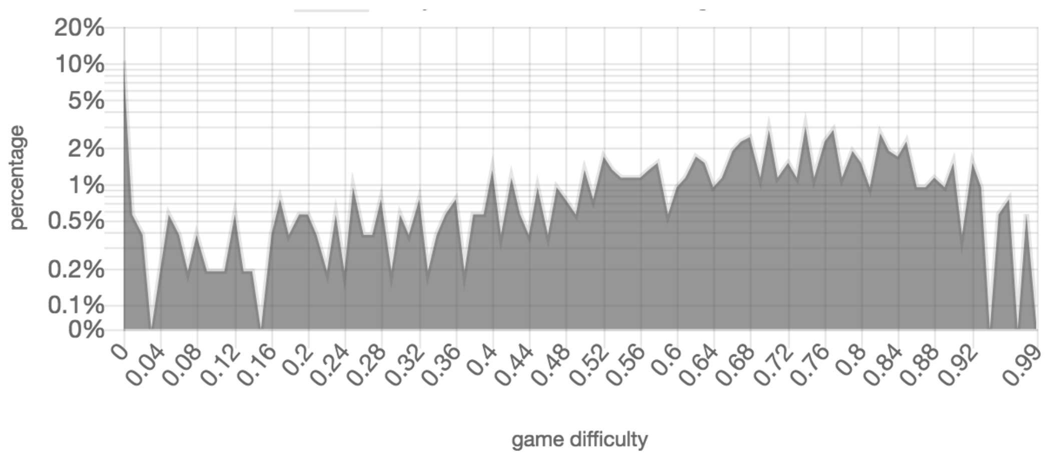


Figure 6.6: This graph shows the distribution of game difficulties generated by the level generator with a logarithmic scale.

Figure 6.7 shows that 89.3% of the randomly generated games have a difficulty standard deviation $\sigma_d = 0$, which means that for all players the game ended at the same position. This can happen in the cases where either all players die at the same position or all players win the game. Therefore, these levels are either impossible to complete or too easy to be played. Hence, the remaining 10.7% of the generated levels are the ones, which are the most interesting levels to play.

Figure 6.8 shows that the amount of games with $\sigma_d = 0$ peaks at $d = 0$ and has a slight increased amount of games in the upper half of d.

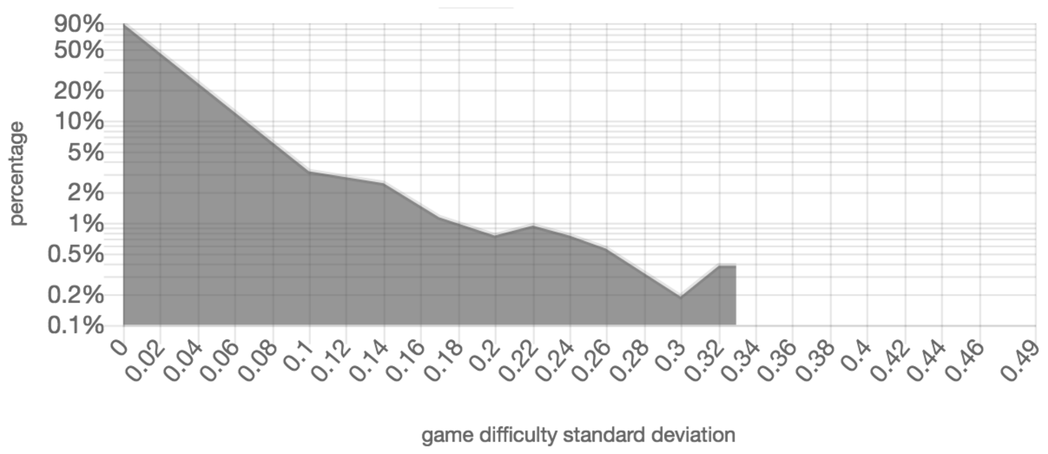


Figure 6.7: This graph shows the distribution of σ_d generated by the level generator with a logarithmic scale.

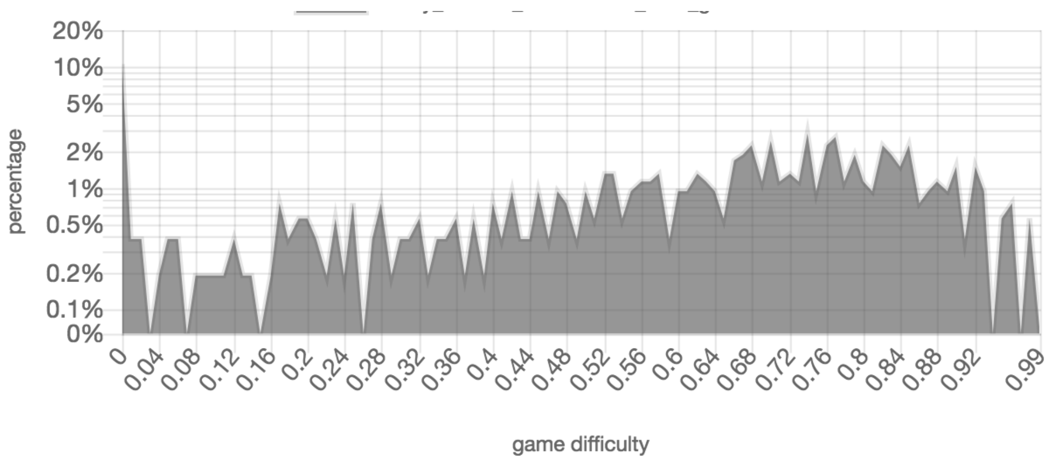


Figure 6.8: This graph shows the distribution of d with $\sigma_d = 0$ generated by the level generator with a logarithmic scale.

