Utrecht University

Master thesis

Generating non-monotone 2D platform levels and predicting difficulty

Erik Koens

supervised by
dr. ing. Jacco Bikker
prof. dr. Marc van Kreveld

December 14, 2015

Contents

Chapter 1

Introduction

1.1 Abstract

This thesis presents a method to procedurally generate levels for 2D side scrolling platformer games. The method yields non-linear gameplay and guarantees bi-directional paths. This method is based on a depth-first search technique to find an arrangement of building pieces, which represent discretized jumps, and that satisfies a set of constraints. We also present a method to estimate the difficulty of the generated levels. A statistical model is constructed for individual challenges, based on three general parameters of the challenges. This model is used to find the path that yields the highest probability of successfully traversing the level. We verify our results by investigating the relationship between the computed difficulty values and the number of deaths measured.

1.2 Problem setting

Many old-skool platformer games yield gameplay where players move horizontally through the levels. On the one hand this is a deliberate game design decision, but on the other hand, games were forced to this by hardware limitations: there were simply not enough computation and storage capabilities to create large open worlds. In the early 90's the first 3D games emerged, such as Doom and Unreal Tournament. Levels of these games were generally small and restricted to a bounded area. A screenshot of Unreal Tournament can be seen in figure 1.1b.

Modern consumer hardware is considerably more capable in terms of compute power and memory capacity compared to the machines in the late 80's and begin 90's. But fast machines are not a solution to everything, these days developers face other problems. The increasing computational power comes with an increasing demand for details and high quality graphics. Although the machines can store wide open worlds, it requires a huge amount of manual labour to create those worlds filled with the tiniest details. GTA V and Borderlands are examples of present-day games that yield large open 3D worlds, created by hundreds of employees and project budgets of millions of dollars. A screenshot of GTA V can be seen in figure 1.1c. Most of the large developers, contracted by publishers, focus these days on 3D games, while most 2D games are created by hobbyists and small independent developers, with a small budget and only a handful of team members. Although the demand for details in 2D games is lower than for 3D games, it is still an extensive task to create large worlds.

Automatically generating content can provide a solution for the limited amount of human resources. Generating levels by a computer program is not new: it was already applied in some of the earliest games in order to reduce the amount of space needed on a storage disk. This idea can by used to create large worlds, and can even be used in modern 3D games to meet the high demands for detail.

This thesis focuses on generating levels for 2D side scrolling games. 2D side scrollers are games where the player runs over platforms and other structures and must jump over gaps and other obstacles in order to survive, while he is pulled down by gravity. An example of such a *platformer* is Super Mario (see screenshot 1.1a). Side scrollers are different from 2D top down games, where the player sees the world from above. In this type of game, the player is not constrained by gravity, and so generating levels requires a different approach.

(a) Super Mario (b) Unreal Tournament (c) GTA V

Figure 1.1

In a side scrolling world the player is expected to move horizontally and vertically through the world. The world must exhibit a non-linear structure, such that the player has the possibility to

chose its own path according to his strategy. The player must also have the possibility to revisit areas where he has already been, so a path between two areas is ought to be wandered in both directions.

Generating a level structure is only half the story. The world is also expected to be challenging, so the world must be populated with traps. The distribution must be done such that each trap forms a fair challenge. Player skill generally increases when playing the game, so the challenges at the start must be less difficult then those further in the world.

1.3 Research Goals

The goal of this thesis is twofold. First, automatically generating levels. Secondly, estimating the difficulty of the generated levels.

The levels must yield horizontal and vertical gameplay, the generated paths in the levels must be traversable in both directions, the level should contain multiple paths from a start to an end and it must be populated with challenges. A well designed game yields a steadily increasing difficulty curve such that the player is challenged during the entire game. Players get frustrated when levels are too difficult and get bored when levels are too easy. Therefore, a level generator should besides generating playable levels also generate levels of a certain difficulty level that meets the players skill level.

This thesis proposes a level generator that meets the following goals:

- Generating levels for 2D side scrolling platform games that yield:
  - non-monotone gameplay;
  - multiple feasible and bi-directional paths;
  - challenges.
- Computing the difficulty of the generated levels by:
  - computing the difficulty of individual challenges;
  - computing the easiest path through the level.

The level generator creates a level structure populated with traps for 2D side scrolling platform games. The player can move horizontally and vertically through the level while he is pulled down by gravity. The goal of generating levels can be considered as achieved when the level generator meets the sub-goals, and when the generator is capable of doing so within a reasonable amount of time such that the generator can be used in practice. A loading screen for a level may take at most 30 seconds, so a level must be generated within that amount of time.

This thesis considers only local dynamic traps. Here, *local* means that the traps can do only harm in a fixed static area, so the traps can't move through the entire level. By *dynamic* is ment that the traps can only periodically hit the player, such that the player needs to time his actions to overcome the trap. A jump over a gap filled with spikes is not considered a dynamic trap, since it does not rely on timing. The player can perform his action at any moment to overcome the gap. A saw moving back and forth within a fixed area is considered a dynamic trap.

The difficulty of a level is quantified as *the probability of successfully completing the level* [16]. It is not the aim to give a metric for player performances, but to give an ordering of the levels with respect to difficulty. The goal of computing difficulty for levels can be considered as achieved when increasing difficulty values correspond to increasing numbers of lost lives, while taking differences in skill level into account.

1.4 Reading guide

Chapter two gives a literature overview. We discuss previous work in the field of procedurally content generation and difficulty measuring in games.

In chapter three, we describe out method to generate levels and to populate these level with traps according to the research goals. It describes a method that uses discretized jumps as building blocks, and it tries to find an arrangement of these building blocks such that a set of constraints is satisfied. After that a graph is built from the level structure, representing the movement possibilities of the player agent in the level. Traps are placed based on a pattern matching process, without violating a set of constraints.

In chapter four, we give a method to estimate the difficulty of the generated levels. A user study is used to gather data about human performances. From this data a model is built that estimates the probability of successfully overcoming a challenge, given some parameters of the challenge. This model is used to built a graph which represents the probability of successfully moving through the level. The difficulty of an entire level is computed as the easiest path through the level.

In chapter five, we give the results of the user study and show a relationship between the estimated difficulty value and the measured numbers of death. In chapter six, we give a discussion and future work and it ends with the conclusion in chapter seven.

Chapter 2

Previous Work

This chapter provides an overview of existing work on the subject of automatically generating levels and measuring difficulty. We start with an overview of desirable properties of generated content, and then we give an overview of different methods. After that we provide an overview of methods to measure difficulty. Some methods focus only on the perceived difficulty of the player, while other methods are closer to the research goal of measuring difficulty of levels. We end with a discussion on which methods can be useful for achieving the research goals.

2.1 Procedural Content Generation.

The field that is involved in generating levels for games is Procedural Content Generation (PCG). Togelius et al. [21] define PCG as *the algorithmical creation of game content with limited or indirect user input*. Here, *content* can refer to a wide range of elements such as levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters etc. *Limited or indirect user input* implies that the user can have control over the result, but that most of the work is performed by the automated process. Although games are explicitly mentioned in this definition, content could also be generated for other fields.

In the early days of computer science PCG techniques were used in games to deal with limited storage space. Rogue is one of the classic games that uses PCG. It is a dungeon-crawling game where the player has to explore a network of rooms where it can find monsters and treasures. A screenshot of Rogue can be seen in figure 2.1a. Each time the game is started an entirely new dungeon is created. Many other games are inspired by Rogue and it has become a class of games itself: Rogue-likes. An example of a modern Rogue-like is Diablo. A screenshot of Diablo 3 can be seen in figure 2.1b.

A recent game that uses PCG is No man's sky, where an entire universe is procedurally generated. The planets are generated based on physical properties like gravity and the distance to the nearest star. Even all life inhabiting the planets is procedurally generated. A screenshot of No man's sky can be seen in figure 2.1c.

(a) Rogue (b) Diablo 3 (c) No man's sky

Figure 2.1

Many different methods exist to procedurally generate content. But regardless of the method used, the content must meet some desired properties. Shaker et al. point out some desired properties in their book on procedural content generation[25]:

- Speed: the time available depends on the application. There is not much time to spend when content needs to be generated during the game or when the game is started, while there is plenty of time to consume when the content is generated offline.
- Reliability: when the generated content plays a keyrole in the gameplay it has to meet certain quality goals. A dungeon must for example have an entrance and an exit. The quality goals for content that plays only a decorative role are less strict, a badly generated piece of decoration has no significant influence on the gameplay.
- Controllability: many applications require some control over aspects of the content, like the difficulty of a level or the style of decorative objects.
- Diversity: the extent to which a method is capable of generating new content. For some applications it is not desirable that a method generates each time roughly the same content with only minor changes, while an instance of a class of plants must always look like a plant of that class.
- Believability: many applications want to induce the feeling that the content was created by a human instead of a computer. This feeling is broken when too many repetitions occur, or when a physically impossible structure is created.

A number of generators have been developed for the game Super Mario Bros, a 2D platformer game in the xy-plane where the player agent has to make jumps to overcome obstacles and enemies. There are also AI agents developed which can autonomously play a level which is helpful to determine whether a level is playable. An advantage of developing a generator for the same game is that the results can be compared, but the disadvantage is that Super Mario Bros does not represent all platform games. The game is very linear and almost fully x-monotone and yields no possibility for the player to explore the game world.

2.1.1 Representation

Some methods use an abstract representation of the level structure for controlling the global structure. For example, a graph representation of the rooms in the level is a broadly used representation.

Some methods do not use an abstract representation, but operate directly on the level space. Examples are a method based on a Markov chain by Snodgrass and Ontanon [23] and a method based on an evolutionary algorithm by Mourato et al. [11]. These methods use a search technique which directly operates in level space, as will be explained in the next section. The method that is used to generate levels for Spelunky [28] uses a maze generator to generate a structure of connected rooms in a grid. The rooms are rectangular and horizontally and vertically aligned with adjacent rooms. The same method is used by Benoit-Koch [2]. An example can be found in figure 2.2.

Dormans [10] and Lefebvre et al. [4] use a graph representation where also each node corresponds to a room, but the rooms are not required to be rectangular. Dormans [10] generates a graph based on gameplay constraints, like the distribution of loot and enemies throughout the level, while Lefebvre et al. [4] take a predefined graph, designed by a game designer, as input.

Smith et al. [27] use a rhythm of jumps to create levels for a 2D side scrolling game. The global structure is made by a rhythm of some basic actions like move and jump for x amount of time. This provides the structure and the context of the level and creates a sense of flow. A level consisting of randomly placed jumps yields most likely an irregular pattern, which yields a lower sense of flow. This is illustrated in figure 2.3.

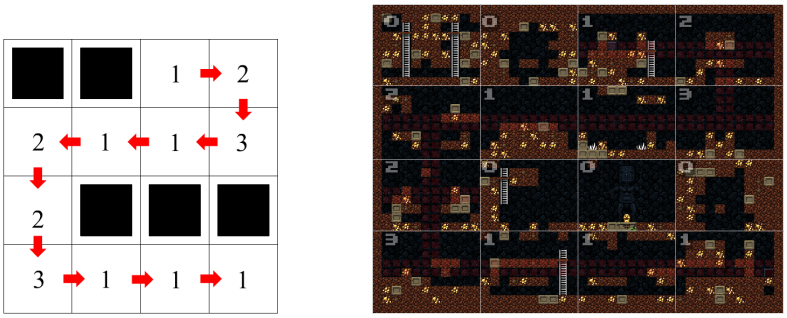
2.1.2 Construction

On a high level two types of methods can be distinguished to generate content. Methods that use a direct approach where the result is directly deduced from a set of input parameters, and methods that use a search process where the result is generated according to a set of objectives and constraints. Besides that, some methods make use of data from existing levels authored by human designers. This data can be gathered through some machine learning technique. These methods are referred as *data-driven*. On the opposite, methods that make use of heuristics and other theories are called *theory-driven*.

Direct approaches

An example of a direct approach are fractal-like algorithms. These take a set of input parameters and produce an output by recursively applying the same steps on different scale levels. The *diamond-square algorithm* is a good example of a fractal-like algorithm that is used to generate mountain environments. The terrain is divided into a patch, and each patch itself is recursively subdivided into more patches. During one step of the diamond-square algorithm, the points of the patch are raised and lowered by some stochastic process with respect to some maximum amplitude. After this, the amplitude is lowered and the step is recursively applied to the sub-patches.

Another direct approach is to use predefined content. The level generator of Spelunky [28] starts by creating a maze of rooms. A type is assigned to each room based on its connections with adjacent rooms. Based on the room type, a predefined template is randomly chosen from a set of templates to fill up the room. This is illustrated in figure 2.2.



(a) (b)

Figure 2.2: Figure 2.2a displays the room layout used by the Spelunky level generator. Figure 2.2b displays how the rooms are filled with predefined templates.

Kerssemakers et al. [24] use agents to generate levels for Super Mario Bros. Each agent is specified by a set of attributes and behaviours that can be parameterised. The level generator consists of 14 to 24 agents. Each agent behaves autonomously and can add and remove elements while wandering through the level. The behaviour of the agents is non-deterministic so the output can vary each iteration even with the same parameters. This idea of agents is very similar to idea of cellular automata, where the value of each cell in a grid is determined based on a set of rules and on the neighbourhood. The results are usually very organic and are useful for natural landscapes like caves and coast lines.

Grammars are also used to generate content. The rhythm based method [27] by Smith et al. uses a rhythm of basic actions to generate a global structure. The resulting level geometry is deduced from this sequence of atomic ations according to a grammar. The same atomic action can be interpreted differently based on the context. For example a jump that lasts for one second can be interpreted as a jump where the end point is at the same level as the starting point, but the end point could also lie above or beneath it. The same applies for a run action, where the player agent can run along a straight line but also over an increasing or decreasing slope. Which geometric building block is chosen for an atomic action depends on the grammar. An example is given in figure 2.3.

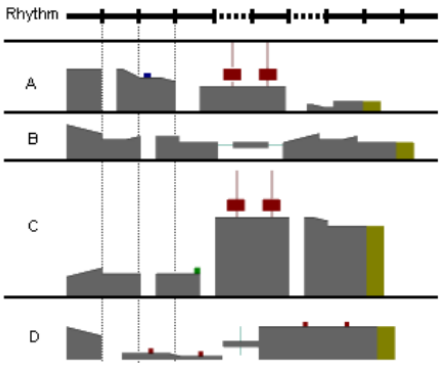


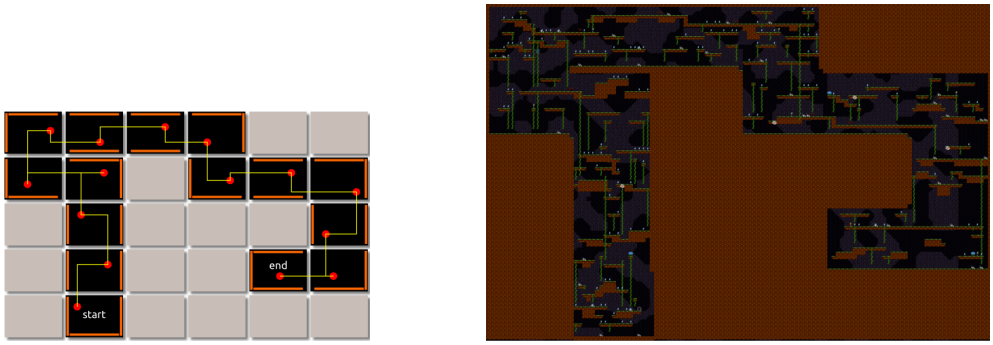
Figure 2.3: Example of how different geometries can be deduced from the same rhythm of basic actions.

Snodgrass and Ontanon [23] describe a method that generates levels for Super Mario Bros and is an example of a data-driven method. It uses manually authored levels to train a higher order Markov chain ordered in a Bayesian network structure. In a higher order Markov chain the current state depends on multiple previous states, this means that the state of a cell in a Bayesian network depends on the states of its neighborhood. To generate levels, the method loops over all cells in the level and assigns a type to it, which is statistically the most probable according to the learned Markov chain.

Search based approaches

Contrary to the direct methods where the results can be directly deduced from the input, the search based methods have to search through the solution space for a solution that meets the user defined objectives and constraints. But the edges between direct and search based methods are sometimes blurred as a direct method can be used to generate solutions during the search process. Also distinctions between the search based approaches can be made: some approaches maintain a feasible solution and expand the current solution throughout the search in a feasible way. Other methods generate candidate solutions, and evaluate whether they meet the objectives and constraints.

A *depth-first search* is an approach that maintains a feasible solution throughout the process. As soon as the solution becomes unfeasible, a series of backtracking steps is performed until the solution is again feasible. This approach is used by Benoit-Koch [2] to generate 2D platform levels. The method starts with generating a maze of rooms, according to the method of Spelunky. After this a depth first technique is used to generate a path of horizontal and vertical line segments, that connects the starting room with the finish room. Regular platforms are placed at the horizontal line segments and ladders are placed at the vertical line segments. This is displayed in 2.4.



(a) (b)

Figure 2.4: Figure 2.4a displays the room layout and the connected line segments used in [2], and figure 2.4b displays the result.

Fisher [12] uses also a depth-first technique to generate levels for a 2D side scrolling game. This method is capable of taking moving obstacles into account. In a static scene it is quite easy to compute which points can be reached from a given point. But the problem gets much harder when the landing spot is moving, the player agent can collide with the platform during his jump or the platform is no longer at the right spot at the time the player agent arrives. During each step of the search process, a new building block is added to a feasible level and a player AI checks whether the level is still feasible. The building block is removed when it interferes with the feasibility and another branch of the search tree is tried.

