Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria


# An AI assisted framework for the design of 2D platformers

Relatore: Prof. Pier Luca LANZI

Correlatore: Ing. Daniele LOIACONO

Tesi di Laurea di:

Antonio Umberto ARAMINI Matr. 836202

Anno Accademico 2015–2016

*Ai miei nonni*

# Ringraziamenti

Ringrazio innanzitutto il Prof. Pier Luca Lanzi, per il tempo dedicatomi e per avermi permesso di svolgere un lavoro di tesi che coniugasse gli studi e la mia passione per i videogiochi. Ringrazio l'Ing. Daniele Loiacono, per i preziosi suggerimenti e il supporto fornitomi.
Un sincero grazie va ai miei genitori, per avermi sempre incoraggiato a puntare al massimo e non avermi mai fatto mancare nulla, anche nelle situazioni più difficili.
Un grazie speciale a Clara, per avermi supportato e sopportato durante lo svolgimento della tesi.
Ringrazio poi tutti i compagni di corso con cui ho avuto la fortuna di condividere questi anni. In particolare, ringrazio Federico, Mauro e Davide per i bei momenti condivisi e i progetti svolti in gruppo; un grazie anche agli amici di Como: Jacopo, Marco, Nadir, Gabriele, Cristina e Alberto.
Ringrazio i miei amici di Stupidi Pixel, Simone, Fabio e Alessio, per l'avventura che stiamo vivendo assieme.
Un grosso grazie va a tutti coloro che hanno dedicato parte del loro tempo per partecipare ai test sui salti.
Infine, ringrazio i miei nonni per tutto l'amore che mi hanno donato e per avermi reso la persona che sono.

*Antonio Umberto Aramini*

# Sommario

Il design di livelli di videogiochi è un compito complesso e di importanza critica. I livelli devono suscitare divertimento e sfida, mentre la frustrazione va evitata a tutti i costi. Strumenti di intelligenza artificiale possono rivelarsi efficaci per supportare il design di livelli godibili e di qualità.
In questa tesi, viene descritto come l'intelligenza artificiale è stata usata per affrontare il problema di supportare il design di livelli di videogiochi, e si offre una panoramica delle metriche utilizzate per valutare i contenuti di gioco. Presentiamo un framework per supportare i designer nella creazione di livelli per platformer 2D. Il nostro framework mette a disposizione dei designer un insieme di strumenti (i) per creare livelli per platformer 2D, (ii) per stimare la difficoltà e la probabilità di successo delle azioni di salto, e (iii) un insieme di metriche per valutare i livelli in termini di difficoltà e probabilità di completamento.
Infine, presentiamo i risultati di un insieme di esperimenti che abbiamo svolto con giocatori umani per validare le metriche incluse nel nostro framework.

# Abstract

The design of video game levels is a complex and critical task. Levels have to elicit fun and challenge while avoiding frustration at all costs. Artificial intelligence tools can prove effective in assisting the design of enjoyable and quality levels.
In this thesis, we discuss how artificial intelligence has been used to approach the problem of assisting level design in video games and overview the metrics employed to evaluate game content. We present a framework to assist designers in the creation of levels for 2D platformers. Our framework provides designers with a toolbox (i) to create 2D platformer levels, (ii) to estimate the difficulty and probability of success of jump actions, and (iii) a set of metrics to evaluate levels in terms of difficulty and probability of completion.
At the end, we present the results of a set of experiments we carried out with human players to validate the metrics included in our framework.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the recent years, artificial intelligence tools have started to play a key role in assisting game designers by providing immediate feedback on the created content and foster the creativity of human authors by suggesting novel solutions. One area of video game development in which artificial intelligence is focusing is *Level Design*, that is the creation of the environments (i.e. levels) that players interact with. Level Design is a critical task for the genre of *2D platformers*, in which the character controlled by the player can move and jump in order to reach a goal location within a level while avoiding obstacles; in this category of games, *platforms* (i.e. blocks on which the character can stand) are the fundamental elements that make it possible for the character to travel from one point to another in a level. Platformer levels have to be designed to achieve the fundamental goal of eliciting fun and challenge; by contrast, frustration must be avoided at all costs and thus, designers put extreme attention to successfully gauge the difficulty of levels. Research in this area of game design has been typically focused on the generation of levels that affect players in terms of emotions [24, 18, 23]. In this context, design metrics are extracted both from structural features of levels (i.e. characteristics of levels based on the placement of game elements) and gameplay characteristics of levels (i.e. data describing players' skill and playing style obtained with sessions of playtesting) to build models for the generation of personalized levels. For instance, Smith et al. proposed a mixed-initiative tool [27, 28] that allows designers to place constraints on the generation of levels, both in terms of the positioning of game elements and on the rhythm at which players must execute actions (e.g. jumping); the tool assists designers by filling in incomplete levels and informing if there is no solution that satisfies the constraints, thus guaranteeing the playability of levels.

## 1.1 Thesis objectives

In this thesis, we propose an artificial intelligence based approach for the design of 2D platformers and introduce a framework to assist level designers in the creation of quality game content. Our approach employs a directed graph representation for game levels and associates each directed edge to the jump action that allows the connection between two platforms. A difficulty value is assigned to each directed edge, indicating how difficult the corresponding jump is; this is achieved by studying the possible jump trajectories starting from an optimal position. Similarly, the probability of success of jumps is associated to the corresponding edge; this probability is obtained by considering a sample of trajectories starting from random points generated, using a noise function, around the optimal position. We apply different noise functions, based on the average reaction time and skill of players, to generate the distribution of random start points. The difficulty and probability metrics associated to directed edges are a form of gameplay features obtained without actually performing sessions of playtesting and provide key feedback on the designed content. Furthermore, the framework provides a tool to design platformers with a set of structural features that can be adjusted by designers to modify the physics parameters (e.g. the gravity or the jump vertical takeoff speed) and other generic game mechanics (e.g. whether the character can run or not); in addition, the platformer design tool can be used as a quick playtesting tool. Our framework is developed as a modular extension of the popular Unity game engine, taking advantage of its physics engine and several other features. We present and discuss the results of a preliminary validation of our approach for the estimation of the probability of success of jumps; the validation involved a set of trials with human players whom were asked to perform several jumps across platforms. The noise functions have been evaluated both in terms of estimates accuracy and on their ability to identify the harder jump in a pair of jumps. At the end, we present an approach for the evaluation of levels in terms of a variety of difficulty metrics and probability of completing the level by extending the approach for single jumps.

## 1.2 Thesis structure

The thesis is structured as follows.

In Chapter 2, we discuss the state of the art of 2D platformers and the related tools used for design tasks. We overview the history of platformers and survey AI-assisted design in videogames and especially in 2D platformers, examining level generation techniques and the tools used.

In Chapter 3, we describe in depth how structural and gameplay features have been used to evaluate created content and to guide the generation of levels in 2D platformers.

In Chapter 4, we introduce our approach and present the framework to assist designers in the creation of levels. Then, we overview the structural features provided by the framework and examine how jump trajectories are used to estimate the difficulty of jumps.

In Chapter 5, we describe how noise functions are employed to estimate the probability of success of single jumps and present the data collected to validate this method. Then, we propose an approach and a set of metrics to evaluate levels in terms of difficulty and probability of completion.

In Chapter 6, we give an overview of the tools provided by our framework within the Unity game engine.

In Chapter 7, we give a summary of this work and describe the plans to extend it.

# Chapter 2

# State of the Art

In this chapter, we discuss the state of the art of 2D platformers and the related tools used for design tasks. At first, we overview the history of platformers; the section is by no means a complete history of the genre, but it is intended to highlight the titles that had a significant impact on the evolution of this category of games. Next, we survey AI-assisted design in videogames, examining the tools used and the published research in this area. Finally, we focus on 2D platformers AI-assisted design.

## 2.1 History of 2D Platformers

Platformers are defined as games where the character controlled by the player can move and jump in order to reach a goal location within a level while avoiding obstacles; in this context, *platforms* (i.e. blocks on which the character can stand) are the fundamental elements that make it possible for the character to travel from one point to another in a level.

### 2.1.1 The 1980s

Platformers have been introduced in the early 1980s. *Space Panic*[1] (1980) and *Crazy Climber*[2] (1980) are considered precursors of this genre, albeit they are solely focused on the mechanic of climbing and have no jumping. The first true 2D platformer is *Donkey Kong*[3] (1981); in this game, which introduced the jump mechanic, the main character has to rescue his kidnapped girlfriend on the top of a building, by moving upward and at the same time avoiding barrels thrown by the antagonist; the game also launched the character Mario, originally named Jumpman. Figure 2.1 shows the first level in Donkey Kong. In the same period,

[1] https://en.wikipedia.org/wiki/Space_Panic
[2] https://en.wikipedia.org/wiki/Crazy_Climber
[3] https://en.wikipedia.org/wiki/Donkey_Kong_(video_game)


Figure 2.1: A screenshot from Donkey Kong.

*Jump Bug*[4] (1981) introduced horizontally scrolled levels and a shooting mechanic to fire simple bullets and kill enemies; a year later, *Pitfall!*[5] (1982) added rope swinging to the mix. Next, Nintendo launched *Mario Bros.*[6] (1983), in which for the first time two players could cooperate to defeat all enemies in single screen and non-scrolling levels. Cooperation is also a huge feature in *Bubble Bobble*[7] (1986), which boasts an evolved shooting mechanic that was conceived so that fired bubbles could trap enemies and also be jumped upon to reach otherwise unreachable areas. *Super Mario Bros.*[8] (1985) shook the 2D platformers world with innovative gameplay elements such as size boosting mushrooms that allows the player to destroy certain blocks in the levels and take an extra hit from monsters without dying. In the same year, *Ghosts 'n Goblins*[9] (1985) emerged as one of the most difficult game of the genre; the game has the player run through levels full of monsters with poor play controls and timed (around three minutes long) lives that allow the character to endure two hits from enemies; in addition, checkpoints are also placed far apart, whereas other games have frequent restart and save points. *Metroid*[10] (1986) features exploration, a dark atmosphere and nonlinear gameplay; it inspired several games giving way to combinations with other genres. In fact, *Metroid* and the *Castlevania*[11] series (first installment in 1986) gave birth to the greatly influencing *Metroidvania* subgenre, characterized

[4] https://en.wikipedia.org/wiki/Jump_Bug
[5] https://en.wikipedia.org/wiki/Pitfall!
[6] https://en.wikipedia.org/wiki/Mario_Bros.
[7] https://en.wikipedia.org/wiki/Bubble_Bobble
[8] https://en.wikipedia.org/wiki/Super_Mario_Bros.
[9] https://en.wikipedia.org/wiki/Ghosts_'n_Goblins
[10] https://en.wikipedia.org/wiki/Metroid_(video_game)
[11] https://en.wikipedia.org/wiki/Castlevania


Figure 2.2: A screenshot of Super Mario Bros. 3 map.

by strong emphasis on story and a large map that has to be explored to unlock new areas and power-ups for the main character. *Mega Man*[12] (1987) represents an important milestone in terms of nonlinear gameplay; the game gives the possibility to play levels in arbitrary order, although the weaknesses of bosses and the power-ups received in each level provides a recommended order for their completion. The third installment installment of the *Super Mario Bros.* series (*Super Mario Bros. 3*[13], 1988) broke new ground by introducing a level progression system in the form of a fully interactive world map, which is a level in itself (Figure 2.2). *Prince of Persia*[14] (1989) introduced a peculiar sword-based combat system and realistic character animations.

### 2.1.2 The 1990s

Next, Sega released *Sonic the Hedgehog*[15] (1991) in an attempt to oppose the *Super Mario Bros.* series domination on the scene; the game features fast paced action thanks to the ability to run through the levels at a much higher speed than other games and curl the character into a ball in order to hit enemies and obstacles. In its sequel (*Sonic the Hedgehog 2*[16], 1992), for the first time in platform games, two players could race against each other through selected stages in a simultaneous split-screen versus mode.
In the 1990s, graphics and technology improvements led videogames to shift to 3D worlds and platformers were no exception, although many 2D platformers continued to be produced. Some examples of successful 3D platformers are *Crash*

[12] https://en.wikipedia.org/wiki/Mega_Man_(video_game)
[13] https://en.wikipedia.org/wiki/Super_Mario_Bros._3
[14] https://en.wikipedia.org/wiki/Prince_of_Persia_(1989_video_game)
[15] https://en.wikipedia.org/wiki/Sonic_the_Hedgehog_(1991_video_game)
[16] https://en.wikipedia.org/wiki/Sonic_the_Hedgehog_2


Figure 2.3: Screenshots from Pandemonium (left) and Crash Bandicoot (right).

*Bandicoot*[17] (1996) and *Super Mario 64*[18] (1996). This transition was not easy and the difficulty of adapting some mechanics gave birth to 2.5D games like *Pandemonium!*[19] (1996) and *Klonoa: Door to Phantomile*[20] (1997), which adopted 3D visuals while retaining 2D gameplay. Figure 2.3 illustrates the 2.5D visuals of Pandemonium! and the 3D world of Crash Bandicoot. *Rayman*[21] (1995) was the first platform game to implement the possibility to make user-created content; in fact, its updated version (*Rayman Gold*, 1997) allows players to create and share levels using the Internet thanks to the *Rayman Mapper* tool.

### 2.1.3 The new millennium

Entering the new century, *Prince of Persia: The Sands of Time*[22] (2003) introduced a time rewinding mechanic and much more freedom of movement in the environment (e.g. running on wall); the latter probably influenced games such as *Mirror's Edge*[23] (2008). The *Jak and Daxter*[24] main saga (2001-2009) merged platforming elements in an open world action adventure featuring racing and puzzle sections. The *Little Big Planet*[25] series (2008-2014) put strong emphasis on content creation and sharing; the games allow to upload levels online so that other players can play, write reviews or comments and share links to levels via social networks.
With the exception of some historic series (e.g. *Super Mario Bros.*), platforming

[17] https://en.wikipedia.org/wiki/Crash_Bandicoot_(video_game)
[18] https://en.wikipedia.org/wiki/Super_Mario_64
[19] https://en.wikipedia.org/wiki/Pandemonium!_(video_game)
[20] https://en.wikipedia.org/wiki/Klonoa:_Door_to_Phantomile
[21] https://en.wikipedia.org/wiki/Rayman_(video_game)
[22] https://en.wikipedia.org/wiki/Prince_of_Persia:_The_Sands_of_Time
[23] http://www.mirrorsedge.com/
[24] https://www.naughtydog.com/games/jak_and_daxter_the_precursor_legacy
[25] http://littlebigplanet.playstation.com/

Figure 2.4: Screenshots from Braid (left) and Super Meat Boy (right).

elements became less and less relevant in major 3D games, leaving the spotlight to intricate stories, cinematic sequences, shooting and much more depending on the context, such as in the *Uncharted*[26] series (2007-2016). With the introduction of mobile devices and their exponential spread, 2D games with simple and immediate mechanics returned to be popular. The simplification of controls gave birth to the new subgenre named *endless runner*, in which levels are procedurally generated and the character is always running and trying to avoid obstacles and gaps; this category of games satisfies the need of short gaming sessions for casual gamers, but also encourages hardcore players to achieve the highest possible scores with online leaderboards; some notable examples for this subgenre are *Temple Run*[27] (2011) and *Subway Surfers*[28] (2012).
In recent years, the independent scene saw the rise of some original games that, while remaining essentially pure 2D platformers, tried to innovate and gave the genre new life. For instance, *Braid*[29] (2008) amazed with the perfect mix of time manipulation powers and puzzle elements. Next, *Super Meat Boy*[30] (2010) offers a peculiar experience in which the loss of lives is an integral part of the game, as the levels are designed to be completed with near perfectly timed actions; also, the online leaderboard recording levels completion times is a huge factor for replayability. Figure 2.4 shows levels from Braid and Super Meat Boy. *Fez*[31] (2012) proposed a 3D world that can be observed from one of the four 2D planes around a cube-like space and has its puzzles focus on this feature. As a final example for the independent scene, *Ori and the Blind Forest*[32] (2015) gives players full freedom on how to manage checkpoints: energy cells scattered around the world can be used to manually create a restart point, but the limited amount of cells

[26] https://www.unchartedthegame.com/en-us/
[27] https://en.wikipedia.org/wiki/Temple_Run
[28] http://www.kiloo.com/games/subway-surfers/
[29] http://braid-game.com/
[30] http://supermeatboy.com/
[31] http://www.fezgame.com/
[32] http://www.oriblindforest.com/

forces the player to carefully use them.
Finally, the recent *Super Mario Maker*[33] (2015) is of particular interest for the research subject, as it gives players the design tools necessary to create original content starting from the basic blocks and elements from the *Super Mario Bros.* series; levels can then be shared using the Internet, encouraging the rise of players and designers communities focused on creation and playing challenges (e.g. world record completion times).

## 2.2 AI-assisted design in videogames

Computer-aided design tools have become widely used in many production environments as they help reduce development time, costs and human efforts, while enabling collaboration between team members. This software category often implements Procedural Content Generation (PCG) of one or more types of artifact; notable examples include *SpeedTree*[34] (IDV, 2002), which allows the automatic generation of vegetation in 3D environments, and Volcano [13], that creates 3D models of swords by exploiting interactive evolution. In the context of 3D artifacts creation, Maddegoda and Karunananda [15] developed a multi agent based approach to generate 3D models starting from agents having a set of simple rules, while Peña et al. [19] presented a mechanism to generate 3D buildings given designer constraints and guidelines.
Especially in the videogames development scene, tools such as *Unreal Engine*[35] (Epic Games, 1998) and *Unity3D*[36] (Unity Technologies, 2005) are greatly supporting the creation of games both in the independent community and in major companies and have established themselves as the standard in the industry. Although these tools are extremely flexible, they limit the human designer actions to the positioning of objects (game elements) in a scene (level) and to the tweaking of parameters (e.g. how fast an enemy runs or how much health it has); they don't provide feedbacks on the player's perspective (i.e. feasibility and difficulty of levels). In addition, initiative for the creation of contents solely depends on the human designer and his creativity, without any concrete input from the software. As a consequence, some works have focused on researching mixed-initiative design, meaning that artificially intelligent tools are designers themselves and take part in the design process at the same level of human authors. In this context, Yannakakis et al. [33] investigated the process of game levels design where a human designer and the computer proactively make contributions to the problem solution, highlighting the fact that computational initiative is useful and can foster human creators' creativity. Browne and Colton [1] studied how to make a

[33] http://supermariomaker.nintendo.com/
[34] http://www.speedtree.com/
[35] https://www.unrealengine.com/
[36] https://unity3d.com/

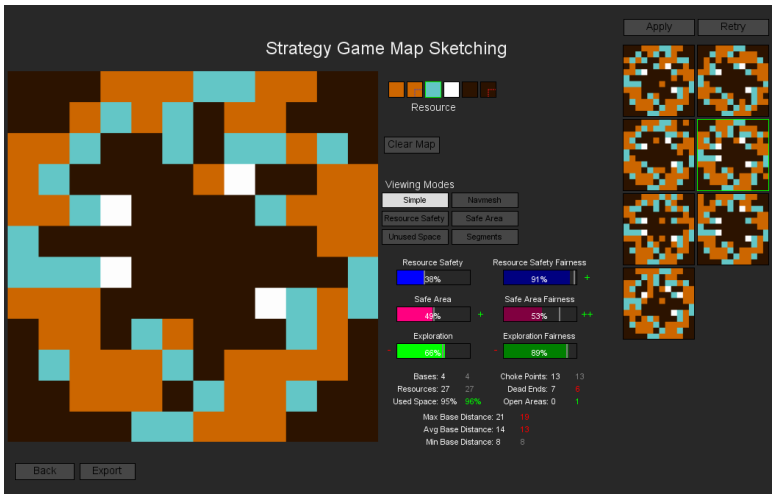


Figure 2.5: A screenshot of Sentient Sketchbook User Interface.

computer emulate the creative process of a human designer, whereas Lopes et al. [14] examined how a mixed-initiative drawing tool can stimulate the creativity of young learners. One of the most relevant work in this field is Sentient Sketchbook [10], a software that supports a designer in the creation of levels for strategy games, by giving the possibility to sketch a map, automating playability checks and evaluations of significant gameplay properties and lastly, using constrained novelty search to provide alternatives to the human's design (Figure 2.5); this research showed how to make an AI tool provide suggestions that are more appropriate to a specific user's style, focus and end-goals by modeling the designer's behavior [9][12]. Tutenel et al. [32] proposed a solution to better understand a designer's intent, combining a semantic class library, which includes information about the objects that can be placed in the scene and their relationships, and a layout solver that places objects in the environment by following a set of rules. Next, different methods of refinement for map sketches of distinct game genres were studied, such as increasing map resolution, linking more sketches together and going from high level evaluations to game-specific estimations; the study in [8] showed promising results on how an AI design tool can iteratively improve the level of detail of a game level, in a similar way to how a human designer goes from the big picture to the fine points.
In the context of mixed-initiative creation, Karavolos et al. used the Ludoscope tool [7] to research the approach of creating a content generator by making a model of the design process; the software at issue defines levels as expressions that are generated by a grammar: the designer declares the alphabet of the grammar and the rules that can transform expressions, then the tool probabilistically executes the rules to create content, while allowing editing from the human author. A similar approach was carried out with BIPED [25], a tool that allows the prototyping of board games by having the designer give a concise game def-

inition; the system also provides access to significant insights: feedback from human players (e.g. hesitation or fun) and automated analysis, which revealed hidden properties of the games. A suite of tools, including PRIME Designer [31], was developed to study how AI can support a common language for the design process, maximize effectiveness of collaboration and teach design methodology. In the specific scope of computer role-playing games (CRPGs), a relevant study was performed with the combination of the Grail Framework, supporting goal-based quests and giving more variety to the actual gameplay, and QuestBrowser, conceived to help designers in finding new and innovative solutions for each quest by using a common-sense knowledgebase of thousands of people [30]. Finally, Liapis et al. showed in [11] how to evaluate the quality of game levels across different games and genres, by defining high level concepts which can be evaluated in various domains.

## 2.3 AI-assisted design in 2D platformers

Many analyses were carried out on the 2D platformer genre, ranging from high level studies about the fundamental elements in this kind of games to advanced PCG algorithms aiming at maximizing one or more metrics. Smith et al. [26] dissected level components in terms of their roles and the structure of 2D platformers, in order to better understand the design behind levels and to provide designers a common vocabulary for the items forming the game worlds. In a similar fashion, Dahlskog and Togelius discussed in [2] the potential roles of design patterns in PCG, identifying them in levels of the original Super Mario Bros.; they also proposed how these patterns can be combined to create new levels that comply with specific constraints, while retaining variety of contents; these classification of patterns was then used to actually generate game levels, by using micro-patterns as building blocks in a search-based algorithm as showed in [3].
In the context of AI-assisted design, the mixed-initiative tool Tanagra [27] was developed with the goal of making a human designer and a computer work together to produce game levels; Tanagra allows the human author to place constraints on a running level generator, both on the geometry layer (e.g. positioning of objects) and on the level's rhythm (Figure 2.6). In fact, Tanagra bases its operations on the concept of "beat", that represents an interval of play time, during which the player must take some action, such as running or jumping. The tool can fill in the rest of the level or inform the designer that there is no solution (i.e. game elements placement) that satisfies the requirements while at the same time guaranteeing playability; the whole process is managed using the reactive planning language ABL to respond to designer input and the constraints solving library Choco to determine the physical placement of components in the level [28]. StreamLevels [6] was developed to support the description of level structures using streamlines, which can be drawn or uploaded as real players trace

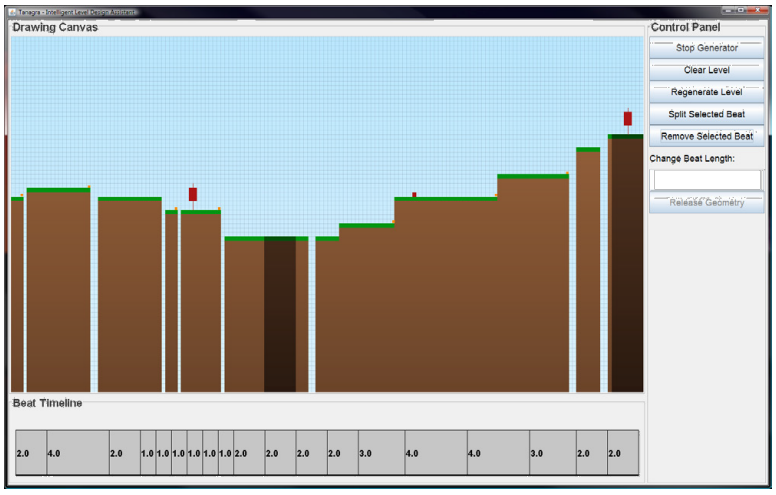


Figure 2.6: A screenshot of Tanagra User Interface.

data; the tool works on a grid and generates levels with a constructive algorithm by placing platforms in each of the tile crossed by the streamline such that there is always a feasible path from the beginning to the final tile.
Moving the focus to PCG techniques, levels generation in 2D platformers has been deeply researched and even competitions, such as the Level Generation Track in the Mario AI Championship, were held to foster researches in the field [22]. One of the main approaches in content creation has been experience-driven, meaning that the goal is to make the player feel a specific emotion while playing. In order to do so, Pedersen et al. investigated in [18] the relationship between level design parameters, individual playing characteristics and player experience; a neural network model was used to map levels structural parameters, gameplay data and reported emotions; results showed that fun is the hardest reaction to predict, but the models were accurate enough to consider deploying them in generating game levels that enhance player experience. Further contributions to this work were given by using a much larger data set and performing validation through algorithms and human players [23]. In addition, investigations were made about having levels with multiple paths and integrating puzzle based content requiring the player to explore [17]. New refinements on these methods were pursued by analyzing the impact of the session size and weighting features (e.g. number of collected coins or number of obtained powerups) and structural patterns, as well as their position in the levels [24].
Another path in PCG was followed by Occupancy-Regulated Extension (ORE) [16], a general geometry assembly algorithm, whose core concept is occupancy, which is expressed as potential positions (defined as anchors) that the player can occupy during gameplay; anchors are used to iteratively expand the existing content by merging human-authored chunks. An approach that minimizes the amount of manually authored content is used in Launchpad [29], a rhythm-based

level generator in which levels are built out of small segments called "rhythm groups".
Ferreira et al. presented in [5] a multi-population genetic algorithm for generating levels by evolving four elements (terrain, enemies, coins and blocks), each one with its own encoding, population and fitness function; at the end of the evolution the best four elements are combined to build the level.
A combinatorial approach is followed with Non-negative Matrix Factorisation (NMF) [21], an algorithm that, using levels from five dissimilar content generators, generates novel content through exploring new combinations of patterns.
Next, Reis et al. [20] proposed a system which uses human computation to evaluate the quality of small portions of levels generated by another system and the tested sections are combined into a full level; results showed that this approach was able to create levels that were better both in terms of visual aesthetics and fun.
Lastly, as a commercial game and design tool, *Super Mario Maker* offers an easy and intuitive interface to create levels; each level is represented as a grid in which game elements (i.e. blocks, enemies and power-ups), selected from a palette, can be placed. The design tools in the game do not include automated checks for playability or difficulty, thus the designer has to beat the levels in order to upload them on the Internet. For what concerns difficulty estimation, a clear rate (e.g. number of completions on number of tries) is displayed for each level and, once a sufficient number of players have played the level, an undisclosed algorithm, probably taking into consideration parameters such as the average number of deaths per player and their locations in the level, assigns a difficulty value from one (minimum difficulty, labelled as *Easy*) to four (maximum difficulty, labelled as *Very Hard*).

