FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Assessing fun in platform games

Diogo Pinela

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Henrique Lopes Cardoso

Co-Supervisor: Prof. Luís Teófilo

July 19, 2015

Assessing fun in platform games

Diogo Pinela

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Dr. Ana Paula Cunha da Rocha
External Examiner: Dr. Pedro Miguel do Vale Moreira
Supervisor: Dr. Henrique Daniel de Avelar Lopes Cardoso
July 19, 2015

Abstract

Platform games are one of the oldest video game genres, and they have recently gained in popularity due to their simple mechanics. However, they can become repetitive once beaten, and thus stand to benefit greatly from procedural content generation, extending their length without boring the player. There are several methods to procedurally generate levels; some assemble low-level components directly, whereas others base their constructions on more abstract representations of the level's structure.

In this dissertation, we have developed a three-phase level generation method for *Sonic* levels, based on abstract representations, using a graph to represent the possible paths through the level. These paths are then allocated space on the map by transposing the graph onto a grid, keeping its topology. Once this is done, the rectangular zone of the level corresponding to each occupied block in the grid is constructed independently from the others. In addition, to add a new element of fun and variety, the developed level generator also includes a boss generator, which allows each level to feature a unique challenge at the end, like the classic *Sonic* games did.

The generator's output can be controlled by a set of parameters, which determine the probabilities of various elements appearing during the game. In order to adjust the generated levels to the player's preferences, a method of inferring and parameterising these preferences from gameplay data has been developed, as well as a way of calculating appropriate parameters for the generator from the preference parameters. The developed game was then subject to a test, where several people were asked to play a set of consecutive levels in a session to assess the effectiveness of the adaptation mechanism. The results indicate that the approach followed shows promise, although there is yet much to improve.

Resumo

Os jogos de plataformas são um dos géneros mais antigos de videojogos, e têm ganho popularidade recentemente devido às suas mecânicas simples. No entanto, podem tornar-se repetitivos depois de terminados, pelo que podem beneficiar significativamente do uso de geração procedimental de conteúdos, estendendo a sua duração sem aborrecer o jogador. Há vários métodos para gerar níveis procedimentalmente; alguns juntam componentes de baixo nível diretamente, enquanto outros baseiam as suas construções em representações mais abstratas da estrutura do nível.

Nesta dissertação, desenvolvemos um método de geração de níveis em três fases para níveis de *Sonic*, baseado em representações abstratas, usando um grafo para representar os caminhos possíveis através do nível. Depois, é alocado espaço no mapa a esses caminhos pela transposição do grafo para uma grelha, mantendo a sua topologia. Feito isto, a zona retangular do nível correspondente a cada bloco ocupado na grelha é construída de forma independente das outras. Adicionalmente, para adicionar um novo elemento de diversão e variedade, o gerador de níveis desenvolvido inclui também um gerador de *bosses*, o que permite a cada nível incluir um desafio único no fim, como faziam os jogos clássicos de *Sonic*.

A saída do gerador pode ser controlada por um conjunto de parâmetros, que determinam as probabilidades de vários elementos aparecerem durante o jogo. Para ajustar os níveis gerados às preferências do jogador, um método de inferir e parametrizar essas preferências a partir de dados de jogo foi desenvolvido, bem como uma forma de calcular parâmetros apropriados para o gerador a partir dos parâmetros de preferência. O jogo desenvolvido foi então sujeito a um teste, em que foi pedido a várias pessoas que jogassem um conjunto de níveis consecutivos numa sessão para avaliar a eficácia do mecanismo de adaptação. Os resultados indicam que a abordagem seguida é promissora, embora haja ainda muito a melhorar.

Contents

List of Figures

LIST OF FIGURES

List of Tables

Chapter 1

Introduction

1.1 Context

Platform games have seen a recent increase in popularity due to their simple mechanics being a good fit for modern mobile devices (smartphones and tablets). Thus, any technological innovations that benefit the development of these games - such as procedural generation of content - have a potentially large commercial impact on the market. In addition, also due to their simplicity, platform games are an ideal medium for testing and evaluating such content generation techniques. Our work aims to take advantage of this, by improving existing techniques for procedural generation of platform game levels, with an emphasis on optimising for player entertainment and the adjustment of difficulty levels in order to enable the generation of a sequence of levels that forms an enjoyable game.