Evolutionary algorithms are another kind of search based approaches. The basis of an evolutionary algorithm is formed by an evaluating function and by a mutation operator. A set of sub-solutions is maintained during the search process. Each sub-solution is evaluated by the evaluating function which is based on the provided objectives and constraints. Some of the worst scoring sub-solutions are discarded, and new solutions are generated by mutating some of the remaining sub-solutions. This process is repeated until a solution meets the objectives and constraints, or when a maximum number of iterations is reached. The methods described by Mourato et al. [11] and Dahlskog and Togelius [8] use an evolutionary algorithm to generate respectively levels for Prince of Persia and Super Mario Bros.

The method [11] that generates Prince of Persia levels uses an evaluation function that is based on the presence of a valid path, on structural properties and the distribution of certain cell types. The mutation operator exchanges parts of the level between two solutions. An example of a generated Prince of Persia level can be seen in figure 2.5.

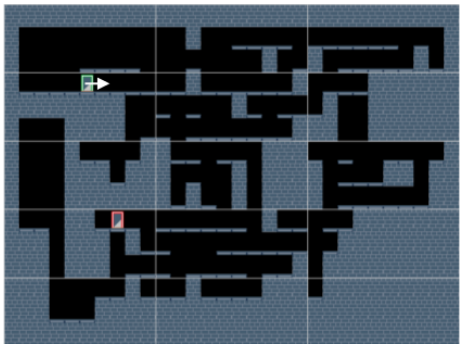


Figure 2.5: Example of a generated Prince of Persia level.

Dahlskog and Togelius [8] describe a multi-level method that is data-driven. The generation takes place at three different levels: micro, meso and macro. Vertical slices of the Super Mario Bros levels form the micro-patterns. These are the elementary building blocks. Meso-patterns consist of multiple micro patters. Each meso-pattern can be constructed in different ways, a valley can consist of 5 or 10 slices but it is still a valley. These meso-patterns are manually identified in their previous paper [26]. Macro-patterns are composed of meso-patters and are used to control global properties like the distribution of resources. The macro-patterns were obtained by analysing existing levels for the presence of meso-patterns. An evolutionary algorithm is used to generate levels. The mutation operator modifies a sequence of slices and exchanges a slice between two candidate solutions. The evaluating function measures the presence and order of patterns.

2.2 Measuring difficulty

It is hard to give a precise definition of difficulty with respect to games as it covers a wide range of psychological aspects like emotional and intellectual capabilities and it involves the complex dynamics of games. A very broad definition can be found in [16] by Levieux et al., where difficulty is defined as *the extent to which the player is capable of overcoming a given challenge regarding the players skills or abilities.* This implies that a given challenge is not always equally hard to overcome, but that the probability of overcoming a challenge also depends on the players capabilities. So to obtain the same level of difficulty throughout the entire game, the difficulty level of the challenges must steadily increase with the players skill level, as the player improves his skills during the game. Game designers face the challenge of creating a design where the difficulty meets the players skills during the entire game. Players loose interest when the game is perceived as too easy or too hard at some point. This concept is proposed by Mihaly Csikszentmihalyis Theory of Flow [7].

The need for measuring difficulty differs per application. An accurate difficulty measure can assist level designers, as they have to rely on their subjective intuition while designing a difficulty curve. Some applications require a system that is capable of intervening in the game in real time to match the players skill level with the difficulty level of the game. Such systems are described with the term *Dynamic Difficulty Adjustment* (DDA). Examples can be found in racing games where the last player is provided a temporarily speed boost to catch up with the other players, or in shooters where extra ammunition is supplied to badly performing players. Measuring difficulty is also very useful for games where the levels are procedurally generated, as one would expect easier levels at the beginning of the game and that the difficulty steadily increases as the game progresses. In all cases a metric is necessary. For the DDA systems it is usually required to measure the performance of the player during a play session, and it involves therefore mainly local parameters like ranking, health and the availability of resources. In the case of procedurally generated levels more global parameters are involved, like the levels geometry, distribution of enemies and possible paths. Some general features that can be measured to indicate the difficulty level are:

- death, hit and nearly hit events of the player;
- number of kills made by the player;
- the possession of resources like health, points or ammunition;
- level geometry, like the length of the path to the exit and the number and type of obstacles;
- distribution and strength of enemies throughout the level;
- The players opinion.

2.2.1 Measuring local difficulty

Examples of systems that measure local difficulty are the Hamlet system [20] proposed by Chapman and Hunicke. Another example is the AI Director system [3] used in the Left for Dead game series. The Hamlet system operates in the Half Life 2 game, which is a *first person shooter* like the Left for Dead games. Screenshots of Half Life 2 and Left for Dead 2 can respectively be seen in figure 2.6a and 2.6b. Both systems try to determine when the play session is too hard or too easy, and intervene in the game in real time.

They both use the players health as an indication of the skill level. The AI Director simply intervenes in the game when the players health drops below a threshold value, while the Hamlet system uses a bit more sophisticated method using a Gaussian normal distribution of the damage done over time. The AI Director has the option to modify the level geometry: by opening or closing passages it modifies the length of the path that needs to be taken by the player, and thereby adjusting the expected difficulty. The AI Director also intervenes with the spawning of enemies and resources in order to match the players skill level. It has the option to spawn a large horde of enemies or special, more lethal, enemies.

(a) Half Life 2

(b) Left for Dead 2

Figure 2.6

The Hamlet system does not intervene in the levels geometry or in the spawning of enemies. Instead, it focuses on modifying parameters that influence the difficulty. When the player is struggling, the Hamlet system can intervene by increasing the impact of the players bullets on the enemies or decreasing the accuracy of the enemies for example. Both systems regulate the availability of ammunition and other resources. The AI Director is designed such that the adaptations are noticed by the player, while the modifications of the Hamlet systems need to stay unnoticed.

2.2.2 Measuring global difficulty

The goal of measuring global difficulty is to determine the difficulty of an entire level, based on game parameters instead of the temporal performance of the player. Determining the global difficulty is usually a hard task due to the dynamic nature of games. It is hard to predict what the influence of a single parameter is on the entire level. One could add a very dangerous pack of enemies sheltered behind some spiky construction, but this has little impact when the player has the opportunity to easily flank these enemies. Determining the global difficulty can be done in multiple ways:

Measuring difficulty using predefined difficulty values

The easiest way to measure the difficulty of an entire level is to assume that the difficulty of each individual component of the level is already known. The difficulty of the entire level is then simply the sum of the difficulties of each component the player traverses. This idea is used by Horswill and Foged [15] who represent the level as a graph and propagate resources through the level subjected to a set of playability constraints. Each node represents a room of a given difficulty.

Bakkes et al. [18] describe a method for generating levels for which the difficulty curves match with user provided difficulty curves. They use an evolutionary algorithm to produce levels according to a given grammar where each terminal is provided with a predefined difficulty value.

Measuring difficulty by mathematical expressions

To fully capture the difficulty of the game dynamics by a mathematical expression is a virtually impossible task. Therefore, existing methods focus only on a small part of the game dynamics and make assumptions about the players behaviour.

Berseth et al. [13] describe a method to measure difficulty based on encounters with enemies along the route of the player, and takes a set of obstacles into account. It starts by computing the routes for both the player and the enemies. These routes can be the shortest path between two points, or some (repeating) patrol path for the enemies. Assumed is that the agents move with a constant velocity along their paths, and that encounters between the player and the enemies only occur when they are close enough to each other. The difficulty at a single point takes into account whether the player is seen by an enemy at some point in time, the distance between them and the skill properties of the enemy. The difficulty of the entire path of the player is measured by taking a line integral along the path with respect to the expected time interval. The paths are highly influenced by the configuration of obstacles. A slight modification can lead to the case where the players path intersects with a lot of enemy paths, which increases the difficulty. They provide a method which creates a configuration of obstacles which leads to a scene of a given difficulty. This method uses an evolutionary strategy for determining the configuration, and the obstacles are only placed within a user defined area to put a boundary on the computation times.

Mourato and dos Santos [19] propose a method to measure difficulty of platform games, based on successfully executing jumps. They distinguish a few scenarios which frequently occur in platform games, and use a graph representation of these scenario's to analyse the difficulty. The edges of these graphs represent atomic actions like moving, climbing along a ladder and jumping over gaps. Moving and climbing are considered as trivial actions which can not fail, so the difficulty is only based on successfully executing jumps. They represent the players jump trajectory by a parabola, and the probability of successfully executing jumps is based on the horizontal and vertical error of the parabola with respect to a given end point. Decreasing errors correspond to decreasing probabilities of successfully executing jumps, and they make the assumption that the probability decreases exponentially. The differences between skill levels are captured by different constants used in the exponential expression.

Baghdadi et al. [22] propose a straightforward method to measure difficulty which is only based on the number of occurrence of traps. They use an evolutionary method to create levels, and use an evaluation function where the difficulty is measured as a weighted sum of the occurrences of different trap types.

Measuring difficulty through experimenting the game

Another approach is to analyse player sessions and measure the influence of game parameters on the difficulty. The advantage of such methods is that no assumptions have to be made about the players performance. On the other hand, the results are usually quite specific to the experimented application and are not easily generalized. In some cases a synthetic player is used, instead of human players. The advantage of using a synthetic player is that he is fast and tireless, thousands of play sessions can be performed with one press on a key. However, since the synthetic player is a simulation of the real player, assumptions about the real player are required and the results of the experiment depend on how well the synthetic player acts like a real player.

Aponte et al. [16] use a synthetic player to play a variant of pacman, but this was not an attempt to accurately measure the difficulty curve of the game but mainly for the sake of example.

Hom and Mark [14] use two synthetic players to play against each other to balance a board game. The game is balanced when both players play an equally hard game, which is based on the number of wins.

The majority of the papers use human players for their experiments, and use some kind of machine learning to analyse the data.

Pedersen et al. [5] investigate the relations between player experiences, play styles and game parameters through experimenting randomly generated levels of Super Mario Bros. They use very generic features like the number of jumps necessary, the time spent on certain gameplay elements, items collected and the number and types of death events. A short survey is used after each two levels to collect data about the players experiences. The collected data is used to train and test a single layered neural network.

Jennings-Teats et al. [17] use player testing to rank pairs of challenges by difficulty. They made a simple side runner game, where the player agent runs horizontally through the screen while jumping over gaps and other obstacles. Each pair of challenges consists of a jump over a gap to the next segment, and possibly some kind of threat in the next segment. The jumps vary in height and length. The player is asked to rate the difficulty after each played segment. The collected data is used to train and test a Multiplayer Perceptron algorithm, which is a multi-layered neural network. As a result they give a ranking of combinations of components by difficulty, based on the players feedback.

R. Vasconcelos de Medeiros and T. Vasconcelos de Medeiros [9] use a machine learning technique to maximize the fun of their game, and to find relations between game play elements and difficulty. Six features are extracted, which are based on specific gameplay elements rather than general game mechanics. The player is asked to rate the difficulty after each level. Cooper [6] also uses machine learning to dynamically adapt difficulty, but some more general features are used like the formation direction of enemies, the number of enemies and which type of path they use. The difficulty is measured by counting death and close calls events.

2.2.3 Abstract framework for measuring difficulty

Aponte et al. [16] provide an abstract frame work for measuring difficulty. During a gameplay session the player faces challenges, which he successfully overcomes or fails to overcome. So the result for a given challenge is either WIN or LOOSE. Before the result is determined, the challenge can be either in the state NOT STARTED or IN PROGRESS.

The difficulty of a challenge is controlled by the skill level of the player and the composition or type of the challenge. Players learn new skills and strategies during the game, making it easier to overcome known challenges. To keep up with the players skill level, new types of challenges can be introduced or previously used challenges can be combined into more difficult challenges. For example, the challenge of performing an action gets more complicated when the action is combined with a strict timing event, like throwing a ball in the basket when a lamp is glowing. In a formal way the challenges are constructed of:

- atomic challenges, whose difficulty can be controlled by a set of parameters;
- partially ordered set of composite challenges, which can be solved by solving the lower level challenges.

During the game the player faces challenges $c \in C$, and learns abilities $a \in A$ and each ability is mastered at a level $l \in L : L = \{bad, average, good\}$. For simplicity a discrete scale is chosen, but

a continuous scale is of course also possible. The outcome of each challenge depends on the skills of the player to overcome that challenge, that is the player level $plevel(c, a) : A \times C \rightarrow L$.

The probabilistic difficulty function for overcoming a challenge c requiring ability a is given by:

$$\delta(c, a) = p\{LOSE(c) \mid plevel(c, a)\}$$

The player level $plevel(c, a)$ is computed based on the players history for the given challenge and the required ability. From the previous encounters the ratio between success and the number of tries is calculated, this value is mapped to a level $l \in L$ to represent the player level.

2.3 Discussion

The proposed level generator of this thesis must be capable of generating levels that yield non-monotone gameplay, multiple bi-directional paths and challenges for 2D side scrolling games. Most 2D side scrolling games are x-monotone, meaning that the player runs horizontally through the levels without going back or having the possibility to explore higher grounds. This gameplay concept turns out to be very successful, so there is no need to support non-monotone gameplay. Therefore the level generators of x-monotone levels do also not focus on generating non-monotone levels. As a result most generators are not applicable to generate such levels. Generating monotone levels is in its simplest form very straightforward: place a level segment, and place another segment next to it with possibly a gap between them. Constructing a non-monotone level is less straightforward, it requires a structure that supports a sequence of multiple jumps to reach higher points. It requires a solution to a global problem, instead of the local problem of generating x-monotone levels.

None of the previous work provides a direct solution for generating non-monotone side scrolling platform levels. The generator for Prince of Persia [11] generates non-monotone levels, but the player agent is allowed to move directly to the cell above him which commits the need for a structure of multiple jumps. The method by Fisher [12] uses a player AI to check whether a part of the level is feasible. That could be used to check whether a candidate solution yields a feasible path. The downside is that this requires considerable computation power.

The Markov chain method [23] and the multi-level method [8] are data-driven and use existing levels to generate new levels. This approach has two problems: there is no dataset of non-monotone levels available and creating one is very time consuming. The second problem is that the generated levels must satisfy the global constraints that come with non-monotone levels. The Markov chain method uses a Markov chain to generate levels, which in theory has the potential to learn from any kind of level structure, also from non-monotone levels. It is however unlikely that it will be able to generate non-monotone levels as it is barely capable of generating x-monotone levels. A substantial portion of the levels are not playable by an AI player. Besides that, different configurations are needed to capture the difference between the lower and higher parts of the level. Also the order of traversing through the level seems to have a large influence on the result. So a lot of human insight is necessary to create something playable. Therefore it is unlikely that this method will be capable of solving the global constraints that come with non-monotone levels. The multi-level method uses building pieces at three different levels: micro, meso and macro. In fact, these building pieces form compounds of other pieces, and the level generator puts the compounds in a sequence according to observed patterns from existing levels. It is a good method to generate x-monotone level but will not be able to generate non-monotone levels and satisfying the global constraints.

The method used in Spelunky is in theory capable of generating non-monotone levels if the templates are designed such that they contain a feasible bi-directional path between each pair of exits of the room. The Spelunky method uses fixed exit points between adjacent rooms, in order to align the room connections. As a consequence, the method suffers from a clearly visible global structure and has limited variety. Solving these issues will include designing a large set of templates rooms for all possible exit configurations.

The method that uses ladders [2] could work but the results are aesthetically unpleasing. The use of only horizontal platforms and the overuse of ladders creates a monotonous and repetitive look and feel. This can be seen in figure 2.4. The rhythm based method [27] itself is not capable of generating non-monotone levels, but the idea of starting with jumps and building the geometry afterwards is interesting.

Besides generating a feasible level structure, the levels must also be populated with challenges. Most of the previous work do not really focus on the placement of traps. Some papers include traps in their atomic building blocks and other papers represent traps as a different kind of cell type, without the need to adjust their algorithms.

After the levels are generated and provided with challenges, the difficulty of the level needs to be determined. The paper by Jennings-Teats et al. [17] and the paper by Pedersen et al. [5] use machine learning techniques to analyse gameplay data, and to train and and test a model. The advantage of such a method is that the players performance is directly mapped onto a model. The accuracy of their models is quite high given the chaotic and unpredictable behaviour of human players, respectively 64% and 77%. However the downside is that these result are quite specific for the tested game, although some general features are used. It is unclear how the model behaves when the levels are scaled, or when the levels become less linear.

These models use every challenge in the level to train the model, whether the challenge is faced or not by the player. The models will most likely fail to notice when a very difficult challenge can be avoided, instead they will rate the level as difficult due to this occurrence. This will not frequently happen in their tested applications since the levels are very linear with little room for detours. But this is a problem when less linear levels are considered.

The models do also not take the differences in skill level among players into account. But the models can easily be adjusted to this, by only learning from data of players of a given skill level.

The paper by Berseth et al. [13] predicts the routes of the player and enemy agents to measure difficulty, which allows them to notice situations where the player avoids challenges. However, this method will not be very practical since the assumption is made that players move with a constant velocity along their path and human players do not usually move in such a predictable way. But this method gives an indication of how difficult a path of a player can be, based on encounters with enemies along its path.

The paper by Mourato and dos Santos [19] gives a measurement of difficulty based on the size of gaps of static challenges. Although it is a simple method, it allows to measure the difficulty of individual challenges rather than the difficulty of the entire level.

Since the levels of this thesis are not necessarily linear, a difficulty measurement is required that is capable of distinguishing between challenges which can be avoided and challenges which have to

be taken. Due to this requirement it is not appropriate to use a method that learns from the entire level, but instead a method is needed that can measure the difficulty of each individual challenge. The difficulty of an entire level is then based on the challenges that are encountered along the easiest path through the level.

A machine learning technique could be used to measure the difficulty of individual challenges. Each challenge can be represented by a patch, representing the local neighbourhood and the placement of the trap. These patches, along with some label whether the challenge is successfully traversed or not, can be fed to a neural network or clustering algorithm to determine a difficulty value. Although high accuracy can be achieved by such a method, it lacks generality and is not easy applicable in other games. Therefore a measurement of the difficulty of individual challenges based on general parameters is preferred. It would be ideal to express the difficulty of a given challenge c by

$$\delta(c) = f(x_0 \dots x_n)$$

where each x_i is a general parameter of the challenge c. Examples of general parameters could be the speed with which the traps move back and forth or the time required to move from the last safe location to the trap. These kind of parameters apply to a wide range of challenges.