## 2.4 Summary

In this chapter, we discussed the state of the art of 2D platformers; we overviewed the history of this category of games, highlighting the most relevant titles. Then, we surveyed AI-assisted design in videogames, examining the tools used and the published research in this field. Finally, we focused on 2D platformers AI-assisted design, describing the techniques used for the generation of levels and the tools employed.

# Chapter 3

# Level design metrics

In this chapter, we introduce the concept of *design metrics* and discuss the goals they try to accomplish in the context of game content creation. First, we define metrics and explain why they are used, also giving an example of how general-purpose metrics can be applied to different game genres. Next, we discuss *structural features* for 2D platformers and their role in the evaluation and generation of levels. Finally, we review *gameplay features* for 2D platformers and how they can be used to improve the player experience within the game.

## 3.1 The importance of design metrics

When designing game levels, it is often useful to analyse created content numerically. This is done using *metrics* that can either describe a specific property of a level or evaluate it as a whole. These numeric evaluators can provide key feedbacks to human designers and drive the automatic content generation process. Liapis et al. [11] proposed a genre-independent method to evaluate game levels in the form of map sketches consisting of a grid layout that involves three high level concepts: *area control*, *exploration* and *balance*. Area control is the feature that quantifies the control over the game's resources; exploration incorporates the "goal of learning the layout of the game world, or locating specific parts or objects in it"; balance "ensure that players have equal opportunities". In addition, Liapis et al. [11] showed that specific and domain-dependent evaluation functions can easily be derived from the three abstract concepts. In fact, they defined the safety of any tile $t$ to a reference tile in the grid (i.e. a tile with a special purpose in the game, such as a checkpoint tile) $i$ as follows:

$$s_{t,i}(S_N) = \min_{1 \leq j \leq N, j \neq i} \{\max\{0, \frac{d_{t,j} - d_{t,i}}{d_{t,j} + d_{t,i}}\}\}$$

Where $N$ is the number of elements in the set $S_N$ of reference tiles and $d_{t,i}$ is the distance from tile $t$ to element $i$. Next, the strategic resource control metric was

derived from it, obtaining the following expression:

$$f_s(S_N, S_M) = \frac{1}{M} \sum_{k=1}^{M} \max_{1 \leq i \leq N} \{s_{k,i}\}$$

Where $M$ is the number of elements in the set $S_M$ of possible target tiles. $f_s(S_N, S_M)$ was then summed with strategic resource control balance, which is expressed as follows:

$$b_s(S_N, S_M) = 1 - \frac{1}{MN(N-1)} \sum_{k=1}^{M} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} |s_{k,i} - s_{k,j}|$$

This allowed to obtain a fitness function which was used to evaluate individuals in a two-population Genetic Algorithm performing map generation for a strategy game.

## 3.2 Structural features

Structural features are the ones based on the placement of game elements and describe levels just by looking at them, without any need to perform playtesting. A trivial example of structural feature can be the number of platforms in the level, although it does not provide any significant insight by itself, thus it should be combined with other metrics to obtain a notable indicator (e.g. use the level width to attain how many platforms there are, on average, for each unit of horizontal space). Structural features are referred as *controllable* if they are used to guide the generation of levels; in [24], [18] and [23] a combination of the following controllable features are exploited to create levels for a modified version of Markus Persson's *Infinite Mario Bros.*[1]:

- Number of gaps in the level; that is the number of empty horizontal spaces between two platforms;
- Average width of gaps; that is the average width of the empty spaces between two platforms;
- Number of enemies in the level;
- Enemies placement (defined by three probabilities which sum to one: $P_x$, $P_g$ and $P_r$):
  - $P_x$: probability of placing an enemy on or under a set of horizontal blocks;

[1] http://indiegames.com/2008/10/browser_game_pick_infinite_mar.html

  - $P_g$: probability of placing an enemy within a close distance to the edge of a gap;
  - $P_r$: probability of placing an enemy on a flat space on the ground.

- Number of power-ups in the level;
- Number of boxes (i.e. game elements that can be pressed to obtain rewards) in the level;
- Entropy of gap-placements expressed as follows:

$$H_g = -\frac{1}{\log G} \sum_{i=1}^{G} \frac{g_i}{G} \log \frac{g_i}{G}$$

  Where $G$ is the number of equally spaced segments and $g_i$ is the number of gap-placements into level segment $i$;

- Number of direction switches (i.e. the number of times the player needs to turn around and go the other way) needed to complete the level.

Most of the listed features are easy to interpret and provide a very direct feedback or goal values for content generation to designers. Entropy of gap-placements provides a measure of how the gaps, or equivalently the jumps to overcome them, are distributed in the various segments of the level; as a result, the entropy metric can be seen as an indicator of the unpredictability of gaps, which is a huge factor for the player experience. The number of direction switches provides a measure of non linearity for the level; if the number of direction switches is zero then the level is extremely linear as the character will always move from left to right; otherwise, as the number of direction switches increases, the character will have to turn around more often in order to go through the level.
A different approach is taken by Launchpad [29], a rhythm-based level generator which works on the concept of *rhythm group*: a small level segment consisting of a rhythm of player actions (also referred as beats) and a level geometry that corresponds to that rhythm; Figure 3.1 illustrates some examples of rhythm groups. Launchpad presents the following set of parameters that the designer can manipulate to obtain different levels:

- Rhythm type:
  - *Regular*: beats are evenly spaced;
  - *Swing*: a short beat is followed by a long one;
  - *Random*: beats are randomly spaced.
- Rhythm density: describes how closely the beats are spaced (can be "low", "medium" or "high");

- Rhythm length: defines the length (in seconds) of the rhythm group;
- Action probabilities: the probabilities of jump and wait actions occurring;
- Jump components: the desired frequency for specific geometry being used in a level for a jump action;
- Wait components: the desired frequency for specific geometry being used in a level for a wait action;
- Repeating geometry: probability that a rhythm group is immediately repeated;
- Line equation: the path that the final level should follow, defined as a set of non-overlapping line segments;
- Number of coins: game elements that can be collected to increase the score in rhythm group;
- Platform length: threshold platform length for coin placement, meaning that any platform with a length greater than this value will have coins placed along it;
- Coin over gap probability: probability that a coin is placed over a gap.

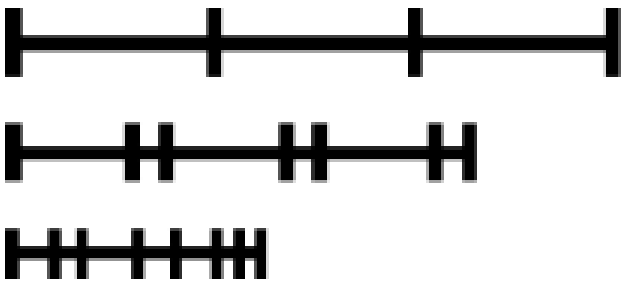

Figure 3.1: Examples of rhythm groups in Launchpad. A 20s long, regular, low density rhythm group (top), a 15s long, swing, normal density rhythm group (center) and a 10s long, random, high density rhythm group (bottom).

In addition to the listed parameters, a level designer can adjust the importance (i.e. assign a weight) of two critics: the *line distance critic* and the *components frequency critic*; the line distance critic measures the difference between the generated level structure and the specified line equation, whereas the components frequency critic measures the discrepancy between the components (i.e. geometry elements corresponding to jump or wait actions) frequency in the generated

level and the desired parameters. The two critics are combined to evaluate generated levels and, since they often contradict each other, the level with the lowest weighted sum is selected.
Finally, a combinatorial approach to cover and widen the expressive range of five different level generators is followed in [21], and, in order to evaluate generated levels, the following expressivity metrics are used:

- *Linearity*: measures how flat a level is (e.g. a level with high linearity presents little to no fluctuations in the height of platforms, meaning that the player will go through the level following a perfectly horizontal line);
- *Leniency*: measures how easy a level is by counting the number of occurrences of rewarding and harmful items such as coins and enemies;
- *Density*: measures the number of platforms stacked on top of each other at each level segment.

## 3.3 Gameplay features

Gameplay features describe a player's skill and playing style and they are usually obtained collecting data with one or more sessions of playtesting on a set of levels and describe players' behavior in a number of ways; for instance, these dynamic metrics can provide the number of times an action was executed, report how players went through the level (both in terms of speed and path followed), or highlight where in the level players encountered the most difficulties; to sum up, these numeric evaluators describe a player's skill and playing style. Furthermore, gameplay features can reveal hidden properties of levels, which may not be captured by structural features, such as points in the levels where players are not sure how to proceed and, as a consequence, they may try random actions in order to advance; recording this type of data allow designers to spot ambiguous or misleading design elements and fix them.
In [24], [18] and [23], the goal was to create personalized content: after profiling players, they generated levels that would fit the players' skill and playing style, eliciting fun and challenge. In order to do that, hundreds of game sessions were performed on levels generated exhausting the controllable features space for Infinite Mario Bros.; a combination of the following gameplay features was collected:

- Generic features:
  - Level completion time;
  - Duration of last life over total time spent on the level;
  - Time spent in Small Mario mode over total time spent on the level;

  - Time spent in Big Mario mode over total time spent on the level;
  - Number of times the player shifted the Mario mode (Small, Big, Fire);
  - Total number of player's deaths;
  - Cause of the last player's death;
  - Number of times the player was killed by an enemy;
  - Number of deaths due to gaps;
  - Whether the level was completed or not (boolean value).

- Actions related features:

  - Time spent ducking over total time spent on the level;
  - Time spent jumping over total time spent on the level;
  - Time spent moving left over total time spent on the level;
  - Time spent moving right over total time spent on the level;
  - Time spent running over total time spent on the level;
  - Number of times the jump button was pressed;
  - Number of times the duck button was pressed;
  - Number of times the run button was pressed;
  - Total number of jumps;
  - Number of jumps over gaps;
  - Number of jumps without any purpose;
  - Difference between the number of gaps and the number of jumps;
  - Number of times the player changed the state between: standing still, running, jumping, moving left and moving right.

- Items related features:

  - Number of collected items over total items in the level;
  - Number of coins collected over total number of coins in the level;
  - Number of empty blocks destroyed over their total number in the level;
  - Number of coin blocks pressed or coin rocks destroyed over their total number in the level;
  - Number of power-ups blocks pressed over their total number in the level;
  - Sum of all blocks and rocks pressed or destroyed over their total number in the level.

- Enemies related features:
  - Number of times the player killed a specific enemy;
  - Total number of enemies killed;
  - Number of enemies killed by stomping;
  - Number of enemies killed by fire shots;
  - Number of enemies killed by unleashing a turtle shell;
  - Number of times the player kicked an opponent shell;
  - Total number of kills over total number of opponents;
  - Number of enemies killed minus number of deaths caused by enemies.

All player actions and interactions with game items and their timestamps were recorded together with the full trajectory of Mario within the level [24][18][23]. In addition to the listed features, a jump difficulty heuristic, proportional to the number of deaths due to gaps, number of gaps and average gap width, was computed to provide an estimation of the level difficulty. Players were also asked to report their experience after playing the levels; a questionnaire was presented to the players asking to report their emotional states among fun, challenge and frustration. In order to analyse the effects of sequential features (i.e. sequence of player actions), the study in [24] applied data mining techniques, such as Generalised Sequential Pattern (GSP), to extract the most frequent sequences of interactions. Next, the relevant subsets of features for predicting each emotional state were selected; this was achieved with different algorithms: *Sequential Forward Selection* (SFS), *n best individual feature selection* (nBest) and *Perceptron Feature Selection* (PFS). The quality of a features subset was evaluated using it as the input for a set of multi-layer perceptron (MLP) models: the performance of each MLP was obtained as the average classification accuracy in three independent runs using 3-fold cross validation across five runs. The resulting neural networks, approximating the function between gameplay features, controllable features, and reported emotional states, presented different number of selected features and performance; the most accurate model was the one predicting challenge (approximately 91% accuracy), whereas fun was determined to be the most difficult emotion to predict from controllable features.
Another interesting approach for personalized level generation has been presented in [17]. As a first step, each player was asked two pre-game questions; the first requested to choose a subgenre preference among *Combat*, *Flow* and *Puzzle*. As the names indicate, Combat indicates the player wish to fight enemies; Flow indicates a preference for level structures that require tactile skills, and Puzzle expresses the wish to explore different paths through the level in order to complete it. The second question asks the player to indicate a difficulty level. Next, the player's abilities were measured with a set of trials (i.e. hand-made levels),

which were shorter than typical levels, but of greater difficulty. The following gameplay features were derived from the trials:

- Time spent to complete the trial;
- Distance travelled on the ground;
- Distance travelled in the air;
- Average speed on ground;
- Average speed in air;
- Number of shots fired;
- Percentage of fireball hits per shot;
- Highest number of kills with a single jump;
- Highest number of kills with a single shell kick;
- Number of enemies killed with fireballs;
- Number of enemies killed by kicking a shell;
- Number of jumps;
- Number of combat jumps (i.e. used to hit enemies);
- Number of portal (i.e. game elements working as teleporters, connecting different parts of a level) usages.

In order to track the player's trajectory, every time the screen rendered a new image the player's position was recorded together with a timestamp. In addition, other events, such as an enemy being killed or the usage of a portal, were recorded with the event position and current system time. This data collection allowed to set the values of the four high-level parameters that were used in the generation phase: combat, flow, puzzle and *clutter*; the latter described the quantity of coins and items that would be placed in the level; as such, a high value of clutter was bound to result in the generation of levels with a lot of rewarding items, which is common in easy levels. The four above-mentioned parameters were all scaled in a $[0, 1]$ interval. Before actually creating a level, a graph was generated as an abstract representation of the final level. A feasible-infeasible two population Genetic Algorithm was used to generate graphs, employing a weighted sum of the four following parameters as the fitness function:

- Average number of outgoing edges;

- Number of starting node connections: the count of outgoing edges from the starting node (ideally low or exactly one);
- Number of final node connections: the count of incoming edges in the final node (ideally as close to one as possible);
- Length of the shortest path from the starting node to the final one.

Furthermore, a set of feasibility constraints was applied to obtain feasible graphs, which were used in the next steps of level generation. The following constraints had to be fulfilled to guarantee feasibility:

- Path length: in order to make sure that the starting and final nodes were not adjacent, the shortest path from these nodes had to traverse at least a fraction of the graph size (i.e. nodes count), defined as $1.0 - 0.3puzzle$;
- Forward reachability: every node had to be reachable from the starting node;
- Backward reachability: every node had to be reachable from the final node.

It must be noted that the backward reachability constraint prevents the presence of points where the player would be forced to fail and restart the level, allowing him to backtrack. Together, the two reachability constraints guarantee that, for every node, there will be a path from the starting to the final node that passes through it. As a final step, the best feasible graph found by the Genetic Algorithm was converted into a grid layout; cells were then iteratively populated with game elements while preserving the graph properties.

## 3.4 Summary

In this chapter we discussed level design metrics. In particular, we focused on structural features and gameplay features for 2D platformers, reviewing their role in the process of game content creation. We examined the most relevant works in this area and their approach to level design. In the next chapter, we discuss how we tackled the level design problem and describe our framework for designing 2D platformers.

# Chapter 4

# Our approach to level design

In this chapter, we review our approach to level design in 2D platformers that extends the previous works in this area. In the next sections, we first introduce the philosophy of our approach. We examine how we conceptually model game levels and present our framework for the design of 2D platformers. Next, we discuss the structural features offered by our framework. Finally, we review our trajectory based approach for the evaluation of gameplay features.

## 4.1 Introduction

In Chapter 3, we reviewed several approaches that employed either structural or gameplay features or both for the evaluation and generation of 2D platformer levels. Our approach is a hybrid: it provides a set of structural features that can be adjusted by designers and employs a method for the estimation of gameplay features, namely the jumps difficulty and their probability of success. In order to accomplish this, we had to use a fitting conceptual model for game levels and to develop a framework that supports the analysis of gameplay dynamics. Considering our needs and goals, a directed graph representation is used to abstract game levels, whereas the framework allows to perform specific simulations on level graphs and playtest the designed content.

### 4.1.1 Conceptual model and abstract representation of a level

As we discussed in Chapter 3, being able to inspect game content using metrics is of key importance to improve its quality. For this reason, employing the appropriate abstract representation for game levels is critical, as it allows to study the created content properties with minimal effort. Thus, we decided to use a directed graph representation for the levels: each graph vertex represents a platform, whereas each directed edge represents the connection between two plat-

forms. Directed edges are essential because we need to know the direction of the connections, as there are cases in which, for example, a platform A is reachable from a platform B, but not vice versa. In the context of 2D platformers, the jump is the fundamental action that allows to move from one platform to another, or, equivalently, from one vertex to another. As a consequence, our approach is based on the concept of *Reachability* through the evaluation of jump trajectories, and provides the estimation of the jumps difficulty and their probability of success. In order to do that, we use the set of physics parameters handling the movement of the character, both on ground and in air. As such, to achieve a flexible and general method, we implemented a 2D platformer as a module of our framework that offers a wide set of adjustable physics parameters and different mechanics (e.g. single or double jump).

### 4.1.2 Our framework

Our framework for the design of 2D platformers consists of two modules: a parametric 2D platformer and a level evaluator. The 2D platformer was designed and developed to support our approach and to serve as a quick testing tool for the designer. In addition, the 2D platformer was implemented keeping in mind the goal of flexibility, and, as such, presents several changeable physics parameters (e.g. gravity) and a set of mechanics with different operating modes, which can be switched on or off. Furthermore, the platformer implementation provides a set of platforms of different types, covering the macro categories of platforms that can be encountered in games of this genre. These elements allow to design levels with a good variety and let us test the generality of our approach and make it possible to virtually apply it to several types of 2D platformers. The level evaluator takes into account the physics and mechanics settings, building a set of constraints that are used to define the possible jump trajectories. These trajectories are evaluated according to whether they allow the character to proceed through the level or not, thus providing a feedback on the jumps difficulty. This approach allows designers to have an estimation of gameplay features, in terms of how difficult or easy a level is, without performing gameplay sessions with test players.

## 4.2 Analysis of structural features

As described in [4], a number of physics parameters are necessary to describe in a correct and complete way the movement, both on ground and in air, and the jump trajectories of an object (in our case the controllable character) in a two-dimensional environment. Thus, it is mandatory, for example, to define parameters which specify the maximum movement speed on ground and the related acceleration. We selected a set of parameters and mechanics to obtain a frame-

work that is as flexible as possible, also taking into account accessibility and ease of use for a human designer. As for the space measurement unit, we use the space unit defined by the physics engine underlying our framework; we refer to it as $u$. In addition, different operating modes for the movement and jump mechanics are provided. Table 4.1 lists and describes the adjustable parameters that regulate the character movement. Table 4.2 lists and describes the adjustable parameters that regulate the character jump.
For what concerns the features that are unrelated to the physical model, the framework provides parameters for three macro areas: management of the character health and of damages inflicted by harmful game elements (described in Table 4.3), management of the character shooting ability (described in Table 4.4), and management of the depletion of a resource (described in Table 4.5). The latter represents an on/off mechanic that involves the gradual consumption (as time passes) of a generic resource; when the value of this resource reaches zero, the character dies and must restart the level from the beginning; activating the resource depletion mechanic is equivalent to setting a time limit for the completion of the level.
All the values of the described parameters can be adjusted by a designer through a user interface.
To clarify the role of the movement parameters (Table 4.1), we describe how they work together. First, the movement input from the player is only horizontal, thus acting solely on the $x$ axis. This axis values increase from left to right; as such, a positive speed indicates a movement toward the right, whereas a negative speed indicates a movement toward the left. Given this consideration, the movement input has a positive value if the player wants to move the character toward the right and a negative value if the desired direction is the left. We suppose to describe a series of character movement interactions on ground; the air movement is managed in the same way, the only difference being the parameters used. The character starts motionless, with zero speed. As soon as a movement input is received, the character accelerates with a constant acceleration equal to *groundAcceleration* (with the same sign as the input); the character moves with a uniformly accelerated motion until his speed is equal to *maxGroundSpeed* (with the same sign as the input) and continues moving with this speed value if the movement input is not changed; In this condition, if the movement input changes its sign, the character starts accelerating in the opposite direction with a constant acceleration equal to *groundTurnAcceleration* until his speed has the same sign as the new input, and the acceleration value goes back to *groundAcceleration*. If the character speed is not zero and the movement input is released, the character decelerates with a constant deceleration value equal to *groundDeceleration* (with the opposite sign of his speed) until his speed reaches zero. If the *hasInstantAcceleration* and *hasInstantDeceleration* parameters are set to *true*, the acceleration and deceleration times are, respectively, reduced to zero. If the character can run, the *runningMultiplier* value provides the parameters for maximum speed and ac-

| Name | Description | Type |
|---|---|---|
| *movementType* | Defines how the character moves on the horizontal axis. It has three possible values:<br>*WALK_ONLY*: the character can only walk.<br>*WALK_AND_RUN*: the character can both walk and run (pressing the run button).<br>*RUN_ONLY*: the character can only run. | Enum |
| *maxGroundSpeed (maxGS)* | Defines the maximum horizontal movement speed on ground. Measured in $u/s$. | Float |
| *groundAcceleration (gndAcc)* | Defines the constant horizontal acceleration for ground movement. Measured in $u/s^2$. | Float |
| *groundTurnAcceleration (gndTAcc)* | Defines the constant horizontal acceleration for ground movement when the movement direction is changed. Measured in $u/s^2$. | Float |
| *groundDeceleration (gndDec)* | Defines the constant horizontal deceleration for ground movement (applied when no movement input is received). Measured in $u/s^2$. | Float |
| *maxAirSpeed (maxAS)* | Defines the maximum horizontal movement speed in air. Measured in $u/s$. | Float |
| *airAcceleration (airAcc)* | Defines the constant horizontal acceleration for air movement. Measured in $u/s^2$. | Float |
| *airTurnAcceleration (airTAcc)* | Defines the constant horizontal acceleration for air movement when the movement direction is changed. Measured in $u/s^2$. | Float |
| *airDeceleration (airDec)* | Defines the constant horizontal deceleration for air movement (applied when no movement input is received). Measured in $u/s^2$. | Float |
| *runningMultiplier (runMult)* | Defines the horizontal speed and acceleration multiplier used when the character is running. | Float |
| *hasInstantAcceleration* | Indicates whether the character instantly reaches the maximum horizontal movement speed (both on ground and in air) when he starts moving. | Boolean |
| *hasInstantDeceleration* | Indicates whether the character instantly stops moving horizontally (both on ground and in air) when no movement input is received. | Boolean |

Table 4.1: Parameters regulating the character movement.

celeration related to running, by multiplying the ones related to the standard movement (e.g. if *maxGroundSpeed* is equal to $3u/s$ and the *runningMultiplier* is equal to 2, then the maximum running speed on ground is $6u/s$).
Moving the attention to the jump parameters (Table 4.2), the jump mechanic allows to move the character on the vertical axis, thus acting solely on the $y$ axis; this mechanic, together with the movement (described in Table 4.1) allow to move the character in the two-dimensional space. The $y$ axis values increase from bottom to top; as such, a positive speed indicates the character is moving upward, whereas a negative speed indicates the character is falling down. The jump physics works in an extremely simple way: when the character is standing on a platform and the jump button is pressed, he is instantly accelerated to *ToS* and starts moving upward; the speed value is gradually reduced by the action of gravity and when it becomes negative, the character starts falling down. If the *hasTerminalFallingSpeed* parameter is set to *true*, the character falling speed caps at *TFS*. If the *isJumpDynamic* parameter is set to *false* the jump height is not influenced by how long the jump button is kept pressed, thus every jump reaches the same height; otherwise, if the *isJumpDynamic* parameter is set to *true*, we chose to make the jump reached height work as in *Super Meat Boy*: if the jump button is released during a jump and the character is moving upward (positive speed), then his speed is instantly set to zero, stopping his ascent; this makes the jump reached height dynamic, with a minimum time interval (*minimumJumpDuration*) in which releasing the jump button does not trigger the reset of the speed; this means that releasing the jump button before the minimum time interval is elapsed is equivalent to the release of the jump button as soon as the time interval elapses. As a consequence, the *minimumJumpDuration*, *ToS* and $g$ parameters define together the minimum jump height. If the *isTakeoffSpeedDynamic* parameter is set to *false*, the takeoff speed is always equal to *ToS* regardless of the horizontal movement speed; otherwise, if the *isTakeoffSpeedDynamic* parameter is set to *true* and the character can run, the takeoff speed is dynamic and proportional to how much the horizontal movement speed is greater than *maxGS*; the following equations describe how the dynamic takeoff speed is computed:

$$speedBonus = \frac{(|v_x| - maxGS)(maxToS - ToS)}{maxGS \times runMult - maxGS} \tag{4.1}$$

$$dynamicTakeoffSpeed = ToS + speedBonus \tag{4.2}$$

Equation 4.1 computes the takeoff speed bonus proportional to the current horizontal movement speed $v_x$, whereas equation 4.2 defines the dynamic takeoff speed by adding the speed bonus to the standard *takeoffSpeed* (*ToS*); as the equations show, if the character is moving at his maximum running speed, then his takeoff speed is equal to *maxTakeoffSpeed* (*maxToS*). All the movement and jump parameters must be positive numbers.
It should be noted that, having a physical model that works on speed and acceleration parameters, allows to have physical behaviors that are not dependent on

| **Name** | **Description** | **Type** |
|---|---|---|
| *gravity (g)* | Defines the gravitational acceleration applied to the character on the vertical axis. Measured in $u/s^2$. | Float |
| *jumpType* | Defines the jumping mode of the character. It has three possible values:<br>*NO_JUMP*: the character cannot jump.<br>*SINGLE_JUMP*: the character can perform a single jump when standing on a platform.<br>*DOUBLE_JUMP*: the character can perform a second jump in the air after having executed a first jump while standing on a platform. | Enum |
| *hasTerminalFallingSpeed* | Indicates whether the character has a terminal falling speed. If set to *true*, when the character reaches his terminal falling speed due to gravity, the character will stop accelerating and maintain his vertical speed constant. | Boolean |
| *terminalFallingSpeed (TFS)* | Defines the terminal falling speed value. Used only if *hasTerminalFallingSpeed* is set to *true*. Measured in $u/s$. | Float |
| *takeoffSpeed (ToS)* | Defines the vertical speed at which the character moves when he takes off from ground (i.e. performing the first jump). Measured in $u/s$. | Float |
| *isJumpDynamic* | Indicates whether the height of the jump is influenced by how long the jump button is kept pressed. | Boolean |
| *minimumJumpDuration* | Defines the time interval from the jump button pression during which releasing the jump button does not affect the height of the jump. Used only if *isJumpDynamic* is set to *true*. Measured in $s$. | Float |
| *isTakeoffSpeedDynamic* | Indicates whether the takeoff speed is dependent on the character horizontal movement speed at the time of the jump button pression. | Boolean |
| *maxTakeoffSpeed (maxToS)* | Defines the vertical speed at which the character moves when he takes off from ground (i.e. performing the first jump) while running at his maximum horizontal speed. Used only if *isTakeoffSpeedDynamic* is set to *true*. Measured in $u/s$. | Float |
| *doubleJumpSpeed (DJS)* | Defines the vertical speed at which the character moves when he starts performing a second jump in the air. Used only if *jumpType* is set to *DOUBLE_JUMP*. Measured in $u/s$. | Float |

Table 4.2: Parameters regulating the character jump.

| Name | Description | Type |
|---|---|---|
| *healthMode* | Defines how the character health is managed. It has three possible values:<br>*ONE_SHOT_KILLED*: the character is killed if he suffers any damage.<br>*TWO_SHOTS_KILLED*: the character is killed when he suffers any damage for the second time.<br>*HEALTH_BAR*: the character is killed when his health value reaches zero. | Enum |
| *maxHealth* | Defines the character health starting value when *healthMode* is set to *HEALTH_BAR*. | Float |
| *knockedTime* | Defines how long the character remains knocked out (i.e. unable to perform any action) after being damaged. Measured in *s*. | Float |
| *recoveryTime* | Defines how long the character remains in a recovery state in which he cannot be damaged after being knocked out. Measured in *s*. | Float |
| *spikesDamage* | Defines the amount of damage inflicted to the character when colliding with a *SPIKED* platform. | Float |
| *enemyContactDamage* | Defines the amount of damage inflicted to the character when making contact with an enemy. This value varies from enemy to enemy. | Float |

Table 4.3: Parameters regulating the character health and damages interactions.

the mass of the character, leaving the computation of the forces to apply to the physics engine.
For what concerns the character health and harmful game elements (Table 4.3), the elements that can harm the character are *SPIKED* platforms and enemies. The platformer implementation provides two types of enemies that can be placed in levels: a static enemy that starts shooting bullets at the character if he enters its visual range and a moving enemy that moves back and forth between two points in a straight line. Both types of enemies can be killed by striking them with a fireball, and the moving enemy can also be killed by jumping on its head. When the character suffers any damage and the parameters values allow him to survive the hit, he gets knocked out for *knockedTime* seconds; after being knocked out, he enters a recovery phase lasting *recoveryTime* seconds, in which he cannot be damaged; finally, after his recovery, he returns to a normal state, in which he can be damaged as usual.
For what regards the character shooting ability (Table 4.4), the platformer implementation offers a set of power-up items that change the fireballs shooting mode of the character, allowing the designer to place these power-ups in strategic

| Name | Description | Type |
|---|---|---|
| *timeBetweenShots* | Defines the minimum time that must elapse between two shots. Measured in *s*. | Float |
| *shootingMode* | Defines the fireballs shooting mode of the character. It has three possible values:<br>*NO_SHOOTING*: the character cannot shoot fireballs.<br>*CLASSIC_SHOOTING*: the character can shoot fireballs that are not affected by gravity. This means that fireballs travel in a straight line in front of the character until they collide with a game element.<br>*MARIO_LIKE_SHOOTING*: the character can shoot fireballs that are affected by gravity. This means that fireballs falls down and, if they hit a platform, they will bounce upwards and start falling again. This is similar to how fireballs shot by Mario in the Super Mario Bros. series behave. | Enum |
| *multipleShotsInterval* | Defines the time interval in which a maximum number of shots can be fired. Measured in *s*. | Float |
| *maxShotInInterval* | Defines the maximum number of shots that can be fired in the time interval set with *multipleShotsInterval*. | Integer |
| *rechargeTime* | Defines the time that must pass to resume shooting after firing the maximum number of shots in the limit interval. Measured in *s*. | Float |

Table 4.4: Parameters regulating the character shooting ability.

points of the levels (e.g. segments where a lot of enemies are present).

| Name | Description | Type |
|---|---|---|
| *isResourceDepleting* | Indicates whether the resource associated to the character is currently depleting. | Boolean |
| *maxResourceValue* | Defines the starting value for the resource. Measured in *ResourceUnits*. | Float |
| *depletionSpeed* | Defines the speed at which the resource depletes. Measured in *ResourceUnits*/*s*. | Float |

Table 4.5: Parameters regulating the resource depletion mechanic.