1.2 Motivation

Procedural content generation is an increasingly common way of extending the replayability of video games, while reducing their development cost, which is especially advantageous to smaller, independent developers who lack the resources to create extensive environments by hand [1]. Platform games especially stand to benefit from this, as they are largely single-player and player-vs.-environment, which makes them quickly become repetitive once the player first completes them.

However, merely extending the length of a game is not sufficient to increase its entertainment value; in fact, levels that are too repetitive or have an inadequate difficulty level will leave the player bored at best, and frustrated at worst. Thus, we need to take into account the player's preferences so that we can create levels that are fun to play for them.

Beyond increasing the replayability of games, this mechanism also has the advantage of personalising games for each individual player, which hopefully should be a boon to their satisfaction as well.

1.3 Goals

Given the problem at hand, the main goals of our work were:

- To create a way of assessing a player's preferred level and gameplay styles during the course of normal gameplay (rather than in a dedicated testing session), to enable the generator to tailor its output to the player.
- To improve upon current level generation techniques by using richer structural design (especially having multiple paths through the level) and gameplay elements (enemies, items, and most interestingly, bosses), to allow machine-generated levels to approach the feature gamut of human-designed ones.
- To develop a method of assembling a set of levels with an adequate difficulty curve to produce a nearly-complete game, to further expedite the game's development process.

By pursuing these goals, we intended to verify the hypothesis that, by using an abstract representation of game levels in conjunction with a parametrised generation process for that representation, it is possible to learn player's preferred level styles and use that information to dynamically create game levels that mantain their interest and fun throughout the game.

Chapter 2

Background

2.1 Platform games

Platform games are a kind of video game whose gameplay consists of guiding a virtual character (also known as an *avatar*) to traverse a series of environments, usually called *levels* or *zones*, by making it run and jump across arrangements of ledges called *platforms*, overcoming or bypassing obstacles, and sometimes also solving puzzles (which may require finding hidden switches or pathways, or performing a sequence of well-timed jumps between moving platforms). Levels are composed of a variety of elements, each providing a different kind of challenge [10]:

- *Platforms*, surfaces atop which the player can stand or move (see Figure 2.1); these may be static or moving and, in some cases, temporary (disappear a set time after they spawn, or the player steps on them).

Figure 2.1: A pair of moving platforms in *Sonic the Hedgehog 3*. These particular platforms swing periodically up and down, allowing access to a higher route through the level.

- *Movement aids*, objects that help the player move from one place to another. Examples include springs, trampolines, ropes, conveyor belts or speed boosters.

- *Obstacles*, objects that in some fashion impede the player's progress or deal damage to it, but otherwise do not change position or attempt to actively attack the player. Gaps between platforms can also be considered obstacles.

- *Enemies*, a type of non-player character (NPC) that can attack the player. Enemies can usually be destroyed, and, like some obstacles, deal damage to the player when touched.

- *Triggers*, objects that, when touched, cause the level to change in some way, such as opening or closing passageways or spawning enemies, collectibles, or power-ups.

- *Collectibles*, objects that serve only to increment the player's score when hit. Rings in *Sonic* games and coins in *Mario* games are examples of this.

- *Power-ups*, objects that grant the player bonus abilities or powers when hit, such as shields, weapons, speed boosts, invunerability, or lives (see Figure 2.2).

Figure 2.2: A set of power-ups in *Sonic 3 & Knuckles*, including a speed boost, invulnerability, and three different types of shield.

- *Bosses*, enemies with special abilities usually encountered at the end of levels or in between them (see Figure 2.1), which are substantially more difficult to destroy than a typical enemy. They are usually associated with the game's main antagonist (for example, most *Sonic* bosses are piloted by Dr. Eggman, Sonic's principal foe) or are the antagonist themselves, and must be defeated in order to progress further into the game.