Such a model must be based on statistical data, which can be gathered through experimenting the game. Some papers measure difficulty based on the opinion of players, but that is risky since humans do not perform well in judging their performances. Large differences occur between different persons, and even between ratings of the same person. Therefore objective data will be gathered from the play sessions, like the number of lives lost.

Chapter 3

Level Generation

This chapter proposes a method to generate levels. The method yields *non-monotone* gameplay and guarantees a feasible and *bi-directional* path through the levels. It also populates the levels with traps in a feasible manner. We start with a formal description of the problem and point out why non-monotone levels can not be created in a straightforward manner. After that we give an algorithm to create non-monotone levels. The global structure is represented by a maze of *rooms* where each room is provided with some exits. The rooms are procedurally generated by finding an arrangement of *building pieces* such that each pair of exits are connected. Each building piece represents a discretized jump. After that we propose a method that explains how the levels are populated with traps, without violating the guarantee on a feasible path through the levels. We start with constructing a *level graph*: a graph that represents the movement possibilities of the player through the world. We also give an abstract representation of traps. Once the representation is explained, interesting routes are found by an algorithm based on cycles in the level graph. Traps are placed along the found routes. An example of a generated level with traps can be seen in figure 3.1.

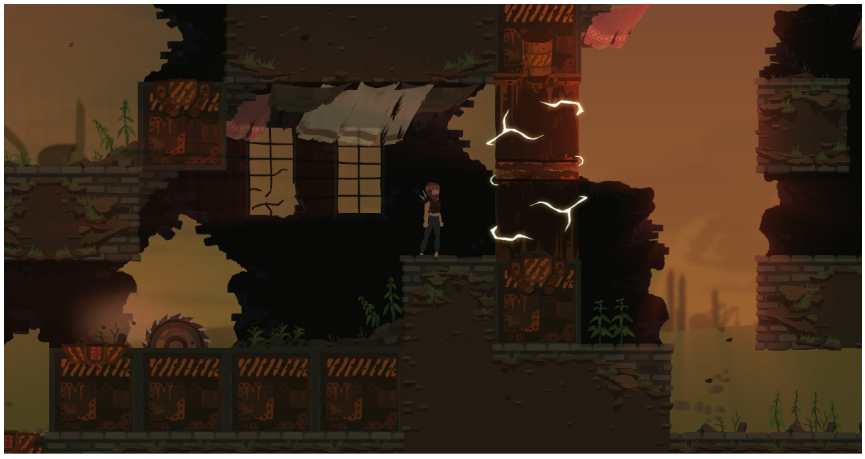


Figure 3.1: Example of a generated level.

3.1 Formal description

In many platformer games the player agent moves from left to right through the levels. The levels of such games are mainly linear with few branching paths. The level layout of these games is x-monotone. In a non-monotone platformer the player has the possibility to go in any direction, and to explore a non-linear level. The goal is to automatically generate non-monotone levels for 2D platform games.

Let E be the set of exits that the player agent must be able to reach, C_{free} the set of positions where the player agent can be without colliding with the levels obstacles and let $C_{obstacle}$ be the set of positions where the player agent collides with the levels obstacles. The goal is to generate a level $L(E)$ such that it yields a feasible path $\pi : E \subseteq \pi \subseteq C_{free}$. This would be trivial if the level is in the xz-plane, in which case the player agent can move in a straight line from exit to exit. In that case a level with $C_{obstacle} = \emptyset$ would yield a valid path.

The problem becomes more complicated when the xy-plane is considered, since gravitational forces make it impossible for the player to move freely in the y-direction. The dynamics of the player agent are constrained by

$$m\frac{d\boldsymbol{x}}{dt^2} = \boldsymbol{f}_p + \boldsymbol{f}_e$$

where m is mass of the player agent, $\frac{d\boldsymbol{x}}{dt^2}$ the accceleration of the player agent, $\boldsymbol{f}_p$ the forces that act on the player agent to control it and $\boldsymbol{f}_e$ are environmental forces that act on the player agent. $\boldsymbol{f}_p$ is constrained by the player agents physical abilities, it has a maximum jump and run force. $\boldsymbol{f}_e$ are external forces such as gravity, friction and drag.

It would still not be very complicated if the start exit is located above the end exit. The player agent could simply fall to reach the end exit. But the player agent would not be able to reach the start exit again. Therefore a bi-directional path between each pair of exits is mandatory, where each exit can be arbitrarily placed.

Now the player agent can not move freely in the xy-plane it is much harder to reach the exits E. It can still move freely in the horizontal direction over obstacles with flat surfaces (which normals point upwards). But to reach higher grounds, the player agent needs to make a sequence of jumps. The obstacles of the level need to be placed such that the player can reach the exits by a combination of horizontal movements and jumps.

The problem of generating a non-monotone level can be described as generating a level $L(E)$ in the xy-plane with an arbitrary set of exits E by finding a set of obstacles such that a feasible bi-directional path π between each pair of exits exists, subjected to the constrained movement of the player agent.

A platformer level must also contain traps to make it challenging. These traps must be placed with great care, as they may not interfere with the levels feasibility of having bi-directional paths between each pair of exits. On the other hand, trap-free paths between exits are also unwanted. The problem of adding traps to a level can be described as generating a set of traps $T(L, E)$ such that it maintains a feasible bi-directional path between each pairs of exits, while also preventing trap-free paths between exits.

3.2 Generating the level structure

This solution adopts the idea of the Spelunky method of creating a maze of rooms, and then generating each room individually. The method of Spelunky generates a y-monotone maze, and ensures that the start room is located at the top of the level and the end room at the bottom of the level. Instead of a y-monotone maze, a non x- and y-monotone maze is generated with the start and end room at arbitrary positions. Instead of using predefined templates, the rooms are procedurally generated and the exits $E(r)$ of room r are placed at arbitrary places at the edges of the room as long as they do not conflict with each other. The number of exits per edge is restricted to be at most one. A conflict would occur when placing two exits in the same upper corner. The exit at the horizontal edge requires a solid cell beneath it for the player agent to stand on, and this solid cell can accidentally block off the exit at the vertical edge of the room. The exits are placed based on the connectivity of a room with it's neighbours such that exits of adjacent rooms are aligned.

The use of this room structure gives control over the global structure and also narrows down the solution space, which makes it faster to compute a solution.

The movements of the player agent are discretized and for each movement a polyomino is obtained, which is a plane geometric figure formed by joining one or more cells edge to edge and where the cells are equally sized squares. Three types of cells can be distinguished. A cell type that represents open space needed by the player agent to execute it's movement, a cell type that represents solid space and a cell type that represents a connection point. A connection cell represents also open space. The player agent starts it's movement at one of the connections points and finishes it's movement at another connection point. These discretized movements are called building pieces and some examples are displayed in 3.2.

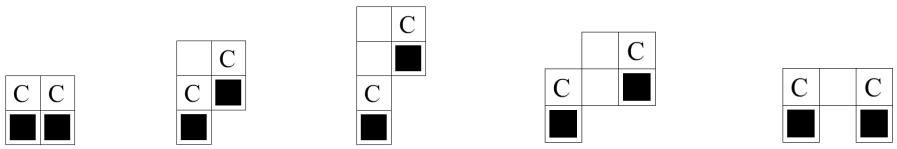


Figure 3.2: The black squares are solid cells, white squares are non-solid cells and the squares which contain a C are connection cells which are also non-solid.

Given a room r with exits $E(r)$, the solution $\pi(r)$ for r is a combination of transformed building pieces such that all exits are contained:

$$\pi(r) = \bigcup_i g_i \cdot b_i : \pi(r) \subseteq E(r), g_i \in G, b_i \in B$$

where g_i is a transformation from the group G that represents horizontal mirroring and translational transformations and b_i is a building piece from the set of available building pieces B. The solution is constrained by the following constraints C:

- C_0 : a transformed building piece must be fully contained by the room;

- C_1 : two transformed building pieces b' and b'' can only be connected if a connection cell of b' coincides with a connection cell of b'', and open cells of b' don't coincide with solid cells of b'' and vice versa;
- C_2 : a transformed building piece must at least add one new connection cell to the path $\pi(r)$;
- C_3 : only connection cells of a transformed building piece are allowed to coincide with an exit cell e of the room;
- C_4 : a consecutive sequence of transformed building pieces must exists between each pair of exits.

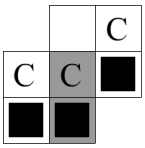


Figure 3.3: An example of how the building pieces of figure 3.6a and 3.2b can be feasibly connected. The gray area represents overlapping of the two pieces.

Evolutionary search approaches are very popular, but they don't suite well for finding a solution $\pi(r)$ subjected to C. As a mutation operation, one of the building pieces could be swapped with a random building piece. But this will most likely violate the constraints C_1 and C_4. The mutated building piece can overlap with other pieces in an illegal way or it breaks a sequence between two exits. Besides that, a set of candidate solutions is required during the evolutionary search process, where each candidate solution is a feasible solution. This yields a contradiction, as an evolutionary search process is used to find a feasible solution in the first place. This can be solved by no longer requiring that each candidate solution is a full solution. An evaluation function could be used to measure how close a candidate solution is to a full solution, but that would be a very complex function.

3.2.1 Algorithm for generating rooms

The solution of a room is not unique and no optimum solution is required. We find that the probability of finding a solution within a reasonable amount of time is quite high, as long as the rooms are relative small. Therefore an exhaustive search method is possible. The proposed algorithm is based on a random depth-first technique.

A cell with the marker e is placed for each exit of $E(r)$ at the corresponding position in the room. The algorithm starts at one of the exits and during each iteration it tries to add one transformed building piece to the path until all exits are contained. Each newly placed building piece adds one or more connection cells to the path. The used building pieces represent individual moves and therefore contain exactly two connection cells, but there is no restriction on the number of connection cells as long as the player agent can move between each pair of connection cells in a

feasible manner. One of the connection cells is used to connect the new building piece to an already placed building piece. The remaining connection cells yield a new branch in the search process.

A building piece is only added to $\pi(r)$ if it does not violate the constraints $C_i : i \in [0, 3]$. The constraint C_4 can only be satisfied on a global level and therefore a newly placed building does not have to satisfy C_4. The algorithm must backtrack when no building piece can satisfy the constraints $C_i : i \in [0, 3]$ given a certain connection cell. A previously placed building piece is removed from $\pi(r)$ and another branch is tried. The number of exits left is decremented when a newly placed building piece contains an exit cell in a feasible way, and when no exits are left the algorithm returns it's result $\pi(r)$. When there are still exits left, all the connection cells that belong to $\pi(r)$ become new branch points. An example of a successful result is displayed in figure 3.4.

Data: Room r, solution π, exits E, exits left E', building pieces B, start points S
Result: solution $\pi = \bigcup_i g_i \cdot b_i$ subjected to C
forall the *start point* $s_i \in perm(S)$ **do**
  forall the *building piece* $b_j \in perm(B)$ **do**
    forall the *transformation* $g_k \in perm(transformations(b_j, s_i))$ **do**
      $b' = g_k \cdot b_j$;
      if $satisfiesConstraints(b', \pi, r)$ **then**
        $\pi = \pi \cup b'$;
        $E'' = containsExits(b', E')$;
        $E' = E' \backslash E''$;
        if $E' = \emptyset$ **then**
          return π;
        end
        if $E'' \neq \emptyset$ **then**
          $S' = possibleStartPoints(\pi)$;
        end
        else
          $S' = possibleStartPoints(b', s_i)$;
        end
        $\pi' = solveRoom(r, \pi, E, E', B, S')$;
        if $\pi' \neq \emptyset$ **then**
          return π';
        end
        else
          $\pi = \pi \backslash b'$;
          $E' = E' \cup E''$;
        end
      end
    end
  end
end
return $\emptyset$;

Algorithm 1: solveRoom

perm Arranges the elements of a set S into a random order.

transformations Returns all transformations such that a building piece can be connected to the given starting point s.

satisfiesConstraints Returns whether a placed building piece b satisfies the constraints $C_i : i \in [0, 3]$.

possibleStartPoints Returns all connection cells contained by π when a path π is given as argument, otherwise it returns the unused connection cells of the given building piece b.

containsExits Returns the exits which are contained by the given building piece.

A non-empty result yields a feasible bi-directional path between each pair of exits. The path is bi-directional since each individual building piece represents a bi-directional jump. The search starts at one of the exits e_0, a path π' between two exits is found as soon as another exit e_1 is contained by π'. A random point of π' is chosen as start point for the search for another exit e_2, which results in a path π''. The path π'' connects e_2 to π' which already connects e_0 with e_1, therefore e_2 is also connected with e_0 and e_1. This inductively connects all exits. The result consists of building pieces composed of solid and non-solid cells. The solid cells form the set of obstacles needed for the player agent to reach each exit, and the non-solid cells form a collision free path. The remaining cells of the room can be filled in freely.

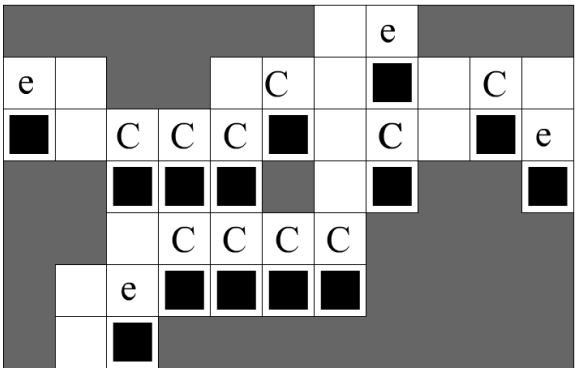


Figure 3.4: Solution for a room of size 11x7 solved with the building pieces of 3.2 with an exit at each side of the room. The cells marked with 'e' are exits and the dark gray area represents unused cells. Note that the marker of the exit at the bottom of the room is moved upwards by one unit, and a solid cell and two non-solid cells are added to make sure that the player agent can leave the room through the bottom side.

3.2.2 Properties

The algorithm uses a search technique where the number of branches grow exponentially since all building pieces of B are candidates to be connected to the last building piece of $\pi(r)$. Consider the case where the placement of a building piece blocks off an exit. The algorithm will search the entire space left before that building piece can be removed during backtracking. In that case the algorithm will consume a huge amount of time before terminating.

On the other hand, the algorithm can also terminate very quickly. Let the room be of size $O(n)$ and each building piece of size $O(1)$. The solution consists of at most $O(n)$ pieces and there exists the possibility that the algorithm finds that solution in $O(n)$.

So the algorithm has the probability of terminating very quickly and also of consuming a huge amount of time. An experiment was conducted to test how likely it is that the algorithm would terminate within a reasonable amount of time. A reasonable amount of time was defined as the time required to perform 5000 iterations, which is on a modern machine a few milliseconds. We find that the probability of quickly terminating is high for small rooms and decreases when the room size increases. A room of size 11x7 yields a 90% probability of terminating quickly.

Since not every attempt results in a quick termination, the attempts that have not found a solution within 5000 iterations are terminated and a new attempt is started until a solution is found. Trying different attempts of limited iterations is still faster than searching the entire space for a solution. However, the very unlikely event exists that a quick solution will never be found. Let p be the probability of finding a quick solution and x be the number of attempts, then the probability of not finding a quick solution $\lim_{x\to\infty}(1-p)^x$ will not become exactly zero. The generator should fall back on predefined rooms when after n attempts still no solution is found. Since each solution depends only on the seed for the random number generator, a solution for each exit configuration can be represented by a seed. This seed can be computed offline.

The fact that the algorithm terminates does not imply that a solution is found, it could also imply that no solution exists. Not finding a solution is a catastrophic failure, it is like a dungeon without an entrance and exit.

An experiment was conducted to show that a solution always exists. The algorithm was used to find a solution for each possible exit configuration of a room of given dimensions. It turned out that a solution always exists for rooms of size 5x6 or larger, and for rooms of size 4x4 and 5x5.

There is not much control over the room solutions as it is fully random process. The only thing that can be controlled is the set of building pieces to be used. A minimum set of building is required to ensure that a solution exists, but additional pieces can be added to this set. On the contrary, the global maze structure of rooms is easily controlled. The length and the amount of branching can be easily adjusted.

3.3 Generating traps

After the level generator has created a level with a feasible path, it is time to make that level more challenging by adding traps to it. The level is represented by a grid with only solid and non solid cells, and some markers which indicate the start and finish point of the level. The problem of adding traps to a level L with exits E is given by generating a set of traps $T(L, E)$ such that it maintains a feasible bi-directional path between each pairs of exits, while also preventing trap free paths between exits.

3.3.1 Graph representation

At all time a feasible path between the exits of the level must exist, therefore the traps can not be placed carelessly in the level. A graph representation of the level is used to analyze the possible routes the player agent can take. The level generator uses predefined building pieces that represent feasible movements of the player agent. These building pieces can also be used to compute routes, but a minor adjustment is required. During the generation process it is not necessary to have information about how the player moves between two points, only a guaranteed path between the two points is necessary. Some movements are in-game composed of two jumps, one jump to a point where the player agent can hang on a ledge and from there a jump to a point where the player agent can stand. So the distinction between nodes where the player agent can hang and stand must be made. Figure 3.5 gives an example of the hang gameplay mechanic and figure 3.6 displays how a single building piece of the generating process is subdivided into two jumps.

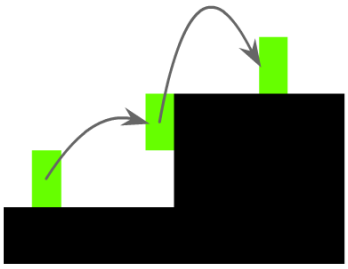
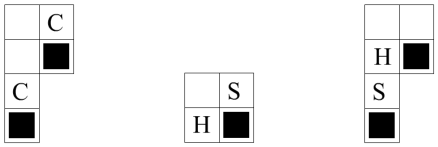


Figure 3.5: Example of the hang gameplay mechanic. The black area represents solid space and the green rectangle represents the player agent. The gray arrows represent jump actions. The player can hang on ledges and has the possibility to perform a jump action when he is hanging.