When looking at the difficulty probabilities of games with $\sigma_d > 0$, Figure 6.9 shows that the amount of games generated for each difficulty has only a slight increase in the amount of games in the upper half of d. Furthermore, no game was generated with $d > 0.86$ and $\sigma_d > 0$.

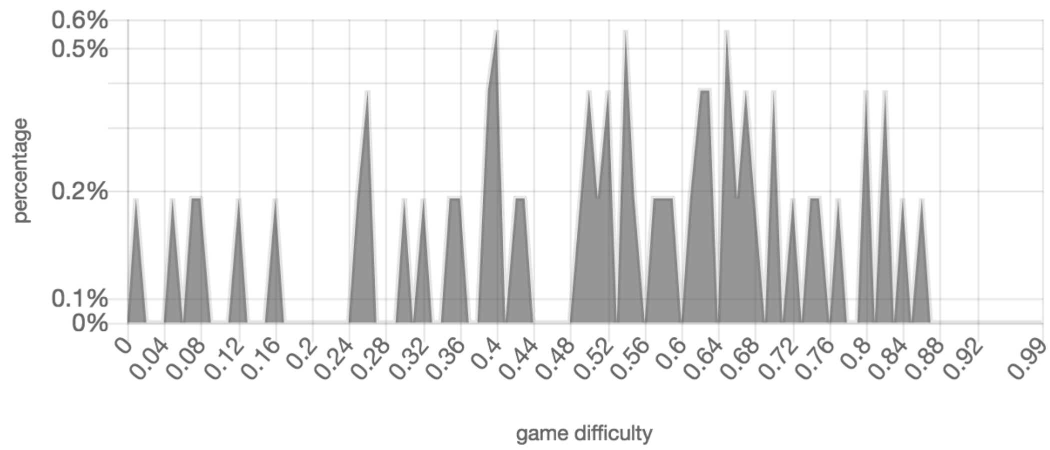


Figure 6.9: This graph shows the distribution of d with $\sigma_d > 0$ generated by the level generator with a logarithmic scale.

6.3 Unique Levels

The random generation of the games provided not only a number of interesting data, but also many unique parameter configurations were found by coincidence. These unique configurations demonstrate the creativity of the random parameter selection, which allows game designers to find inspiration and possible new level variations of the same game. This section of the paper highlights these kind of games and further discusses games that emerged when changing parameters over their defined limits.

6.3.1 Random Generated Games

The game with the parameters as described in Listing 6.1 has a moon-like feeling to it because of the relatively low gravity. The game has a difficulty $d = 0.47$ and $\sigma_d = 0.14$ and is quite difficult to play, because the player jumps over multiple blocks.

Listing 6.1: The so called "Moon Game" with a low gravity.

```
{
"p_flat":"0.40",
"p_hole":"0.50",
"p_obstacle":"0.10",
"speed":"12.8",
"force":"10.1",
"gravity":"0.6",
"obst_height":3,
"block_length":7,
"seed":103454134
}
```

Figure 6.10 shows a game where a human player would not be able to predict the landing position and therefore has to be lucky. This makes the game hard to play and therefore it has a difficulty of $d = 0.71$ and $\sigma_d = 0.1$. The parameters of this game are described in Listing 6.2.

Listing 6.2: The so called "Monte Carlo Game". A human player is not able to predict where the player will land.

```
{
"p_flat":"0.00",
"p_hole":"1.00",
"p_obstacle":"0.00",
"speed":"10.9",
"force":"20",
"gravity":"0.8",
"obst_height":4,
"block_length":3,
"seed":111841765
}
```

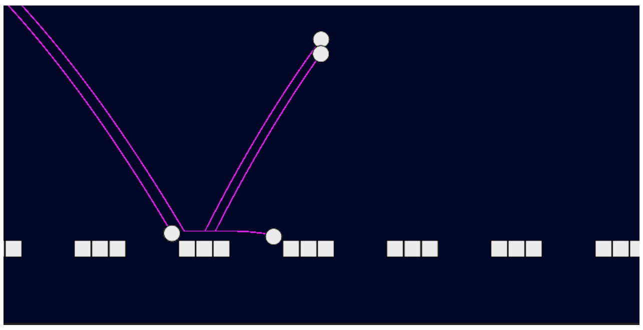


Figure 6.10: The so called "Monte Carlo Level" needs a lucky hand to be won.

A game with a low level of interest emerges if low speed is combined with a high l_{block}, as described in Listing 6.3. In this game neither a hole nor an obstacle can be passed, which results in making the level pointless to play for human players.