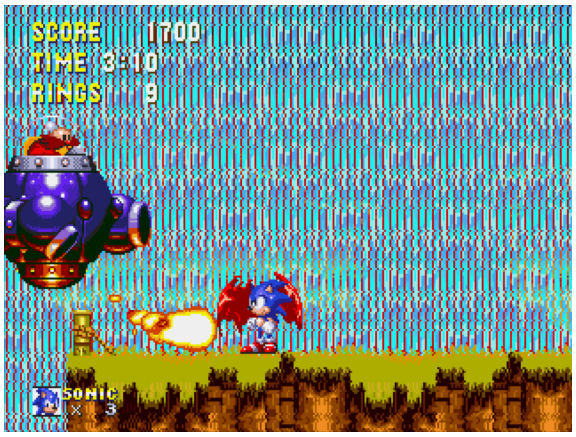

Figure 2.3: Sonic facing a boss in *Sonic the Hedgehog 3*.

In order to traverse the level, the player controls a character, which typically has at least the ability to walk or run across the ground (left or right in a 2D game, in all directions in a 3D one), as well as the ability to jump into the air, and jump onto, out of or between platforms. In action platformers, the character is also often provided with weapons with which they can attack enemies [10].

To allow players some margin of error, they are usually provided with a number of attempts to beat the game, almost universally called lives; taking fatal damage will decrement the life count and return the player to an earlier point in the game. Gaining additional lives is accomplished by picking up a life-granting item, or, sometimes, by collecting a certain amount of collectibles (such as collecting 100 rings in a *Sonic* game).

Platform games can have both 2D and 3D environments; the former is typical of 1990s-era games and the latter of more modern ones. Some games (like the 2008 title *Sonic Unleashed*) combine both kinds of environment, or use levels with a 2D logical structure (that is, the player's movement is restricted to a plane, as in pure 2D games), but with 3D graphics; these are known as "2.5D".

2.2 "Fun"

The concept of "fun" is commonly understood to be synonymous with a person's enjoyment of a given activity, in this case playing a video game. Since it is well-known that people have wildly varying preferences, it follows that the concept is highly subjective; for this reason, some authors have proposed the use of more specific factors that can make a game fun, each of which corresponds to a different source of enjoyment [3]:

- *Sensation*, derived from experiencing the game environment.
- *Fantasy*, derived from enacting one's imaginary adventures within the game.
- *Narrative*, derived from experiencing the story of the game's characters and environment.

- *Challenge*, derived from overcoming certain obstacles, enemies, or otherwise difficult areas.
- *Fellowship*, derived from social interactions centered around the game.
- *Discovery*, derived from exploring and discovering unknown areas in the game environment.
- *Expression*, derived from the exploration and expression of one's own personality and tastes within the context of the game.
- *Submission*, related to the use of the game as a pastime.
- *Immersion*, related to the degree to which the player feels removed from the real world and involved with the game experience instead.

  Of these factors, the most relevant for this work are *sensation* and *challenge*, as the goal of the developed level generator is to create levels that, on one hand, provide an adequate (not too easy or too difficult) challenge to players, and, on the other, feel like fresh, engaging and interesting environments.

2.3 Procedural content generation

As the name indicates, procedural content generation (or PCG for short) is the automated generation of the challenging or interesting elements of a game, which ordinarily are produced by human designers: levels, bosses, enemies, and other kinds of non-player characters (NPCs).

Chapter 3

State of the art

3.1 Level generation techniques

Over the past few years, several different approaches to procedural level generation for platform games have been pursued. The approaches we have observed appear to fit into two categories: *bottom-up* methods that simply combine small, low-level elements, usually in an incremental fashion; and *top-down* methods that first construct an abstract, high-level model of the level's structure and then translate it into actual level objects and platforms (in much the same way as a compiler translates human-readable source code into machine-readable instructions).

3.1.1 Bottom-up methods

Bottom-up methods of level generation, as stated above, rely on concatenating or combining small pieces of levels to do their job. These pieces may be as small as individual pieces of ground, hills, pipes, enemies, blocks and coins (as in the *ProMP* generator described in [9]), which are placed independently of each other; or small hand-composed arrangements of platforms and enemies, named *chunks* [5], which are then concatenated seamlessly (that is, without introducing noticeable edges between components) into an entire level.