### 4.2.1 The platforms

Platforms (i.e. concrete blocks where the character can lean on) are the fundamental elements that make it possible for the character to travel from one point to another in a level. Given their key role, the platformer implementation provides a set of parameters that characterize the type, shape and other type-related properties of platforms. This features give wide choice to the designer and allow him to create levels of great variety.
Table 4.6 lists and describes the parameters that define every platform. To clarify on the *length* parameter role, all platform types, with the exception of the *STATIC_GROUNDED* and *SPIKED_GROUNDED* types, are formed by a single layer of tiles horizontally placed next to each other; as such, the *length* parameter defines the number of tiles composing each layer of a platform.
The *movSO*, *movEO* and *movSp* parameters are used only if the platform is of *MOVING* type. *MOVING* platforms are designed to move back and forth between two points in space in a straight line; we refer to these two points as *beginningPoint (BP)* and *endPoint (EP)*. *BP* and *EP* are computed using the platform initial position *IP* (i.e. the position in the level space where it has been placed by the designer) and the *movSO* and *movEO* parameters. In order to simplify our analysis, *MOVING* platforms are allowed to move either with a a horizontal trajectory (constant $y$ coordinate) or with a vertical trajectory (constant $x$ coordinate). Given this constraint, *MOVING* platforms are set to have a horizontal trajectory if any of the $x$ components of *movSO* or *movEO* is not zero, whereas a vertical trajectory is set if both the $x$ components of *movSO* or *movEO* are equal to zero. Equations 4.3 define the condition for a horizontal trajectory and the related *beginningPoint* and *endPoint*. Equations 4.4 define the condition for a vertical trajectory and the related *beginningPoint* and *endPoint*.

$$\begin{cases} movSO.x \neq 0 \quad \vee \quad movEO.x \neq 0 \\ BP = (IP.x - movSO.x, IP.y) \\ EP = (IP.x + movEO.x, IP.y) \end{cases} \tag{4.3}$$

$$\begin{cases} movSO.x = 0 \quad \wedge \quad movEO.x = 0 \\ BP = (IP.x, IP.y - movSO.y) \\ EP = (IP.x, IP.y + movEO.y) \end{cases} \tag{4.4}$$

For what regards *FADING* platforms, their behavior is quite simple: when the character makes contact with a platform of this type, the alpha channel (i.e. the value defining the transparency of an image) of the platform texture is gradually decreased with a rate equal to *fadSp*; once the alpha channel reaches a predefined threshold (we set it to 0.1), the platform disappears and, if the character is standing on it, he will fall. The platform is then restored to its normal state after a short time interval.

| **Name** | **Description** | **Type** |
|---|---|---|
| *platformType* | Defines the platform type. It has six possible values:<br>*STATIC_FLOATING*: a static floating platform.<br>*STATIC_GROUNDED*: a static multilayer platform that acts as a pillar planted in the ground.<br>*MOVING*: a moving platform.<br>*SPIKED_FLOATING*: a spiked static floating platform that damages the character on contact.<br>*SPIKED_GROUNDED*: a static multilayer platform that acts as a pillar planted in the ground and damages the character on contact.<br>*FADING*: a static floating platform that starts fading when the character collides with it. | Enum |
| *length* | Defines the length of the platform. This value represents the number of tiles composing the platform, except when it is multilayered. | Integer |
| *platformConnections* | Defines the platform appearance to fit one or more possible linked platforms. It has four possible values:<br>*NO_CONNECTIONS*: a platform that has no linked platforms.<br>*CONNECTED_LEFT*: a platform with the appropriate appearance to have a platform linked on its left.<br>*CONNECTED_RIGHT*: a platform with the appropriate appearance to have a platform linked on its right.<br>*CONNECTED_LEFT_RIGHT*: a platform with the appropriate appearance to have a platform linked on its left and another on its right. | Enum |
| *isStartingPlatform* | Indicates whether this is the platform where the character will start the level. | Boolean |
| *isFinalPlatform* | Indicates whether this is the final platform the character has to reach to complete the level. | Boolean |
| *hasCheckpoint* | Indicates whether this platform has a checkpoint on itself. | Boolean |
| *movementStartOffset (movSO)* | Defines the offset from the position the platform is placed to the starting point of its path. | Vector2 |
| *movementEndOffset (movEO)* | Defines the offset from the position the platform is placed to the final point of its path. | Vector2 |
| *movementSpeed (movSp)* | Defines the speed at which the platform moves along its path. Measured in $u/s$. | Float |
| *fadeSpeed (fadSp)* | Defines the speed at which the platform fades. Measured in $alpha/s$. | Float |

Table 4.6: Platforms parameters.

Figure 4.1: *Bounds Boxes* (in red) for a *STATIC_FLOATING* platform (left) and for a *MOVING* platform with a horizontal trajectory (right).

## 4.3 Analysis of gameplay features

We based our platforms reachability analysis on the study of the possible jump trajectories; the result of this analysis is a directed graph of the level, which allows us to make specific considerations on how the level can be tackled by the player, both in terms of the available paths for reaching the final platform and on their difficulty.
One module of our system is responsible for the computation of physics constraints deriving from movement and jump parameters defined for the character. For instance, the pair of values assigned to *takeoffSpeed* and *gravity* determine a specific maximum reachable height for a single jump. This set of constraints is fundamental for the generation of jump trajectories that are coherent with the physical model of the character.
As a preliminary step that guarantees the correctness of the reachability analysis, our method checks that there are no overlapping platforms in the level; in order to achieve that, for every platform in the level, we compute the related *Bounds Box*, which represents, for all the platform types, with the exception of the *MOVING* type, the margins of the platform texture; in the case of *MOVING* platforms, the *Bounds Box* specifies the area of space within which the platforms move; as a consequence, this prevents that other platforms are placed inside the range of movement of each *MOVING* platform. Figure 4.1 shows the *Bounds Boxes* for a *STATIC_FLOATING* and a *MOVING* platform.

### 4.3.1 Trajectory definition

A trajectory is formally defined as the path followed by a moving object. In our case, the trajectory represents the sequence of points in the bidimensional space followed by the feet of our character, considering the feet as a point-like object; this is because the landing on a platform occurs with the lower part of the character body; thus, in order to check if the character can reach a platform, the feet must be considered as the moving element that actually lands on the platform surface.

We chose to represent trajectories as piecewise defined functions; this choice derives from the fact that physics parameters related to movement can determine the variation of the laws of motion in different space intervals; for example, in the first space interval the character moves with a uniformly accelerated motion on the $x$ axis, whereas in the following space interval his speed is constant; as such, the functions describing the motion in the two intervals are different. In order to simplify the reachability check of a point in space (i.e. verify if a certain point is reachable with a trajectory), we chose to define each function composing a trajectory as a $y(x)$ (i.e. a function where the $x$ is the independent variable). Given these considerations, we define a trajectory $T$ as a piecewise defined function in the following form:

$$T([y_1(x), (x_{1,s}, x_{1,e})], [y_2(x), (x_{2,s}, x_{2,e})], ..., \\ ..., [y_{n-1}(x), (x_{n-1,s}, x_{n-1,e})], [y_n(x), (x_{n,s})], direction) \quad (4.5)$$

Or with the following equivalent compact form:

$$T(y_1(x), y_2(x), ..., y_{n-1}(x), y_n(x), direction) \quad (4.6)$$

Where:

- $y_i(x)$ is the i-th function composing the trajectory. Although there is no limit to the number of functions composing the trajectory, the various physical behaviors involve a maximum of four functions forming a trajectory.
- $x_{i,s}$ defines the starting $x$ value for the interval of the i-th function.
- $x_{i,e}$ defines the final $x$ value for the interval of the i-th function. It must be noted that the n-th function composing the trajectory has no final $x$ value because the associated physical behavior does not change after $x_{n,s}$.
- *direction* indicates the direction (i.e. left or right) of the trajectory. If the direction is *right*, the $x$ values of intervals increase (i.e. $x_{i,s} < x_{i,e}$), whereas if the direction is *left*, the $x$ values of intervals decrease (i.e. $x_{i,s} > x_{i,e}$).

In particular, polynomial functions of the first and second degree (i.e. lines and parabolas) are used. This formulation allows to check if a generic point $P(x_p, y_p)$ is reachable by a trajectory $T(y_0, y_1, ..., y_{n-1}, y_n, direction)$ with the following condition:

$$y_i(x_p) > y_p \quad (4.7)$$

Where:

- $x_{i,s} \leq x_p \leq x_{i,e}$ if $direction = right$
- $x_{i,e} \leq x_p \leq x_{i,s}$ if $direction = left$

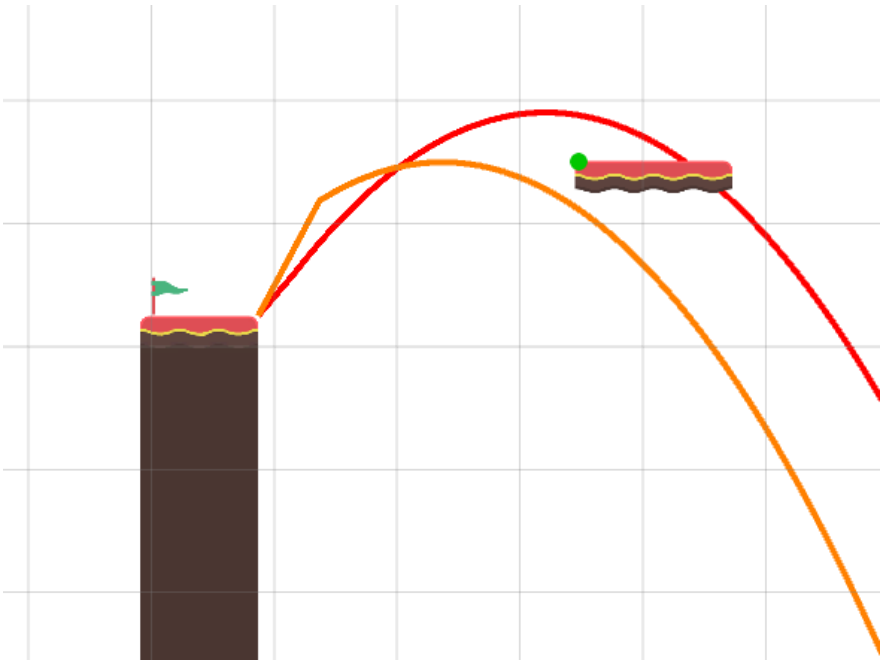

Figure 4.2: Two generated trajectories. Target point in green. The orange trajectory fails to reach the target point. The red trajectory is able to reach the target point.

Basically, we consider the function that covers the interval in which $x_p$ belongs and check if the function value is greater than $y_p$. The inequality 4.7, which is referred as *reachability check* from now on, formalizes the fact that, in order for the character to land on a specific point, he must be in a more elevated position when horizontally aligned with the target. Figure 4.2 shows two trajectories generated from the border of the platform on the left and a target point (in green), that is part of the border of the platform on the right; the orange trajectory fails the reachability check as it is below the target, whereas the red trajectory passes the reachability check as it is above the target.

### 4.3.2 Trajectory generation

In order to generate a trajectory, the following set of parameters is used:

- *startPoint (SP)*: the starting point of the trajectory. This is the position in which the jump is executed.
- *verticalTakeoffSpeed* ($v_{y,Takeoff}$): the character speed on the $y$ axis at the moment of takeoff.
- *horizontalTakeoffSpeed* ($v_{x,Takeoff}$): the character speed on the $x$ axis at the moment of takeoff.

- *direction*: the direction of the trajectory. Can be either *left* or *right*. The generation algorithm works by considering the horizontal movement input (i.e. -1 for *left* and 1 for *right*) constant for the entire trajectory.

- *isRunning*: boolean parameter indicating whether the run button is pressed at the moment of takeoff and for the entire trajectory. The generation algorithm works by considering the run button state constant for the entire trajectory.

- *jumpDuration*: how long the jump button is kept pressed. Given how the jump works if *isJumpDynamic* is set to true, this defines the time at which the movement speed on the $y$ axis is set to zero. This is an optional parameter: if not specified, the trajectory reaches the maximum reachable height.

Together with these parameters, which determine a specific trajectory in space, also the character movement and jump parameters are used for the generation. Based on how the physical model of the character has been configured, both the motion on the $x$ axis and on the $y$ axis can either be uniformly accelerated (or decelerated) or at constant speed; it follows that there are four possible combinations for the overall motion and the framework has been designed to recognise each one of them to output the proper trajectory.

#### 4.3.2.1 Accelerated (or decelerated) horizontal movement and decelerated vertical movement

If the absolute value of the horizontal speed at which the character takes off from the ground is less or greater than the maximum horizontal reachable speed in air ($maxAS$) then the horizontal movement is, respectively, uniformly accelerated or uniformly decelerated. For what concerns the movement on the $y$ axis, the character vertical speed is decelerated due to gravity. This combination of motion, if part of the trajectory to generate, is always in the first section of the piecewise defined function ($y_0$). In this context, the following equations describe, respectively, the variation of the $x$ and $y$ coordinates and the horizontal movement speed with respect to time:

$$x(t) = \frac{1}{2} airAcc \times t^2 + v_{x,P_0} \times t + x_0 \tag{4.8}$$

$$y(t) = \frac{1}{2} g \times t^2 + v_{y,P_0} \times t + y_0 \tag{4.9}$$

$$v_x(t) = airAcceleration \times t + v_{x,P_0} \tag{4.10}$$

Where:

- *airAcc* is the constant acceleration value for the horizontal movement. The sign of this value reflects the direction of the trajectory. This parameter is replaced by *airDec* if the horizontal takeoff speed is greater than *maxAS*, thus the character decelerates.
- $g$ is the constant deceleration value for the vertical movement.
- $P_0(x_0, y_0)$ is the point in space at which the overall motion starts following the set of physical laws at issue. For this combination of motion, this parameter correspond to the *startPoint* of the trajectory.
- $v_{x,P_0}$ is the horizontal speed of the character at $P_0$.
- $v_{y,P_0}$ is the vertical speed of the character at $P_0$.

As this type of motion always characterizes the first section of the trajectory, $v_{x,P_0}$ and $v_{y,P_0}$ are, respectively, the horizontal and vertical takeoff speed. The horizontal movement stays uniformly accelerated (or decelerated) until $v_x = maxAS$. Given that our goal is to obtain a function in the form $y(x)$, we observe that for this combination of motion it is not trivial nor direct to find an expression for $t$ that encapsulates the $x$ and allows us to achieve the desired expression. Furthermore, the fact that the time interval in which the horizontal movement is accelerated (or decelerated) is supposed to be small, we approximate the trajectory for this combination of motion as a line connecting $P_0(x_0, y_0)$ and $P_1(x_1, y_1)$, where the latter is the point at which the character starts moving at a constant speed ($v_x = maxAS$). Considering $t_0 = 0$ as the instant of jump, the following equation, derived from expression 4.10, is used to compute the time at which the horizontal movement speed becomes constant:

$$t_1 = \frac{maxAS - v_{x,P_0}}{airAcc} \tag{4.11}$$

As noted above, the *airAcc* parameter is replaced by *airDec* if the character horizontal movement is decelerated in the air. Using equation 4.11, we can obtain $P_1$ coordinates by substituting $t_1$ in expressions 4.8 and 4.9 as $x_1 = x(t_1)$ and $y_1 = y(t_1)$. We can then use the two points to define the approximating line with the following equations:

$$y(x) = \frac{y_1 - y_0}{x_1 - x_0}x + c \qquad c = y_0 - x_0\frac{y_1 - y_0}{x_1 - x_0} \tag{4.12}$$

#### 4.3.2.2 Constant speed horizontal movement and decelerated vertical movement

In this combination of motion, the horizontal movement is at constant speed, whereas the vertical movement is decelerated due to gravity and, as such, follows

equation 4.9. This type of motion characterizes the first section of a trajectory if the acceleration (or deceleration) is instantaneous, thus making the character reach the maximum horizontal speed in air immediately; otherwise, this combination of motion is typical of the second section of a trajectory, after the character has passed an acceleration phase (motion described in 4.3.2.1). The following equation describes the variation of the $x$ coordinate with respect to time:

$$x(t) = maxAS \times t + x_0 \tag{4.13}$$

Where $x_0$ is the $x$ coordinate of $P_0$, the point at which this combination of motion starts. In this case, we can easily obtain an expression of $t$ from equation 4.13 that allows us to achieve the proper $y(x)$ form of the motion at issue. The following equation defines how $t$ can be expressed in terms of $x$ (derived from equation 4.13):

$$t = \frac{x - x_0}{maxAS} \tag{4.14}$$

By substituting expression 4.14 in equation 4.9, and after expanding it to highlight the parameters that define each coefficient of the polynomial form, we obtain the final $y(x)$ expression:

$$y(x) = (\frac{g}{2maxAS^2})x^2 + (\frac{v_{y,P_0}}{maxAS} - \frac{g \times x_0}{maxAS^2})x + (\frac{g \times x_0^2}{2maxAS^2} - \frac{v_{y,P_0} \times x_0}{maxAS} + y_0) \tag{4.15}$$

#### 4.3.2.3 Constant speed horizontal movement and constant speed vertical movement

In this combination of motion, the movement on both the axes is at constant speed. This happens when the *hasTerminalFallingSpeed* parameter is set to true and, as such, during the descending phase of the jump the character reaches a constant falling speed. This type of motion is typical of the last section of a trajectory. We refer to equation 4.13 to describe the $x$ coordinate, whereas the variation of the $y$ coordinate with respect to time is expressed by the following equation:

$$y(t) = TFS \times t + y_0 \tag{4.16}$$

It must be noted that, as the character is falling down, the *terminalFallingSpeed* parameter ($TFS$) is always negative.
As for the case studied in 4.3.2.2, we can substitute $t$ with the expression 4.14 in equation 4.16 to obtain the $y(x)$ form that describes the motion:

$$y(x) = \frac{TFS}{maxAS}x + (y_0 - \frac{TFS}{maxAS}x_0) \tag{4.17}$$

#### 4.3.2.4 Accelerated (or decelerated) horizontal movement and constant speed vertical movement

The final case for the motion is the one in which the horizontal movement is uniformly accelerated (or decelerated) and the vertical movement happens at constant speed (terminal falling speed). This combination of motion is very unusual, as it occurs if the character takes more time to reach his maximum horizontal speed in air than to get to the jump descending phase and reach the terminal falling speed. Given the facts that the physics parameters configuration should avoid that this type of motion occurs, and that it is not direct to obtain a $y(x)$ expression to describe it, we approximate the motion in a very rough way by using two lines. The first line connects the trajectory starting point $P_0(x_0, y_0)$ with the highest point $P_1(x_1, y_1)$ (i.e. the apex of the jump) of the trajectory; the second line connects $P_1$ with the point $P_2(x_2, y_2)$ at which the horizontal movement speed becomes constant. It must be noted that the first section of such a trajectory would normally be approximated following the rules described in 4.3.2.1, but that would result in a single line connecting $P_0$ and $P_2$, excluding the apex of the jump from the generated trajectory. The following equations describe the two lines approximating the trajectory:

$$y_{P_0,P_1}(x) = \frac{y_1 - y_0}{x_1 - x_0}x + c_1 \qquad c_1 = y_0 - x_0\frac{y_1 - y_0}{x_1 - x_0} \tag{4.18}$$

$$y_{P_1,P_2}(x) = \frac{y_2 - y_1}{x_2 - x_1}x + c_2 \qquad c_2 = y_1 - x_1\frac{y_2 - y_1}{x_2 - x_1} \tag{4.19}$$

The module in charge of computing the physics constraints determines the times at which the type of motion changes, then they are ordered to define the proper sequence of motions for the trajectory.
In order to give a clearer view of the trajectory generation process, we provide a non-numeric example of the algorithm performing this task. We assume that the *hasTerminalFallingSpeed* parameter is set to true, thus the character falling speed caps at *TFS*; in addition, we suppose the jump is directed toward the right. First, the physics module computes the following two values:

- *timeToMaxSpeed* ($t_{MS}$): the time needed to reach the maximum horizontal speed in air.
- *timeToTerminalFallingSpeed* ($t_{TFS}$): the time needed to reach the terminal falling speed.

We assume that the following conditions hold true for this example:

$$t_{MS} \neq 0 \quad \wedge \quad t_{MS} < t_{TFS} \tag{4.20}$$

Given this fact, the trajectory is composed by three functions, each one defined in a specific interval of $x$ values. The first function $y_1$ describes a motion characterized by a uniformly accelerated horizontal movement and a uniformly decelerated

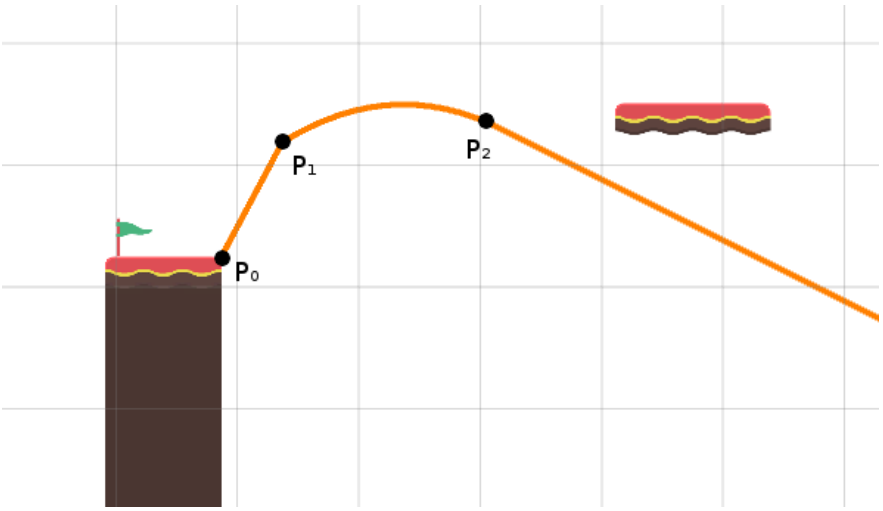


Figure 4.3: A trajectory composed by three functions. $P_0$, $P_1$ and $P_2$ (in black) are the starting points of each function.

(due to gravity) vertical movement (4.3.2.1). The second function $y_2$ describes a motion characterized by a constant speed horizontal movement and a uniformly decelerated (due to gravity) vertical movement (4.3.2.2). Finally, the third function $y_3$ describes a motion at constant speed on both the axes (4.3.2.3). The three points at which the functions respectively start being valid are $P_0(x_0, y_0)$, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. The algorithm generates the functions composing the trajectory $T(y_1(x), y_2(x), y_3(x), right)$ as follows:

$$y_1(x) = \frac{y_1 - y_0}{x_1 - x_0}x + c \qquad c = y_0 - x_0\frac{y_1 - y_0}{x_1 - x_0} \tag{4.21}$$

$$y_2(x) = (\frac{g}{2maxAS^2})x^2 + (\frac{v_{y,P_1}}{maxAS} - \frac{g \times x_1}{maxAS^2})x + (\frac{g \times x_1^2}{2maxAS^2} - \frac{v_{y,P_1} \times x_1}{maxAS} + y_1) \tag{4.22}$$

$$y_3(x) = \frac{TFS}{maxAS}x + (y_2 - \frac{TFS}{maxAS}x_2) \tag{4.23}$$

Figure 4.3 shows the final trajectory generated by the algorithm and the starting point of each function.