Listing 6.3: A game with small v and high l_{block}. No hole or obstacle is reached.

```
{
"p_flat":"0.10",
"p_hole":"0.80",
"p_obstacle":"0.10",
"speed":"5",
"force":"9.9",
"gravity":"2.8",
"obst_height":1,
"block_length":25,
"seed":107467817
}
```

In the following level the AI is able to jump onto an obstacle and continue the game by jumping from obstacle to obstacle. This level is impossible for human players because it they aren't able to jump from the top of one obstacle to the next one. Figure 6.11 shows this game being played by the AI. Listing 6.4 contains the configuration of this game.

Listing 6.4: A game where the AI is able to jump from obstacle to obstacle.

```
{
"p_flat":"0.00",
"p_hole":"0.10",
"p_obstacle":"0.90",
"speed":"6.37",
"force":"6.09",
"gravity":"1.91",
"obst_height":1,
"block_length":2,
"seed":93542070
}
```

The next type of games that emerged during game generation, were games with optional user input, such as the one described in Listing 6.5. These

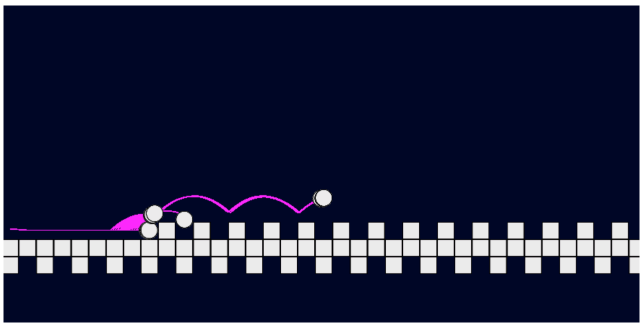


Figure 6.11: This game contains only obstacle blocks and is impossible for human players because it is too difficult to jump from obstacle to obstacle

games can be easily played by humans without pressing any button. The AI on the other hand is not smart enough to detect games that do not need any user input, and consequently will perform much worse than a user. Because the AI is not able to learn from past mistakes it repeats the same tries. Humans on the other hand can learn from their mistakes and try the game again. This means they have another clear advantage over machines. In the context of playability, these kinds of levels should be avoided to ensure a satisfying user experience. A game of this type can be seen in Figure 6.12.

Listing 6.5: In this game no player input is needed since the player moves so fast that he jumps over holes anyway.

```
{
"p_flat":"0.60",
"p_hole":"0.40",
"p_obstacle":"0.00",
"speed":"19.13",
"force":"13.39",
"gravity":"1.1",
"obst_height":2,
"block_length":2,
"seed":74131239
}
```

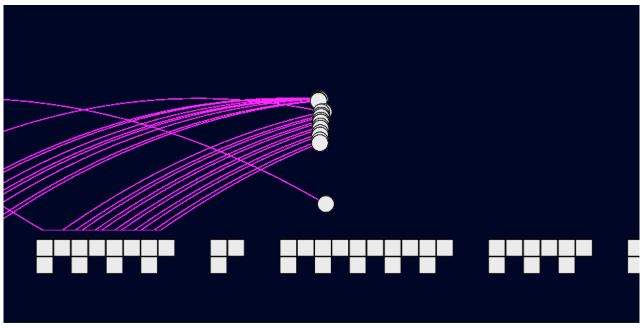


Figure 6.12: In this game the player would jump over holes without any action because l_{block} is small and v is high. The AI doesn't detect this and jumps anyway.

6.3.2 Changing Parameters Over Its Limits

When exceeding the defined limits for parameters, new interesting games can be created. However, the amount of unplayable levels also augment.

Games with $l_{block} = 1$ can be quite intriguing, for example Figure 6.13 (a) shows a game with $P_{hole} = 1$ and a low gravity, making it look as if the players are moving in a herd. A similar behaviour is observable for games with $P_{obstacle} = 1$, as seen in Figure 6.13 (b), where players continuously jump on the obstacles.

Listing 6.6: Game where players move in a herd like fashion.

```
{
"p_flat": 0,
"p_hole": 1,
"p_obstacle": 0,
"speed": 8.82,
"force": 1,
"gravity": 0.1,
"obst_height": 5,
"block_length": 1,
"seed": 46839795
}
```

Games with a negative gravity parameters are not possible to complete. The player could never jump over obstacles because he never touches the ground yet floats in the space.

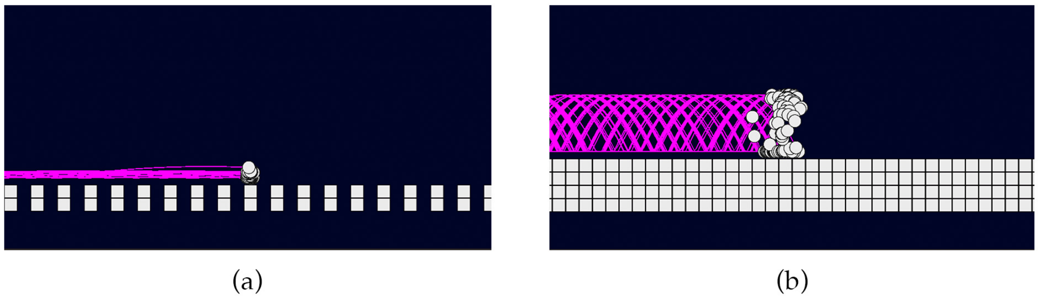


Figure 6.13: Two games with $l_{block} = 1$, looking as if the players are moving in a herd.