Another example of this kind of method can be found in a recent MSc thesis by Nelson Oliveira [7], who implemented a level generator that works by stringing together *cells*, small sections of level each generated randomly based on one of six types (the last two of which are available only on the highest difficulty setting provided):

- Straight lines (*Straight*)
- Zones with gaps (*Jump*)
- Zones with hills (*Hill*)
- Straight lines with a row of randomly-chosen enemies (*Dunking*)
- Zones featuring a ladder-shaped platform for the player to climb (*Manoeuvre*)

- Zones requiring the player to climb a vertical chute by leaping from one wall to the other repeatedly (*Wall Jump*)

In addition, after selecting the two best levels according to the fitness function used by the author, the generator combines each pair of corresponding cells of these two levels into a cell containing characteristics of both. Any two types can be combined, for a total of 15 types of cell in the final output.

The *ProMP* generator exemplifies a seemingly-inherent issue with this kind of generator: it is able to generate levels that have a high density of features, or "microstructure" [9], but that lack "macrostructure", that is, a set of clearly-defined and separated paths through the level and junctions where one may switch between them.

3.1.2 Top-down methods

Rhythm-based level generation [11] is an example of how to generate levels in a high-level fashion. It starts by determining the sequence of actions that the player is expected to perform (called a *rhythm group*); then, it places a series of platforms and objects that match the rhythm. The chunks of level generated from these rhythm groups are then joined by small "rest areas" where the player can pause before proceeding to the next challenge.

To enable control over the characteristics of the generated levels, this generator uses two mechanisms, the first of which are style parameters. These include values such as the length and frequency of jumps, number of coins included in a rhythm group, space between coins.

The other mechanism is a set of *critics*, which measure how far a particular property of the level is from the ideal. The authors implement two critics: one that judges the overall distance between the sequence of platforms and the idealised path through the level; and one that compares the component distribution of each level to the style parameters used. Several levels are generated, and the one for which a weighted average of all critics is lowest is chosen.

This approach allows the generator to ensure the presence of features that are deemed interesting to players, such as sections that require leaping over obstacles or gaps. This still does not address the issue of levels lacking multiple paths or distinct bonus sections, but it contributes to making the level less of a random assemblage of pieces.

Another approach, using a genetic algorithm, has been proposed by Diaz-Furlong and Solis-Gonzales [1]. The authors propose the use of a genetic algorithm to create levels for a game of their own design, which consists of traversing tube-shaped levels while avoiding obstacles and holes, and collecting items. While the game is not a platform game, it shares several concepts with the genre, and the algorithm was designed to be usable for any kind of game. The fitness function used is based on the mean square error between an idealised difficulty curve, and the estimated difficulty curve of the generated level, based on a measure of the individual difficulty of each segment:

$$f(c) = 1 - \sum_{i=0}^{n-1} \left(d\left(\frac{i}{n-1}\right) - d_m(c,i) \right)^2 \tag{3.1}$$

In Equation 3.1, c is the generated level, i is the index of each segment of the level (from 0 to n - 1), n is the number of segments, d is a function giving the value of the pre-defined difficulty curve at a point (between 0 and 1) in the level, and d_m is a function giving the estimated difficulty of segment i of the generated level.

At last, an unusual type of generator has been proposed by Kerssemakers *et. al.* [4]: a "procedural procedural level generator generator", that is, a generator of level generators. The authors describe a method to describe a level generator parametrically, by combining a set of *agents* that concurrently move through the level space and randomly place platforms, obstacles and other objects according to certain parameters, such as where they spawn, how they move and how frequently, when they trigger (place an object), and what they do when triggered.

An interactive genetic algorithm is then used to determine the parameters; at each step, the user can evaluate the current generation of generators - each represented by a sample of levels generated by it, which can be visualised or even played -, and choose which ones will serve as the basis for the following generation. Each generator also receives a playability rating, determined by having an AI attempt to play a sample of levels from it; the rating is the proportion of those levels that the agent successfully completes.

3.1.3 Decoration

While the methods for generating a level's main structural platforms vary, one common aspect of most of them is that smaller items, such as power-ups and coins, are placed after the main geometry and platforms have been created. Their placement is typically governed by a probability value that controls how frequently the generator will place such items in the player's path.