### 4.3.3 Types of trajectory

Our method evaluates the reachability by considering two platforms at once; the starting one, from which the character performs the jump; the target one, on which the character intends to land. If the analysis outcome is positive (i.e. the target platform is reachable from the starting platform), the result in the graph representation is a directed edge from the vertex corresponding to the starting platform to the one associated to the target platform; in case of non-reachability,

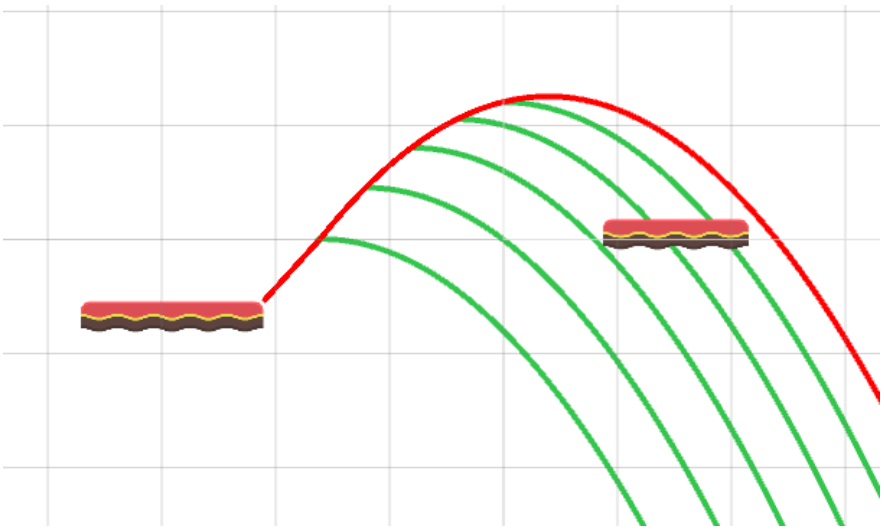

Figure 4.4: A boundary trajectory (in red) and a subset of its covered trajectories (in green).

the graph does not have an edge connecting the two vertices. Once the reachability is evaluated in a direction, the system also evaluates it in the opposite direction.
Based on how the two platforms are positioned, we identified four possible configurations, from which result four methods for the generation of the possible trajectories. It must be noted that we are considering *boundary* jump trajectories: trajectories reflecting the extreme approaches to the jump action (e.g. jumping from a stationary position without pressing the run button or jumping with a running start while pressing the run button). These trajectories assume a maximum jump duration (i.e. the player keeps the jump button pressed to maximize the jump height) and a constant horizontal input. Given that the player has control over the jump duration and the horizontal speed in the air, the boundary trajectories cover a wide set of possible player's actions. In addition, the boundary trajectories are generated from one of the two terminal points of the starting platform, which we call *optimal takeoff points*; this because, ideally, the player tries to jump in correspondence of these positions (i.e. performing a pixel perfect jump) in order to maximize the width of the trajectory and try to reach the most distant points from the starting platform. Thus, the boundary trajectories are, in fact, boundaries for all possible jumping approaches. Figure 4.4 shows a boundary trajectory in red and a subset of trajectories it covers (resulting from different jump duration values) in green. By counting the number of boundary trajectories that pass the reachability check, we are able to provide an estimation of the jump difficulty (see section 4.3.4). Note that, the framework does not employ any checks on whether the generated trajectories are blocked by other platforms; this is because its main goal is to provide feedbacks on human

authored content and designers can identify the issue using a specific tool of the framework to view the boundary trajectories (see Section 6.3). If this work is extended to include the automatic generation of levels that must satisfy specific properties (i.e. search-based content generation) then the issue of blocked trajectories should definitely be addressed.
Note that, our approach works by considering the platforms as static (i.e. with fixed positions), but, as our platformer implementation also offers *MOVING* platforms, the reachability analysis is designed to effectively handle the cases in which one or even both the two considered platforms are of *MOVING* type. In general, the *Bounds Box* (i.e. the area of space within which the platform moves) of a *MOVING* platform is discretized according to the direction of its trajectory. If the platform moves with a horizontal trajectory, the discretization step is equal to the platform width; If the platform moves with a vertical trajectory, the discretization step is equal to the platform height. This means that the reachability analysis is performed considering all the possible positions of the *MOVING* platform inside its *Bounds Box* using the platform width or height as a step between adjacent positions. An ad hoc method is followed in the positions discretization if both the platforms are *MOVING*; in fact, the first step in this scenario is to compute each platform movement half-period, which is the time needed to go from the *beginningPoint* to the *endPoint*. If the two platforms have the same half-period value, they are synchronized, meaning that their mutual positions stay the same at every movement cycle. Thus, if the two platforms are synchronized, the reachability analysis performs the discretization of their positions with respect to time, generating the possible trajectories for their mutual positions over a time interval equal to their period. In case the two *MOVING* platforms are not synchronized, the discretization process is executed normally (i.e. using their width or height as a step) and all the possible combinations of positions are tested. In order to determine the boundary trajectories if at least one of the two platforms considered in the reachability analysis is of *MOVING* type, for each one of the discretized positions, the jump difficulty is evaluated (see 4.3.4) and the boundary trajectories are the ones corresponding to the position with the lowest jump difficulty.
As a preliminary step for the reachability analysis, the physics module computes the maximum reachable height with a jump, taking into account the involved parameters (e.g. if the character can perform a double jump or not); if the target platform is above the starting platform and the vertical distance between them is greater than the maximum reachable height then it is not possible to reach the target, thus the analysis stops and returns a negative output. Next, the optimal takeoff and landing positions (i.e. points used for the reachability check) are computed from the starting and target platforms parameters; as we are considering the terminal points of each platform, we come up with the following:

- *startLeft (SL)*: left terminal point of the starting platform.

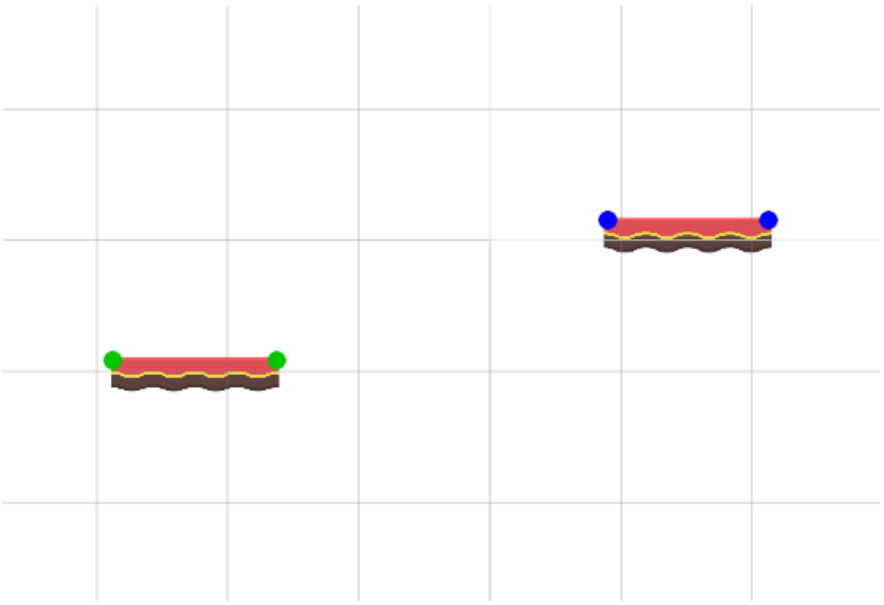

Figure 4.5: Starting platform on the left and target platform on the right. Optimal takeoff points highlighted in green and landing points highlighted in blue.

- *startRight (SR)*: right terminal point of the starting platform.
- *targetLeft (TL)*: left terminal point of the target platform.
- *targetRight (TR)*: right terminal point of the target platform.

Figure 4.5 shows the optimal takeoff points and landing points for a pair of platforms.

#### 4.3.3.1 *Trivial* trajectories

If one of the two platforms is above the other and the projections on the $x$ axis of its terminal points overlap with the respective projections of the other platform (Figure 4.6), then we identify this case as a *Trivial* trajectory scenario. This is because, theoretically, there is no possibility of failing the jump in this situation, as no gap is present between the two platforms. Equation 4.24 formalizes the condition for a *Trivial* trajectory when the target is above the starting platform. Equation 4.25 formalizes the condition for a *Trivial* trajectory when the target is below the starting platform.

$$SL.x \leq TL.x \leq SR.x \quad \vee \quad SL.x \leq TR.x \leq SR.x \tag{4.24}$$

$$TL.x \leq SL.x \leq TR.x \quad \vee \quad TL.x \leq SR.x \leq TR.x \tag{4.25}$$

In this scenario, it is not necessary to actually generate a trajectory. If the target platform is above the starting one, the character just needs to perform a jump

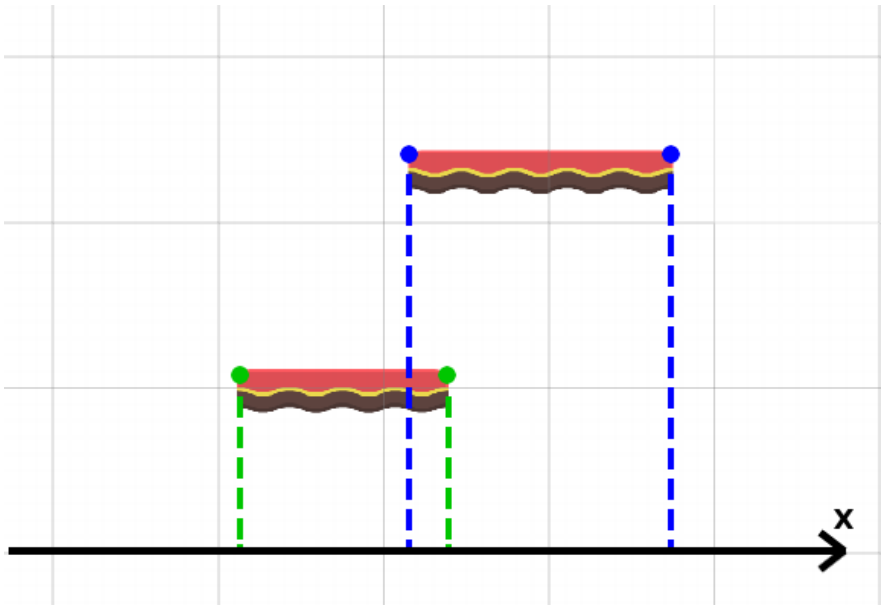


Figure 4.6: Two platforms with overlapping projections.

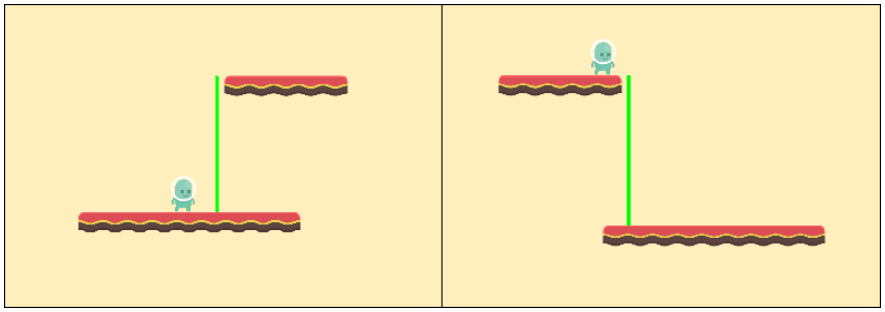

Figure 4.7: Boundary trajectories (in green) for two *Trivial* configurations.

while standing still and slightly move toward the target when he reaches the same height; if the target platform is below the starting one, the character just needs to fall straight to land on the target. Figure 4.7 illustrates two instances of *Trivial* trajectories.

#### 4.3.3.2 *Simple* trajectories

If the two platforms projections are not overlapping, then we identify this case as a *Simple* trajectory scenario. This is the most common configuration in 2D platformers. The gap between the two platforms represents the obstacle the character needs to overcome to move from one platform to the other. Figure 4.8 shows an example of *Simple* configuration. Equation 4.26 formalizes the condition for this scenario.

$$TL.x > SR.x \quad \vee \quad TR.x < SL.x \tag{4.26}$$

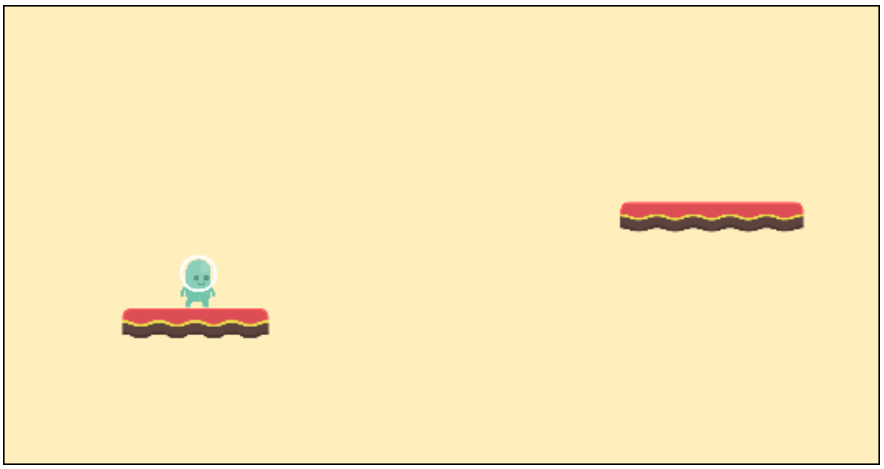

Figure 4.8: An example of *Simple* configuration. The character, standing on the starting platform (left), needs to jump over the gap to reach the target platform (right).

For what concerns the trajectory generation, the start point ($SP$) of the trajectory and the target point used for the reachability check are determined according to how the two platforms are positioned. If the target is to the right of the starting platform ($TL.x > SR.x$) then $SP = SR$ and the target point is equal to $TL$; if the target is to the left of the starting platform ($TR.x < SL.x$) then $SP = SL$ and the target point is equal to $TR$. A big distinction is made by the framework according to the character jumping mode. If the jumping mode is set to *SINGLE_ JUMP*, then a maximum of four boundary trajectories are generated, covering the majority of possible approaches to the jump. Using Figure 4.9 as a reference, the following four boundary trajectories are defined:

- *Green trajectory*: corresponds to the trajectory with $v_{y,Takeoff} = ToS$, $v_{x,Takeoff} = 0$ and $isRunning = false$. This case represents the approach to the jump without a running start (i.e. horizontal movement speed equal to zero) and without pressing the run button while performing the trajectory. This trajectory is generated only if the movement mode is not set to *RUN_ ONLY*.

- *Yellow trajectory*: corresponds to the trajectory with $v_{y,Takeoff} = ToS$, $|v_{x,Takeoff}| > 0$ and $isRunning = false$. $v_{x,Takeoff}$ is computed by considering a running start on the 90% of the starting platform length without pressing the run button. This case represents the approach to the jump with a running start and without pressing the run button while performing the trajectory. This trajectory is generated only if the movement mode is not set to *RUN_ ONLY*.

- *Orange trajectory*: corresponds to the trajectory with $v_{y,Takeoff} = ToS$,

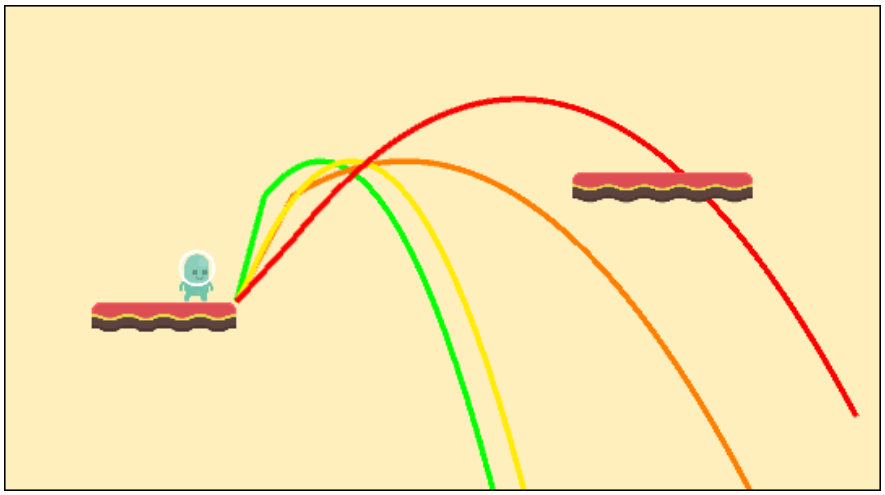

Figure 4.9: The four boundary trajectories generated in case of *SINGLE_JUMP* and *WALK_AND_RUN* mechanics.

$v_{x,Takeoff} = 0$ and $isRunning = true$. This case represents the approach to the jump without a running start, but with the pression of the run button while performing the trajectory. This trajectory is generated only if the movement mode is not set to *WALK_ONLY*.

- *Red trajectory*: corresponds to the trajectory with $v_{y,Takeoff} \geq ToS$, $|v_{x,Takeoff}| > 0$ and $isRunning = true$. $v_{x,Takeoff}$ is computed by considering a running start on the 90% of the starting platform length while pressing the run button. If the $isVerticalTakeoffSpeed$ parameter is set to true, $v_{y,Takeoff}$ is based on the $v_{x,Takeoff}$ value and computed using equations 4.1 and 4.2. This case represents the approach to the jump with a running start and with the pression of the run button while performing the trajectory. This trajectory is generated only if the movement mode is not set to *WALK_ONLY*.

The *direction* parameter for the trajectories is set according to the position of the target with respect to the starting platform. Basically, the trajectories color acts as an indicator of their starting horizontal speed and running settings.
If the character jumping mode is set to *DOUBLE_JUMP*, then, for each one of the four boundary trajectories described above, two additional trajectories are generated only if the respective initial trajectory (i.e. describing the first jump) fails the reachability check: the first starting at the apex of the first jump and the second starting at the same $y$ coordinate of the first jump takeoff point on the descending phase of its trajectory. The double jump trajectories parameters are defined as follows:

- $v_{y,Takeoff} = DJS$.

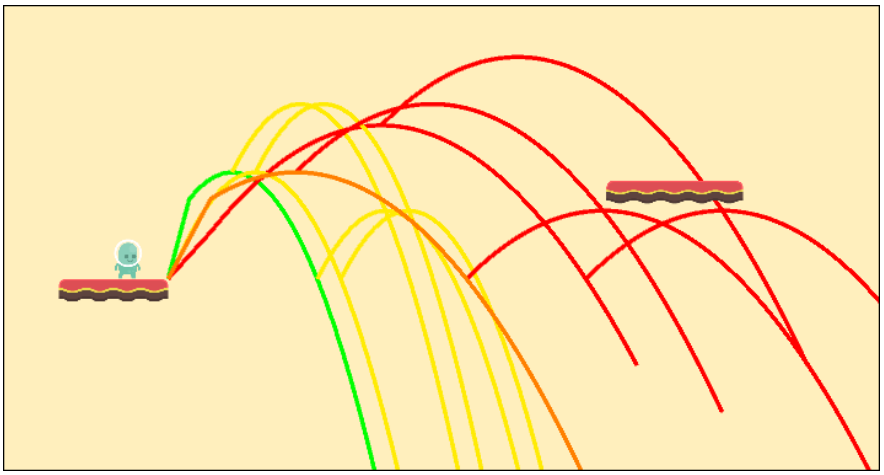

Figure 4.10: The maximum number of boundary trajectories (twelve) that can be generated in case of *DOUBLE_JUMP* and *WALK_AND_RUN* mechanics.

- $v_{x,Takeoff}$ is set to the horizontal movement speed reached at that point on the respective first jump trajectory.
- The *isRunning* parameter keeps the same value assigned to the respective first jump trajectory.
- The *direction* paramater keeps the same value assigned to the respective first jump trajectory.

These two trajectories guarantee a good spatial coverage both horizontally and vertically. Figure 4.10 shows an instance of *Simple* configuration with all the twelve possible trajectories generated.

#### 4.3.3.3 *Falling* trajectories

If the target platform is below the starting one and the projections of the former are completely contained in the projections of the latter, then we identify this case as a *Falling* trajectory scenario. Figure 4.11 shows an example of *Falling* configuration. Equation 4.27 formalizes the condition for this configuration.

$$TL.x > SL.x \quad \wedge \quad TR.x < SR.x \tag{4.27}$$

As a preliminary step for trajectories generation, the framework determines the side (i.e. left or right) of the starting platform from which the character performs the fall. This is done by computing the euclidean distances between the terminal points on the same side of the two platforms; in order to maximize the chances of landing on the target, the side with the smaller distance is chosen. The target point used for the reachability check is the terminal point of the target platform

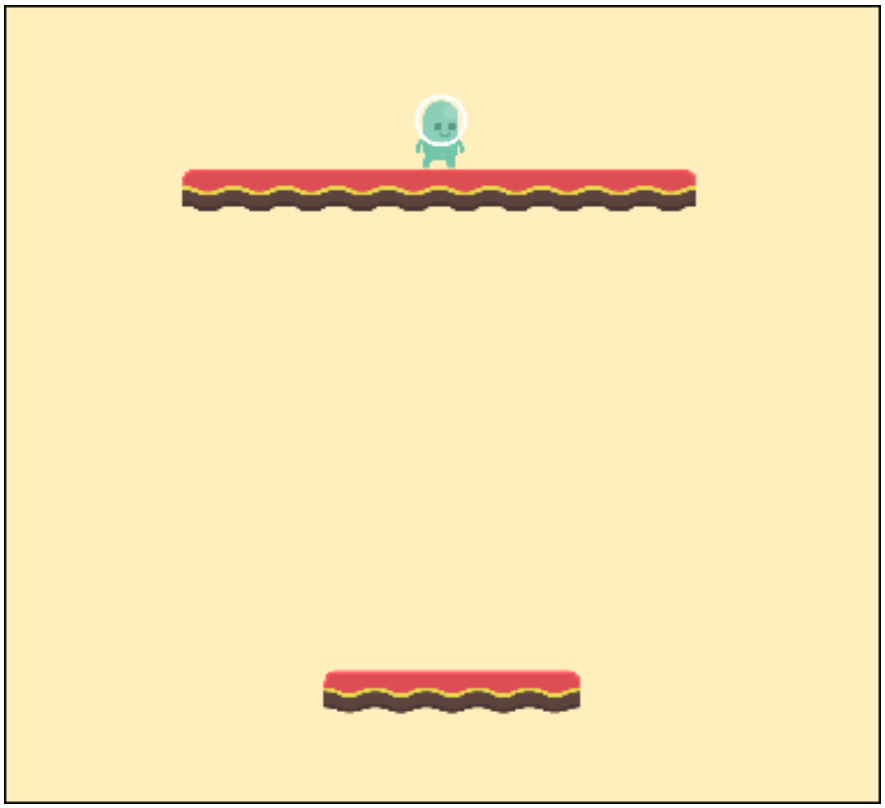

Figure 4.11: An example of *Falling* configuration. The character, standing on the starting platform (top), needs to fall down and land on the target platform (bottom).

on the chosen side.
According to the character jumping mode, two different approaches are followed for the trajectories generation. If the jumping mode is set to *SINGLE_JUMP*, we consider a free fall trajectory without the use of the jump. Also, we want the character to keep the horizontal distance from the target platform to a minimum while falling next to the starting platform. As a consequence, the start point *SP* of the trajectories, assuming the left side as the one chosen to start the fall, is computed as follows:

$$SP = (SL.x - \frac{charWidth}{2}, SL.y - platHeight - charHeight) \tag{4.28}$$

Where:

- *charWidth* is the character width, measured in $u$.
- *charHeight* is the character height, measured in $u$.
- *platHeight* is the starting platform height, measured in $u$.

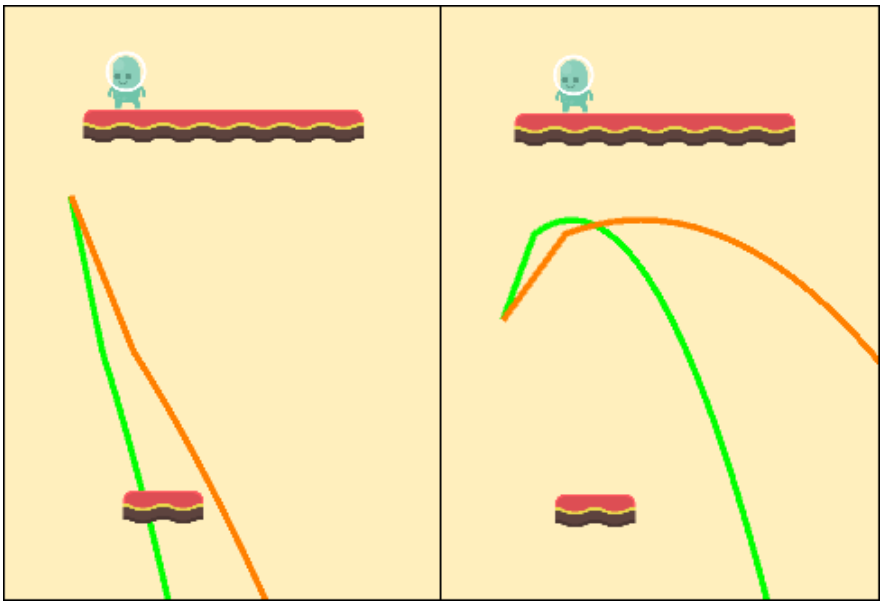

Figure 4.12: Boundary trajectories generated for the same *Falling* configuration with two different mechanics settings. On the left, *SINGLE_JUMP* and *WALK_AND_RUN* settings. On the right, *DOUBLE_JUMP* and *WALK_AND_RUN* settings.

It must be noted that *SP* represents the position at which the character can start moving horizontally toward the target platform without colliding with the border of the starting platform. In addition, the framework needs to know the vertical movement speed at *SP* to define the correct trajectories, thus the following equations are used:

$$t_{SP} = \sqrt{2\frac{-verticalOffset}{g}} \tag{4.29}$$

$$v_{y,SP} = g \times t_{SP} \tag{4.30}$$

Where $t_{SP}$ is the time it takes to the character to reach *SP*, whereas $verticalOffset = platHeight + charHeight$. It must be noted that, if the *hasTerminalFallingSpeed* parameter is set to true and the time needed to reach the terminal falling speed is less or equal to $t_{SP}$, then $v_{y,SP} = TFS$. Given these considerations, the following two boundary trajectories can be generated in case of *SINGLE_JUMP*:

- *Green trajectory*: corresponds to the trajectory with $v_{y,Takeoff} = v_{y,SP}$, $v_{x,Takeoff} = 0$ and $isRunning = false$. This trajectory is generated only if the movement mode is not set to *RUN_ONLY*.
- *Orange trajectory*: corresponds to the trajectory with $v_{y,Takeoff} = v_{y,SP}$, $v_{x,Takeoff} = 0$ and $isRunning = true$. This trajectory is generated only if the movement mode is not set to *WALK_ONLY*.

The trajectories *direction* is the opposite of the side chosen to start the fall (e.g. if the left side of the starting platform is chosen then the trajectories *direction* is equal to *right*).
If the jumping mode is set to *DOUBLE_JUMP*, the character can exploit the second jump to have a better control in the air and maximize his chances of landing on the target. As such, the basic assumption for the definition of the boundary trajectories is that the character performs a first jump with $jumpDuration = minimumJumpDuration$ to start the free fall and to enable the double jump; then, when he is at half the vertical distance between the two platforms, he performs the second jump, directed toward the target. As a consequence, the *startPoint* of the trajectories, assuming the left side as the one chosen to start the fall, is computed as follows:

$$SP = (SL.x - \frac{charWidth}{2}, \frac{SL.y - platHeight + TL.y}{2}) \tag{4.31}$$

Equations 4.29 and 4.30 are used to compute $v_{y,SP} = 0$, but in this case $verticalOffset = SL.y + heightMinJump - SP.y$, where *heightMinJump* is the height reached with the first jump. As in the *SINGLE_JUMP* setting, the same considerations about the terminal falling speed also hold true in this case. The two trajectories that can be generated in the *DOUBLE_JUMP* case are the same as in the *SINGLE_JUMP* situation, with the difference being their $v_{y,Takeoff}$, that is equal to *DJS*. Figure 4.12 compares the trajectories generated in case of *SINGLE_JUMP* and *DOUBLE_JUMP*.