6.4 Comparison Of Games With Identical Difficulty d

Games with an identical difficulty do not necessarily need to have the same parameters e.g. l_{block} can differ between minimum and maximum possible value while keeping the same difficulty. This section discusses the differences and similarities of levels with $d = 0$, $d = 0.98$, $d = 0.5$ and $\sigma_d = 0.33$.

6.4.1 Games With Difficulty $d = 0$

For games with $d = 0$ the survival diagram, as shown in Figure 6.14, is always identical — a flat line, since all players win the game. Therefore, all of these games have $\sigma_d = 0$. Furthermore, most of these games have a high l_{block}.

6.4.2 Games With Difficulty $d = 0.98$

Games with $d = 0.98$, the highest achieved value of d, have an identical survival diagram, the line goes to 0 survivors immediately at the start since all players go game over already at the first obstacle, as seen in Figure 6.15. The block length l_{block} is 2 for all games with $d = 0.98$, because only with $l_{block} = 2$ and a high speed the players are able to reach the first obstacle fast, which results in such a low score. This is also backed by the fact that σ_d is 0 for every game.

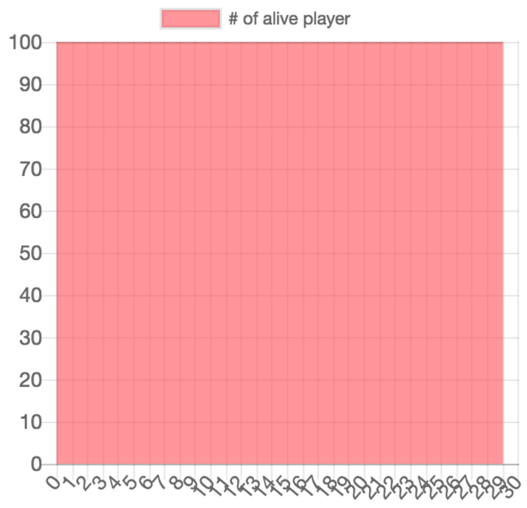


Figure 6.14: A survival diagram of a game with $d = 0$. All players won the game.

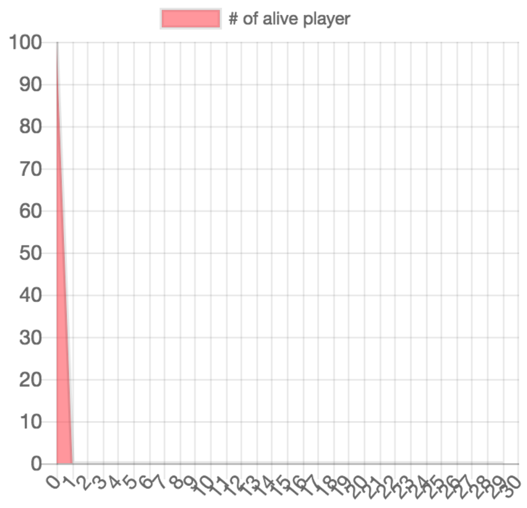


Figure 6.15: A survival diagram of a game with $d = 0.98$. All players reached a low score.

6.4.3 Games With Difficulty $d = 0.5$

Such a consistency as with $d = 0$ or $d = 0.98$ cannot be observed for games with $d = 0.5$. Games with this difficulty can be quite contrasting, parameters like l_{block} and v can range from their minimum to their maximum value.

Games With $\sigma_d = 0$

There exist games with $\sigma_d = 0$, for example the game described in Listing 6.7, where all players die in the middle of the game at one point, as the survival diagram in Figure 6.16 shows. This game is playable until half into the level, where after jumping over an obstacle all players directly fall into a hole and lose the level.
Another reason why games with $\sigma_d = 0$ and $d = 0.5$ exist is, due to the fact that at the middle of the game the players encounter an impossible block sequence. This could be an obstacle that is too high, a hole that is too wide or a combination of both, where no player has a chance to overcome them.

Listing 6.7: JSON of a game with $d = 0.5$ and $\sigma_d = 0$

```
{
"p_flat":"0.60",
"p_hole":"0.20",
"p_obstacle":"0.20",
"speed":"11.10",
"force":"16.72",
"gravity":"2.29",
"obst_height":5,
"block_length":11,
"seed":41062718
}
```

Games With $\sigma_d > 0$

Games with a $\sigma_d > 0$ can have a different survival diagram and consequently have a completely different gameplay feeling. The game described in Listing 6.8 has a relatively uniform distribution of scores as is indicated by the survival diagram shown in Figure 6.17. On the other hand the game described in Listing 6.9 has two points in the game where many players die, as can be seen in the survival diagram in Figure 6.18. About 50% of the players die at a score of 8 and the other half dies at a score of 23, yet the game still has a $\sigma_d = 0.26$.