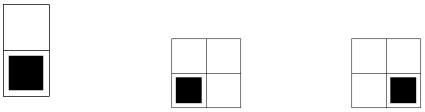
(a) (b) (c)

Figure 3.6: The building piece displayed in 3.6a is decomposed into the separate jumps 3.6b and 3.6c which are used to represent the level by a graph. Cells with an S represent places where the player agent can stand and cells with an H represent places where the player agent can hang on a ledge.

The player agent can hang on ledges which are located at the left side of a cell as well as on ledges which are located at the right side of a cell. Therefore three types of nodes are distinguished:

- stand node;
- left hang node;
- right hang node.

The graph is built in two phases, first the nodes are determined and then edges between the nodes. The nodes are found by a pattern matching process. Each node type is represented by a patch containing solid and non-solid cells as displayed in 3.7. A node is placed in the level at the point where the patch matches with the level geometry. The edges are computed by a similar process. Each jump is represented by a patch which also contains information about the node types, an example of a jump patch is displayed in 3.6b. The cells that contain an S must overlap with cells in the level that contain a stand node and an H must overlap with a cell that contains a hang node. Since there are two types of hang nodes, the patches of the hang nodes are matched with the jump patch to determine the correct node type.



(a) (b) (c)

Figure 3.7: A stand node used the patch displayed in 3.7a and the left and right hang nodes use respectively the patches displayed in 3.7b and 3.7c.

The weight of an edge is given by the Manhattan-distance between the two nodes it connects. The player agent is able to jump a distance of multiple cells in the horizontal direction. Since the Manhattan-distance is a true metric, the cost for taking a single jump of multiple cells in the

horizontal direction is equal to the sum of walking multiple cells one by one. This may lead to problems when computing properties like the shortest path because a single jump skips a number of nodes. These nodes will not be part of the shortest path while it can be very useful to actually have them part of it. Giving a lower weight than the Manhattan-distance to two horizontal adjacent nodes solves this problem. The weight $w_{i,j}$ between the pair of nodes n_i and n_j is given by:

$$w_{i,j} = \begin{cases} \frac{1}{2}, & \text{if } |\Delta x| = 1 \wedge |\Delta y| = 0 \\ |\Delta x| + |\Delta y|, & \text{otherwise} \end{cases}$$

where Δx is the horizontal distance between p_i and p_j and Δy the vertical distance between p_i and p_j. The resulting graph is undirected as for each pair (i, j) also a pair (j, i) exists. Whether the graph is connected depends on the result of the level generator, it will not be connected when the level contains isolated areas.

3.3.2 Trap representation

A trap, or a sequence of traps, is represented by a graph and by a set of modifications to the level structure. The graph is used to express a path for the player to traverse the trap. The graph is composed of nodes and edges of the same type as the graph that represents the level structure. All traps alter the level structure in some way. At least the value of one cell of the level structure is turned into a value that represents a trap. Two types of traps are distinguished: a trap that is at a fixed location and is not traversable for the player agent, and a trap that is only periodically active which gives the opportunity for the player agent to traverse the trap. That a trap is periodically active can imply that the trap is periodically turned on and off or that a trap moves over multiple cells, such that not every cell constantly creates a danger for the player agent. Other modifications that are necessary for most traps can be the requirement for some cells to be solid or to be open. When a player is expected to jump over a dangerous obstacle, the cell above the obstacle must be open and the cell beneath must be solid to support the obstacle.

A cell can have one of the following values:

- [#] : the cell is currently solid, but is not obliged to;
- [*o*] : the cell is currently open, but is not obliged to;
- [=] : the cell is obliged to be solid;
- [.] : the cell is obliged to be open;
- [*t*] : the cell contains a periodically active trap;
- [*d*] : the cell contains a trap that can't be traversed.

Modifiers are used to alter the level structure, each modifier corresponds to a cell in the level structure and has a value into which the cell must be altered. The value of a modifier can be one of the cell values. But there are two extra values which correspond to digging cells and to adding solid cells:

- [\] : turns an open cell into a solid cell;
- [+] : turns a solid cell into an open cell.

A modifier is not always allowed to alter the level the structure, it depends on the value of the modifier and on the value of the cell of the level structure. A truth table is given in figure 3.8.

	[#]	[o]	[=]	[.]	[t]	[d]
[#]	true	false	false	false	false	false
[o]	false	true	false	false	false	false
[=]	true	false	true	false	false	false
[.]	false	true	false	true	false	false
[t]	false	true	false	false	false	false
[d]	true	true	false	false	false	false
[\]	true	true	false	false	false	false
[+]	true	true	false	false	false	false

Figure 3.8: Truth table of the modifiers. Possible cell values are positioned along the x-axis and the modifier values along the y-axis.

This representation does not determine which exact types of traps end up in the game, and puts no limitations on the kind of traps to be used. The only drawback of this representation is that a cell can contain at most one trap. This is an acceptable limitation, since this is the case in most designs of platform games. A composite that has a trap on the floor and also a trap on the ceiling can be represented by one area, into which the individual components are added later on, since they together form one trap. But when one insists on having multiple trap values per cell, the cells can be subdivided into multiple areas. A trap, or more generally a trap sequence, is formally given by

$$S_i = (G_i, M_i)$$

where G_i is a graph and M_i is the set of modifications that must be performed on the level structure.

3.3.3 Routes

When traps are placed in a sequence, it is very useful to place them along a path that is likely to be travelled by the player. The shortest path will most likely be such a path. But when the shortest path is covered with traps, the player will try to take an easier detour. Therefore alternative routes must be detected to place traps on. An alternative path indicates a cycle in the graph since a start and end node are connected by two different paths. So the problem of finding alternative routes can be solved by finding simple cycles in a undirected graph. A simple cycle of graph G is a sequence of vertices starting and ending at the same vertex where each two consecutive vertices in the sequence are connected by an edge of G and no repetitions of vertices occur in the sequence except for the start vertex.

The undirected graph representing the level contains a lot of simple cycles since the player can jump multiple units in the horizontal direction, and walk back in the opposite direction until he reaches his start point again. These small cycles are not very useful to place trap sequences on. Therefore larger cycles are favoured over smaller ones.

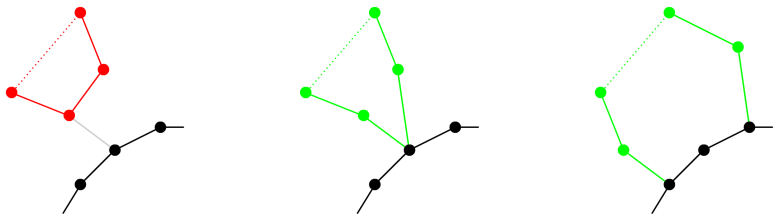
Detecting cycles

A simple cycle in an undirected graph is indicated by a back edge during a depth first search. When a vertex v_i is again visited during the search process through an edge e_{ji} starting at vertex v_j, then the edge e_{ji} is a back edge and a simple cycle exists with v_i as a start. A lot of unwanted small cycles will be detected with this method. A minimum spanning tree is used to solve that problem. A spanning tree $ST(G)$ of graph G is a subgraph of G which connects all vertices of G together. The total weight of the spanning tree is the sum of the weights of all edges that belong to $ST(G)$. A minimum spanning tree $MST(G)$ of G is a spanning tree such that there exists no other spanning tree with less total weight. Multiple minimum spanning trees can exist for the same graph.

Instead of using a depth first search to find back edges, the MST can be used to find back edges. Each edge that doesn't belong to the MST is an back edge. Many back edges will indicate very small cycles, but the edges that connect two end vertices of the MST indicate larger cycles. An end vertex of the MST is a vertex which is only connected by one edge. There are multiple methods to compute a minimum spanning tree. A very straight forward one is Prim's algorithm. That is a greedy algorithm which adds one edge to the minimum spanning tree during each iteration. The edge with the lowest weight is added, from the set of edges that connects a vertex that does already belong to the MST with a vertex that does not. The running time is $O(m + n \log n)$ where m is the number of vertices and n the number of edges. There are methods that run in linear time, but since the graph contains only a few hundred vertices the computational cost is relatively low and has no noticeable impact on the computation time. The computation time is dominated by the algorithm used for the level generation, which runs worst-case in exponential time.

Alternative routes algorithm

Not every simple cycle is an alternative route. Let R be the set of already found routes, which is initialized to the shortest route from a start vertex v_{start} to an end vertex v_{end}. A simple cycle c can only be an alternative route when $c \cap R \neq \emptyset$, otherwise it is just an isolated cycle without a connection to the main path. Not the entire cycle will be an alternative route, only the part of the cycle that does not occur in the already found routes, so the new alternative route r is given by $c \backslash R$. This is displayed in 3.9.



(a) (b) (c)

Figure 3.9: The black nodes and edges belong to already found routes and the gray ones belong to nothing yet. The red nodes and edges form a simple cycle that results in a invalid route. The green nodes and edges represent a valid route. The dotted edges display back edges.

As mentioned before, longer routes are preferred over smaller ones since the probability that

they can facilitate a sequence of traps is larger. However, it is not necessary to find the largest possible route since that is much harder to find. It is a combinatorial problem involving merging multiple cycles to one large cycle.

During each iteration all back edges e_i are evaluated whether they result in a valid alternative route. From all possible routes, the longest route r is chosen and is added to the set of already found routes R. The length of a route r is given by the sum of the weights of the edges in r. This process continues until no new route can be found. It is not guaranteed that R contains all vertices of G at the end, the missing vertices indicate dead ends. This method is not really efficient as it runs in quadratic time since all edges are evaluated during each iteration. But it has no noticeable impact on the running time. The algorithm to find alternative routes is given below:

Data: undirected graph G, start vertex v_{start}, end vertex v_{end}
Result: routes R
$mst = MST(G, v_{start})$;
$r = shortestPath(mst, v_{start}, v_{end})$;
$R = r$;
while $r \neq \emptyset$ **do**
  $E = computeBackEdges(R, mst)$;
  $r = \underset{i}{\operatorname{argmax}} \|route(e_i \in E)\|$;
  $R = R \cup r$;
end
return R;

Algorithm 2: computeRoutes

MST Returns the minimum spanning tree of the given graph G starting at the given start vertex v_{start}.

shortestPath Returns the shortest path between the given vertices v_{start} and v_{end}. This operation runs in linear time since backtracking from v_{end} to v_{start} in the MST results in the shortest path.

computeBackEdges Returns all edges e_i that have at most one vertex that occur in the given routes R, such that they produce a simple cycle $c_i : c_i \cap R \neq \emptyset$.

route Returns the route $r_i = c_i \backslash R$, where c_i is the simple cycle that is produced by the back edge e_i. The length of r_i is given by the sum of the weights of all edges in r_i.

3.3.4 Placing traps

At this point the level structure L, represented by the graph G, and a set of routes R through the level are available, the next step is to find a configuration $T(L, G, R)$ of trap sequences:

$$T(L, G, R) = \bigcup_i S_i : S_i = (G'_i, M_i)$$

that satisfies the following constraints:

- C_0 : G'_i matches with a part of a route $r_j \in R$;
- C_1 : M_i alters the level structure in a feasible way;
- C_2 : there exists a node $v' : v' \in predecessor(S_i), trap(v') = \emptyset$;
- C_3 : there exists a node $v' : v' \in successor(S_i), trap(v') = \emptyset$;
- C_4 : there exists a feasible path between each pair of exits of the level after applying M_i;
- C_5 : $\parallel \bigcup_i G_i \parallel < \alpha \parallel G \parallel$.

The constraints C_0 and C_1 are used for fitting the trap sequence into the level. The trap sequence does not fit when these constraints are not satisfied. The constraints C_2 and C_3 check whether S_i does not conflict with other sequences, by ensuring that S_i has a trap-free start node and a trap free end node. When a trap sequence is placed, it alters the level in some way. Constraint C_4 ensures that the level is still playable by checking whether each pair of exits can still be reached by the player agent. C_5 is used to regulate the maximum number of traps that can be placed, it forces that the sum of all nodes that belong to any sequence $\parallel \bigcup_i G_i \parallel$ is less than a proportion α of the number of nodes $\parallel G \parallel$ in the graph G.

The configuration is found by a simple random search algorithm without backtracking. Backtracking is not necessary since the constraints do not require a minimum of sequences to be placed. Therefore all branches of the search tree lead to a valid solution. The algorithm loops over all routes and tries to place traps on them as long as it is possible without exceeding the maximum amount. The satisfaction of C_5 is ensured by constraining the number of trap nodes along a route r_i to be less than the proportion α of the number of nodes $\parallel r_i \parallel$ of the route r_j

```
Data: level structure L, graph G and routes R
Result: trap configuration S
S = ∅;
for r_i ∈ R do
    n = 0;
    Δn = ∞;
    while Δn > 0 do
        S' = possibleTraps();
        Δn = 0;
        for (G_j, M_j) ∈ S' do
            if fit(G_j, M_j, r_i, L) then
                applyModifications(M_j, L);
                updateGraph(L, G);
                Δn =‖ G_j ‖;
                n = n + Δn;
                if existPaths(L, G) ∧ n < α ‖ r_i ‖ then
                    S = S ∪ (G_j, M_j);
                    break;
                else
                    undoModifications(M_j, L);
                    updateGraph(L, G);
                    n = n − Δn;
                    Δn = 0;
                end
            end
        end
    end
end
return S;
```

Algorithm 3: computeRoutes

possibleTraps Returns a set of possible trap sequences to be placed, in a random order. Each trap sequence S_j consists of graph G_j and a set of modifications M_j.

fit Checks whether the constraints C_0, C_1, C_2 and C_3 are met. Although C_2 and C_3 only require only one trap free node, this function checks whether all successors and predecessors are trap free.

applyModifications Alters the cell values according to the given set of modifiers.

updateGraph Updates the graph G by adding and removing nodes and edges according to the new cell values of the level structure L.

existPaths Checks whether C_4 is met.

The result is a set of trap sequences. Although each trap sequence is represented as a single graph and a single set of modifiers, the trap sequence can internally be subdivided into multiple traps along with their graphs and modifiers. When a single trap sequence contains multiple traps

which are periodically activated, care must be taken when configuring the periods of the traps. When two consecutive traps inside a sequence are badly configured, the situation can arise where the player impossibly can traverse both traps. For example, at the time that the player traversed the first trap, he will get killed by the second trap.

The time that is required to move between two consecutive traps must be taken into account. Let q_i and q_{i+1} be two consecutive traps and $t(q_i, q_{i+1})$ be the time required for the player agent to move between them. Then q_{i+1} must be delayed by $t(q_i, q_{i+1})$ to ensure a feasible pass through. The nodes need to have a position in world space to compute the time required between two nodes. The position of the hang nodes corresponds with the position of the center of mass of the player agent when he hangs on a ledge, and the position of the stand nodes correspond with the position when the player stands at the center of a cell. The time required between two nodes can be computed by fitting a parabola which corresponds to the gravity force and the physical properties of the player agent. However, the results diverge enormously with the in-game measured values. This is the consequence of using a semi-implicit integration method to compute trajectories in the game. In a real life situation is the trajectory of an object with no initial velocity given by

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \int_t \int_t \boldsymbol{F}(t) m^{-1} \, dt \, dt$$

where $\boldsymbol{x}$ is the position of the object and $\boldsymbol{F}$ a function which gives the net force for each point in time. But the forces that an object in the game experiences are not known at forehand for each point in time. The forces are influenced by the players input and by collisions between objects. Therefore numerical integration methods are used, they approximate the exact trajectory. But the approximated trajectory differs from the exact solution, and this error builds up over time. The error caused by the semi-implicit integrator becomes so high that the resulting trajectory can't be described by a parabola. Therefore numerical integration is also used to measure the time required between two points.

Chapter 4

Measuring Difficulty

This chapter explains how the difficulty of a generated level can be estimated. We start with a formal description of the problem. We explain a *challenge model* to estimate difficulty of individual challenges, and a *level model* to estimate difficulty of entire levels. The challenge model is based on statistical data. This data is obtained by performing a user study. We briefly explain the gameplay mechanics and the traps that have been implemented.

Relevant data is extracted from the user study results to build the challenge model. An *event trace* is constructed for each encounter with a challenge. These event traces can not be used directly for the challenge model. The event traces are mapped onto the level graph such that each challenge can be expressed as a *sub-graph* of the level graph. A set of general parameters is subtracted from these sub-graphs. This set of parameters, and a label whether the challenge is successfully traversed or not, are used to construct the challenge model.

This challenge model is used to construct the level model. The level model is based on a *probability graph*. This probability graph represents the probability of successfully reaching a node of the level graph. Here, each edge of the probability graph has a weight assigned by the challenge model. The difficulty of an entire level is given by the easiest path through this probability graph.

4.1 Formal description

Let a challenge c be represented by a set parameters $\boldsymbol{x} = (x_0 \dots x_n)$, then the probability that the player successfully overcomes challenge c is given by:

$$\delta(\boldsymbol{x}) = p(SUCCES \mid \boldsymbol{x})$$

Let G be the graph that represents a level L. In order to determine the difficulty of L, the graph G must be altered into G' such that G' represents the probability graph of L. The graph G' represents what the probability is that a player successfully reaches a node of G. Each edge $e_{ij} \in G'$ has a weight $w_{ij} : 0 \leq w_{ij} \leq 1$ which represents the probability that the player successfully traverses that edge. $w_{ij} = 1$ when the edge is not a part of a challenge, and $w_{ij} = \delta(\boldsymbol{x})$ when it is part of a challenge represented by $\boldsymbol{x}$.