#### 4.3.3.4 *Reentrant* trajectories

If the target platform is above the starting one and the projections of the latter are completely contained in the projections of the former, then we identify this case as a *Reentrant* trajectory scenario. In this configuration, the character needs to jump and follow a reentrant trajectory to land on the target platform. Figure 4.13 shows an example of *Reentrant* configuration. Equation 4.32 formalizes the condition for this configuration.

$$TL.x < SL.x \quad \wedge \quad TR.x > SR.x \tag{4.32}$$

As the character needs to follow a reentrant path while in the air, having access to a second jump is mandatory to succeed in reaching the target platform. As such, if the jumping mode is not set to *DOUBLE_JUMP*, the reachability analysis returns a negative output without proceeding to the trajectories generation step.
Similarly to the *Falling* configuration case discussed in 4.3.3.3, the side of the starting platform from which the character takes off is defined by computing the distances between the platforms terminal points on the same side and choosing the one with the smaller value. The possible first jump trajectories are the same

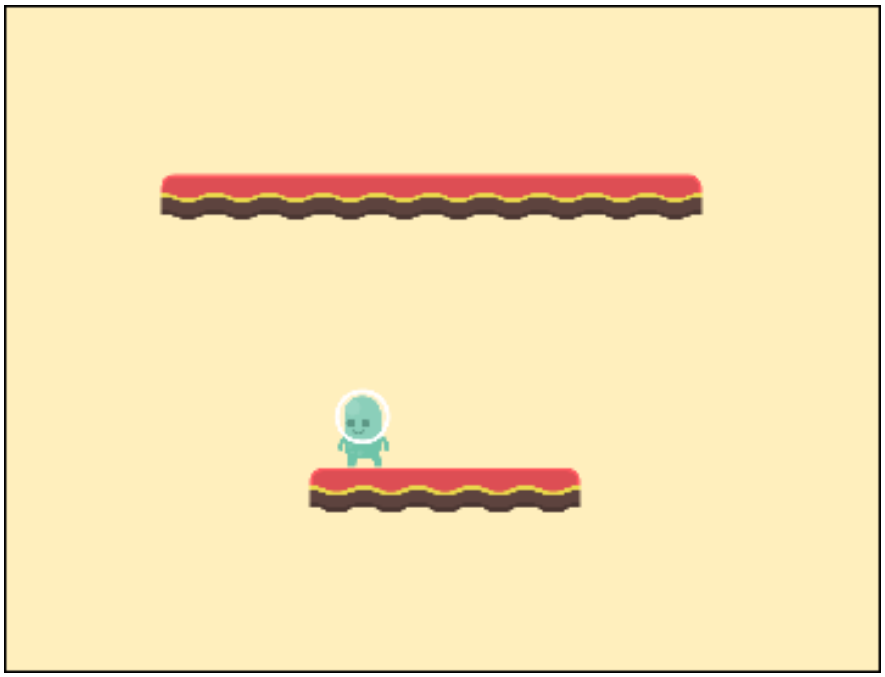

Figure 4.13: An example of *Reentrant* configuration. The character is standing on the starting platform (bottom), with the target platform above him.

described in 4.3.3.2 for the takeoff; in addition, the first jump trajectories *jumpDuration* is set to guarantee the maximum reachable height and at the same time avoid the collision with the upper platform; this is because, if the character collides with the target platform, he is pushed down and unable to follow the generated trajectory. Ideally, the character needs to change his direction while in air and perform the second jump in correspondence of the first jump apex. Clearly, if the character acceleration is not instantaneous, it takes some time to change direction as the air turn acceleration ($airTAcc$) influences the character horizontal speed; while $airTAcc$ is in effect, the character still moves in the direction of the first jump trajectory and this determines a horizontal offset for the start point $SP$ of the double jump trajectory. Given these considerations, we assume that the character starts changing his movement direction at the first jump apex and that he performs the second jump at the time his horizontal speed becomes zero (entire $airTAcc$ effect). With these assumptions, we make the following computations:

$$t_{ZS} = \frac{-v_{x,Apex}}{airTAcc} \tag{4.33}$$

$$horizontalOffset = \frac{1}{2} airTAcc \times t_{ZS}^2 + v_{x,Apex} \times t_{ZS} \tag{4.34}$$

Where:

- $t_{ZS}$ is the time needed for the horizontal speed value to reach zero due to

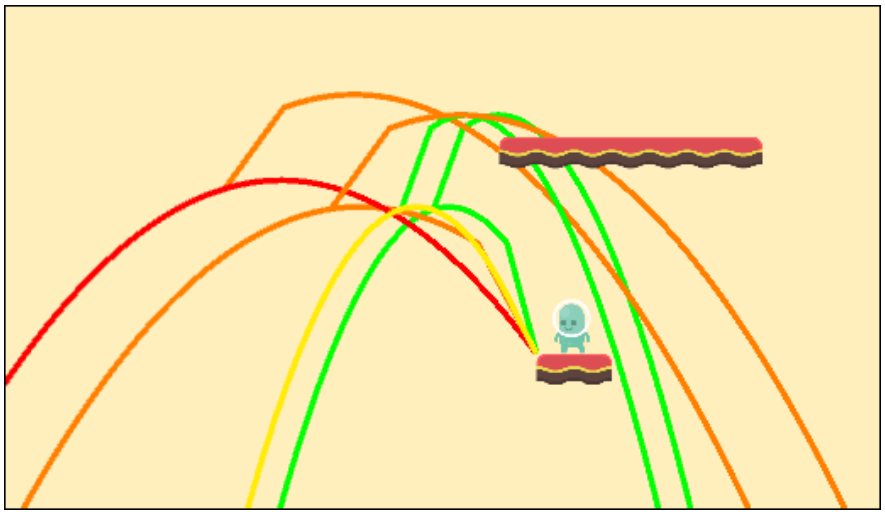

Figure 4.14: The set of boundary trajectories generated for a *Reentrant* configuration.

*airTAcc*. Measured in $s$.

- $v_{x,Apex}$ is the horizontal speed of the character at the first jump apex. This parameter has the opposite sign of *airTAcc*, hence the minus in the fraction. Measured in $u/s$.
- *horizontalOffset* is the horizontal distance travelled by the character while changing direction. Measured in $u$.

As such, the second jump $SP$ is taken on the first jump trajectories with *horizontalOffset* distance from the apex. To sum up, the parameters that define the double jump trajectories are the following:

- $v_{y,Takeoff} = DJS$
- $v_{x,Takeoff} = 0$
- The *isRunning* parameter keeps the same value assigned to the respective first jump trajectory.
- The *direction* is the opposite of the one assigned to the respective first jump trajectory.

Figure 4.14 shows the trajectories generated for a *Reentrant* configuration, in case of $WALK\_AND\_RUN$ movement mode. Figure 4.15 illustrates the trajectories generated for a *Reentrant* configuration that causes the algorithm to generate multiple times a specific trajectory (the first jump green one) to avoid the character collision with the upper platform, in case of $WALK\_AND\_RUN$ movement mode.

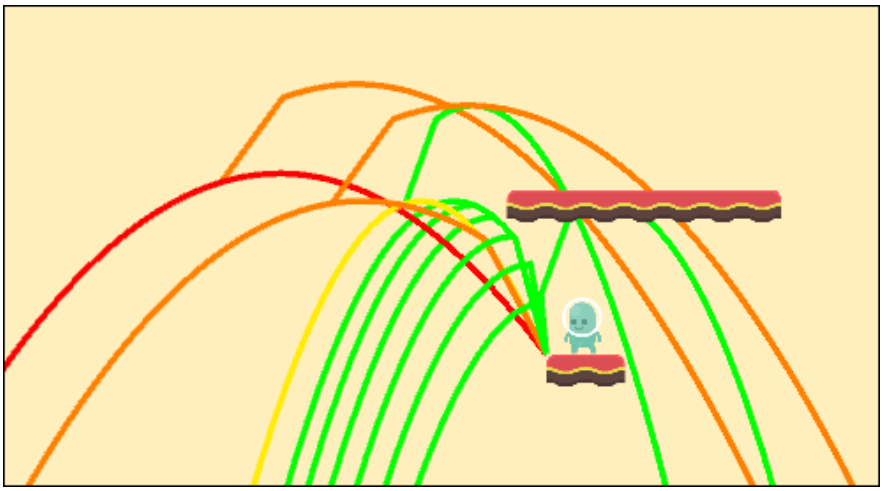

Figure 4.15: The set of boundary trajectories generated for a *Reentrant* configuration. The first jump green trajectory has been generated multiple times to avoid the collision with the upper platform.

### 4.3.4 Jump difficulty evaluation

As we previously mentioned, the goal of the reachability analysis is not only to conclude if a platform can be reached from another platform, but also to provide information on the jump difficulty. As such, the reachability analysis performed on two platforms outputs a directed edge which comes with two indicators: a *weight* and a *probability*. The edge probability and its meaning in the context of jumping from one platform to another are discussed in depth in Chapter 5. The edge weight provides an evaluation of the jump difficulty based on the generated boundary trajectories. The formula for the computation of the edge weight is the following:

$$Weight = \frac{generatedTrajectories}{trajectoriesReachingTarget} \times DC \tag{4.35}$$

Where:

- *generatedTrajectories* is the number of generated trajectories for the current two platforms configuration.
- *trajectoriesReachingTarget* is the number of generated trajectories that pass the reachability check (i.e. allow the character to reach the target platform).
- *DC* is the difficulty coefficient measuring how much context-specific properties (e.g. if the target platform is *MOVING*) change the overall jump difficulty.

The edge weight is clamped between 1 and 12, with 1 indicating trivial jumps and 12 indicating maximum difficulty jumps. This interval of values has been

chosen for two reasons; first, to give designers a range that is not too wide, thus making their life easier in classifying different jumps; secondly, as the maximum number of trajectories that can be generated for a *Simple* configuration is twelve, the extreme case that still guarantees the reachability is the one in which only one of the twelve trajectories passes the reachability check, resulting in $generatedTrajectories/trajectoriesReachingTarget = 12$.
For what concerns $DC$, it is computed as follows:

$$DC = MOVING_{Start} \times MOVING_{Target} \times FADING_{Start} \times Falling_{Traj} \quad (4.36)$$

Each one of the factor in equation 4.36 depends on platforms specific properties and on their configuration (i.e. trajectory type). The next paragraphs review the meaning of these factors and how they are computed.

#### 4.3.4.1 *MOVING* platforms effect

If one or both the two platforms is of *MOVING* type, the jump difficulty is clearly influenced by their speed of movement. Intuitively, the faster a platform moves, whether it is the starting one or the target, the more difficult it is for the character to successfully perform the jump and land on the target. Given this consideration, we consider the movement speed parameter of *MOVING* platforms ($movSp$) to define the $MOVING_{Start}$ and $MOVING_{Target}$ factors of equation 4.36 as follows:

$$MOVING_{Start} = \sqrt{movSp_{Starting} + 1} \quad (4.37)$$

$$MOVING_{Target} = \sqrt{movSp_{Target} + 1} \quad (4.38)$$

Where $movSp_{Starting}$ and $movSp_{Target}$ are the movement speed parameters of the starting and target platforms respectively. The $+1$ under the square root is used so that the two factors are always greater than 1, even if $movSp \leq 1$. The jump difficulty is directly proportional to the movement speed of *MOVING* platforms, but we observed that their relation is not linear, thus we employed the square root to slow the factors growth.
If the starting or the target platform is not of *MOVING* type, the respective factor is set to 1, having no influence on $DC$.

#### 4.3.4.2 *FADING* platforms effect

If the starting platform is of *FADING* type, the character has a limited amount of time to perform a jump and land on another platform before the platform he is standing on disappears. As such, the shorter the time needed for the *FADING* platform to disappear, the less time the character has to accurately jump and thus the more difficult it is to safely reach another platform. Due to this consideration,

having a *FADING* platform as target has no influence over the jump difficulty. The following equations are used to compute the $FADING_{Start}$ factor:

$$t_{Dis} = \frac{alphaTotalDecreaseToDisappear}{fadSp} \tag{4.39}$$

$$FADING_{Start} = \frac{1}{\log_{10}(t_{Dis} + 1)} \tag{4.40}$$

Where:

- $t_{Dis}$ is the time needed for the *FADING* platform to disappear since the collision with the character. This value is always greater than 0. Measured in $s$.
- *alphaTotalDecreaseToDisappear* is the alpha decrease that cause the *FADING* platform to disappear. As we set the alpha threshold for disappearing to 0.1 and the alpha value starts at 1, the *alphaTotalDecreaseToDisappear* is equal to 0.9.
- *fadSp* is the *FADING* platform parameter that defines the rate at which the alpha value decreases. Measured in $alpha/s$.

The logarithm in equation 4.40 allows to obtain a difficulty factor that is high when $t_{Dis}$ is close to 0, whereas when $t_{Dis}$ becomes bigger the factor is slightly greater than 1. In fact:

$$t_{Dis} = 0.9s \rightarrow FADING_{Start} = \frac{1}{\log_{10}(1.9)} \approx 3.6$$

$$t_{Dis} = 3s \rightarrow FADING_{Start} = \frac{1}{\log_{10}(4)} \approx 1.66$$

If $t_{Dis} > 9s$, then equation 4.40 is not applied and $FADING_{Start}$ is automatically set to 1, as we assume that the time needed for the starting platform to disappear is long enough to not affect the jump difficulty. If the starting platform is not of *FADING* type, the $FADING_{Start}$ factor is set to 1, having no influence on the *DC*.

#### 4.3.4.3 *Falling* trajectories effect

We observed that the difficulty related to *Falling* trajectories configurations is influenced by at least two parameters: the target platform width and the vertical speed at which the character falls. Intuitively, the wider the target the easier it is for the character to land on it; by contrast, the faster he falls, the more difficult it

is to reach the target platform. Given these considerations, the following equation defines the difficulty factor assigned to *Falling* trajectories:

$$Falling_{Traj} = \frac{|v_y|}{2\log_5(length_{Target} + 1)} \tag{4.41}$$

Where:

- $v_y$ is the vertical speed value defined with equation 4.30. Measured in $u/s$.
- $length_{Target}$ is the *length* parameter of the target platform, indicating the number of tiles that are horizontally placed next to each other to form the platform.

The jump difficulty is considered linearly dependent from the vertical speed, whereas the influence of the target length tends to fade as it increases, hence the use of the logarithm. In addition, if the target platform is formed by more than four tiles (i.e. $length > 4$), then this has the effect of slightly easing the jump. If the two platforms are not in a *Falling* configuration, then the $Falling_{Traj}$ factor is set to 1, having no influence on $DC$.

## 4.4 Summary

In this chapter we discussed our approach for designing 2D platformer levels. First, we examined the conceptual model for game levels. Next, we introduced and described our framework. We presented the structural features offered by the platformer implementation. Finally, we reviewed our method for the trajectories generation and discussed how it supports the evaluation of the jumps difficulty.

# Chapter 5

# Modeling level success probability

In this chapter, we introduce how we employed noise functions to estimate the jumps probability of success. Next, we examine the experimental data we collected to validate our approach for single jumps. Finally, we discuss how the estimate for single jumps was extended to evaluate the difficulty and probability of success of a level.

## 5.1 Jumps start point noise functions

Since players can tackle a jump in different ways, there are multiple trajectories that they can follow when performing a jump between two platforms. To evaluate the probability of success of a jump, we generate for each successful boundary jump trajectory (i.e. reaching the target landing point; Section 4.3.3) a set of random takeoff points. These are generated around the optimal position using a noise function based on the following parameters:

- the average reaction time of the player $Rt$ which defines the quickness in reacting to game events and, as such, it is critical for the survival of the character. We employed this parameter with values ranging from 0.01s to 1s.

- the player skill value $Ps$ which defines how good at playing 2D platformers the player is, the higher the better. We employed this parameter with values ranging from 1 to 50. For some noise models, this parameters is weighted by $k$, a factor determined empirically.

- the absolute value of either the horizontal speed or the vertical speed at which the character approaches the jump: $|v_x|$ or $|v_y|$; the choice between the two components depends on the jump type, whereas the speed value is bound to the considered optimal trajectory.

Intuitively, the higher $Rt$ and $|v_x|$ (or $|v_y|$) are the more the noise affects the sample trajectories, moving them away from the optimal ones. By contrast, the higher $Ps$ is the lower the effect of the noise becomes.
For each random takeoff point, we generate a sample trajectory. Thus, the probability of success of the jump is given by the percentage of successful sample trajectories. In addition, the probability is weighted by the inverse of the difficulty coefficient $DC$ (Section 4.3.4) to take into consideration the different properties of the two platforms. It must be noted that, for unsuccessful optimal jump trajectories, the framework counts the respective sample trajectories as unsuccessful without actually generating them. Furthermore, if the target platform is of *MOVING* type, the probability is computed as the average of the estimated probabilities obtained considering the target optimal position (i.e. the position resulting in the lowest difficulty) and the adjacent ones.
The framework supports three noise functions, each one with its peculiar features: *Uniform*, *Gaussian without resampling* and *Gaussian with resampling*. Note that, since *Trivial* configurations do not involve the generation of trajectories (see Section 4.3.3.1), they make no use of noise functions for jumps, thus their probability of success is simply computed as the inverse of the directed edge weight (i.e. the inverse of the difficulty evaluation).

### 5.1.1 *Simple* trajectory jumps

If the two platforms involved in the jump for which we want to estimate the probability of success are in a *Simple* trajectory configuration (Section 4.3.3.2), the noise function affects the $x$ coordinate of the optimal takeoff point to generate random start points for the sample trajectories. When the *Uniform* noise model is used, the random start points are selected with a uniform probability distribution from a interval of width $\delta = 2(|v_x| + \epsilon)/Ps$, where $\epsilon$ is a constant value used to avoid that $\delta$ is zero in case $v_x = 0$. The formula to compute $\delta$ reflects the fact that, the faster the character is approaching the jump, the more likely he is to perform the jump far from the optimal point. By contrast, the higher the player skill is, the more likely she is to perform the jump close to the optimal point. If we consider the extreme case of jumping from a stationary position ($v_x = 0$), a skilled player should always be able to jump from the optimal takeoff point. Note that $\delta$ caps at half the starting platform length.
Using Figure 5.1 as a reference, we see the four boundary trajectories (defined with different parameters, see Section 4.3.2) considered for a *Simple* jump scenario. The red trajectories fail to reach the target platform, whereas the green one is successful. The optimal takeoff point $P^*(x^*, y^*)$ is highlighted in blue. The interval $\delta$ associated to the successful optimal trajectory is displayed in black. Figure 5.2 shows the sample trajectories obtained using the *Uniform* noise function to generate the random start points: the red trajectories are unsuccessful, whereas the green ones are successful. Given the number of successful sam-

ple trajectories and the ones that are unsuccessful (also due to the unsuccessful boundary trajectories) the estimated probability of success for the jump is 0.075, which is uninfluenced by the difficulty coefficient in this scenario.

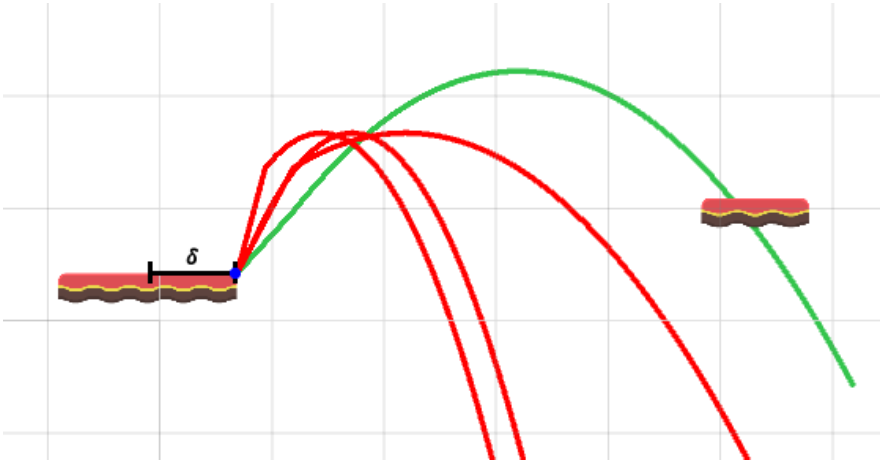


Figure 5.1: The boundary trajectories for a *Simple* jump configuration. The only successful boundary trajectory is the one in green. The optimal takeoff point is highlighted in blue. The interval width $\delta$ associated to the successful trajectory is displayed in black.

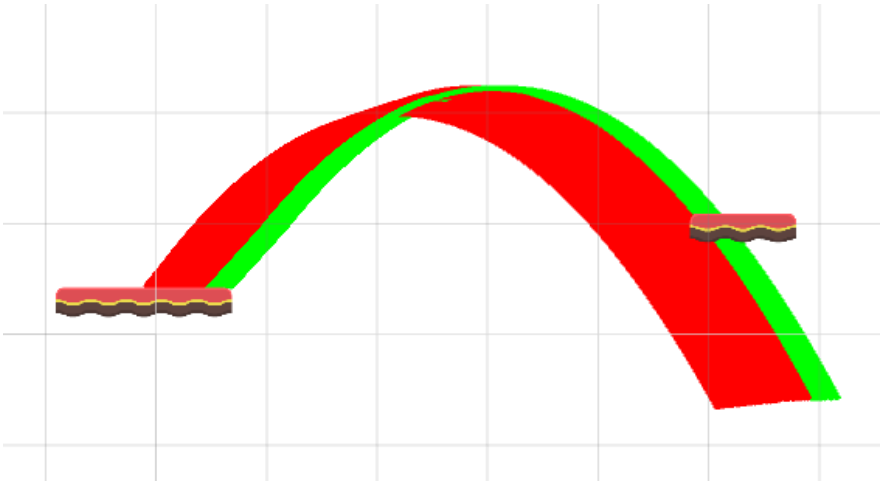

Figure 5.2: The generated sample trajectories for the scenario of Figure 5.1 using the *Uniform* noise function.

When the *Gaussian without resampling* or the *Gaussian with resampling* noise function is used, the sample trajectories random takeoff points are sampled from a Gaussian distribution. These distributions have a standard deviation $\sigma = (|v_x| + \epsilon)/(k \times Ps)$ and a mean value $\mu = x^* - Js \times Rt \times |v_x|$, where $x^*$

is the $x$ coordinate of the optimal takeoff point and $Js$ is either equal to 1 or -1 if the jump direction is, respectively, *right* or *left*. If $\mu$ exceeds the platform half then it is set to be equal to the $x$ coordinate of the platform center. Since in this case we sample without bounds, the random takeoff points can be off the starting platform. The key difference between the two Gaussian noise functions lies in how they manage the above-mentioned issue: the *Gaussian without resampling* noise function counts the takeoff points that are off the starting platform as unsuccessful sample trajectories, whereas the *Gaussian with resampling* noise function generates a new random takeoff point until it is on the starting platform. By doing that, the *Gaussian without resampling* noise function takes into account the possibility of failing a jump because the player pressed the jump button too late while moving toward the optimal takeoff point. The only instance for which the random takeoff points are constrained to be on the platform is for $v_x = 0$: we assume that from a stationary position there is no possibility of failing a jump.
In Figure 5.3, we see the four boundary trajectories considered for a *Simple* jump scenario. The red trajectories fail to reach the target platform, whereas the green one is successful. The optimal takeoff point $P^*(x^*, y^*)$ is highlighted in blue. The Gaussian distribution for the random takeoff points associated to the successful boundary trajectory is displayed in black. Figures 5.4 and 5.5 show the sample trajectories obtained using, respectively, the *Gaussian without resampling* and the *Gaussian with resampling* noise function to generate the random takeoff points: the red trajectories are unsuccessful, whereas the green ones are successful. Figure 5.4 also shows the random takeoff points that are off the starting platform (in red) and result in unsuccessful sample trajectories. The two estimated probabilities for the jump success are 0.035 (*Gaussian without resampling*) and 0.043 (*Gaussian with resampling*), which are uninfluenced by the difficulty coefficient in this scenario.

If the character can perform a second jump while in the air (i.e. double jumping), we need to account for this event to estimate the probability of success of a jump. In this case, a jump is considered as the composition of two trajectories: the first jump trajectory and the double jump trajectory. Referring to Figure 5.6, which shows for simplicity only the optimal trajectories that involve the player pressing the run button, we see the two double jump trajectories generated for each initial jump trajectory. The two points from which the double jump trajectories originate are the apex of the first jump $P_1^*(x_1^*, y_1^*)$ (highlighted in blue) and the point at the same $y$ coordinate of the first takeoff point on the first jump trajectory $P_2^*(x_2^*, y_2^*)$ (highlighted in yellow). As usual, the successful trajectories are displayed in green, whereas the unsuccessful ones are in red. Since we consider the jump as the composition of two trajectories, to estimate its probability of success, we apply the noise function to the takeoff points of both the first jump and the double jump trajectories. As such, we have sample double jump trajectories originating from sample first jump trajectories. This means that the random takeoff points for the double jump trajectories are generated

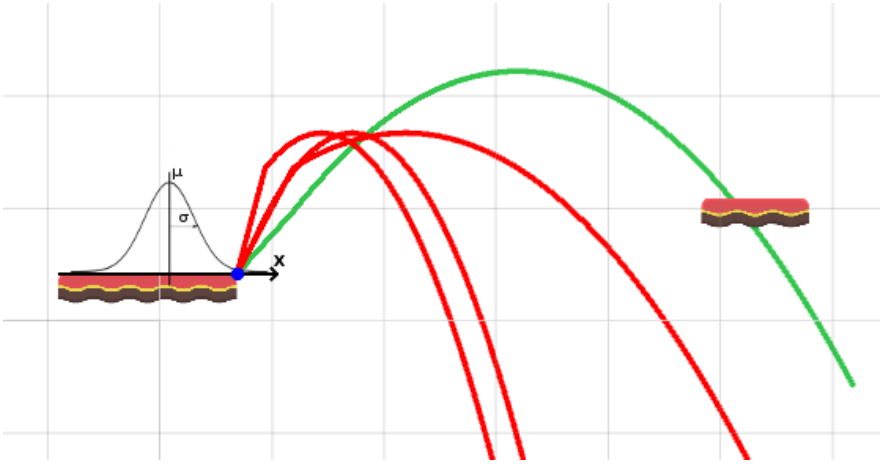


Figure 5.3: The boundary trajectories for a *Simple* jump configuration. Successful trajectory in green and unsuccessful trajectories in red. The optimal takeoff point is highlighted in blue. The Gaussian distribution associated to the successful trajectory is displayed in black.

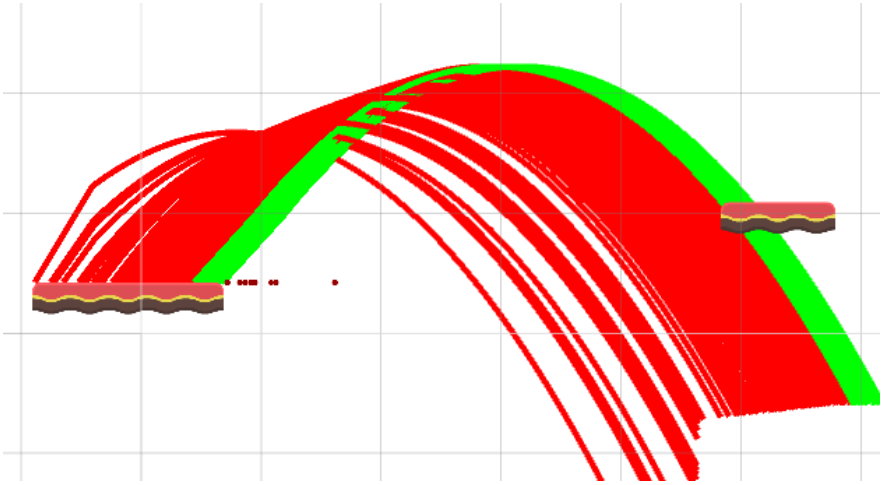

Figure 5.4: The generated sample trajectories for the scenario of Figure 5.3 using the *Gaussian without resampling* noise function.