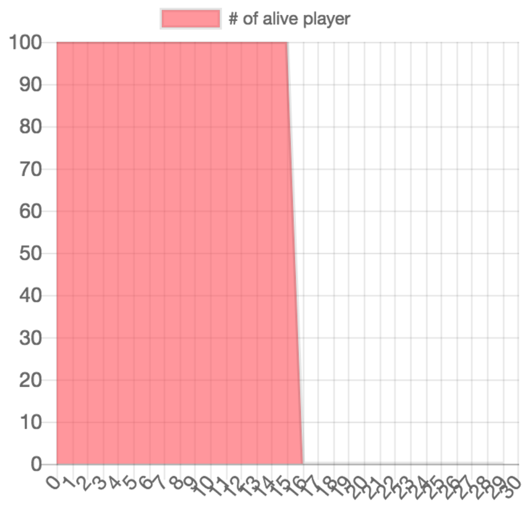


Figure 6.16: A survival diagram of a game with $\sigma_d = 0$ and $d = 0.5$. All players die in the middle of the game.

Listing 6.8: JSON of a game with $d = 0.5$ and a uniform player score deviation

```
{
"p_flat":"0.10",
"p_hole":"0.00",
"p_obstacle":"0.90",
"speed":"10.91",
"force":"14.57",
"gravity":"0.84",
"obst_height":1,
"block_length":9,
"seed":63320135
}
```

Listing 6.9: JSON of a game with $d = 0.5$ and two difficult points

```
{
"p_flat":"0.30",
"p_hole":"0.30",
"p_obstacle":"0.40",
"speed":"15.60",
```

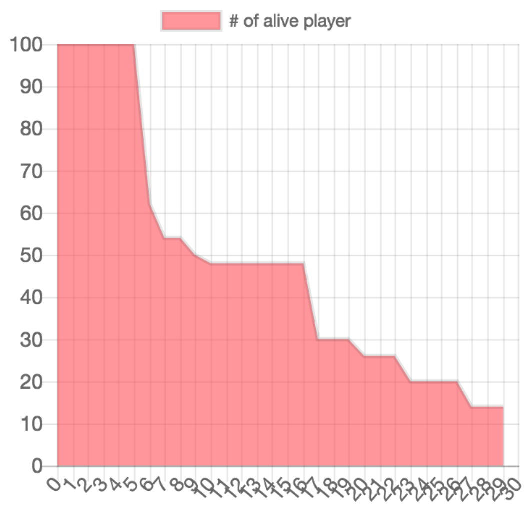


Figure 6.17: A survival diagram of a game with $\sigma_d > 0$. The scores are distributed relatively uniform.

```
"force":"16.10",
"gravity":"0.95",
"obst_height":4,
"block_length":21,
"seed":9109592
}
```

This shows that the survival diagram does not only enable game designers to tell the difficulty of a game, but also allows to get an overall feel on how the level behaves.

6.4.4 Games with difficulty $\sigma_d = 0.33$

The maximum σ_d of the games simulated for this work was $\sigma_d = 0.33$ and was achieved by two different games. While one game has $P_{hole} = 1$ and the other one $P_{hole} = 0.2$, they both have in common that most players jump just far enough to make it over holes, leading to some players dying at every hole. The game with $P_{hole} = 1$ has a `BlockType.Flat` block as every second block because of the rule that a `BlockType.Hole` cannot be immediately followed by another `BlockType.Hole` (Figure 6.19).

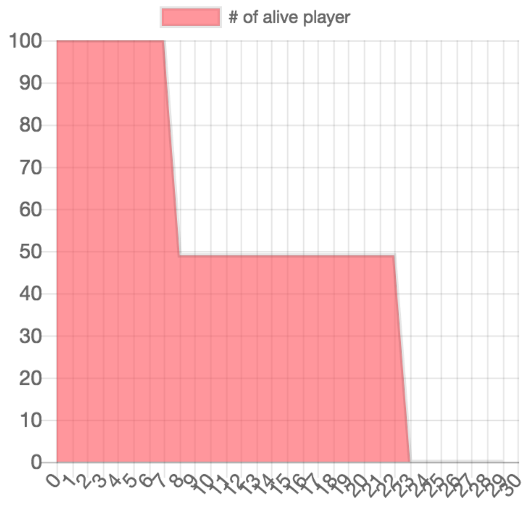


Figure 6.18: A survival diagram of a game with $\sigma_d > 0$. There are two main points where players die.

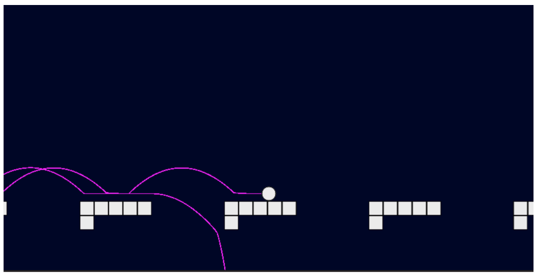


Figure 6.19: A game with $P(hole) = 1$ has an alternating block sequence, because two subsequent `BlockType.Hole` aren't allowed by the game.

7 Discussion

7.1 Simulation Speed Inaccuracies

In an optimal simulation the simulation speed does not have an influence on the calculated game difficulty. In the case of this work the simulation speed is changed by altering the Unity variable `Time.timeScale`. When the simulation speed is raised too high, the game starts to drop frames and inaccuracies start to occur.

To further analyze this observation 20 games with an average difficulty $\overline{d} = 0.5$ and a standard deviation of $s_d = 0.31$ and an average difficulty standard deviation $\overline{\sigma_d} = 0.15$ with a standard deviation $s_{\sigma_d} = 0.11$ were selected. Each game was simulated 15 times with different simulation speed values ranging from 0.5 to 8.0.