Let $\pi_{min}(G')$ be the path between the start and the end of the level L with the highest probability of success, then the difficulty of the level L is given by the product of all weights of $\pi_{min}(G')$:

$$\delta(L) = \bigcap_i weight(e_i) : e_i \in \pi_{min}(G')$$

4.2 User study

A user study is performed to gather gameplay data, which is used to build a model for measuring difficulty. The previously explained level generator is used to generate levels for the user study. Our own 2D game engine written in C/C++ is used to play the levels. The idea of the game is very simple, the player controls a player agent through a world consisting of solid and non solid cells. The size of a cell is 2 by 2 meters and the frame rate of the game is 60 fps, so a single time step takes approximately 16 ms. The player agent has the following physical characteristics:

- Horizontal movement: the right and left arrow keys are used to control the player agent in the horizontal direction. The player agents accelerates almost instantly to the maximum horizontal speed of $9.5ms^{-1}$. The player agent can move as well as on the ground as in the air and the player needs to hold the arrow key down to keep moving.

- Vertical movement: the arrow up key is used to jump. When the player agent touches the ground and the player agent presses the arrow up key, an upwards force is applied to the player agent for exactly one time step. This force changes the vertical velocity instantly to $16ms^{-1}$. The player can control the height of the jump by holding the up arrow key down, when the arrow up key is released before the player agents reaches it's highest point, the player agents vertical velocity is set to zero and he will start falling down.

- Hanging: when the player agent is falling down, and close enough to ledge, the player automatically grabs that ledge to hang on. The player can leave the hang state by either a jump action or a release action which is performed by pressing the arrow down key.

- Gravity: the player agents experiences each time step a vertical acceleration of $-40ms^{-2}$ due to gravitational forces. Such large gravity forces are normal for platform games to make the gameplay fast.

- Friction: a special case friction force acts on the player agents body to simulate friction and drag. The maximum value of the friction force calculated during the collision process depends on the magnitude of the normal force. This can lead to situations where it seems like that the player sticks to a wall, which is physically correct but unwanted in platform games. This phenomena occurs due to the fact that the player agent is able to move in the air, so horizontal forces are applied to the agent while he is in the air. A horizontal normal force will act on the agent when he hits a wall to resolve the collision. As a consequence, the friction force acts in the vertical direction preventing the agent from sliding down the wall. The special case friction force is designed to remove only a part of the horizontal velocity and acts only when the player doesn't press the left or right arrow key. The special case friction force removes 20% of the horizontal velocity when the agent is in the air to simulate drag, and removes 80% when the agent touches the ground, resulting in an almost immediate stop.

- Player body: the rigid body belonging to the player agent has a mass of 1 kg and it's inertia tensor is set to infinity, implying that the angular momentum is kept constant and the player

agent can't rotate. The size of the shape that represents the player agent during the collision process is 0.66 m in width and 1.5 m in height, but the shape is not rectangular. Instead the shape has a pointed roof, to make certain jumps easier.

The previously explained trap generator is used to populate the level with traps. The following traps are implemented:

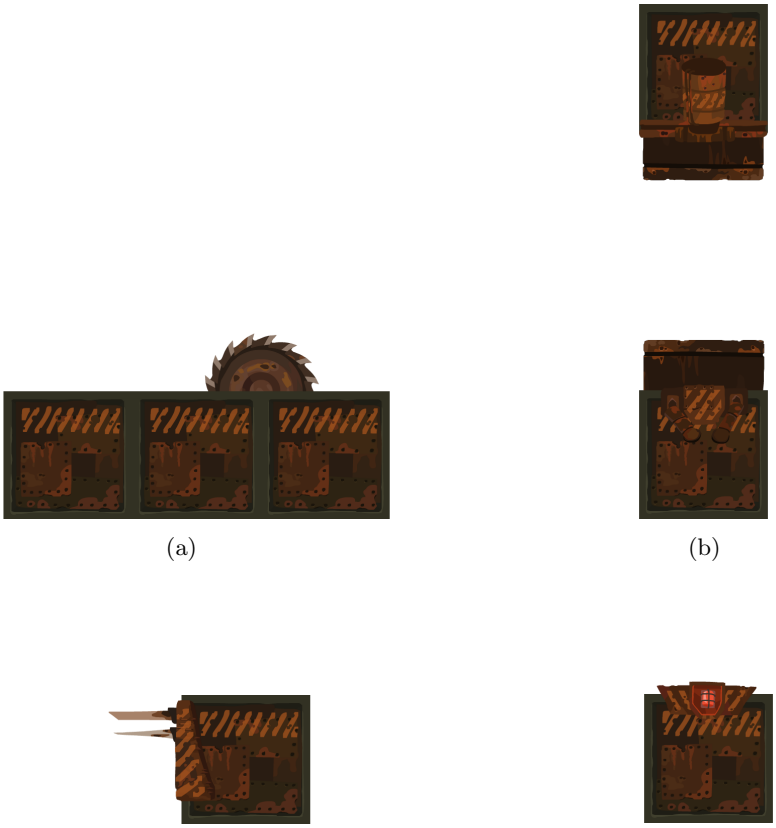
- Saw trap: this is a rotating saw that moves horizontally back and forth inside 3 cells. The player agent dies when he hits the saw and he can jump over the saw to successfully overcome the challenge. This kind of trap occurs in many platformer games. Super meat boy has very similar saws like this one, and in games like Super Mario it occurs as an enemy how walks back and forth.

- Wall spikes: sharp spikes come periodically out of a wall. The player agent dies when he hits the spikes, this can happen when he jumps into them or when he is hanging on a ledge. Spelunky is an example where this trap also occurs.

- Squeeze trap: this trap is composed of two metal blocks that vertically slam onto each other. The player agent dies when he is squeezed by the metal blocks. The squeeze event occurs periodically and lasts only for one single frame while the other traps have a larger window of time in which the player can die. To increase the window of time, the metal blocks are electrified when they hit each other and stay like that for a couple of frames. A horizontal movement is required to overcome the challenge. A very similar trap occurs in Prince of Persia and many other games have variants, like laser beams which are periodically activated.

- Mine trap: a bomb is activated when the player hits an activation area, after some amount of time the bomb explodes and can kill the player. This trap prevents the player agent to stay too long in the activation area. The activation area is exactly the size of one cell. This kind of trap occurs in many games like Spelunky.

The saw, squeeze and wall spike traps are atomic challenges. Each of these challenges needs another atomic movement to overcome. The mine trap is only used in combination with one of the other traps. The purpose of this trap is to increase the timing by the player. The number of mine traps before another trap is limited to two, more mines could lead to the situation where the player is no longer able to see the primary trap on the screen. Overcoming such a challenge is no longer based on timing but on luck. The traps are only periodically harmful to the player. The parameter t_{cycle} determines the cycle speed of each trap.



(a) (b)

(c) (d)

Figure 4.1: Examples of the implemented traps. The saw trap, squeeze trap, wall spikes and the mine trap are respectively displayed in 4.1a, 4.1b, 4.1c and 4.1d.

Initially, composite challenges were also implemented like a saw trap followed by a squeeze trap. But it is for the testers nearly impossible to overcome these challenges as it requires a considerable amount of skills. These skills can only be developed by playing the game for an extended period of time. Note that also the classic jump over a gap filled with spikes is not implemented. These kind of traps do not require a timing aspect and are already researched by others.

The experiment starts with two fixed tutorial levels. In the first level the controls are explained to the player, and the player needs to find an exit gate. The second tutorial is populated with traps and the player is asked to jump in some traps such that he understands how the traps work and when he gets killed. After that, the player must play ten levels populated with traps. Each level has a linear room layout: the exit gate is always located in the most right room. A linear room layout is chosen because the goal is to measure the difficulty of the easiest path. When a full maze generator is used for the room layout, it is unclear what path the player will take. The player might get stuck in dead ends. He might get killed in traps which he did not even had to take.

Although the room layout is linear, the level can still contain multiple routes to reach the exit. But it is restricted to the choice of taking or avoiding a challenge if possible. The parameter t_{cycle} is set randomly for each trap. The interval for the possible values increases linearly from $[2, 2]$ to $[1.2, 2]$ during the first five levels, and will remain like that for the last five levels.

During the play session some events are registered and saved. These events will be explained in the next section. Also the seed which is used to generate the level and all key presses are saved.

4.3 Event trace

From the recorded data it is possible to regenerate the play sessions. The recorded seed is used to generate the exact same level as in the original play session. The recorded key presses are fed to the automaton of the player agent, generating the exact same physics simulation as in the original play session. Besides the position and velocity, also other information is available like collision impulses and contact points. So it is possible to extract all necessary information from the player and the level.

The challenge model is based on the ratio between successfully overcoming a challenge and the number of tries. An unsuccessfull try results in a death event. These death events are simply registered. The successful events are not registered and need to be deduced from the players trajectories. Event traces are used to deduce these events from the players trajectories.

The game has a grid based layout and the traps are also placed in a grid based manner. Each trap covers one or more cells. When the player hits a cell that contains a trap, the player is considered to be no longer at a safe location. This does not immediately imply that the player agent is hit by the trap. The traps move back and forth, so the exact location of the traps depend on time.

Each time the player agents hits the ground while he is at a safe location, a new event trace is constructed. All events are added to the event trace until the player reaches a new safe location, or dies in a trap. The following events are registered:

- E_{pos}: the position of the player agent;
- E_{trap}: the player hits a cell that contains a trap;
- E_{node}: the closest node to position of the player agent when he touches the ground;
- E_{death}: the player is killed by a trap.

Each event gets a time stamp at the moment it is registered. When a finished event trace does not contain any trap or death event, it is removed and is not used in the statistics. The timing required to overcome a challenge probably influences the difficulty of that challenge. This timing value can be measured as the time between the last node event and the first trap event. However this value is not always representative for the actual timing required to overcome a given challenge. Some players make large jumps to overcome relative easy challenges. When the players succeed, large jumps relate to high probabilities of success. However, this relationship is not true. The actual relationship is that challenges which require less timing relate to high probabilities of success.

4.3.1 Mapping the event trace

During the generation process a graph is used to represent the level structure and to place traps in the level. The event traces are mapped onto this level graph, so relations between death events and sub-graphs can be used to construct the challenge model. An event trace already contains node events. However the sub-graph built from these nodes does not necessarily represent the shortest way to overcome the challenge. When players take unnecessary large jumps, some nodes are missing to construct the shortest path to overcome a challenge. Large timing values do still relate to challenges which can be easily overcome. An example is given in figure 4.2. Here, the players trajectory and the shortest sub-graph are displayed. The shortest sub-graph requires less timing than the large jump the player took.

(a) (b)

Figure 4.2: Example of the shortest sub-graph for overcoming a challenge, given a start location. The black cells represent solid cells, and the gray nodes and edges display the graph that represents the level structure. The dashed curve represents the players trajectory, jumping from a save location into a trap. The purple cells represents mine traps and the red cells saw traps. The yellow and the green part of the graph in 4.2a display the shortest path between the last safe location and the first encounter with a trap. The green part of the graph display the shortest sub-graph to overcome a challenge. In 4.2b is a situation displayed where the method fails to find the shortest sub-graph. This only happens when the player takes a large jump, which is also the shortest route. Therefore it is not considered as a problem.

The sub-graph is computed by the shortest path between the nodes v_{safe} and $v_{encounter}$. Here, v_{safe} represents the last safe location and $v_{encounter}$ represents the first encounter with a challenge. When the player successfully overcomes a given challenge, $v_{encounter}$ is the node where the player touches the ground for the first time while he is inside the challenge area. When a player dies during a challenge, $v_{encounter}$ is the closest node to the player agents position inside the challenge area. At this point, the sub-graph can contain more nodes than necessary. The sub-graph should only consist v_{safe}, $v_{encounter}$, possibly with some nodes which represent mine traps in the case that the challenge contains mines. All other nodes must be filtered from the sub-graph, such that the sub-graph starts with v_{safe}, zero or more mine nodes and ends with $v_{encounter}$. Note that a node representing a mine trap is not considered as a trap node, since mine traps are only placed to increase the timing aspect and are not a challenge on their own. The procedure is given below:

Data: graph G, event trace t_c
Result: sub-graph G'
$v_{safe} = firstNodeEvent(t_c)$;
$v_{encounter} = firstTrapEvent(t_c)$;
$P = shortestPath(G, v_{safe}, v_{encounter})$;
$G' = \emptyset$;
for $v_i \in P$ **do**
  if $type(v_i) == \emptyset \wedge type(next(v_i)) \neq \emptyset$ **then**
    $G' = G' \cup v_i$;
  end
  if $type(v_i) == mine$) **then**
    $G' = G' \cup v_i$;
  end
  if $type(v_i) \neq \emptyset \wedge type(v_i) \neq mine$ **then**
    return $G' \cup v_i$;
  end
end
return $\emptyset$;

Algorithm 4: computeSubGraph

firstNodeEvent Returns the node corresponding to the last safe location according to the event trace t_c.

firstTrapEvent Returns the node corresponding to the first encounter with an atomic challenge to the event trace t_c. Note that nodes corresponding to mine traps are ignored.

shortestPath Returns the shortest path between the two given nodes in the given graph.

type Returns the trap type $t \in T : T = \{\emptyset, mine, saw, squeeze, wallspike\}$ to which the given node corresponds.

next Returns the successor of the given node in the given path.

4.4 General parameters

Sub-graphs are used to express the difficulty of a challenge. Rather than using the nodes and edges, some features of the sub-graph and the challenge are used as parameters. The following parameters are used:

- t_{cycle} : the cycle time of the trap, or the time that is needed by the trap to reach its initial state again. The value is a property of the trap, and can be directly used as a parameter.
- t_{timing} : the time required by the player agent to move from the last safe node to the first trap node. This value is influenced by jumps that the player agent needs to make and by the presence of mines. This value can be deduced from the shortest sub-graph of a given challenge. Each node n_i corresponds to a position x_i in world space. The value for t_{timing} is computed as:

$$t_{timing} = \sum_{i=0}^{n-1} t(x_i, x_{i+1})$$

where $t(x_i, x_{i+1})$ is the time required by the player agent to move from the point x_i to x_{i+1} and n is number of nodes in the shortest sub-graph. The time between two points is computed in the same manner as is explained in the trap generator. When two nodes lie horizontally next to each other, thus when the horizontal distance between two nodes is exactly one unit and the vertical distance is zero, then the time required to move between these two points is given by $\parallel (x_i - x_{i+1}) \parallel / v_x$ where v_x is the maximum horizontal speed of the player agent.

- j : the number of jumps required by the player agent to traverse the shortest sub-graph. This value can be calculated by evaluating each two consecutive nodes in the sub-graph. When two nodes lie horizontally next to each other in world space, then no jump is required to move between these two nodes. In all other cases a jump is needed. An extra jump is added when the last node has no connection to a horizontally adjacent node besides its predecessor.

4.5 Challenge model

The challenge model predicts the difficulty of individual challenges. Each challenge is represented by the three parameters t_{cycle}, t_{timing} and j. So the challenge model must map $\mathbf{x} = (t_{cycle}, j, t_{timing})$ to $p(SUCCES \mid \mathbf{x}) : 0 \leq p \leq 1$. This model can be created based on the intuition of a game designer, in that case the model is theory-driven. It is likely that the probability of success decreases when the timing and number of jumps required increase. But it is unclear how fast the difficulty increases. Therefore a data-driven model is chosen.

The data gathered form the user study is used to built the model. Each event trace corresponds to a successful or unsuccessful walkthrough of a challenge. The result is a set of measurements $(\mathbf{x}_i, y_i)$, where each $\mathbf{x}_i$ consists of the three parameters and y_i is label which is either $SUCCES$ or $FAILURE$. However the model is expected to return a continuous value between 0 and 1 rather than a discrete label. Therefore the set of measurements $(\mathbf{x}_i, y_i)$ must be transformed into a set of measurements $(\mathbf{x}'_i, y'_i)$, where each y_i is a value between 0 and 1. A histogram is constructed which hashed each parameter $\mathbf{x}_i$ to a bin along with its measured label y_i. Each bin contains multiple measurements, according to these measurements a probability value between 0 and 1 can be computed. This results in a set of points $(\mathbf{x}'_i, y'_i)$ which can be used to built the model.

The challenge model can be obtained by fitting a function to the points $(\mathbf{x}'_i, y'_i)$.

4.5.1 Least squares

When fitting a function to a set of points, some measurement is required to determine how well the functions fits. Least squares is a method that is often used to determine the quality of the fit. The error ϵ is given by the squared sum of the differences between the computed values $f(x_i)$ and the measured values y_i:

$$\epsilon = \sum_i (f(x_i) - y_i)^2$$

Taking the square of the difference has the advantage that no distinction is made between points above and beneath the curve. And points that lie further away from the curve contribute way more to the total error than points that lie only a small amount away from the curve.

The best fit of a function is obtained by minimizing the error ϵ. Each function is composed of constants and variables where the constants are used to control the output of the function. The best fit is given by the values of these constants that minimize the square error. The best fit can be found be setting the gradient of the function to zero. So the partial derivative of each constant c_i is set to zero:

$$\frac{\partial \epsilon}{\partial c_i} = 0$$

This partial derivative gives the rate of change of the function when the constant c_i is changed by one unit, while keeping all other constants and variables constant. So it gives the rate of change only due to the change of c_i. A rate of change of zero implies that no better fit exists for the constant c_i. Performing this for each constant results in the best fit.

When a function consists of m constants, a system of m equations needs to be solved to obtain the values that minimize the error. Various functions can be used to model the difficulty of the individual challenges. A linear and an exponential function are good candidates. Higher order polynomials could yield better fits, but there is a reasonable chance that these models would overfit. The resulting curve can get distorted due to the goal of satisfying each point as best as possible. As a result the curve will include the unwanted noise.