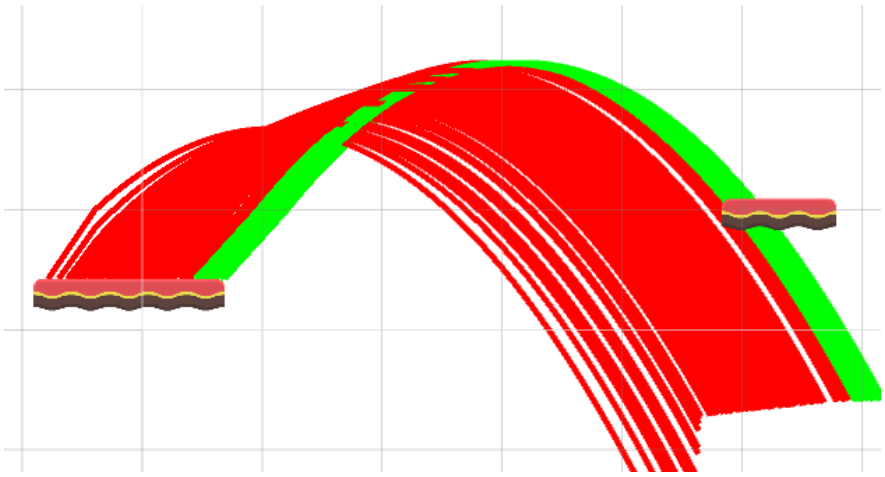

Figure 5.5: The generated sample trajectories for the scenario of Figure 5.3 using the *Gaussian with resampling* noise function.

around the points $P_1^*$ and $P_2^*$ of the sample first jump trajectories. If the *Uniform* noise function is used, the interval width is computed as in the single jump case ($\delta = 2(|v_x| + \epsilon)/Ps$). The first jump takeoff points are sampled using the uniform interval employed in the single jump scenario, whereas two intervals for the double jump takeoff points are obtained for each sample first jump trajectory: $[x_1^* - 0.5\delta, x_1^* + 0.5\delta]$ (centered on $P_1^*$) and $[x_2^* - 0.5\delta, x_2^* + 0.5\delta]$ (centered on $P_2^*$). Figure 5.7 shows the intervals obtained for two sample first jump trajectories. Note that, we sample the $x$ coordinate of the double jump random takeoff points from these intervals, whereas the $y$ coordinate is obtained evaluating the respective first jump trajectory in the sampled $x$ coordinate. Figure 5.8 shows the sample trajectories obtained using the *Uniform* noise function to generate the random takeoff points; the estimated probability of success for the jump in this scenario is 0.208, which is not affected by the difficulty coefficient.

When using the *Gaussian without resampling* or the *Gaussian with resampling*, the random takeoff points for the first jump and double jump trajectories are sampled from a Gaussian distribution. The takeoff points for the first jump trajectories are sampled with the Gaussian distribution described for the single jump case. For what concerns the Gaussian distribution of the double jump takeoff points, the standard deviation is $\sigma = (|v_x| + \epsilon)/(k \times Ps)$, whereas the mean value is $\mu = x^* + Rs \times Rt \times |v_x|$, where $x^*$ is the $x$ coordinate of either $P_1^*$ or $P_2^*$ (the two optimal takeoff points on the respective first jump trajectory) and $Rs$ is randomly 1 or -1 so that the mean can be, respectively, on the *right* or *left* of the optimal takeoff point. Figure 5.9 shows two sample of initial jump trajectories and their $P_1^*$ and $P_2^*$ points (in blue and in yellow respectively); the Gaussian distributions for the random takeoff points of the double jump trajectories are centered around $P_1^*$ and $P_2^*$. Figures 5.10 and 5.11 show the sample trajectories

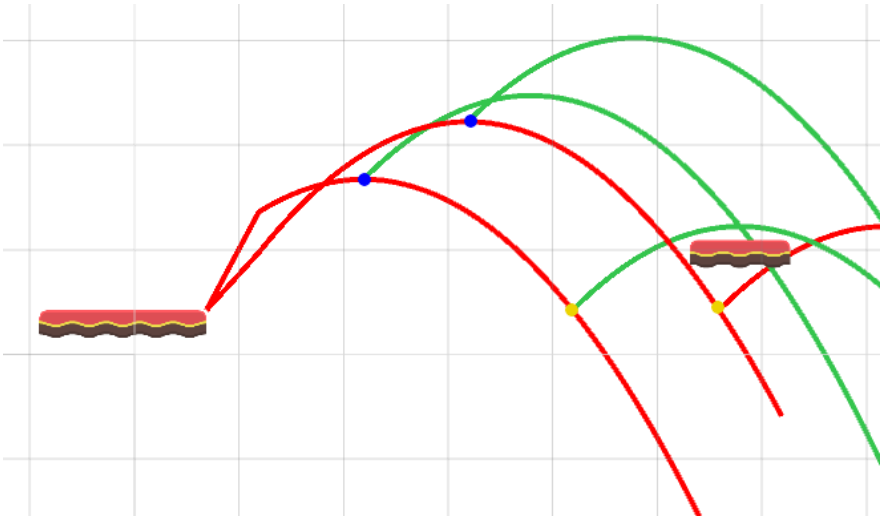

Figure 5.6: The boundary trajectories for a *Simple* jump scenario.

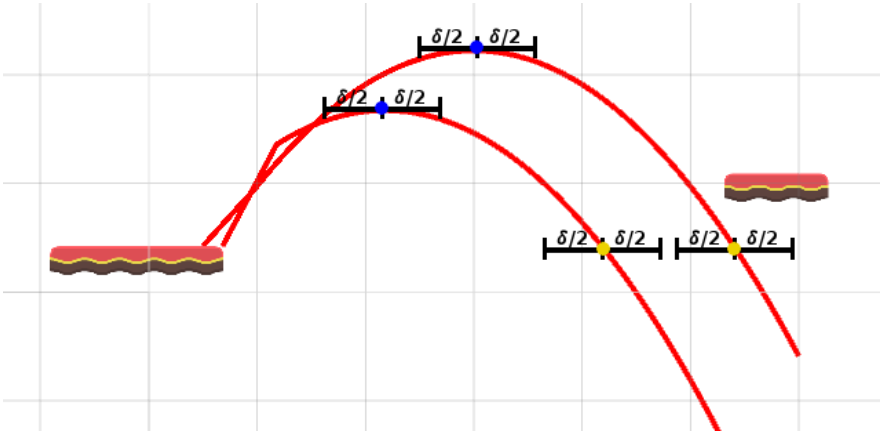


Figure 5.7: The *Uniform* noise function intervals for two sample first jump trajectories.

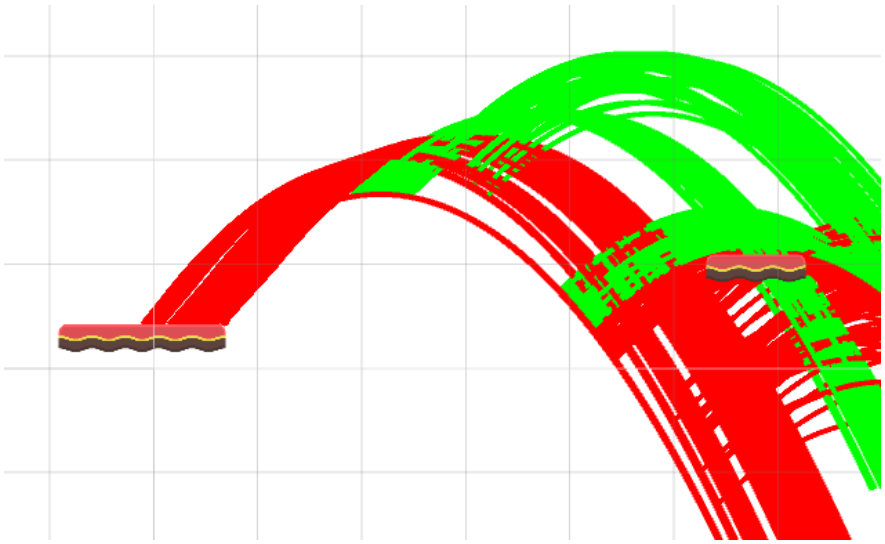

Figure 5.8: The sample trajectories generated for the scenario of Figure 5.6 using the *Uniform* noise function.

obtained using, respectively, the *Gaussian without resampling* and the *Gaussian with resampling* noise function to generate the random takeoff points; note that, in the former case, the red points that are off the starting platform are counted as unsuccessful sample trajectories. The two estimated probabilities for the jump success are 0.098 (*Gaussian without resampling*) and 0.123 (*Gaussian with resampling*); since the target platform is *MOVING* with a movement speed of 1 ($movSp = 1$), the probabilities are computed by weighting the percentage of successful trajectories with $1/\sqrt{2}$ as $DC = \sqrt{movSp + 1}$.

To recap, Figure 5.12 illustrates the difficulty and probability associated to the directed edge computed for a *Simple* trajectory configuration, using the *Uniform* noise function. The same platforms configuration is used in Figure 5.13 to compare the directed edge parameters between the *Gaussian without resampling* and the *Gaussian with resampling* noise functions. Notice how the edge difficulty value is the same among the three cases, as the considered optimal trajectories remains the same, whereas the probability value changes according to the employed noise function. Also, the *Gaussian without resampling* characteristic of considering random takeoff points off the starting platform as unsuccessful trajectories in the edge probability computation results in the most pessimistic probability estimate.

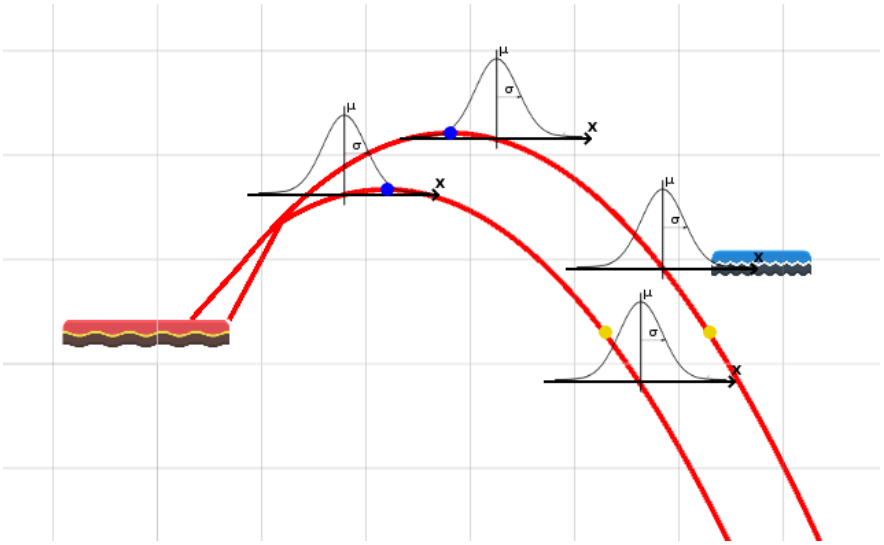

Figure 5.9: Two sample first jump trajectories with the respective Gaussian distributions for the random takeoff points of the double jump trajectories. The optimal takeoff points for the double jump trajectories are displayed in blue and in yellow.

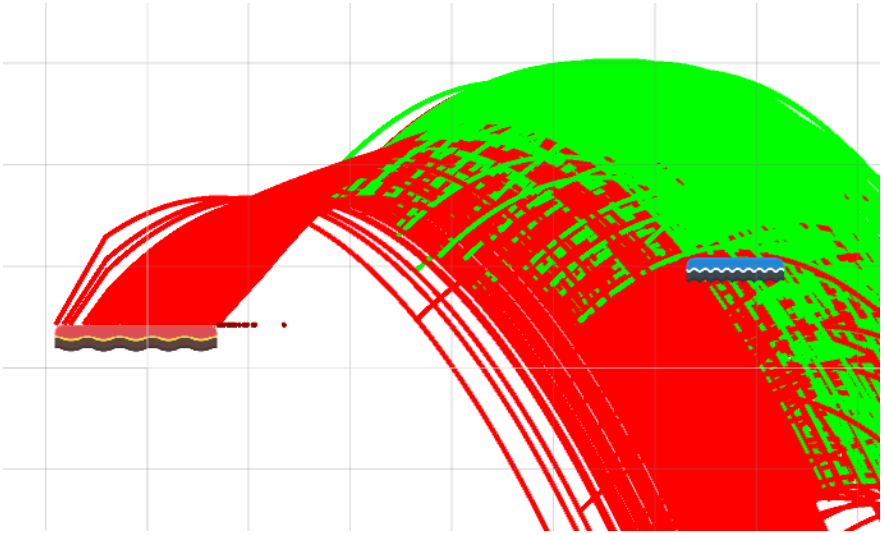

Figure 5.10: The sample trajectories generated for the scenario of Figure 5.9 using the *Gaussian without resampling* noise function.

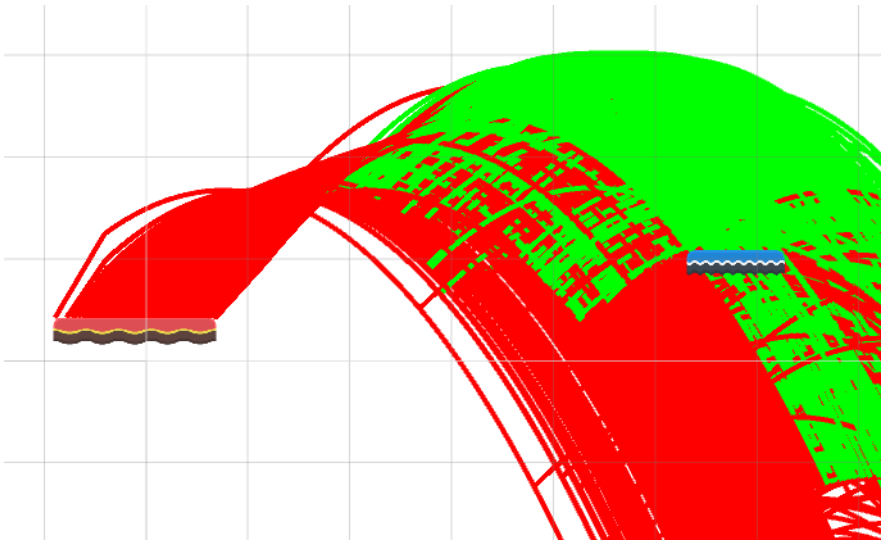

Figure 5.11: The sample trajectories generated for the scenario of Figure 5.9 using the *Gaussian with resampling* noise function.

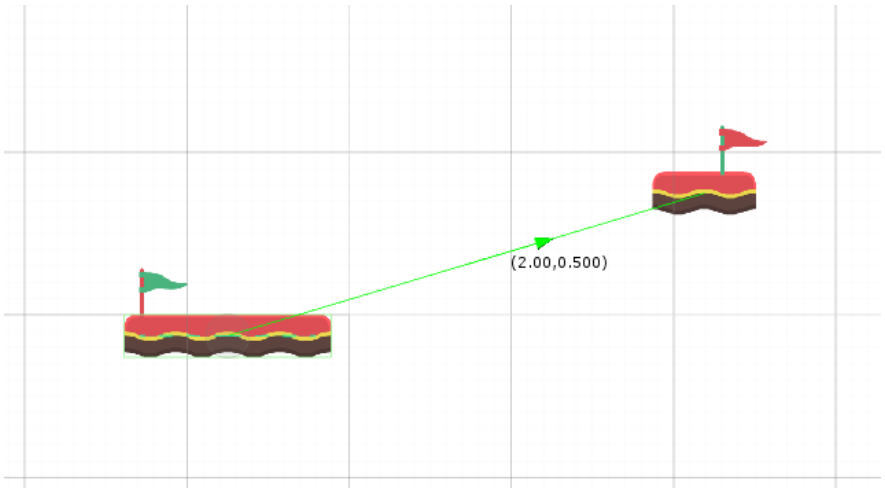


Figure 5.12: The directed edge computed for a *Simple* trajectory configuration. The difficulty and probability values of the edge are displayed. The probability is computed using the *Uniform* noise function.

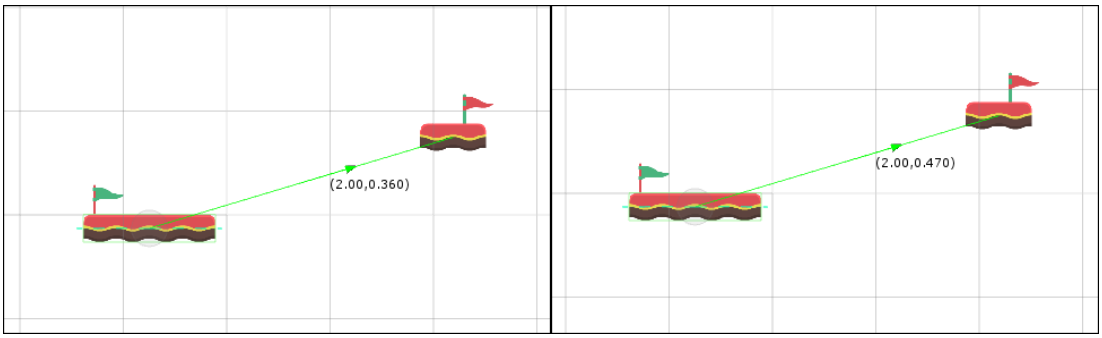


Figure 5.13: The directed edge computed for the same *Simple* trajectory configuration is displayed together with its difficulty and probability values. On the left, the edge probability is computed using the *Gaussian without resampling* noise function. On the right, the edge probability is computed using the *Gaussian with resampling* noise function.

### 5.1.2 *Reentrant* trajectory jumps

If the two considered platforms are in a *Reentrant* trajectory configuration (Section 4.3.3.4), the character must be able to double jump in order to reach the target platform. Figure 5.14 shows two boundary jump trajectories for a *Reentrant* configuration; the optimal takeoff points for the first jump and the double jump are highlighted in yellow and in blue respectively. To estimate the jump probability of success for this configuration, the noise function affects the $x$ coordinate of the takeoff points of both the first jump and the double jump trajectories. Similarly to the *Simple* trajectory case (Section 5.1.1), the sample double jump trajectories originate from sample first jump trajectories. For a *Reentrant* configuration, a single double jump trajectory originates from a first jump trajectory. The main difference with respect to the *Simple* trajectory scenario is the direction change of the double jump trajectory, which causes the optimal takeoff point for the second jump to be moved away from the apex of the first jump. When the *Uniform* noise function is used, the random takeoff points are sampled with a uniform probability distribution from a interval of width $\delta = 2(|v_x| + \epsilon)/Ps$. Figure 5.15 shows a sample reentrant trajectory and the uniform intervals generated around the optimal takeoff points. When using one of the two Gaussian noise models, the random takeoff points are sampled from a Gaussian distribution having standard deviation $\sigma = (|v_x| + \epsilon)/(k \times Ps)$. Note that, as in the *Simple* trajectory configuration with double jump, the takeoff points for the first jump trajectories are sampled from a different Gaussian distribution with respect to the takeoff points for the double jump trajectories. In fact, for the first jump trajectories the mean of the distribution is $\mu = x^* - Js \times Rt \times |v_x|$, whereas for the double jump trajectories $\mu = x^* + Rs \times Rt \times |v_x|$. As in the *Simple* trajectory case, the first jump random takeoff points are constrained to be on

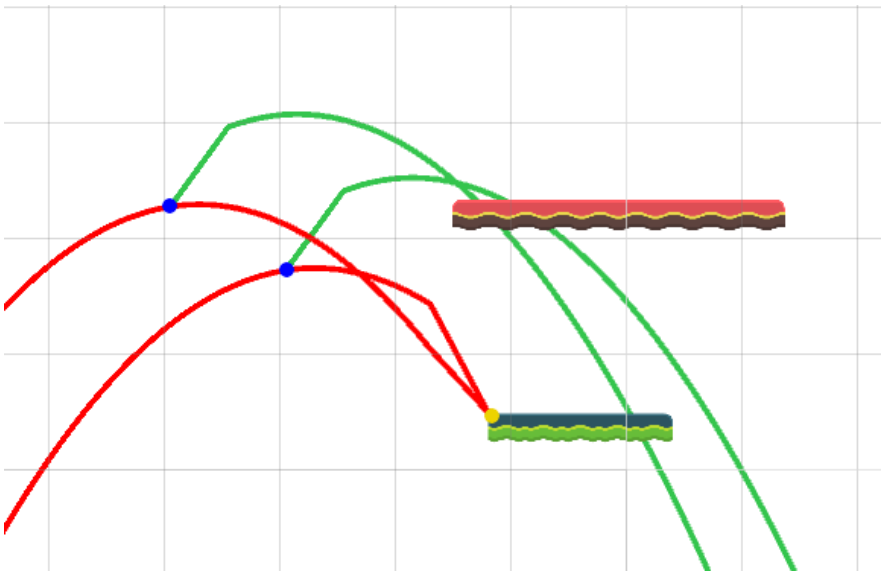

Figure 5.14: Two boundary trajectories for a *Reentrant* configuration. The optimal takeoff points for the first jump and the double jump are highlighted in yellow and in blue respectively.

the platform if $v_x = 0$. Figure 5.16 shows a sample reentrant trajectory and

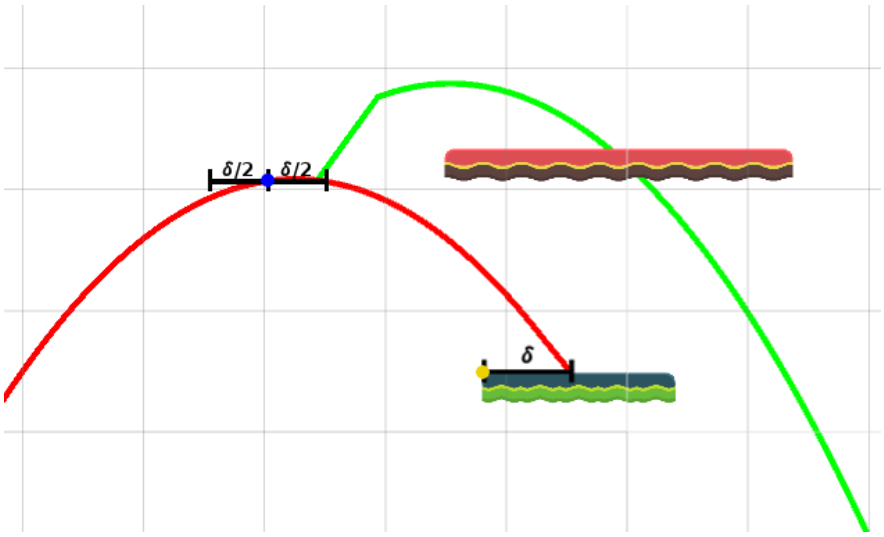


Figure 5.15: A sample *Reentrant* trajectory and the two intervals used by the Uniform noise function to generate the random takeoff points.

the Gaussian distributions for the random takeoff points. In the scenario of Figure 5.14, the three estimated probabilities are 0.376 (*Uniform*), 0.161 (*Gaussian without resampling*) and 0.232 (*Gaussian with resampling*); since the starting platform is *FADING* with a fade speed of 0.55 ($fadSp = 0.55$), these estimates

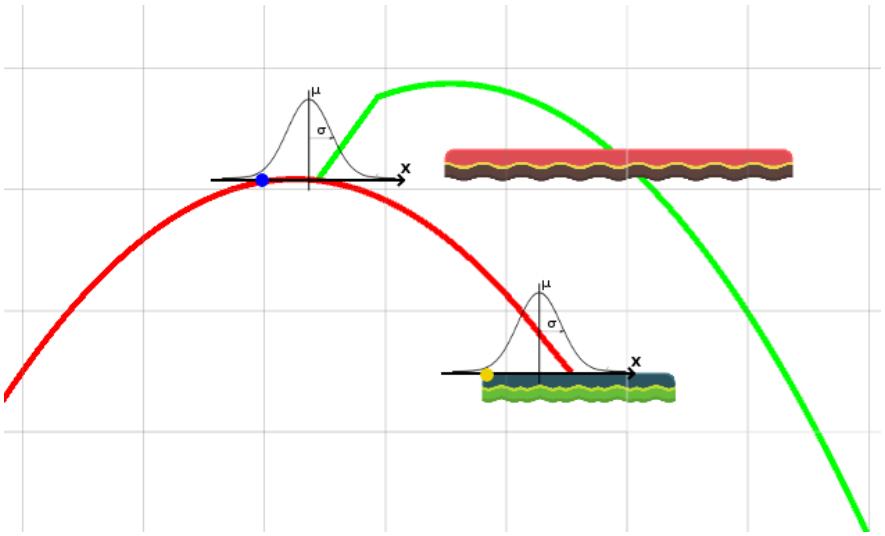


Figure 5.16: A sample *Reentrant* trajectory and the Gaussian distributions used to generate the random takeoff points.

are computed by weighting the percentage of successful trajectories with 1/2.37 as $DC = 1/(\log_1 0(t_D is + 1))$ and $t_D is = 0.9/fadSp$. Note that the estimated probability of the *Gaussian without resampling* model is the lowest, as it accounts for the possibility of failing the first jumps.

### 5.1.3 *Falling* trajectory jumps

If the two considered platforms are in a *Falling* trajectory configuration (Section 4.3.3.3), the noise function affects the $y$ coordinate of the boundary trajectories optimal start point to generate sample trajectories. Since the player tackles this category of jumps by falling down, we consider the vertical speed $v_y$ as the approach speed for the computation of the parameters that define the noise functions. As we discussed in Section 4.3.3.3, depending on whether the character can double jump or not the considered boundary trajectories and the optimal start points are different. When the *Uniform* noise function is used, the interval width for the random start points is $\delta = 2(|v_y| + \epsilon)/Ps$. If we call the optimal start point $P^*(x^*, y^*)$, the interval is $[y^* - \delta, y^*]$ if the character cannot double jump, whereas the interval is $[y^* - 0.5\delta, y^* + 0.5\delta]$ if the character can double jump. Note that the interval upper bound in case of double jump is limited to minimize the possibility of collision with the starting platform when performing the second jump. Figure 5.17 shows the boundary trajectories for a *Falling* configuration in case of single jump and double jump modes, the respective optimal start point (highlighted in blue) and the *Uniform* noise function intervals. Figure 5.18 shows and compares the sample trajectories in case of single jump and double jump for the scenario of Figure 5.17. When using one of the two Gaussian noise models, the

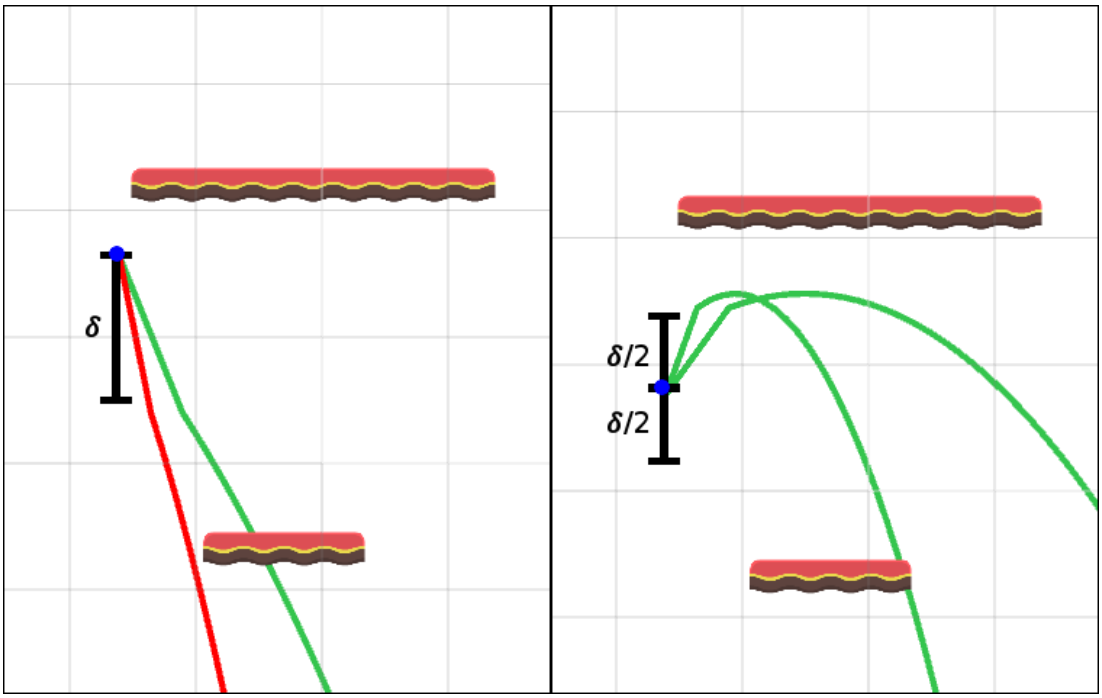


Figure 5.17: Boundary trajectories and *Uniform* noise function intervals for a *Falling* configuration. On the left, the character jumping mode is not set to *DOUBLE_JUMP*; on the right, the character jumping mode is set to *DOUBLE_JUMP*.