The decoration of levels may be aided by an abstract representation of the level structure, such as a graph [6] representing the possible paths through the level. The authors propose a method of extracting such a graph from an already-generated level and using the information to inform the placement of enemies, collectibles, power-ups and triggers (enabling additional paths through the level), such as using dead-ends to house bonus items, or ones required to pass through the level (so that the player must pass through these areas).

3.1.4 Visuals

While current research on procedural level generation focuses on the placement of platforms, terrain and objects, another important aspect of any platform game level is its visual theme. The elaborateness of the game's graphics varies considerably depending on the game's art direction and the platform's capabilities; for example, while classic *Mario* games (Figure 3.1) from the early 1990s featured relatively simplistic art, contemporary *Sonic* games of the same period (Figure 3.2) used more ornate graphics and visual effects.

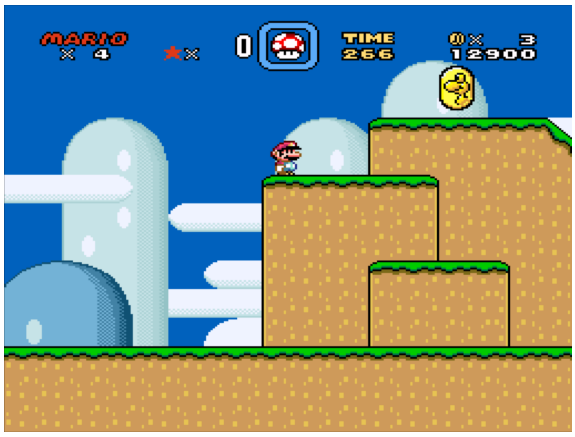

Figure 3.1: The first level of *Super Mario World*.

Figure 3.2: The first level of *Sonic the Hedgehog 3*. The underwater area has a ripple effect not visible in this static picture.

There is research on the subject of procedural generation of decorative elements for computer game levels in general, but it seems to focus on creating realistic-seeming graphics and replicating real-life artistic patterns, such as Gothic or Islamic ones; some authors propose the creation of a computational representation of decorative styles (and not specific patterns), in such a form that it is possible to manipulate, modify or combine them, and thus construct a generator which can generate new ones [14]. This capability could be useful for generating styles for platform games, which frequently feature sets of levels with a similar visual theme.

3.2 Playstyle determination

To generate levels which keep the player entertained, and given that, as stated before, different players find different things entertaining, we must first find out the target player's preferences. One way to accomplish this [8] is to have them play some randomly-generated levels, while recording statistics such as the number of jumps performed, enemies destroyed, coins collected, etc.; and then ask them how fun, frustrating or challenging they found the level. With this data, it is possible

to create a model (such as a neural network) that predicts the player's level of predicted fun from the generator's input parameters and the collected statistics.

To test the effectiveness of this method, *in [8]*, the authors feed random parameters into the generator, and have two different AIs created by Robin Baumgartner and Sergio Lopez (presented as entries to the 2009 Mario AI Championship's gameplay track) traverse the resulting levels. Using the gameplay statistics recording from those playthroughs, they generate levels adapted for each AI and compare the predicted fun value to that obtained in the random levels. For both AIs, the result is an improvement in fun; this improvement is more pronounced for Lopez's AI, which the authors attribute to it having a more human-like behaviour, and the fact that the model used to generate the levels was initially trained using data from human players.

Similarly, the generator described in [7] bases its evaluation mechanism on player surveys done during its development. The author had the players play several levels from the original *Super Mario Bros.*, with varying difficulty levels, and surveyed them on how challenging and how fun each type of zone (see Section 3.1.1) was, in a scale of 1 to 5, as well as what they deemed to be the ideal number of gaps, hills, enemies, and blocks. From this, he assigned ratings to each type of zone; the generator computes the fitness function for a candidate level as the sum of the ratings of the zones it contains.