A linear function is given by $y = c_0 + c_1 x$ and its error is given by $\epsilon(c_0, c_1) = \sum_i (c_0 + c_1 x - y_i)^2$. The following system must be solved to obtain c_0 and c_1:

$$\begin{array}{l} \frac{\partial \epsilon}{\partial c_0} = \sum_i 2(c_0 + c_1 x - y_i) = 0 \\ \frac{\partial \epsilon}{\partial c_1} = \sum_i 2(c_0 + c_1 x_i - y_i) x_i = 0 \end{array}$$

An exponential function is given by $y = Ae^{kx}$ but it is not linear in its constants, and therefore no linear system can be directly constructed to obtain values for the constants. The exponential function can be made linear in its constants by taking the natural logarithm at both sides: $ln(y) = ln(A) + kx = c_0 + c_1 x$. After solving the best linear fit, the constants of the exponential function can be obtained by $A = e^{c_0}$ and $k = c_1$.

A Gaussian-Jordan elimination algorithm is implemented to solve the systems. It is a straightforward algorithm that finds the solution by performing simple row operations. Before the system can be solved by Gaussian-Jordan elimination, it needs to be in the form $\mathbf{Ax} = \mathbf{b}$ such that $\mathbf{x}$ holds the constants.

As mentioned before the input parameters consist of three values: $\mathbf{x} = (t_{cycle}, j, t_{timing})$, instead of a single valued parameter. So the systems must be extended with some extra rows and columns according to the linear equation:

$$y = c_0 + c_1 t_{timing} + c_2 j + c_3 t_{cycletime}$$

Once the fitting of a function f is completed, the probability of successfully overcoming a challenge that is represented by the parameters $\mathbf{x}$ is given by:

$$f(\mathbf{x}) = p(SUCCES \mid \mathbf{x})$$

4.6 Level model

The level model predicts the difficulty of an entire level. This level model is based on a probability graph which represents the probability of successfully reaching a node of the level graph. When an edge is not part of a challenge, its weigth is 1. When an edge is part of a challenge, its weight is assigned by the challenge model. The difficulty of the level is given by the path through the level that yields the highest probability of success.

For each challenge a difficulty value can be obtained according to the challenge model. A challenge is represented by a sub-graph which contains multiple edges. However, the difficulty value relates to the entire challenge. Therefore it is not possible to assign a difficulty value to each individual edge of that challenge. The level graph must be altered such that it only contains edges which represent entire challenges and edges which do not belong to any challenge.

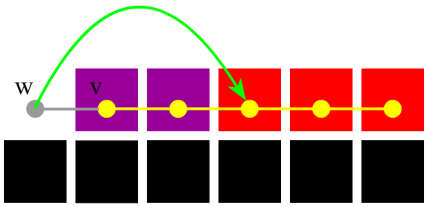


Figure 4.3: Example of a substitution of a challenge walkthrough by one single edge. The black cells represent solid cells, purple cells represent mine traps and the red cells saw traps. The yellow nodes and edges represent the sequence of nodes belonging to the challenge. Node w is an adjacent node of v, and the green line represents the new edge, starting at w and ending at the first node that represents a trap other than a mine.

The neighbourhood of each challenge must be inspected and appropriately altered. This means that an entire walkthrough of the challenge must be substituted by a single edge with a probability value according to the challenge model. An example is given in figure 4.3.

Let the challenge c be represented by the sub-graph, or path, P. Then each node $w \notin P$ that is adjacent to a node $v \in P$ indicates a starting point of a walkthrough of the challenge c. The walkthrough P' of c with w as a starting point does not entirely contain P. It starts with w, followed by a sequence of nodes of P starting at v and ending at the first node that represents a trap other than a mine trap. An edge e' is constructed between w and the last node of P' with a probability value that is deduced from P'. The procedure is given below:

```
Data: graph G, challenge c
Result: substitute edges E
P = challengePath(c, G);
E = ∅;
for v_i ∈ P do
    W = adj(v_i);
    for w_j ∈ W do
        if w_j ∈ P then
            continue;
        end
        P' = w_j ∪ challengeSubPath(v_i, P);
        δ' = δ(P');
        e' = edge(w_j, last(P'), δ');
        E = E ∪ e';
    end
end
return E;
```

Algorithm 5: computeSubstituteEdges

challengePath Returns a sequence of nodes that represent the given challenge.

adj Returns all adjacent nodes to the given node.

challengeSubPath Returns a sub-sequence of the given path, starting at the given node and ending at the first node that represents a trap other than a mine trap.

δ Returns a difficulty value that corresponds to the probability of overcoming the challenge, where the challenge is represented as the given sequence of nodes. The difficulty value is deduced from the given sequence, by subtracting parameters from it and feeding these parameters to a function that computes the difficulty.

last Returns the last node of the given sequence.

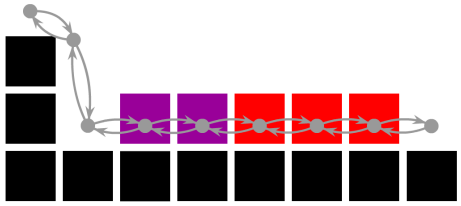
edge Returns a edge between the two given nodes, with the given weight value.

Besides adding new edges, some edges need to be removed from the level graph. The substitute edges $E_{substitute}$ represent the possible ways to overcome a challenge. All other edges that connect a node outside a trap area with a node inside a trap area must be removed. Some edges have none of the end points inside a trap area but still provide a way to overcome a challenge. So all edges that intersect with a trap cell must be removed. These edges form the set E_{trap}.

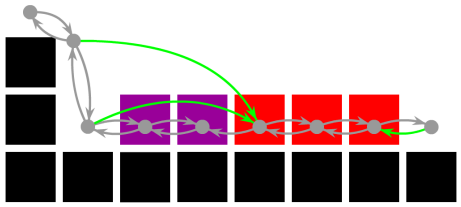
The edges from $E_{substitute}$ provide only a way to enter a trap area from an outside node, but not a way to leave the challenge. E_{traps} contains all edges that intersect with a trap, so E_{traps} also contains edges that are useful to leave a challenge and to move inside a challenge. These edges must be preserved. Let these edges form the set $E_{leaving}$ then the resulting graph G' is given by:

$$G' = G/E_{trap} \cup E_{substitute} \cup E_{leaving}$$

All edges besides the edges from $E_{substitute}$ have a weight of 1, since only the edges from $E_{substitute}$ provide a way to enter a trap area.



(a)



(b)

Figure 4.4: Example of the reconstruction of the graph representing the level structure. The black cells represent solid cells, purple cells represent mine traps and the red cells saw traps. For simplicity some redundant edges are omitted. Figure 4.4a displays the initial graph and figure 4.4b displays the altered graph. The green edges are from the set $E_{subsitute}$ and provide the only way to enter the area of the saw trap. The edges inside the saw trap area are maintained to move inside the saw trap area. Note that the mine traps are not considered as challenges on their own, and therefore no edges had to be substituted to enter the mine areas.

4.6.1 Computing the easiest path

The level graph G' is altered into a probability graph G', and a shortest path algorithm can be used to find the path with highest probability of successfully traversing the level. However, the shortest path algorithm needs to be slightly modified. Let u and v be two nodes connected by the edge e_{uv}. When v is reached from u through e_{uv}, the value of v is determined with an additive operator by the standard shortest path algorithms:

$$value(v) = value(u) + weight(e_{uv})$$

But since G' represents a probability graph, a multiplier operator is used. However, this raises a problem. Let the value of the start node be initially zero, which implies that the probability for failing is zero. Then all adjacent nodes of the start node will get the value zero since a multiplication of zero results in zero. Therefore the value of v, when reached from u through e_{uv} is given by:

$$value(v) = 1 - ((1 - value(u)) \cdot weight(e_{uv}))$$

The value of the node represents the probability of failing. Before the value is multiplied with the weight of the edge, the value is transformed into the probability of success. This is done by negating the value: $1 - value(u)$. After the multiplication, the value is again transformed into the probability of failing.

Dijkstra's algorithm is used to solve the shortest path problem. It is a straightforward algorithm. It uses two sets, a closed set and an open set. The closed set contains all nodes for which the distance to the start node is already determined. The open set contains all nodes that can be reached from the closed set. The value of the start node is initialised to zero and all other nodes to infinity. The closed set is initially empty and the open set contains initially the start node. During each iteration the node with the lowest value is removed from the open set, and is added to the closed set. All adjacent nodes to this node, which are not already in the closed set, are added to the open set and their values are updated. This process continues until the closed set contains the end node.

When the shortest path algorithm is terminated, the value of the end node of the level stores the probability of failing, and thus indirectly the probability of successfully traversing the level. This value is used to express the difficulty of a level.

Chapter 5

Results

A user study was conducted to gather data. In total 53 people participated, most of them were men between 20 and 30 years old. The gaming experience of the participants was diverse. Some of them had experience playing the game which is used for this study, or played games on a regular basis. Others had less gaming experience or only played games when they were younger. However, all of the participants were familiar with the concept of platform games. Each participant played 10 randomly generated levels, which is called a *session*. They had the opportunity to skip to the next level in case they got stuck in the current level. This is called a *switch event*. The user study was sent to the participants by mail, so that they could play the game in their own familiar environment. The statistics of the user study are given below in figure 5.1:

number of sessions:	53
number of levels:	530
number of event traces:	6812
number of switch events:	19

Figure 5.1: User study statistics.

5.1 Challenge difficulty model

The gathered event traces are used to construct a model to predict difficulty of individual challenges. This *challenge model* predicts the probability of successfully traversing an individual challenge. A switch event occurs when a player is stuck and wants to skip the current level. Such an event can give important information about challenges, but it is only registered 19 times and some of them were reported as an accident. These events have therefore been ignored in the analysis.

First a histogram is constructed to obtain sample points and measurements, then a mathematical model is fit to these samples points and measurements. Each challenge of an event trace is expressed by the parameters $(t_{timing}, j, t_{cycletime})$. These parameters are used as a key when inserting an event trace into the histogram. Each axis of the histogram relates to one of these parameters. The axis for the timing parameter t_{timing} is divided into 5 bins in the interval $[0, 1.25]$, the axis for the number of jumps parameter j is divided into 4 bins in the interval $[0, 4]$ and the axis for the cycle time of

a challenge $t_{cycletime}$ is divided into 3 bins in the interval $[1.2, 2]$. In total the histogram consists of 60 bins. The event traces yield either a successful or an unsuccessful result. The fitting process requires continuous values rather than binary values. For each bin the probability of success is given by the proportion between the number of successful events and the total number of events in that bin. The histogram obtained from all player sessions is given below in figure 5.2:

$t_{cycletime} \in [1.2, 1.46]$					
	$t_{timing} \in [0, 0.25]$	$t_{timing} \in [0.25, 0.5]$	$t_{timing} \in [0.5, 0.75]$	$t_{timing} \in [0.75, 1]$	$t_{timing} \in [1, 1.25]$
j = 3	-	-	-	0.53	0.40
j = 2	-	0.58	0.52	0.41	0.26
j = 1	0.65	0.58	0.59	0.54	-
j = 0	0.79	-	0.73	-	-
$t_{cycletime} \in [1.46, 1.73]$					
	$t_{timing} \in [0, 0.25]$	$t_{timing} \in [0.25, 0.5]$	$t_{timing} \in [0.5, 0.75]$	$t_{timing} \in [0.75, 1]$	$t_{timing} \in [1, 1.25]$
j = 3	-	-	0.56	0.48	0.37
j = 2	-	0.64	0.62	0.46	0.43
j = 1	0.84	0.58	0.73	0.58	-
j = 0	0.78	0.70	0.65	-	-
$t_{cycletime} \in [1.73, 2]$					
	$t_{timing} \in [0, 0.25]$	$t_{timing} \in [0.25, 0.5]$	$t_{timing} \in [0.5, 0.75]$	$t_{timing} \in [0.75, 1]$	$t_{timing} \in [1, 1.25]$
j = 3	-	-	0.59	0.58	0.38
j = 2	-	0.65	0.68	0.54	0.48
j = 1	0.70	0.73	0.75	0.63	-
j = 0	0.75	0.80	0.69	-	-

Figure 5.2: Histogram containing probability values of successfully traversing challenges, where each challenge is expressed by the parameters t_{timing}, jumps j and $t_{cycletime}$. This histogram was obtained from all player sessions.

This histogram is used to gather samples for the data fitting process. Each bin corresponds to a sample point $\boldsymbol{x}_i = (t_{timing}, j, t_{cycletime})$, and its accompanying probability value is used as the measurement y_i. Some of the bins do not contain any event trace. These bins are therefore not used as samples for the fitting process. Bins containing less than 10 event traces are also not used as they sometimes yield abnormal probability values.

Two candidate models are considered: a linear model and an exponential model. At first glance one would expect that the probability of success relates in an exponential way to the challenge parameters. Initially the probability of success is high, but when the values of the challenge parameters increase, the probability of success decreases more and more rapidly.

However such a trend can not be seen in the histogram. The fitting process confirms this visual intuition: the exponential model does not score much better than the linear model. The exponential model yields an error of 0.128818 while the linear model yields an error of 0.136169.

This can have multiple reasons. The performance of the players is very noisy, each axis of the histogram is only divided into a few bins and different trap types are combined into one model. Maybe each individual trap type yields a exponential pattern and this pattern is lost when adding the results of different traps types together. Constructing different models for each trap type is

not possible, as this would lead to a major loss of generality. Multiple factors can possibly have influenced the results. Nevertheless, we chose a linear model for the challenge difficulty model.

5.2 Training set

A part of the data is used to obtain a challenge model. This challenge model is used by the process that predicts difficulty of levels.

The data is divided into two sets: a training and a verification set. The training set is used to obtain the challenge model, and the verification set is used to verify the predicted difficulty. The size of these sets influences the results: the quality of the challenge model can be too low when the training set is too small and on the other hand the verification results can be poor when the verification set is too small.

Assume that the best model can be obtained from all available data. Then it is possible to compute the minimum required size of the training set before the quality of the challenge model becomes too low. The quality of the model is quantified as the square error ϵ by the fitting process, which is given by:

$$\epsilon = \sum_{i=0}^{n} (f(x_i) - y_i)^2$$

where $f(x_i)$ is the value computed by the model and y_i is the measured value. Assume that the error is equally distributed over the model and the difference d for each sample point is equal, than the error can be given by:

$$\epsilon = \sum_{i=0}^{n} d^2 = nd^2$$

where n is the number of samples. To compute whether the quality of a model is high enough, first a model $M_{reference}$ is fitted to all available data. The quality of another model M_i is high enough when the average difference between each sample of $M_{reference}$ and M_i is less than a value d. The sample points must be taken such that they are equally distributed over the model. A sample point is taken at each bin of the previously explained histogram. This results in a set of sample points which is equally distributed. The histogram consists of 60 bins, So the number of samples n is 60. When d is chosen to be 0.02, then the error between $M_{reference}$ and M_i must be smaller than 0.024 in order for M_i to be a high quality model. It turns out that the training set must consist of 25 play sessions at least.

5.3 Verification

The obtained challenge model is used by the process that predicts the difficulty of each level. This is called the *level model*. The level model is based on a shortest path in a probability graph. Each edge of this graph has a weight assigned by the challenge model.

The data from the verification set is used to verify the level model. A difficulty value is computed for each level from the verification set. Each level already has a value holding the measured number of deaths.

Theoretically the difficulty value corresponds with the probability of successfully traversing the level. However it is not possible to verify this value. This possibility value is near to zero for many levels. It requires a large amount of data to obtain a percentage of players who successfully traversed a level of a given difficulty.

However, it is possible to compute the average number of deaths. Although some difficult levels yield a high average number of deaths, there is still a chance that some players lose only a few lives. On average the players will lose some lives, but that does not imply that each individual player loses exactly that number of lives.

So verification based on averages is not suitable for predicting the exact number of deaths for each individual player, only for large groups. A histogram is constructed which uses the estimated difficulty value as a key and stores the average number of deaths per bin. Such a histogram is constructed for the training set and also for the verification set. The difference between those histograms provides information about the consistency and predicting capabilities of the difficulty estimation. However, little difference between the two histograms does not imply that the level model correctly predicts difficulty. A mean model is a model that contains only noise and no relationship between the values on the axis. The difference between the histograms will be small when both histograms represent a mean model.

Therefore, a relationship between the predicted difficulty values and the measured numbers of deaths should also be verified. This can be quantified by performing linear regression. Although it is not the aim to predict the average number of deaths with this linear model, it provides information about the strength of the relationship between difficulty and the number of measured deaths. In the absence of any relationship between difficulty and the number of deaths, linear regression results in a model with a horizontal slope. This implies that the level model does not correctly predict difficulty. When the slope significantly differs from zero, it implies there is a relationship.

To verify the existence of a relationship between difficulty and the number of deaths based on linear regression, the following two hypotheses are constructed:

- H_0: The slope of the regression line is equal to zero.
- H_1: The slope of the regression line is not equal to zero.

To verify the null hypothesis, the t-score of the regression line is calculated. This t-score, along with the degrees of freedom of the data set, corresponds to a p-value according to a table[1]. A two-tailed significance test of this p-value is performed against the significance level to accept or reject the null hypothesis. A significance level of 0.05 is used.

The t-score is obtained by dividing the slope b by the standard error of the slope $SE(b)$ of the linear model. This t-score reflects the strength of the slope with respect to the distribution and range of the points along the x and y-axis. It can be seen as a normalization step of the slope. The standard error of the slope $SE(b)$ is given by:

$$SE(b) = \frac{S_{res}}{\sqrt{\sum_i (x_i - \overline{x})^2}}$$

where S_{res} is given by:

$$S_{res} = \sqrt{\frac{\sum_i (y_i - f(x_i))^2}{n-2}}$$

where $\overline{x}$ is the mean of the x-values and $f(x_i)$ gives the y-value according to the linear model for a given x_i.

To ensure that the partition of the training and verification set does not accidentally yield a good result, 25 random partitions are performed and the results are averaged. The result of linear regression of one of the partitions is displayed in figure 5.3.