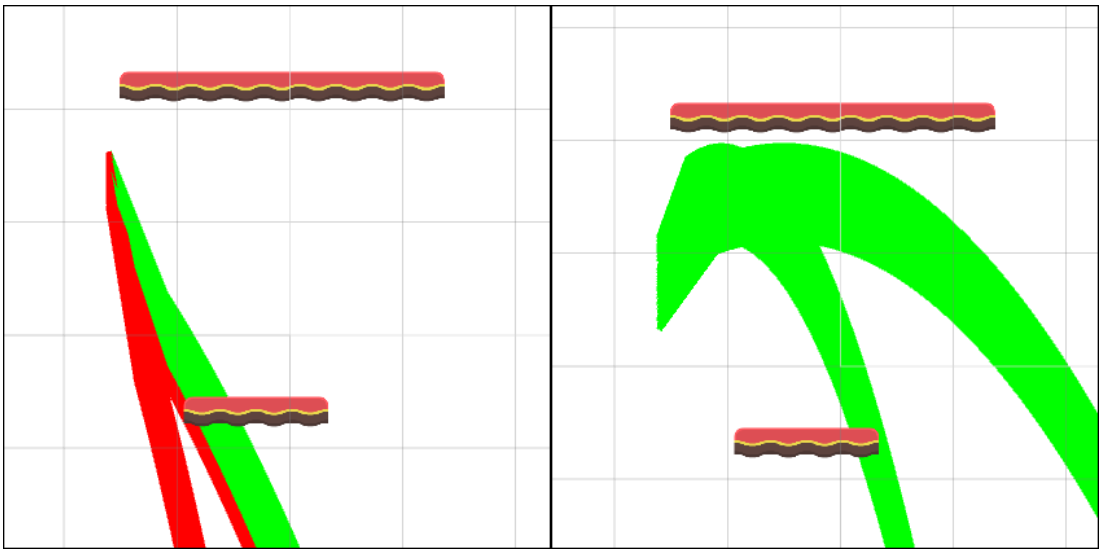

Figure 5.18: Sample trajectories generated with the *Uniform* noise function for the scenario of Figure 5.17.

random start points are sampled from a Gaussian distribution having standard deviation $\sigma = (|v_y| + \epsilon)/(k \times Ps)$. The mean value $\mu$ of the distribution depends on whether the character can double jump or not: if only a single jump can be performed $\mu = y^* - Rt \times |v_y|$; if double jumping is possible $\mu = y^* + Rs \times Rt \times |v_y|$, where $Rs$ is randomly 1 or -1 so that the mean can be, respectively, above or below the optimal start point. It must be noted that, for *Falling* configurations, the two Gaussian noise functions are equivalent. Figure 5.19 shows the boundary

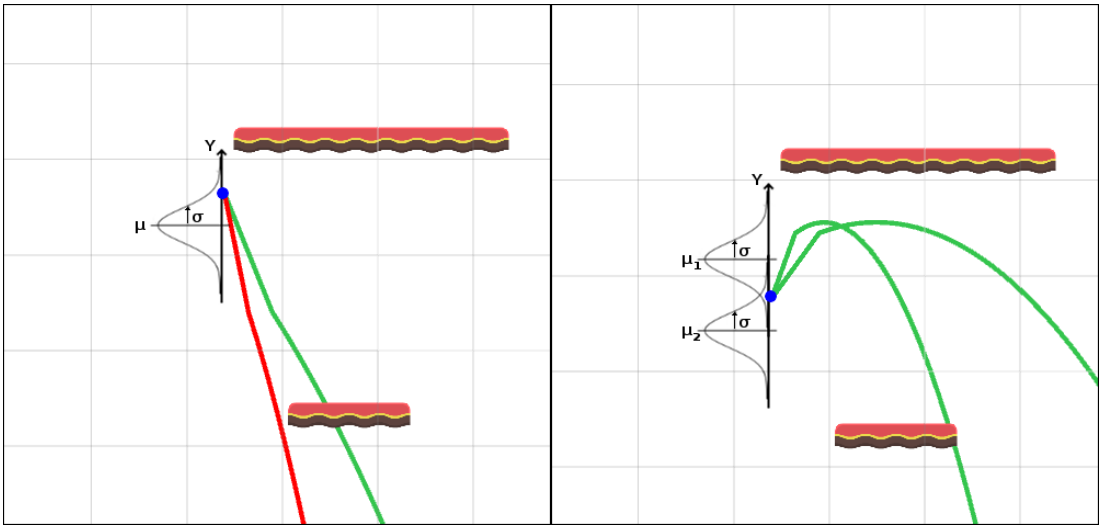


Figure 5.19: Boundary trajectories and Gaussian distributions for the random start points for a *Falling* configuration. On the left, the character jumping mode is not set to *DOUBLE_JUMP*; on the right, the character jumping mode is set to *DOUBLE_JUMP*.

trajectories for a *Falling* configuration in case of single jump and double jump modes, the respective optimal start point (highlighted in blue) and the Gaussian distribution for the generation of the random start points of sample trajectories: note that, in case of double jump, two Gaussian distributions are displayed to highlight the fact that they can be centered either above or below the optimal start point; as a matter of fact, the player can perform the double jump either too early or too late with respect to the assumed optimal position. Figure 5.20 shows and compares the sample trajectories in case of single jump and double jump for the scenario of Figure 5.19. The four estimated probabilities are 0.161 (*Uniform*, single jump), 0.500 (*Uniform*, double jump), 0.104 (*Gaussian*, single jump) and 0.500 (*Gaussian*, double jump); these estimates are computed by weighting the percentage of successful trajectories with the inverse of the difficulty factor associated to *Falling* trajectories (see Section 4.3.4.3).

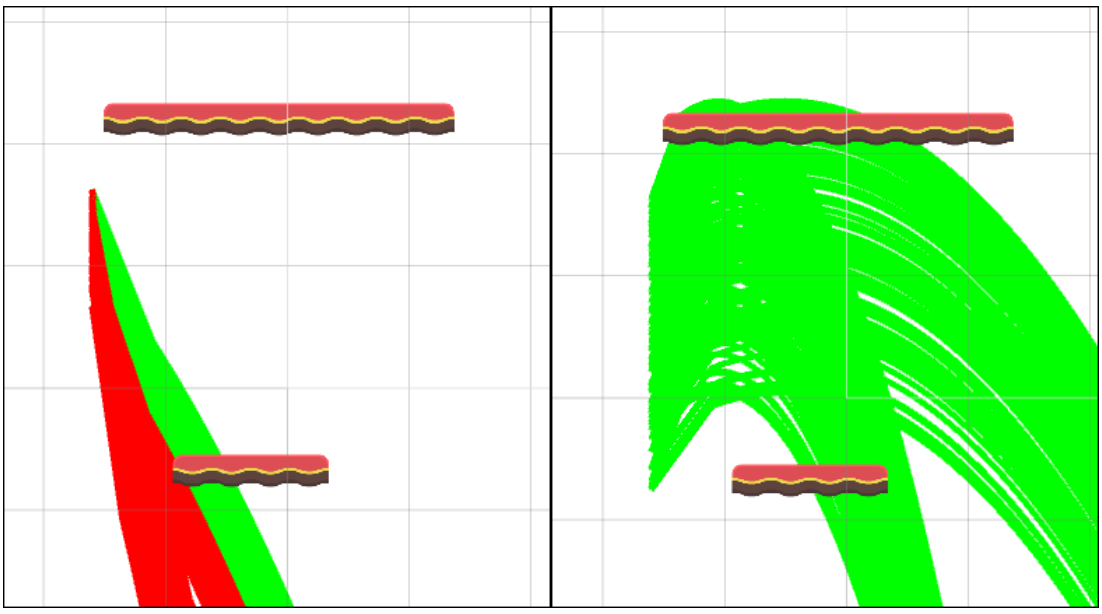

Figure 5.20: Sample trajectories generated with a Gaussian noise function for the scenario of Figure 5.19.

## 5.2 Experimental validation of single jumps probability

In order to validate our approach for the computation of the single jumps probability of success and establish which of the noise model provides the best estimates, we carried out an experiment to collect data. We asked several users with different skill levels to perform from 10 up to 50 jumps for each experiment session. Users could repeat the experiment as many times as they wanted. The experiment was carried out online, using a web build of our platformer implementation; in particular, the build was configured to allow the character to double jump and run. The experiment involved a set of 16 game screens, each one asking the players to jump across two platforms positioned in different configurations: (i) 2 screens involved *Trivial* jumps across two nearby platforms with no possibility of failing; (ii) 6 screens involved *Simple* trajectory jumps across two platforms, which were positioned to have a gap of variable width between them; (iii) 4 screens involved *Falling* trajectories with the starting platform positioned above the target one; (iv) 4 screens involved *Reentrant* jumps across two platforms, with the starting platform below the target one, requiring good skills and timing to be overcome. Users tackled a sequence of jump tests randomly sampled from the 16 trials set, with the constraint of never presenting the same jump trial twice in a row. The following information were recorded for each jump performed: (i) the *trialID*, that is the identifier of the jump trial; (ii) the takeoff point coordinates; (iii) the landing point coordinates (only if the user managed to land on the target platform); (iv) the horizontal speed at which the character took off for the jump; (v) whether the character successfully reached the target platform with this jump or not; (vi) the array of points forming the trajectory followed by the character during the jump. A total of 58 users took part in the experiment, 26 of them also filled in a form to provide their age, how much they liked platformer and how difficult the jump trials felt. The users who provided additional information were between 20 and 40 years old and reported an above the average liking of platformers. A total of 2361 jumps were recorded, 1477 of which performed by users that filled in the form. Table 5.1 shows a summary of the information collected with the experiments for each jump screen (identified by trialID) available to the players: the jump type (Trajectory Type), the total number of times the jump screen was presented to the users (#Jumps), the total number of times the jump screen was successfully completed by the users (#Successes) and the ratio of successes for the jump screen (Experimental Probability). We observe that almost all jump trials have an experimental probability of success greater than 30%, whereas the two most difficult jumps (experimental probability equal to 21.6%) were trials #4 and #8; the former required jumping with a running start and performing a well timed double jump; the latter was extremely tricky due to the *Reentrant* configuration featuring a *MOVING* starting

| trialID | Trajectory Type | #Jumps | #Successes | Experimental Probability |
|---|---|---|---|---|
| 0 | Simple | 156 | 136 | 0.872 |
| 1 | Simple | 131 | 80 | 0.611 |
| 2 | Simple | 158 | 114 | 0.722 |
| 3 | Simple | 159 | 104 | 0.654 |
| 4 | Simple | 153 | 33 | 0.216 |
| 5 | Reentrant | 150 | 53 | 0.353 |
| 6 | Reentrant | 139 | 76 | 0.547 |
| 7 | Reentrant | 134 | 43 | 0.321 |
| 8 | Reentrant | 148 | 32 | 0.216 |
| 9 | Trivial | 157 | 139 | 0.885 |
| 10 | Falling | 151 | 142 | 0.940 |
| 11 | Falling | 156 | 143 | 0.917 |
| 12 | Trivial | 157 | 156 | 0.994 |
| 13 | Falling | 140 | 128 | 0.914 |
| 14 | Falling | 141 | 93 | 0.660 |
| 15 | Simple | 131 | 64 | 0.489 |

Table 5.1: A summary of the data collected for each jump trial.

platform. The two *Trivial* configurations reported a probability that is close to one; in particular, the jump with trialID 12 was failed only once out of 157 tries. In general, *Reentrant* configurations seemed to be the hardest to tackle, whereas *Falling* configurations appeared to be the easiest.

We estimated the probability of success of each jump trial using the three noise models and employing different combinations of average reaction time $Rt$ and player skill $Ps$. Note that the *Uniform* noise function employs only the $Ps$ parameter, thus its performance are compared solely with regard to the player skill value. To compare the probability estimates provided by our framework against the experimental data, we used two metrics: the *Mean Absolute Error* (MAE) and the *Difficulty Ordering Error Ratio* (DOER). MAE provides a measure of how much the probability estimates obtained with a noise model differ from the experimental data; it was computed as:

$$MAE = \frac{\sum_{i=0}^{N-1} |P_{i,exp} - P_{i,noise}|}{N} \tag{5.1}$$

Where $N$ is the total number of jump trials (16), $P_{i,exp}$ is the experimental probability obtained for jump trial $i$ and $P_{i,noise}$ is the probability estimated using the

noise model for jump trial $i$.
DOER provides an evaluation of the noise models based on their accuracy in ordering jump trials according to their probability of success. In particular, given a pair of trials, a good noise model should be able to report which of the two is the harder (or the easier) using the probability estimates. Thus, DOER provides a measure of how much the noise model fails at deciding which is the harder jump between two jump trials and it is computed as:

$$DOER = \frac{WronglyOrderedPairs}{PairsCount} \tag{5.2}$$

Where *WronglyOrderedPairs* is the number of jump trial pairs that the noise model ordered in the wrong way (e.g. $P_{0,noise} < P_{1,noise}$ when $P_{0,exp} > P_{1,exp}$) and *PairsCount* is the total number of possible jump trial pairs. In our case, *PairsCount* is equal to the 2-combinations of 16 elements (120).
MAE and DOER result from the estimates obtained on the 16 jump trials. Figure 5.21 shows MAE values for the *Uniform* noise function. Tables 5.2 and 5.3 report MAE values and the respective standard error computed for the two Gaussian noise models according to different combinations of *Rt* and *Ps*. Figures 5.23 and 5.22 compare MAE values for each Gaussian noise model. Figure 5.24 shows DOER values for the *Uniform* noise function. Table 5.4 reports DOER values computed for the two Gaussian noise models according to different combinations of *Rt* and *Ps*. Figures 5.26 and 5.25 compare DOER values for each Gaussian noise model.
By observing the performance metrics, we see that the three noise models keep MAE values below 0.32. In particular, for the *Uniform* noise model, the minimum is 0.227, corresponding to $Ps = 5$; for what concerns the Gaussian models, the minimum MAE value is 0.217, corresponding to the *Gaussian with resampling* noise function with $Ps = 1$ and $Rt = 0.1s$. For what concerns DOER, the *Uniform* noise model provides the best performance among the three models with $Ps = 1$ and an error rate equal to 17.5%; the Gaussian models have minimum DOER values corresponding to $Ps = 1$ and $Rt = 0.5s$: 26.7% for *Gaussian without resampling* and 27.5% for *Gaussian with resampling*.

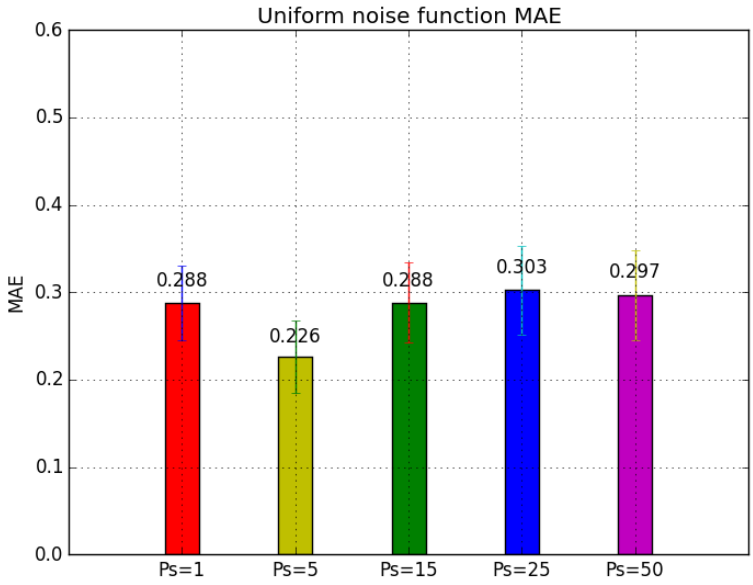


Figure 5.21: Bar chart comparing Mean Absolute Error (MAE) values for the *Uniform* noise function.

| **MAE** | Rt=0.01s | | Rt=0.05s | |
|---|---|---|---|---|
| | **GNR** | **GR** | **GNR** | **GR** |
| Ps=1 | $0.229 \pm 0.044$ | $0.233 \pm 0.042$ | $0.225 \pm 0.044$ | $0.224 \pm 0.043$ |
| Ps=5 | $0.269 \pm 0.044$ | $0.295 \pm 0.050$ | $0.302 \pm 0.050$ | $0.303 \pm 0.051$ |
| Ps=15 | $0.286 \pm 0.048$ | $0.293 \pm 0.051$ | $0.306 \pm 0.053$ | $0.306 \pm 0.053$ |
| Ps=25 | $0.292 \pm 0.050$ | $0.293 \pm 0.051$ | $0.306 \pm 0.053$ | $0.306 \pm 0.053$ |
| Ps=50 | $0.293 \pm 0.051$ | $0.293 \pm 0.051$ | $0.306 \pm 0.053$ | $0.306 \pm 0.053$ |

Table 5.2: Mean Absolute Error (MAE) values for the *Gaussian without resampling* (GNR) and the *Gaussian with resampling* (GR) noise functions with respect to different combinations of $Rt$ and $Ps$.

| **MAE** | Rt=0.1s | | Rt=0.5s | |
|---|---|---|---|---|
| | **GNR** | **GR** | **GNR** | **GR** |
| Ps=1 | $0.220 \pm 0.046$ | $0.217 \pm 0.045$ | $0.231 \pm 0.050$ | $0.225 \pm 0.049$ |
| Ps=5 | $0.297 \pm 0.047$ | $0.298 \pm 0.047$ | $0.260 \pm 0.045$ | $0.260 \pm 0.045$ |
| Ps=15 | $0.315 \pm 0.051$ | $0.315 \pm 0.051$ | $0.262 \pm 0.047$ | $0.262 \pm 0.047$ |
| Ps=25 | $0.316 \pm 0.051$ | $0.316 \pm 0.051$ | $0.262 \pm 0.047$ | $0.262 \pm 0.047$ |
| Ps=50 | $0.316 \pm 0.051$ | $0.316 \pm 0.051$ | $0.261 \pm 0.047$ | $0.261 \pm 0.047$ |

Table 5.3: Mean Absolute Error (MAE) values for the *Gaussian without resampling* (GNR) and the *Gaussian with resampling* (GR) noise functions with respect to different combinations of $Rt$ and $Ps$.

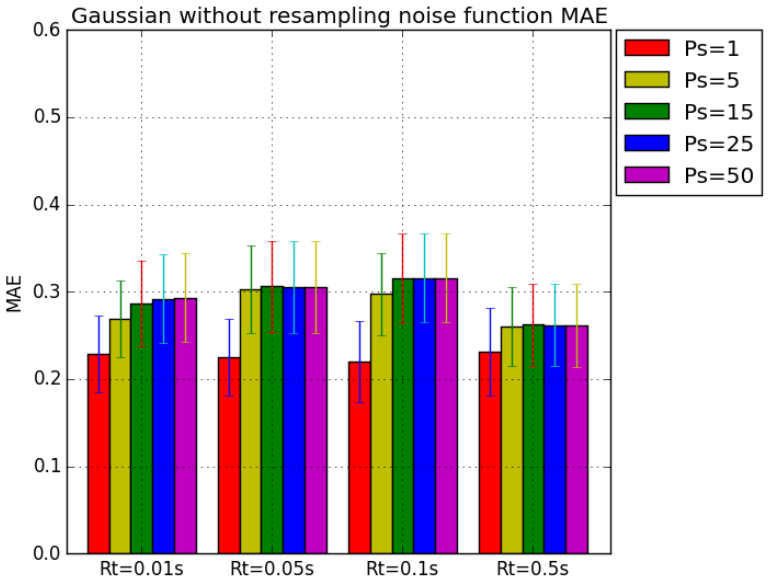


Figure 5.22: Bar chart comparing Mean Absolute Error (MAE) values for the *Gaussian without resampling* noise function.

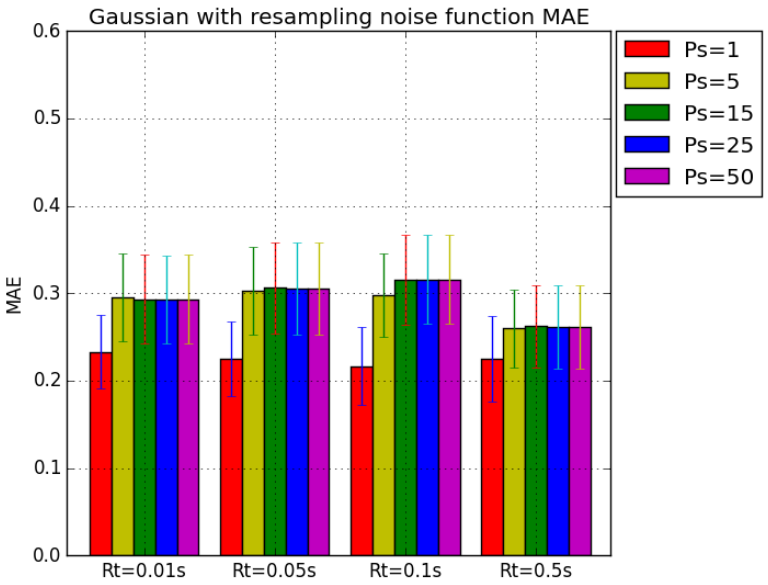


Figure 5.23: Bar chart comparing Mean Absolute Error (MAE) values for the *Gaussian with resampling* noise function.

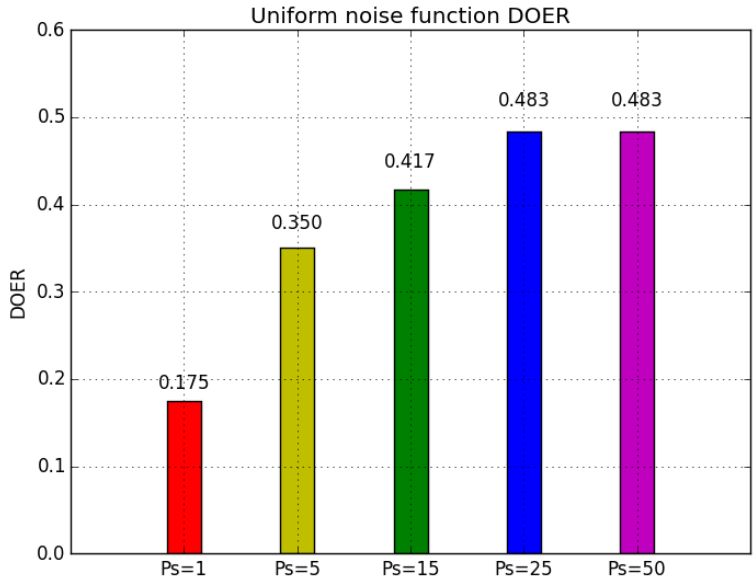


Figure 5.24: Bar chart comparing the Difficulty Ordering Error Ratio (DOER) values for the *Uniform* noise function.

| **DOER** | Rt=0.01s | | Rt=0.05s | | Rt=0.1s | | Rt=0.5s | |
|---|---|---|---|---|---|---|---|---|
| | **GNR** | **GR** | **GNR** | **GR** | **GNR** | **GR** | **GNR** | **GR** |
| Ps=1 | 0.358 | 0.375 | 0.342 | 0.358 | 0.300 | 0.317 | 0.267 | 0.275 |
| Ps=5 | 0.417 | 0.433 | 0.425 | 0.433 | 0.417 | 0.417 | 0.383 | 0.383 |
| Ps=15 | 0.408 | 0.483 | 0.525 | 0.525 | 0.483 | 0.483 | 0.383 | 0.383 |
| Ps=25 | 0.417 | 0.483 | 0.525 | 0.525 | 0.525 | 0.525 | 0.358 | 0.358 |
| Ps=50 | 0.483 | 0.483 | 0.525 | 0.525 | 0.525 | 0.525 | 0.358 | 0.358 |

Table 5.4: Difficulty Ordering Error Ratio (DOER) values for the *Gaussian without resampling* (GNR) and the *Gaussian with resampling* (GR) noise functions with respect to different combinations of $Rt$ and $Ps$.

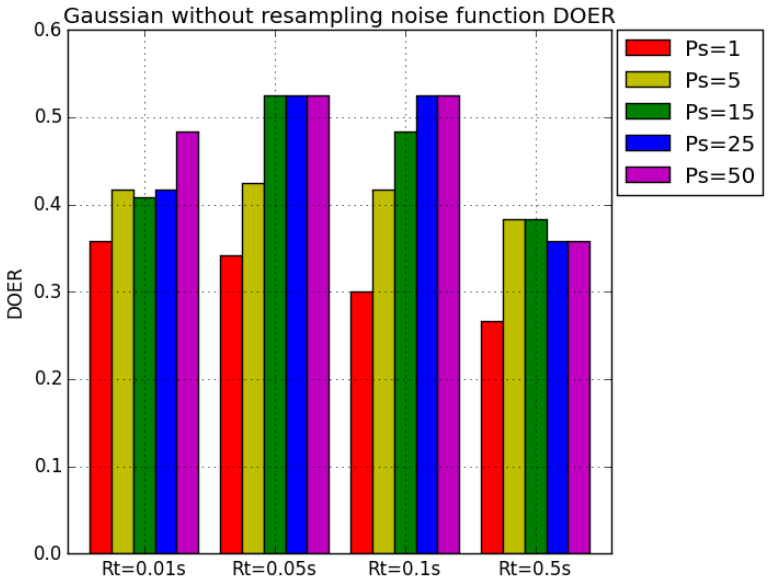


Figure 5.25: Bar chart comparing the Difficulty Ordering Error Ratio (DOER) values for the *Gaussian without resampling* noise function.

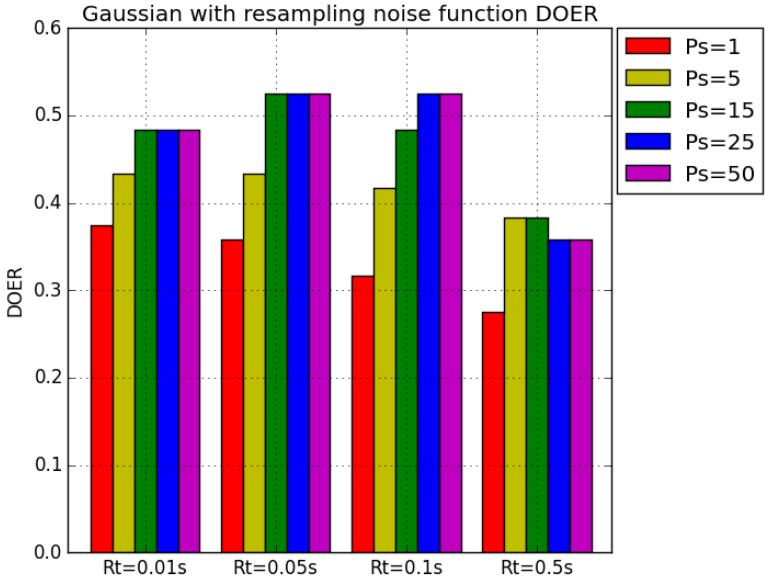


Figure 5.26: Bar chart comparing the Difficulty Ordering Error Ratio (DOER) values for the *Gaussian with resampling* noise function.