Figure 7.1 shows that with an increased simulation speed the average calculated difficulty d, visualized by the red line, starts to decrease. The area surrounding the red line depicts the average calculated difficulty standard deviation σ_d, which is as well decreasing on increased simulation speed. The average calculated difficulty $\overline{d}$ drops from $\overline{d} = 0.47$ at a simulation speed of 0.5 to $\overline{d} = 0.35$ at a simulation speed of 8.0. The average difficulty standard deviation $\overline{\sigma_d}$ drops from $\overline{\sigma_d} = 0.15$ at a simulation speed of 0.5 to $\overline{\sigma_d} = 0.06$ at a simulation speed of 8.0. This implies that simulations with a higher simulation speed are inaccurate and therefore all simulations in this work are done using a simulation speed of 1.

Figure 7.2 shows how much the simulation speed affects the difficulty d for each simulated game. The average standard deviation of the difficulty d in this test is `0.08`, demonstrating the inaccuracies caused by the simulation speed.

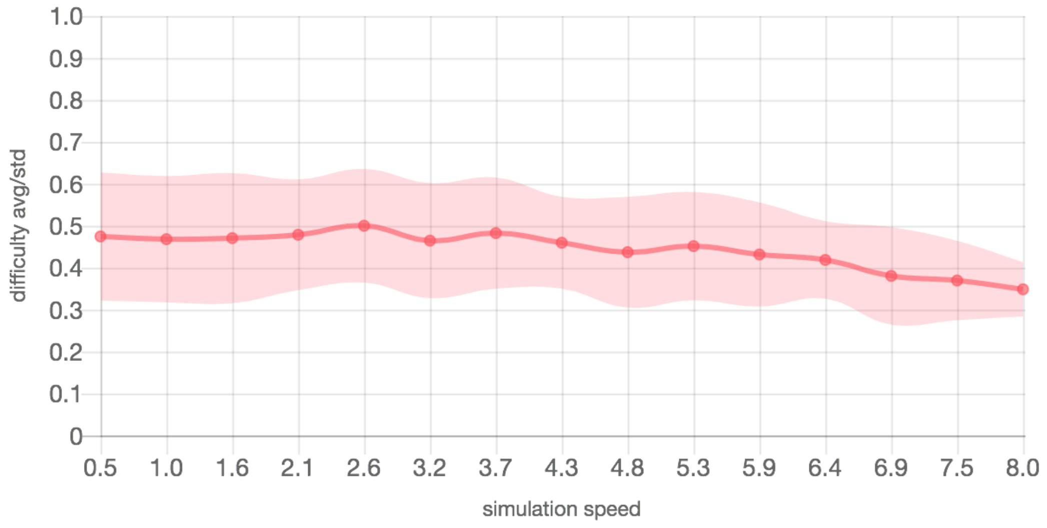


Figure 7.1: Game difficulty changes with varying simulation speed. The line shows the average difficulties d calculated and the area around the line indicates the average σ_d calculated.

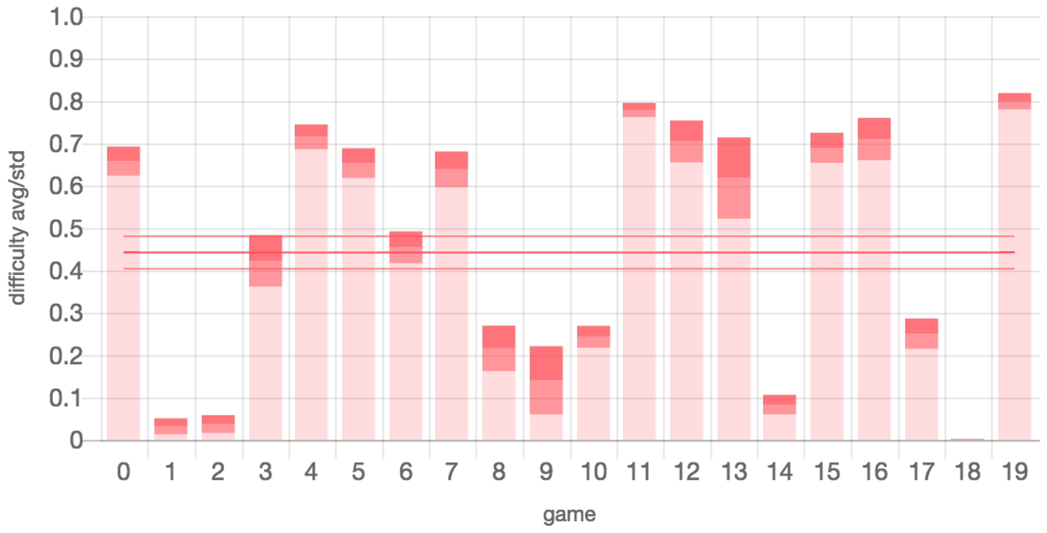


Figure 7.2: The deviation of d for each tested game when sampling the simulation speed. The lines show the average difficulty and difficulty deviation.