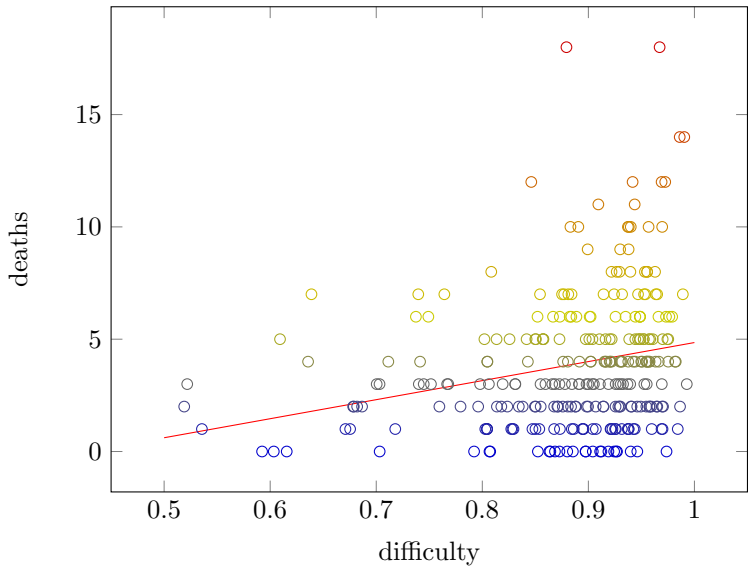


Figure 5.3: Scatter plot of the computed difficulty values and the accompanying measured number of deaths. Also the line of regression is displayed.

Although a clear increment of the line of regression can be noticed, the data points yield a noisy pattern rather than an increasing pattern. This perception is caused by the discrete scaling of the number of deaths. Many high numbers of deaths are measured for high difficulty values, but also some low number of deaths are measured. As a result a dot is drawn at each death value. A histogram gives a better impression of the increasing numbers of deaths as is displayed in figure 5.4.

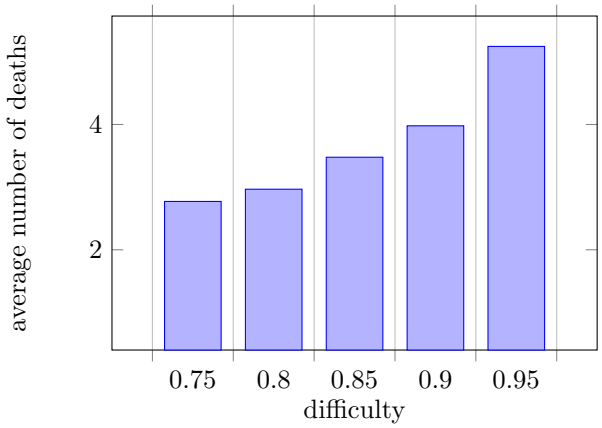


Figure 5.4: Histogram of average number of deaths.

After performing 25 random partitions it turns out that the average accuracy is 82%, and the average t-score is 4.4. The verification set contains 280 levels, so the degrees of freedom are 278. This t-score and degrees of freedom correspond to a p-value of 0.000. Since a two-tailed test is conducted, the p-value must be less than 0.025 in order to reject H_0. The p-value is smaller than this value, and therefore the null hypothesis can be rejected and we can conclude that increasing difficulty values correspond significantly with increasing numbers of deaths.

5.4 Different skill levels

We intuitively assume that players improve their skills during the play session. However, the results show no improvement in skills. Figure 5.5 displays the average score per level. Here, the score is determined as the proportion between successfully traversing a challenge and the total number of encountered challenges. The participants only played for 15 minutes, this is probably too short to improve skills.

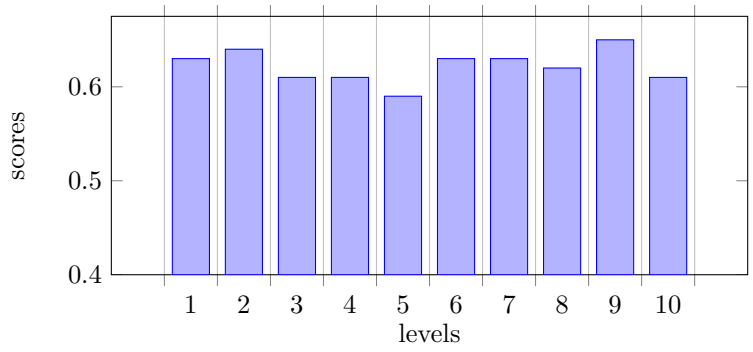


Figure 5.5: Average scores for different levels.

We also intuitively assume that some players are more skilled than others. This is actually

confirmed by the results. The list of players is sorted on their score and divided into three groups, representing the players of the skill levels $GOOD$, $AVERAGE$ and BAD. There is a clear difference between the scores of the different groups, as can be seen in figure 5.6.

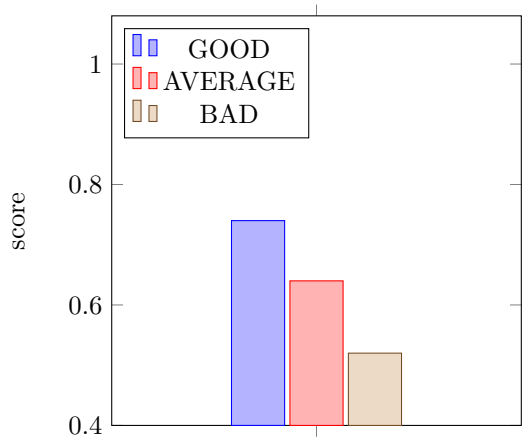


Figure 5.6: Average scores for different groups of skill level.

Such a distribution of scores can also occur when all players are of the same skill level. In that case it is caused by a difference in difficulty of the played levels. Some players played more difficult levels than others, and these players are therefore labeled as BAD regardless of their actual skill level. But this is not the case according to the distribution of levels over the players with different labels. This can be seen in figure 5.7.

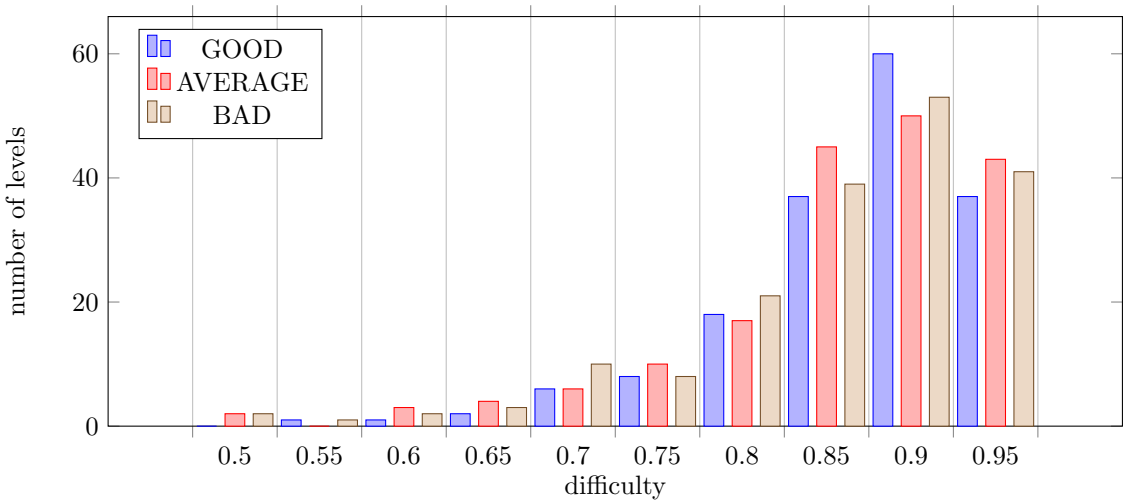


Figure 5.7: Level distribution

According to the differences in scores and the absence of differences in the level distribution, we

can safely assume that some players are more skilled than others.

During the same experiment, each partition was also split into three groups (*GOOD*, *AVERAGE* and *BAD*) based on the scores of the players. A clear difference can be noticed. The average histograms per skill group over 25 runs are displayed in figure 5.8. The accuracy, t-scores and p-values are also computed per skill level group, the results are displayed in 5.9.

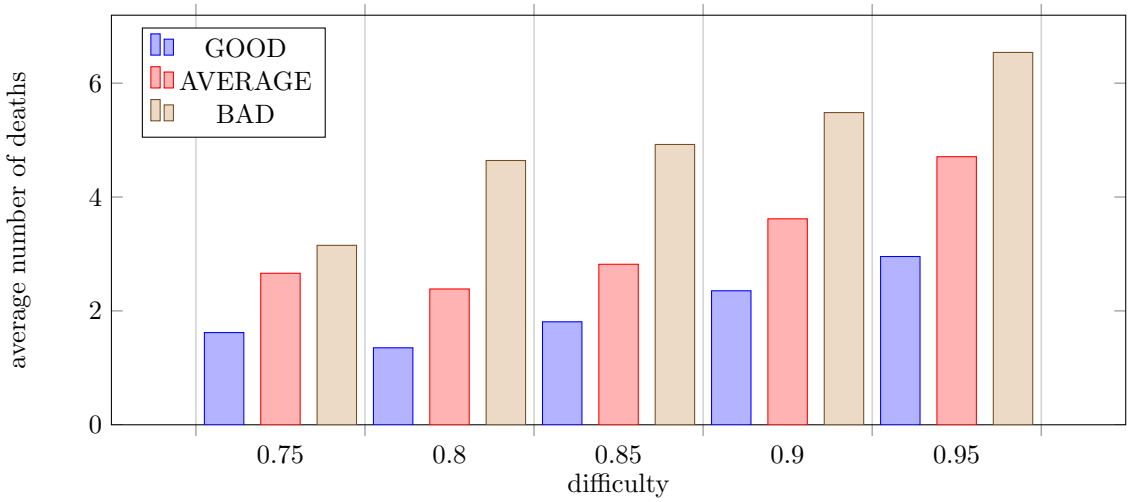


Figure 5.8: Histogram of average number of deaths per skill group.

	ALL	GOOD	AVERAGE	BAD
t-score:	4.4	2.7	2.9	3.4
p-value:	0.0000	0.0041	0.0023	0.0005
accuracy:	82%	64%	70%	77%

Figure 5.9: Average accuracy values and t-scores and p-values over 25 runs.

The p-values for each skill level group are lower than the threshold of 0.025 used in the significance test, therefore the null hypothesis can be rejected and we can conclude that there is a significant relationship between difficulty and the numbers of death for each skill level group. It can be seen that the results of the different skill level groups are slightly worse than the results of the overall group. This is most likely explained by a shortage of data per group. Each bin contains approximately only 15 data samples, which is quite low given the high variance in the performances of the players.

When more data would have been available, one could expect that the results per skill group are better than the results of the overall group. This would be caused by less noise due to less differences in the performances of the players.

But one could also argue that performances of the GOOD group would be more noisy than the performances of the overall group. The good players lose a life less often, even in the more difficulty challenges. On the other hand, the good players do lose sometimes a life due to impatience and inattentiveness as is observed from the replays. This results a noisy performance of the *GOOD*

group. Another reason for the lower accuracy value of the *GOOD* group can be found in the low average number of deaths per level, only one to three. An outlier has a relative high impact on the accuracy.

On the other hand, the *BAD* group makes a clear difference between easy and hard challenges as they almost always lose a life in the more difficult challenges. Also the average number of deaths is quite high, and it differs a lot per difficulty bin. This makes it less vulnerable for outliers. So a high accuracy value could be expected when more data would have been available.

5.5 In-game results

Besides quantitative results we also present some visual results. The presented level generator only generates the level geometry and adds traps to the level. To enrich the levels we added some decoration like plants and rocks. This can be seen in the screenshots below. One of the research goals was that each level had to yield a feasible path. Also after adding traps to a level a feasible path has to be maintained. We did not receive any messages from the participants that a level was impossible to complete. Only that some levels were very difficult to complete. We also never encountered an impossible level. Figure 5.10 displays a part of a level that contains multiple traps, but still guarantees a feasible path.

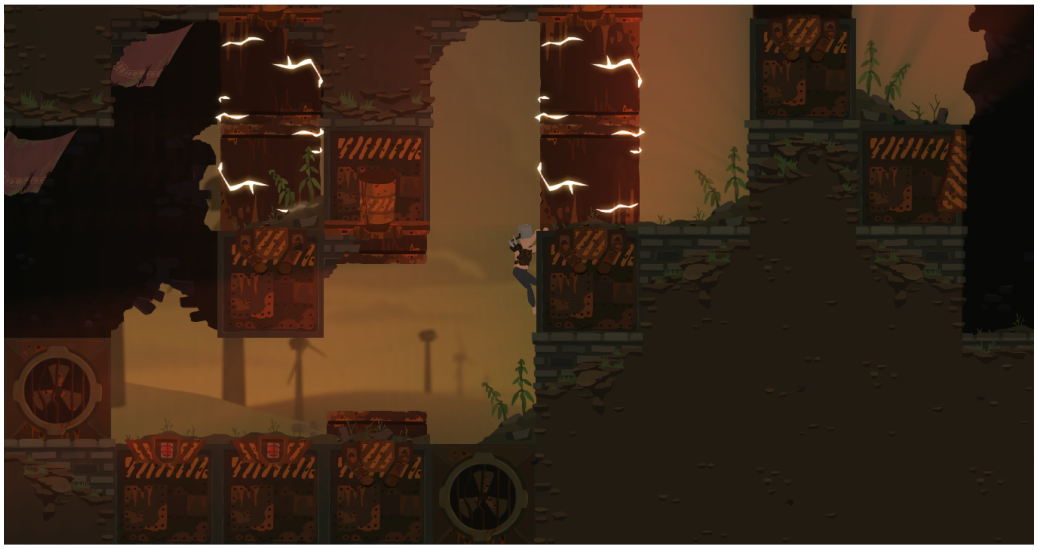


Figure 5.10: Example of a level which contains multiple traps and still guarantees a feasible path.

Another goal was that the levels had to yield bi-directional non-monotone paths. An example of a level yielding vertical gameplay is given in figure 5.11.

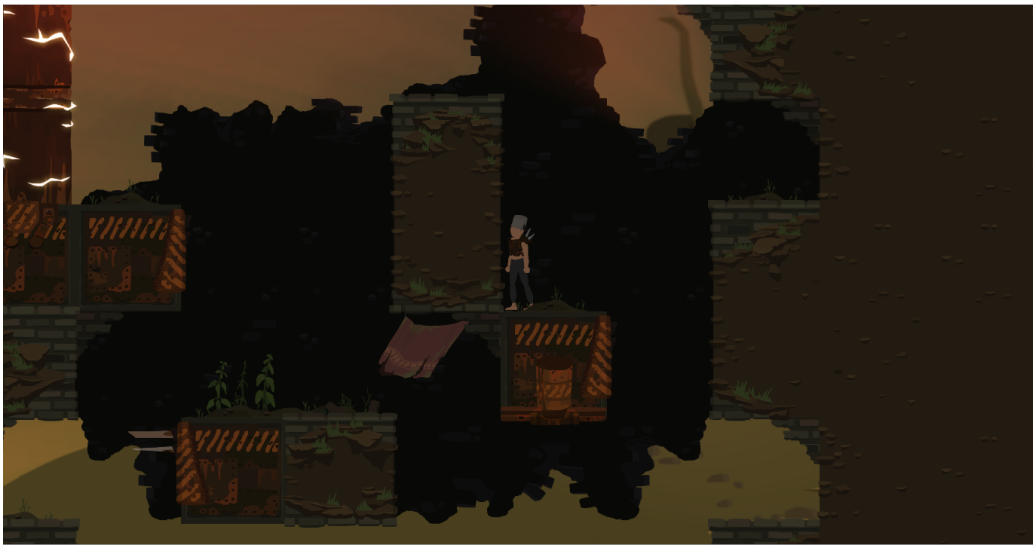


Figure 5.11: Example of a level yielding vertical gameplay.

A graph of the level is used to place traps in the level and this graph is also used to estimate difficulty of levels. Figure 5.12 displays such a graph and also the routes deduced from that graph.

Figure 5.12: Example of a graph that represents the level. The yellow edges represent routes through the level and the white edges represent arbitrary edges.

To estimate the difficulty of a level two models are used: a challenge model and a level model. The challenge model estimates the difficulty of individual challenges. Figure 5.13 displays a chal-

lenge that is estimated as easy and figure 5.14 displays challenges that are estimated as difficult. This seems to be correct according to our intuition.

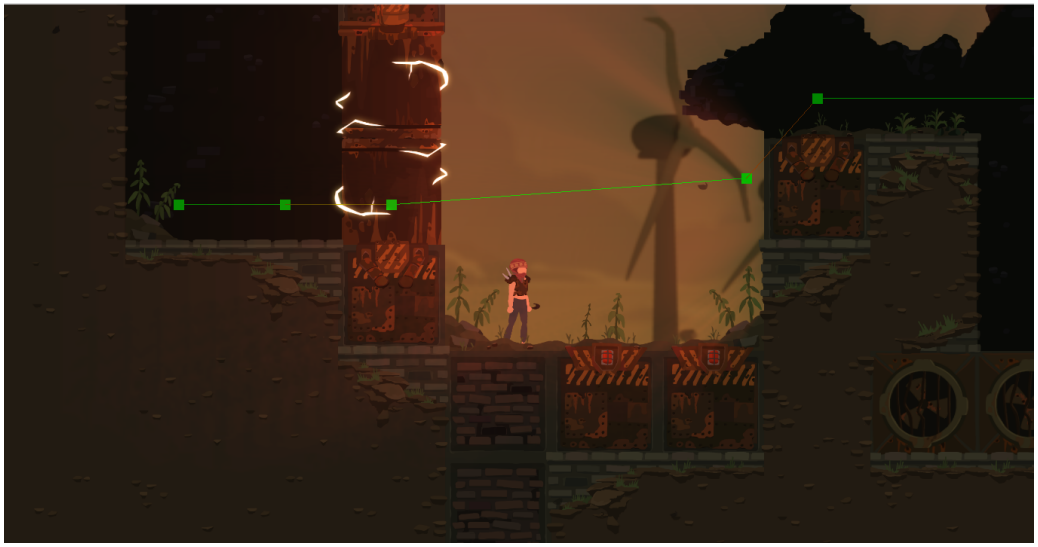


Figure 5.13: The squeeze trap at the left side of the picture is an example of a challenge that is estimated as easy, as can be seen by the green edge.