## 5.3 Level difficulty and probability of success

Designers aim at building levels that elicit quality gameplay experiences. The role of our framework is to assist the designers during this process. Accordingly, we extended the evaluation of difficulties and probabilities of success from the single jump instance to the evaluation of entire levels viewed as multiple sequences of consecutive jumps. In order to do this, we defined a *path* as the sequence of jumps that allows the character to move through a level; a *path* $P$ is defined as a list of directed edges $(e_1, e_2, ..., e_L - 1, e_L)$, where $e_i$ is the i-th directed edge of path $P$, which is of length $L$. In addition, we only consider acyclic paths, meaning that a vertex of the graph (or equivalently a platform) can be visited only once in a path. In order to evaluate the difficulty and the probability of success of a path, we assumed that each jump is independent from the others. As such, we introduced the following metrics to evaluate a path $P$ of length $L$:

$$p(P) = \prod_{e_i \in P} p(e_i) \tag{5.3}$$

$$d_t(P) = \sum_{e_i \in P} d(e_i) \tag{5.4}$$

$$\overline{d}(P) = 1/L \sum_{e_i \in P} d(e_i) \tag{5.5}$$

$$d_M(P) = Median(d(e_1), d(e_2), ..., d(e_{L-1}), d(e_L)) \tag{5.6}$$

$$d_w(P) = \frac{\sum_{e_i \in P} d(e_i)p(e_i)}{\sum_{e_i \in P} p(e_i)} \tag{5.7}$$

$$d_i(P) = \sum_{e_i \in P} d(e_i)\alpha^i \tag{5.8}$$

As implied by our independence hypothesis, the probability of success $p$ of a path $P$ (equation 5.3) is computed as the product of the probability of each edge $p(e_i)$ composing the path. The total difficulty $d_t$ of a path $P$ (equation 5.4) is given by the sum of the difficulty of each edge $d(e_i)$ composing the path. The $d_t$ metric is agnostic to the position of the jump in the level (e.g. the first jump has the same weight of the last jump in the path), thus it provides a rough estimate of the path difficulty, also in terms of its length; in addition, $d_t$ can be considered as a metric that models the player's fatigue. The average difficulty $\overline{d}$ of a path $P$ (equation 5.5) is the mean value of the edges difficulties and represents a basic indicator of the overall difficulty of the path, but provides no information on the path length. The median difficulty $d_M$ of a path $P$ (equation 5.6) is the median value of the edges difficulties and provides an indicator of the path difficulty that is not affected by extreme (i.e. very low or high) edge difficulty values. The weighted difficulty $d_w$ of path $P$ (equation 5.7) weighs the edges difficulties by

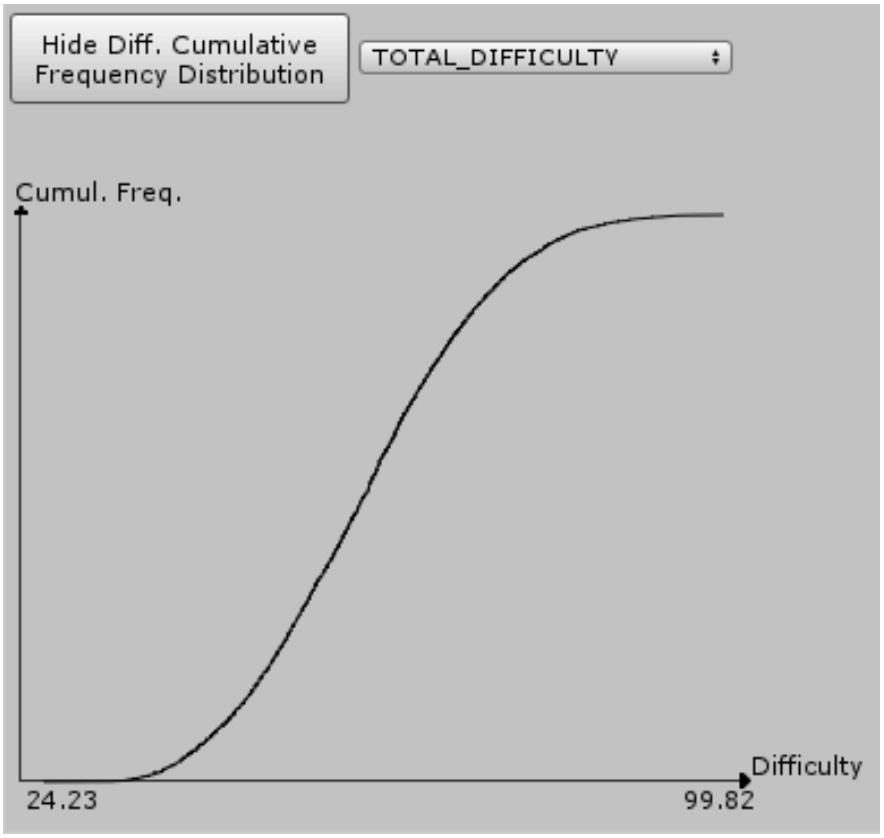


Figure 5.27: An example of cumulative frequency distribution displayed by the framework.

their probability and in doing so balances the effect of the difficulty value. The increasing difficulty $d_i$ of a path $P$ (equation 5.8) weighs the difficulty of each edge by an increasing factor $\alpha^i$, thus modeling the player's fatigue: a jump at the end of a long path is considered more difficult than a jump with the same estimated difficulty but placed at the beginning of the path; the $\alpha$ coefficient represents the rate at which the player gets tired and as such impacts the increase of the jump difficulty.
Since there are usually multiple paths that traverse a level from the starting platform to the final one, our framework features a method to extract all the available paths that connect the beginning and the end of the level. This method applies a Depth First Search (DFS) on the level graph and accounts for the character health mode and harmful platforms to find only the paths that are actually traversable without dying. Our framework provides a set of tools to assist designers in the creation of quality levels; in particular, to evaluate the probability and difficulty of the available paths in a level in an aggregate form, the framework provides a cumulative frequency diagram to display the distribution of the selected metric. Figure 5.27 shows an example of cumulative frequency


Figure 5.28: An example of minimum difficulty path (in red) displayed by the framework.

distribution for the total difficulty metric $d_t$. The diagram displays the minimum and maximum values for the selected difficulty metric on the $x$ axis. Furthermore, our framework supports the visualization of the minimum difficulty path, where the difficulty metric can be selected by the designer among the set of metrics that we previously discussed. Figure 5.28 shows the minimum difficulty path (in red) with respect to the $d_t$ metric for an example level; each one of the edge composing the path is displayed together with its difficulty and probability of success.

## 5.4 Summary

In this chapter, we discussed how we estimated the probability of success of single jumps using noise functions. Next, we described the experiments carried out to perform a preliminary validation of such estimates. Finally, we proposed an approach to evaluate platformer levels in terms of difficulty (total, average, median, weighted and increasing) and probability of reaching the final platform by extending the approach for single jumps.

# Chapter 6

# Framework tools

In this chapter we overview the tools provided by our framework and describe the interfaces the designers can use to customize platforms and to evaluate levels in terms of difficulty and probability of completion.

## 6.1 Introduction

Our framework was designed and developed as a modular extension of the popular game engine Unity. As such, it is integrated within the development environment and takes advantage of its physics engine and several features; our framework extends Unity standard features to provide dedicated tools for the design of 2D platformers. For what concerns the actual platformer implementation that is part of our framework, all the main game elements (e.g. the character and platforms) are accessible as Prefabs (i.e. template objects that can be reused in different levels) so that designers can easily replicate them to build levels. The physics parameters regulating movement, jump and the other variables associated to the platformer implementation (Section 4.2) can be accessed and modified through the inspector view (i.e. the user interface provided by Unity to edit objects). Our framework provides two dedicated interfaces to assist designers in the creation of levels: a *Custom Editor* for platforms and a *Platformer Design Tools* window to access levels related features.

## 6.2 Platforms Custom Editor

Our framework employs a customized inspector view to edit platforms properties and obtain information about their reachability in the level. Figure 6.1 shows the custom editor interface for a *STATIC_FLOATING* platform; the interface allows to edit the appearance of the platform through the type, length and connections attributes; note that, if the platform type is changed, the editor interface dynamically updates its structure in order to display the current platform type

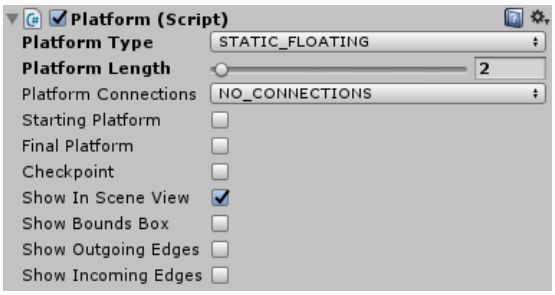


Figure 6.1: Custom editor interface for a *STATIC_FLOATING* platform.

specific properties. Figure 6.2 shows the editor interface for a *MOVING* platform: movement related parameters are displayed and can be edited. In addition, the platform can be set as a special element of the level (i.e. starting, final or checkpoint platform). The framework applies the constraint of a single starting platform and a single final platform in a level, thus the editor interface is updated accordingly. The editor allows to hide the platform from the scene view (i.e. the interface in which the level is being built) and to show its *Bounds Box* (see Section 4.3). Finally, the interface provides the visualization of the outgoing and

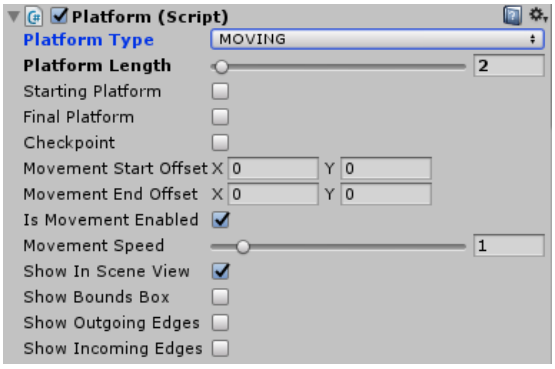


Figure 6.2: Custom editor interface for a *MOVING* platform.

incoming edges computed with the approach described in Chapters 4 and 5 for the selected platform. Figure 6.3 shows the outgoing and incoming edges together with their difficulty and probability estimates for the selected platform.

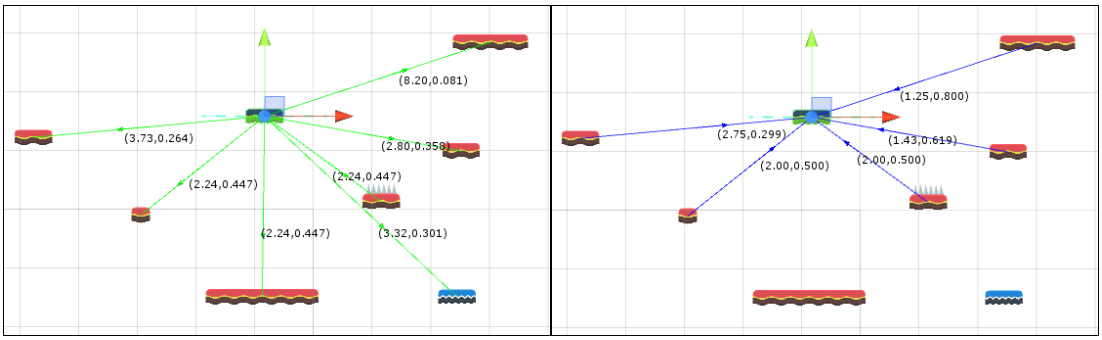


Figure 6.3: Visualization of the outgoing edges (left) and incoming edges (right) for the selected platform.

## 6.3 Platformer Design Tools window

Our framework employs a customized window that provides a set of tools for the design of platformers. The main view of the design window is shown in Figure 6.4. The user interface elements in the green rectangle provide the following set of features: (i) the selection of the noise function used to estimate jumps probability of success among the three described in Section 5.1; (ii) the generation of levels with a selected maximum length (in game units $u$); in particular, the generation can be completely random or with the constraint of feasibility (i.e. a path must exist between the starting platform and the final one); (iii) the possibility to show or hide all platforms in the scene view. Figure 6.5 shows an example of level generated with the feasibility constraint. Note that, the automatic generation of levels is not the focus of this work; as such, this feature is implemented to provide a first rough sketch of a level. In addition, the absence of a check for blocked trajectories (see Section 4.3.3), implies that levels that are generated as feasible might not satisfy the constraint, especially due to the platforms occupying a lot of space vertically. The interface elements in the blue rectangle provide the view of the boundary trajectories (see Section 4.3.3) for the desired pair of platforms; note that, the framework automatically names the platforms using an increasing index (Platform0, Platform1, Platform2 and so on), thus designers only need to input the indexes of the two platforms to obtain the associated boundary trajectories. The button in the red rectangle allows to compute and display the graph of the current level in the scene view. Figure 6.6 shows the graph computed for a small level with the information associated to each directed edge. If any changes are applied to the level (e.g. changing the position of a platform), the graph is marked as dirty and the design tools interface forces the update of the graph before further analysis are performed. Once the level graph has been computed, the design interface is extended to offer additional analysis tools for the level. Figure 6.7 shows the extended interface; the elements in the green rectangle of this extended view allow to view a heatmap of the level based on a

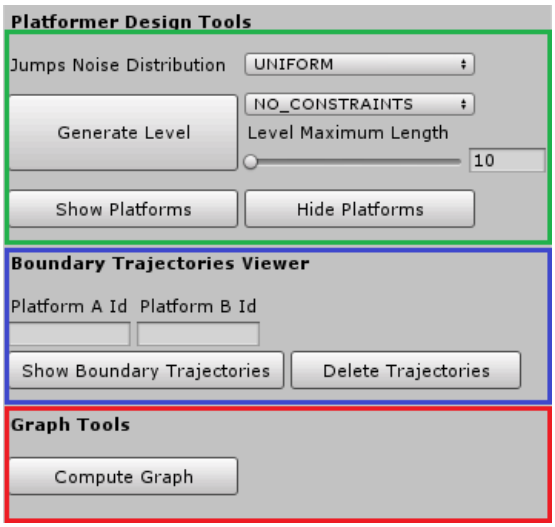


Figure 6.4: A screenshot of the design tools main view.

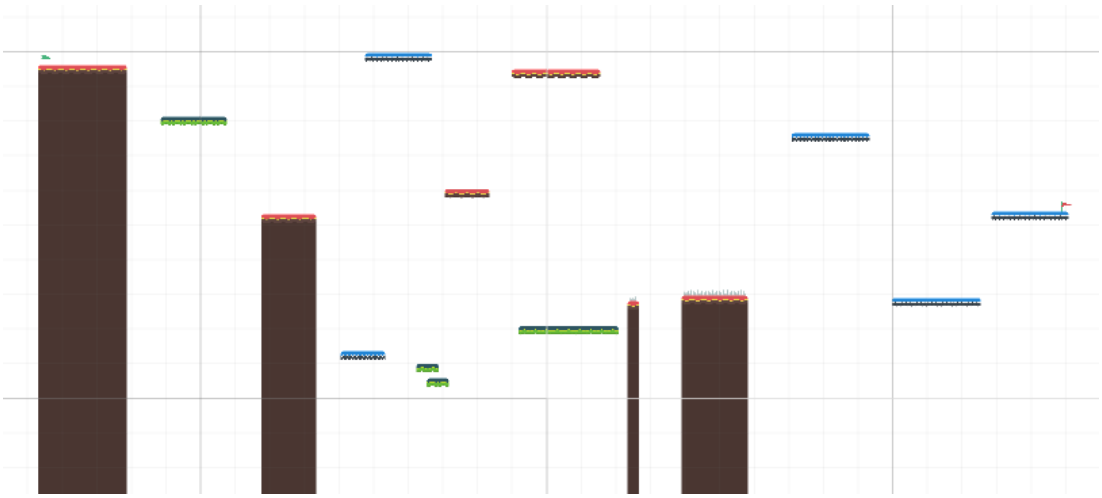

Figure 6.5: An example of level generated with the feasibility constraint.

selected metric (e.g. average probability of the incoming edges); Figure 6.8 shows the heatmap of a level based on the probability of the outgoing edges: the higher the probability the more the color tends to green; viceversa the color tends to red, indicating a critical point in the level. The elements in the blue rectangle of the extended interface provide the following features: (i) show the number of paths available to reach the final platform from the starting one; (ii) view the minimum difficulty path either ignoring harmful elements or considering them; (iii) display the cumulative frequency distribution of the paths probabilities or difficulties (the difficulty metric can be selected among those described in Section 5.3).

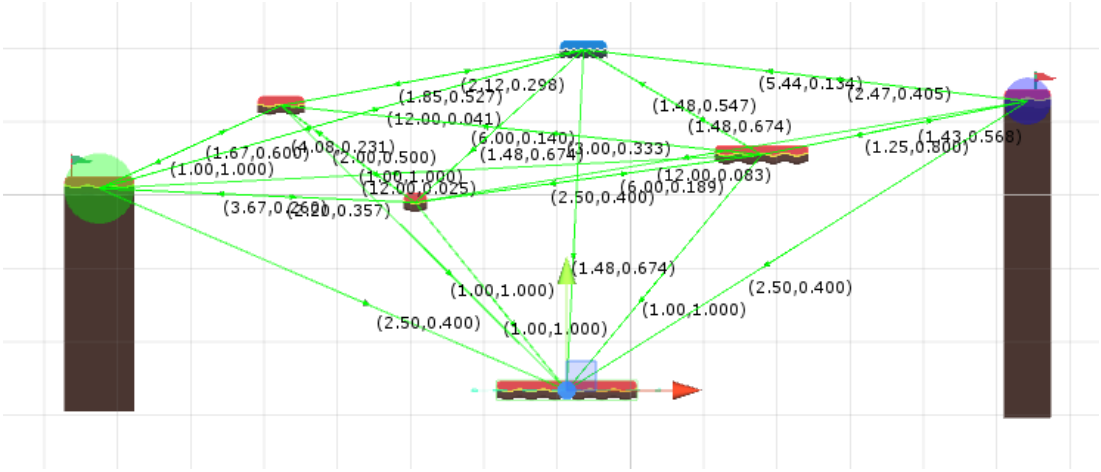


Figure 6.6: The graph computed for a small level.

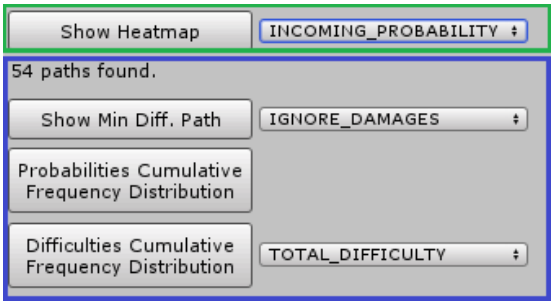


Figure 6.7: A screenshot of the design tools extended view.

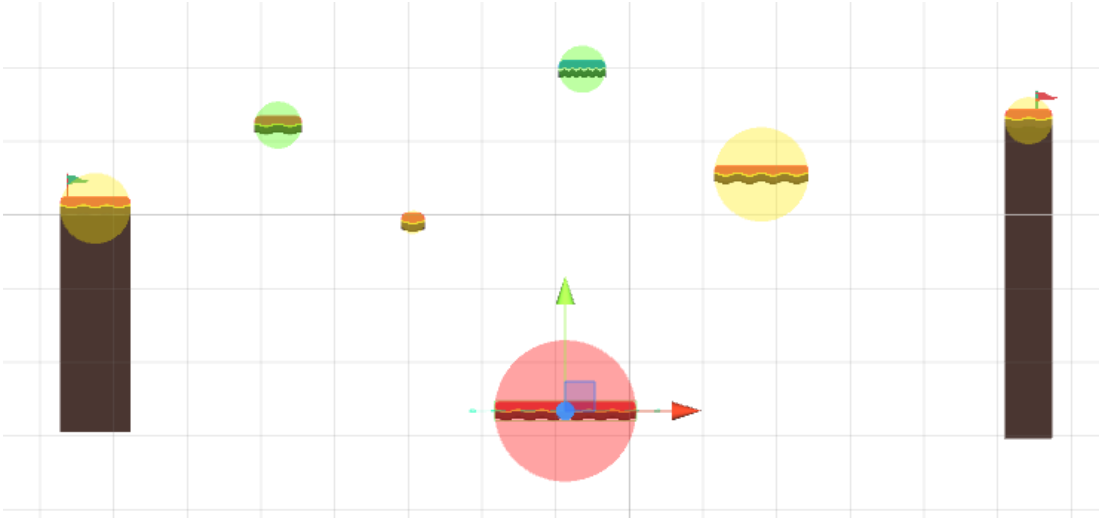

Figure 6.8: A heatmap computed for a level based on the probabilities of the outgoing edges of each platform.

## 6.4 Summary

In this chapter, we overviewed the tools provided by our framework within the Unity game engine and described the interfaces the designers can use to customize platforms and evaluate levels in terms of difficulty and probability of completion.

# Chapter 7

# Conclusions

In this thesis we proposed an approach for the design of 2D platformers and introduced our framework to assist designers in the process of game content creation. We discussed the state of the art for AI-assisted design in videogames and then focused on design metrics employed for the evaluation and procedural generation of platformer levels. Next, we presented the conceptual model of our framework and reviewed how it manages the structural and gameplay features. We examined our method for the evaluation of the jumps difficulty, involving the generation of boundary jump trajectories. Then, we extended this approach to estimate the jumps probability of success using noise functions to generate random takeoff points for sample trajectories around the optimal takeoff point. We collected gameplay data from several users to validate our approach for the estimation of the jumps probability of success and compared the performance of the employed noise models. We proposed a method and a set of metrics to evaluate the difficulty and probability of success of levels, by extending the single jumps scenario under the assumption of independent jumps. Finally, we reviewed the tools provided by our framework in the form of customized interfaces for Unity.

We plan to extend this work in two ways. First, we want to validate the method proposed for the evaluation of levels by collecting gameplay data on levels with distinct features and involving difficulty spikes in specific points; in addition, we plan to verify how our approach handles combinations of different jump types (see Section 4.3.3) by collecting gameplay data on screens involving two consecutive jumps (employing all possible combinations of jump types). Second, we plan to add to our framework a search-based procedural content generator in order to create levels with specific properties or metrics values.

# Bibliography


[1] C. Browne and S. Colton. Computational creativity in a closed game system. *IEEE Conference on Computational Intelligence and Games*, 2012.

[2] S. Dahlskog and J. Togelius. Patterns and procedural content generation revisiting mario in world 1 level 1. *Proceedings of the Workshop on Design Patterns in Games (DPG 2012)*, 2012.

[3] S. Dahlskog and J. Togelius. A multi-level level generator. *IEEE Conference on Computational Intelligence and Games*, 2014.

[4] M. Fasterholdt. You say jump, i say how high? Master's thesis, IT University of Copenhagen, 2015.

[5] L. Ferreira, L. Pereira, and C. Toledo. A multi-population genetic algorithm for procedural generation of levels for platform games. *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion (GECCO Comp '14)*, 2014.

[6] L. N Ferreira. Streamlevels: Using visualization to generate platform levels. *ACM Computers in Entertainment*, 2015.

[7] D. Karavolos, A. Bouwer, and R. Bidarra. Mixed-initiative design of game levels: Integrating mission and space into level generation. *Proceedings of the International Conference on the Foundations of Digital Games*, 2015.

[8] A. Liapis and G. N. Yannakakis. Refining the paradigm of sketching in ai-based level design. *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2015.

[9] A. Liapis, G. N. Yannakakis, and J. Togelius. Designer modeling for personalized game content creation tools. *Proceedings of the AIIDE Workshop on Artificial Intelligence & Game Aesthetics*, 2013.

[10] A. Liapis, G. N. Yannakakis, and J. Togelius. Sentient sketchbook: Computer-aided game level authoring. *Proceedings of the 8th Conference on the Foundations of Digital Games*, 2013.

[11] A. Liapis, G. N. Yannakakis, and J. Togelius. Towards a generic method of evaluating game levels. *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2013.

[12] A. Liapis, G. N. Yannakakis, and J. Togelius. Designer modeling for sentient sketchbook. *IEEE Conference on Computational Intelligence and Games*, 2014.

[13] D. Loiacono, R. Mainetti, and M. Pirovano. Volcano: An interactive sword generator. *Games, Entertainment, and Media*, 2015.

[14] P. Lopes, A. Liapis, and G. N. Yannakakis. The c2create authoring tool: Fostering creativity via game asset creation. *IEEE Conference on Computational Intelligence and Games*, 2014.

[15] R. Maddegoda and A. S. Karunananda. Multi agent based approach to assist the design process of 3d game environments. *The International Conference on Advances in ICT for Emerging Regions*, 2012.

[16] P. Mawhorter and M. Mateas. Procedural level generation using occupancy-regulated extension. *IEEE Conference on Computational Intelligence and Games*, 2010.

[17] N. Nygren, J. Denzinger, B. Stephenson, and J. Aycock. User-preference-based automated level generation for platform games. *IEEE Conference on Computational Intelligence and Games*, 2011.

[18] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience in super mario bros. *IEEE Conference on Computational Intelligence and Games*, 2009.

[19] J. M. Peña, J. Viedma, S. Muelas, and A. LaTorreand L. Peña. Designer-driven 3d buildings generated using variable neighborhood search. *IEEE Conference on Computational Intelligence and Games*, 2014.

[20] W. M. P. Reis, L. H. S. Lelis, and Y. Gal. Human computation for procedural content generation in platform games. *IEEE Conference on Computational Intelligence and Games*, 2015.

[21] N. Shaker and M. Abou-Zleikha. Alonewe can do so little, togetherwe can do so much: A combinatorial approach for generating game content. *Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014.

[22] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten. The 2010 mario ai championship: Level

generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[23] N. Shaker, G. N. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[24] N. Shaker, G. N. Yannakakis, and J. Togelius. Digging deeper into platform game level design: Session size and sequential features. *Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications)*, 2012.

[25] A. M. Smith, M. J. Nelson, and M. Mateas. Prototyping games with biped. *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*, 2009.

[26] G. Smith, M. Cha, and J. Whitehead. A framework for analysis of 2d platformer levels. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 2008.

[27] G. Smith, J. Whitehead, and M. Mateas. Tanagra: A mixed-initiative level design tool. *IEEE Transactions on Computational Intelligence and AI in Games*, 2010.

[28] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[29] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[30] A. Sullivan, M. Mateas, and N. Wardrip-Fruin. Questbrowser: Making quests playable with computer-assisted design. *Proceedings of the Digital Arts and Culture Conference*, 2009.

[31] S. L. Tanimoto, T. Robinson, and S. B. Fan. A game-building environment for research in collaborative design. *IEEE Conference on Computational Intelligence and Games*, 2009.

[32] T. Tutenel, R. M. Smelik, and R. Bidarra. Using semantics to improve the design of game worlds. *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*, 2009.

[33] G. N. Yannakakis, A. Liapis, and C. Alexopoulos. Mixed-initiative co-creativity. *Proceedings of the 9th Conference on the Foundations of Digital Games*, 2014.