7.2 End Position Variation

The quest of finding the perfect end position in subsection 5.4.4 revealed an interesting behavior of the difficulty d regarding the variation of the end position. If the end position S_{max} is increased, the average difficulty $\overline{d}$ also increases, as seen Figure 7.3. This can be explained due to the property that in a game with an infinite end position, players would always achieve a similar score, therefore altering the end position does only scale the difficulty d. Meaning the larger S_{max}, the less player win the game and therefore d increases. Assuming that for a game with a fixed end position and $\sigma_d > 0$, at least one player did not reach the end position, the amount of players not reaching the end position approximately doubles if the end position is doubled. If the end position approaches infinity no player reaches the end position and therefore d approaches 1. Nevertheless, this does not imply that the average difficulty $\overline{d}$, as seen in Figure 7.3, would approach 1 if the end position approaches infinity because of games with $\sigma_d = 0$, which might retain $d = 0$ even if the end position approaches infinity.

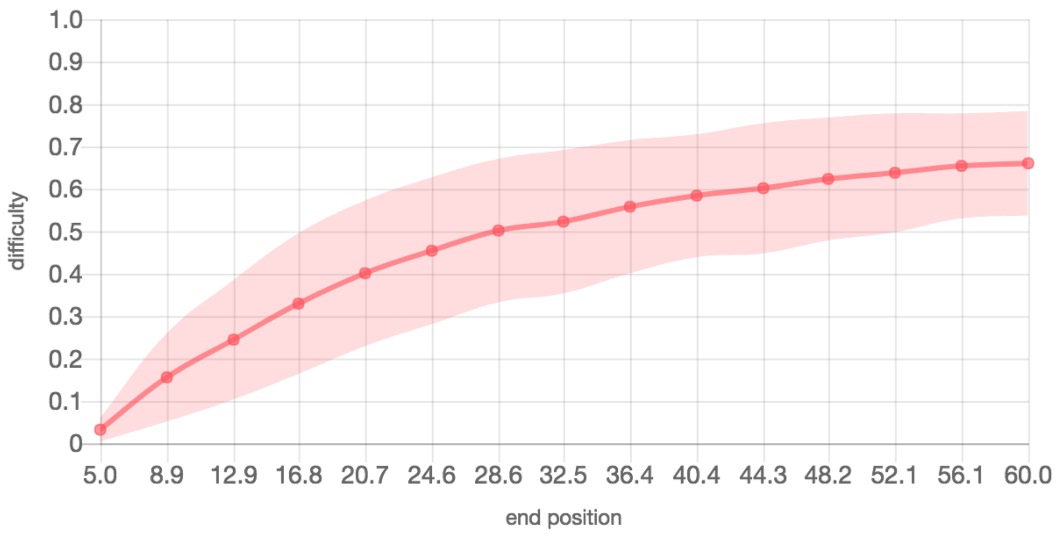


Figure 7.3: This graph shows how the average difficulty d behaves if the end position is varied. The area around the line represents σ_d.

8 Future Work

This work parameterizes the game Line Runner and analyses the randomly generated levels. It serves as an introduction to automated playtesting using an implementation of Line Runner as an example to illustrate the process.

8.1 Improved Difficulty Calculation

The definition of difficulty in this thesis consists of the two values d and σ_d. This limits the ability to sort the levels by its difficulty, as it is only possible to sort the list of levels either by d and disregarding σ_d or the other way around. The thesis "Exploring Automated Game Testing" approaches this issue by introducing a single difficulty value for each game.

8.2 Dynamic Difficulty Adjustment

This thesis analyses games with constant parameters. In order to achieve a game that becomes increasingly challenging over time, the parameter vector must also change. The sequence of used parameter vectors need to have a rising difficulty. "Exploring Automated Game Testing" modifies Line Runner to add dynamic difficulty adjustment (DDA) to it. It demonstrates two different approaches on how DDA can be applied to endless runner games (Larch, 2018).

1. By analyzing how each parameter influences the game's difficulty a rising difficulty adjustment can be achieved. For example the speed parameter can be increased over time and the game becomes more difficult to play.
2. By chaining different levels with different parameter vectors and different difficulties, difficulty adjustment can also be reached.

8.3 AI Improvements

Figure 8.1 illustrates the problem of having two subsequent obstacle blocks. The AI is able to observe the first obstacle block and calculates the perfect jumping position accordingly. When the players jump at this position they crash into the second obstacle block, leading to a game with $\sigma_d = 0$. Human players on the other hand are able to observe both obstacle blocks and guess the perfect jumping position to make it past both obstacles.
The AI should be able to do the same as human players and try to center the trajectory path at the center of both obstacle blocks. Since this does not happen frequently during the random generation of the levels, it has been neglected in this work.

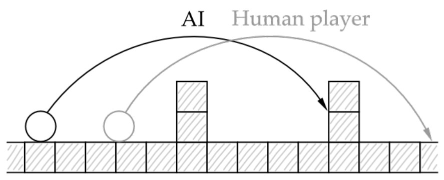


Figure 8.1: Human players can jump over two sequential obstacles, while the AI cannot.

8.4 Level Generation Improvements

The Level Generator developed in this thesis generated 89.3% games where $\sigma_d = 0$, which means that in all these levels all players either died at the same position or made it to S_{max} and won the game. Both cases are unwanted, since levels are either too easy or impossible. Due to this high percentage, the conclusion can be made that it is very inefficient to create playable levels when using random generation. The thesis "Exploring Automated Game Testing" explores the possibilities of enhancing the level generation efficiency in order to get more playable levels (Larch, 2018).