Figure 5.14: Example of two challenges that are estimated as difficult due to the presence of mines, as can be seen by the red edges.

The level model is used to find the easiest path through the level. This path avoids challenges

if possible and finds the easiest way to overcome a challenge. Some challenges can be overcome from multiple starting positions. Some starting positions yield a more difficult challenge due to the presence of mines. The level model finds a path that avoids mines if possible. Figure 5.15 displays a path that takes a detour in order to avoid a challenge and figure 5.16 displays a path that avoids mines in a sequence with another trap.

Figure 5.15: Example of the easiest path that takes a detour to avoid a challenge.

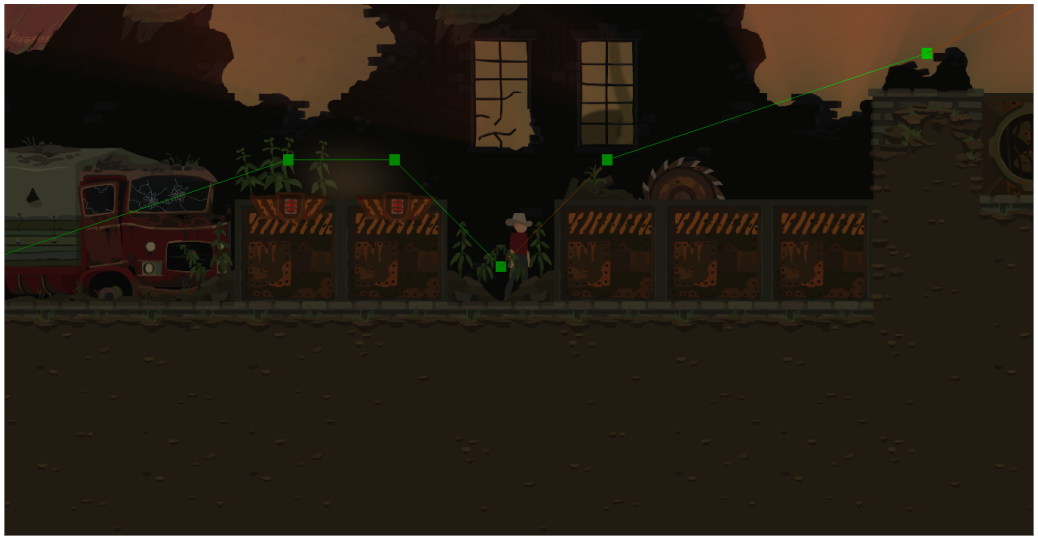


Figure 5.16: Example of the easiest path that avoids mines in a sequence with another trap. After the two mines it finds a safe spot before it continues with the saw trap.

Chapter 6

Discussion and Future work

We focused on generating levels and predicting the difficulty of the generated levels. The levels yield non-monotone gameplay and bi-directional paths.

In the previous work section we stated some desired properties of generated content. We will now discus how well our level generator meets these properties, or how we can extend our level generator to meet these properties.

- Speed: the level generator has limited time available to generate levels since the levels must be generated during the play session. The levels used for the user study are almost instantly generated. When a full size level is considered, the generation process requires at most a few seconds. This is fast enough for practical use.
- Reliability: when the generator does not generate a feasible level, it is considered as a catastrophic failure. We performed an experiment to show that for each feasible exit configuration a solution exists and a solution can be found by our algorithm.
- Controllability: a certain level of control over the generated levels is required in order to use the generator in practice. Our method provides control over the global path, but does not provide any control over an individual room solution. We do not consider this as a problem, since it is more important to have control over the global level layout. Heuristics can be used during the process of generating an individual room to obtain control over individual rooms. Also the set of building pieces can be extended to obtain more control. A minimum set of building pieces is required to guarantee a solution. Adding other building pieces to this set gives some control over individual rooms. However, when extending the set of building pieces one must ensure that a high probability of finding quickly a solution is maintained.
- Diversity: we did not measure how diverse the generated levels are, but in our opinion the results yield a considerable extent of diversity. The diversity can be increased by extending the set of building pieces and by applying filters. These filters can simply turn cells on and off. To quantitatively measure the diversity a distance function like earth movers distance can be used or a user study can be performed.
- Believability: we did not focus on creating levels which can not be distinguished from human created levels. Besides level geometry the design of a high quality level also includes a clever

distribution of resources and decoration. The distribution of resources can be used to guide the player through level and to provide interesting gameplay, like a reward after a difficult challenge. Also the player must sometimes have multiple options to overcome a challenge, like the possibility to backstab an enemy. Pieces of decoration must be placed such that they make sense in a context. For example a room that is used as a garage should contain only decorative objects that fit in that context. Our method uses possible routes through the level to place traps. These routes provide a powerful tool to distribute gameplay elements over the level and to create high quality levels. The quality can be measured by a user study.

Our level generator was specifically designed to create 2D levels in a grid. However, our method is not limited to 2D and a grid structure. It can easily be extended to 3D by using 3D building pieces consisting of voxels. The computation time will increase as the search space is larger, since the rooms are larger and there are more building pieces to fit.

Our method can be extended to create non grid-based levels by using smoothing filters or by using polygons as building pieces. When applying a smoothing filter one must take into account that not every edge can be smoothed, as some edges are used as hang points or take off points for jumps. Rather than cell-based building pieces polygon-based building pieces can be used. These polygons can be used to represent the possible jumps and the required solid spaces. The same fitting strategy can be used, only during each placement must be verified whether the polygon representing a jump does not intersect with polygons representing solid space. These intersection tests will have a considerable impact on the computation time.

Our method is created for 2D platformers, but it can also be used in some 3D games. For example it can be used to create caves in Minecraft or to create dungeons of predefined rooms in Diablo. In the latter case, each room is represented as a building piece with multiple connection points. However, our method will not be suitable for making 3D landscapes.

We also focused on predicting difficulty of the generated levels. The strength of our method is that it is based on finding the easiest path, making it capable of avoiding difficult challenges if possible. This method has also some limitations. Only local traps are considered, this method is not suitable for coping with global challenges like enemies who chase the player once he is spotted. Global challenges are not obliged to stay inside a certain area, but can go anywhere in the level. Therefore finding the easiest path is no longer possible, since global challenges can be potentially anywhere. The easiest path could still avoid encounters with global challenges when these have a static initial position and only exhibit global behaviour once there are triggered. This method is also not capable of coping with puzzle based challenges or with boss fights, unless a specific model is built for the puzzles and bosses. This model could also be based on simple features, like is done for the challenges.

There are also some limitations with respect to the obtained difficulty values. As difficulty is based on challenges encountered along a path, a consequence is that longer paths become more difficult. This makes it hard to compare difficulty values between different path lengths. Which level is more difficult, a level consisting of many relatively easy challenges or a short level with only a few difficult challenges? The players will sometimes lose a life due to one of the easy challenges. At the end, the players could have lost an equal number lives in both levels. But are both levels equally difficult? Or is the level with harder challenges more difficult than the longer level? There is no clear answer to this question, as it is very subjective.

We were able to show that a significant relationship exists between the predicted difficulty values

and the measured numbers of deaths per level. We did this for groups of mixed skill levels and also for groups of the same skill level. The accuracy for the mixed group is quite high: around 80%. The accuracy for the different skill levels is considerably lower. This is likely to be caused by a shortage of data and the high amount of noise. However, increasing the number of play sessions yields another problem: the limited variety of different levels that can be generated. Currently around 500 levels are used during the user study. It is unlikely that two exactly the same levels are generated. Each level contains approximately a hundred nodes to place traps on, so enough possibilities are available. But some levels could be considerably similar. When the training and the verification sets contain many similar levels, the outcome of the results will become less relevant. It is an obvious conclusion that two histograms of two similar sets are also similar. Instead a fixed set of levels in a random order could be played by all participants. During the partitioning one must ensure that similar levels are contained by the same partition.

The participants played only 10 levels and no improvement was noticed during their play sessions. It is interesting to research the learning curve of the participants. In order to do that, the participants should play more levels. However, it is quite difficult to find a group that is willing to spend so much time on an experiment.

Instead of increasing the amount of data, reducing the amount of noise can also lead to better results. Some sources of noise experienced during the user study are:

- Difference in skill level: the skill level of the players is quite diverse, as some players play games very often while others do not. This results in a inconsistent measurement of numbers of death. The less skilled players lost a lot more lives, while some of the best performing players almost did not lose a life at all. This results in a high variance in the number of deaths for the more difficult levels: some very skilled players lost only one life while some of the less skilled players lost more than ten lives.

- Inconsistent performances: there is a high variance in the performance of individual players. Although a player is labeled as a bad performing player, he sometimes manages to complete a difficult level without losing many lives. He just had a bit of luck in that level. On the other hand, some of the more skilled players lost lives due to impatience and inattentiveness, while they almost did not lose any life in the more difficult challenges.

- Easiest path: the difficulty computation is based on the easiest path through the level. The levels yield some possibilities for alternative routes. This implies that the predicted difficulty is sometimes based on another route than the player actually took. But the levels were designed such that there are not much possibilities for alternative routes, and only very local. It is restricted to the choice of taking a challenge or avoiding that one.

- Side effects: sometimes a jump required for a challenge results in a death event of another challenge. Although this happens now and then, it is not expected to have a large impact on the results since more than 6000 event traces are considered.

- Differences between trap types: the levels contain three different type of traps, the saw trap, squeeze trap and the wall spikes. It could be that a difference in difficulty exists between these trap types. Some of the players mentioned that the wall spikes were especially hard to overcome. These wall spikes require at least two jumps and quite some timing, and are therefore automatically rated as quite difficult. But it could also be that this trap is more difficulty than others. However, it is no option to construct a model for each trap as that would be a major loss in generality.

The proposed method yields some visually nice results. It seems to correctly judge when a challenge is quite difficult and is capable of avoiding difficult challenges if possible. Some changes could be made to the experiment setup in order to achieve better quantitative results:

- User provided difficulty rating: there is a high variance in the measured numbers of deaths per level, which is an important source of noise. A user provided difficulty is most likely more consistent. When a player loses a lot of lives due to inattentiveness, he still can rate the level as easy. On the opposite a player can rate a level as difficult when he accidentally overcame some difficult challenges. Since the rating is very subjective and differs a lot between players, some normalization would be required. However, such approach can not be used when the aim is to automatically gather data from a released game. This can be used to make a better difficulty model for the next game.
- Larger difficulty spreading: most of the generated levels yield a difficulty value around the 0.8 a 0.9, but only a few levels yield a difficulty value lower than 0.7. For the analysis it would have been better to have more easier levels. Unfortunately this wasn't possible since a difficulty measurement would have been required, which is exactly the aim of this thesis. A temporarily heuristic should have been used, or multiple user studies should have been conducted.
- Strictly linear levels: the choice for players to avoid sometimes a challenge can cause noise. This noise can be omitted when strictly linear level would have been used. The players are forced to take a certain set of challenges, without any possibility of side effects. Although some higher accuracy would have been achieved, the method would be less suitable to be used in practice when levels are not linear and side effects can occur.

The proposed method can also be improved to obtain more accurate difficulty values:

- Higher grid resolution: the traps are placed in a grid based manner. Some parts of the cells can be traversed without the risk of getting hit by the trap. The saw trap is only half a cell in height and the wall spikes only half a cell in width. A higher resolution grid yields a better trap representation, and provides a more precise indication when the player is at risk.
- Jump difficulty model: the focus of this thesis was to determine difficulty of moving challenges within a static area. Static challenges like jumping over a gap were not considered, but a model for determining difficulty of jumps can improve the difficulty computation. Failing some jumps can cause a death event when the player falls in a trap. It would also be interesting to research the effect of a difficult challenge which also requires a difficult jump.
- Directional movements: the current model considers only bi-directional movements, but some players overcome a challenges by letting them falling down. This is a directional movement, since the player can not jump so high to reach the starting position again. Such directional movements can improve the difficulty computation.
- Other parameters: currently only three parameters are used to represent challenges, but more parameters could be used. For example the size of a challenge, the strength of a challenge when the player can defeat it, and the damage done by the challenge when a hit is not an instant kill. The amount of extra space to overcome a challenge would be an interesting parameter. This sounds rather vague, but overcoming a challenge in a narrow space could be more difficult than overcoming a challenge with more space available.

The proposed method was only tested in a game with three different kind of traps. Both the training set and verification set contained levels with these traps. The method would be really put to the test when the training set contains different traps than the verification set. The accuracy values will probably be very low, but that is not a problem. It is more important that increasing difficulty values still relate to increasing numbers of deaths. The method should also be tested on other games like Spelunky and Super Mario.

Chapter 7

Conclusion

We presented a method to generate levels that yield non-monotone gameplay and bi-directional paths. The generated levels are populated with traps and the levels always guarantee a feasible path. So the goals with respect to the generating part are achieved. However, the generated levels do not yet yield the quality of human authored levels. To reach that level of quality the distribution of gameplay elements must be included. The graph that represents the level can be a powerful tool to reach this goal.

We also presented a method to estimate the difficulty of the generated levels. We presented a model to estimate difficulty for individual challenges and a model to estimate difficulty of entire levels. The aim was not to provide a metric for difficulty, but an ordering of the levels with respect to difficulty. We have shown that increasing difficulty values correspond significantly with increasing numbers of measured deaths. This applies to a mixed group of skill levels as well as to groups of the same skill level. Unfortunately we were not able to observe a learning curve. Nevertheless, the goals with respect to the difficulty estimation part can also be considered as achieved.

The results can be improved by reducing the amount of noise in our measurements included in the user study, and by extending the difficulty model with a difficulty estimation for individual jumps, more challenge parameters and a higher resolution trap representation.

Bibliography

[1] t-distribution table. http://stattrek.com/online-calculator/t-distribution.aspx.

[2] F. Benoit-Koch. Procedural Level Generation for a 2D Platformer. Available on http://fbksoft.com/procedural-level-generation-for-a-2d-platformer/, 2014.

[3] M. Booth. The ai systems of left 4 dead. Available on http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf, 2009.

[4] S. Lefebvre C. Ma, N. Vining and A. Sheffer. Game level layout from design specification. In *Computer Graphics Forum*, Volume 33, Issue 2, 2014.

[5] J. Togelius C. Pedersen and G. N. Yannakakis. Modeling player experience in super mario bros. In *Computational Intelligence and Games*, 132-139, 2009.

[6] N. Cooper. Machine learning for auto-dynamic difficulty in a 2-d space shooter. Available on cs229.stanford.edu/proj2005/Cooper-DynamicDifficulty.pdf, 2005.

[7] M. Csikszentmihalyi. Flow: The psychology of optimal experience. Available on http://www.amazon.com/Flow-Psychology-Experience-Perennial-Classics/dp/0061339202, 1991.

[8] S. Dahlskog and J. Togelius. A multi-level level generator. In *Computational Intelligence and Games (CIG)*, 1-8, 2014.

[9] R. Vasconcelos de Medeiros and T. Vasconcelos de Medeiros. Procedural level balancing in runner games. In *Computer Games and Digital Entertainment*, 109-114, 2014.

[10] J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Article No. 1, 2010.

[11] F. Mourato M. P. dos Santos and F. Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, Article No. 8, 2011.

[12] J. Fisher. How to Make Insane, Procedural Platformer Levels. Available on http://www.gamasutra.com/view/feature/170049/how_to_make_insane_procedural_.php, 2012.

[13] M. Kapadia G. Berseth, M. Haworth and P. Faloutsos. Characterizing and optimizing game level difficulty. In *Proceedings of the Seventh International Conference on Motion in Games*, 153-160, 2014.

[14] V. Hom and J. Mark. Automatic design of balanced board games. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, 25-30, 2007.

[15] I. Horswill and L. Foged. Fast procedural level population with playability constraints. In *Conference on Artificial Intelligence and Interactive Digital Entertainment*, 20-25, 2012.

[16] G. Levieux M. Aponte and S. Natkin. Measuring the level of difficulty in single player video games. In *Entertainment Computing*, 205-213, 2011.

[17] G. Smith M. Jennings-Teats and N. Wardrip-Fruin. Polymorph: A model for dynamic level generation. In *Conference on Artificial Intelligence and Interactive Digital Entertainment*, 138-143, 2010.

[18] S. Bakkes M. Traichioiu and D. Roijers. Grammar-based procedural content generation from designer-provided difficulty curves. In *Foundations of Digital Games Conference*, 2015.

[19] F. Mourato and M. dos Santos. Measuring difficulty in platform videogames. In *Conferncia Nacional em Interaco Pessoa-Mquina*, 173-180, 2010.

[20] V. Chapman R. Hunicke. Ai for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 429-433, 2005.

[21] J. Togelius E. Kastbjerg D. Schedl and G. N. Yannakakis. What is procedural content generation? mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, Article No. 3, 2011.

[22] W. Baghdadi F. S. Eddin R. Al-Omari 1 Z. Alhalawani M. Shaker and N. Shaker. A procedural method for automatic generation of spelunky levels. In *Applications of Evolutionary Computation*, 305-317, 2015.

[23] S. Snodgrass and S. Ontanon. Experiments in map generation using markov chains. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*, 2014.

[24] M. Kerssemakers J. Tuxen J. Togelius and G. N. Yannakakis. A procedural procedural level generator generator. In *Computational Intelligence and Games (CIG)*, 335-341, 2012.

[25] N. Shaker J. Togelius and M.J. Nelson. Procedural content generation in games. Available on `http://pcgbook.com/`, 2015.

[26] S. Dahlskog J. Togelius. Patterns and procedural content generation revisiting mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, Article No. 1, 2012.

[27] G. Smith M. Treanor J. Whitehead and M. Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 175-182, 2009.

[28] D. Yu and A. Hull. Spelunky. Independent, 2009.