8.5 Difficulty Validation

The difficulty is calculated using the average of all scores of the players of a simulated level. These difficulties are only an estimation of the real difficulty of a level. Therefore, the difficulties calculated in this thesis should be further validated by making a comparison to the scores of human players. For this purpose Larch conducts a user study on said issue and validates the results of the calculations (Larch, 2018).

8.6 General Survival Analysis

This type of research can be applied to other endless runners, where the objective is to come as far as possible. The more complex the game becomes, the more complex the whole analysis as well as the AI becomes. Simple one tap games like Line Runner, Flappy Bird (Isaksen, Gopstein, and Nealen, 2015) etc. can be driven by a rather simple AI that calculates the next best game move to be carried out. This is done by imitating human behavior in the decision-making.
While it would be interesting to apply such an approach to a completely different and more complex games like chess, it would need a completely different approach to automated game testing. In chess the sequence of consecutive moves is the key criteria to winning the game. It needs a completely different approach in modeling the player interaction with the game.

8.7 Computational Creativity

Computational creativity targets to find novel and appropriate solutions using computational means (Duch, 2013). The analysis chapter in this thesis yields interesting game configurations of Line Runner. This questions whether it is possible to explore the game space to find interesting types of levels which share the same game mechanics, but appear like a different game. Isaksen et al. solved this task by searching the parameter space to find a subset of games that maximizes the minimum euclidean distance between all points in the 2D point cloud of all generated games.

9 Conclusion

This thesis presents and analyzes the automated difficulty estimation of a parameterized version of Line Runner by using Monte Carlo simulation and player score analysis.
It demonstrates how the level generation of a parameterized version of Line Runner is implemented and explains the mechanics of the developed AI, that is able to play these games. Using the results of 500 randomly generated and simulated games, the Level Generator's abilities to the difficulty prediction is analyzed.

Even though the games used for analysis were generated completely random, the influence of the parameters is still visible in the graphs. For example, the faster speed of the player in the generated level, the higher the probability that the game will be more difficult.
Despite the simplicity of the difficulty estimation the results indicate their validity. The capability of the Level Generator is analyzed using a difficulty probability analysis, showing that 10.7% of the generated games have a $\sigma_d > 0$. These games are subject to the analysis because they provide the best playability.

This paper further shows unique game configurations, found using the random level generation, such as the Moon Level. This level has a low gravity parameter and therefore gives a moon-like feeling to the game.

In addition, the similarities and differences of game parameters and the survival diagram of games with identical difficulties are discussed. While games with $d = 0.98$ have an identical l_{block} and survival diagram, games with $d = 0.5$ can have completely different parameters and survival diagrams. The survival diagram allows game designers to easily interpret a game's properties such as the overall difficulty and get an overview of the playability.

The AI in this work calculates the jump position depending on the next visible obstacle. It could be improved by observing multiple subsequent obstacles to calculate a better jump position.

Automated playtesting is a powerful method for game designers to ease the process of finding the optimal game experience of their game, while also finding new ideas by exploring the whole game space.

Appendix

Bibliography

Aponte, M.-V., Levieux, G., & Natkin, S. (2011). Measuring the level of difficulty in single player video games. *Entertainment Computing*, *2*(4), 205–213.

Duch, W. (2013). Computational creativity. *Encyclopedia of Systems Biology*, 464–468.

Holmgard, C., Green, M. C., Liapis, A., & Togelius, J. (2018). Automated playtesting with procedural personas with evolved heuristics. *IEEE Transactions on Games*.

Isaksen, A., Gopstein, D., Togelius, J., & Nealen, A. (2015). Discovering unique game variants. In *Computational creativity and games workshop at the 2015 international conference on computational creativity*.

Isaksen, A., Gopstein, D., Togelius, J., & Nealen, A. (2018). Exploring game space of minimal action games via parameter tuning and survival analysis. *IEEE Transactions on Games*, *10*(2).

Isaksen, A., Gopstein, D., & Nealen, A. (2015). Exploring game space using survival analysis. In *Fdg*.

Isaksen, A. & Nealen, A. (2015). Comparing player skill, game variants, and learning rates using survival analysis. In *Eleventh artificial intelligence and interactive digital entertainment conference*.

Karimovich, G. S., Turakulovich, K. Z., & Ubaydullayevna, H. I. (2017). Computer's source based (pseudo) random number generation. In *Information science and communications technologies (icisct), 2017 international conference on* (pp. 1–6). IEEE.

Larch, J. (2018). *Exploring automated game testing* (Bachelor's Thesis, Technical University of Graz).

Park, S. K. & Miller, K. W. (1988). Random number generators: good ones are hard to find. *Communications of the ACM*, *31*(10), 1192–1201.

Payne, W., Rabung, J. R., & Bogyo, T. (1969). Coding the lehmer pseudo-random number generator. *Communications of the ACM*, *12*(2), 85–86.

Raychaudhuri, S. (2008). Introduction to monte carlo simulation. In *Simulation conference, 2008. wsc 2008. winter* (pp. 91–100). IEEE.