Another way, which requires less work on the part of the player, is to compute, from each generated level, a formal measure that is believed to be correlated with the player's enjoyment. In [13], the authors take an estimate of challenge based on the distance and margin of error of a jump between the platforms found at each time step, and integrate this over time to obtain a curve of the player's *anxiety* (a) over time, as shown in Equation 3.2 - $c(t)$ is the aforementioned challenge estimate at time step t, and c_{decay} is a constant which indicates how fast anxiety falls off in the absence of challenge:

$$\frac{\mathrm{d}a}{\mathrm{d}t} = c(t) - c_{decay} \tag{3.2}$$

This curve is then computed for several levels from the original *Super Mario Bros.*, in order to discover patterns characteristic of human-designed levels. Such levels, as the authors note, frequently have a curve with a lower slope in the early parts and a peak near the end; this is consistent with our experience with platform games, which tend to feature their most challenging and climatic sections at that point.

The authors go on to create two mathematical models of "fun" (f) based on the notions of challenge and anxiety, and use a genetic algorithm to generate levels using them as fitness functions (c_{skill} is a constant representing the player's skill level):

$$\frac{\mathrm{d}f}{\mathrm{d}t} = -(c_{skill} - c(t))^2 \tag{3.3}$$

$$\frac{\mathrm{d}f}{\mathrm{d}t} = m\frac{\mathrm{d}a}{\mathrm{d}t} \tag{3.4}$$

The first model (Equation 3.3) seeks to minimise the overall disparity between challenge and skill; however, it is found to be ineffective, since it produces levels that appear to be "a chaotic smattering of platforms" and have a linear anxiety curve.

The second model (Equation 3.4) seeks to correct this by introducing a factor *m* that is 1 when anxiety is below a certain threshold, and -1 otherwise. The result is closer to those observed in the *Mario* levels, though the characteristic peak at the end is still missing.

3.3 Fun evaluation

Once we have a model of the player's preferences, and a generator using that model, we must discover how well that generator accomplishes our goal of entertaining the player. The most obvious way to do this is having human players play-test levels created by the generator, and then measuring their response in some way, such as a survey.

Surveying approaches vary; for instance, in Oliveira's evaluation surveys [7], play-testers were asked not only whether generated levels of 3 different difficulty levels were fun and challenging, but also, like in the previous survey from that thesis, asked them how fun and how challenging each type of zone was (in a scale of 1 to 5), to enable comparisons with the corresponding measures regarding human-designed levels. In addition, players were asked directly whether they felt that the generated levels would "blend in" among original *Mario* levels. The results seem to have confirmed the efficacy of the generator, although the fact that 100% of the testers responded affirmatively to the last question may indicate that a larger sample of players is needed to obtain a realistic result.

Similarly, in the surveys described in [1], the players were instead asked to rate six generated levels according to three parameters: how fun and how challenging they were, and how good their level design was. The results confirmed the expectations of the authors regarding the relationship between the difficulty curve used to generate the levels and the players' satisfaction with them; levels with a more varied curve tended to fare better.

Chapter 4

Game engine

The game engine underlying this work, called SonicKit, was originally developed by the author to serve as a framework for creating classic Sonic[1]-like games. To this end, it provides implementations of the essential physics and controls from the classic games, common objects (such as rings, springs or item monitors), and auxiliary structures for the purpose of loading resources (sprites, sounds, and level layouts) and arranging objects into levels. The engine is intended to be extensible, allowing the addition of new kinds of object with new behaviours, as well as platforms with arbitrary shapes.

4.1 Architecture and object model

SonicKit uses an object-oriented architecture, in order to simplify the task of extending it with new objects, as well as ease understanding of the game's core logic. As such, all entities present in the game's levels, as well as the player itself, are modelled by objects, as well as the levels themselves.

The most important classes of the engine are:

- `Runner` - responsible for executing the main loop of the game, initially receiving input events and periodically calling the methods for updating and rendering the game state.
- `Player` - implements Sonic's generic movement physics and controls, as well as handling his collision with platforms such that he can run and jump correctly across them. Specific characters and their animations are defined by subclasses of this.
- `Object` - the base class for all objects found in levels. While implementing little concrete behaviour itself, it defines the interface for all other objects to follow. The object module, where this is located, contains several subclasses (for example, `SpriteObject`) which implement features common to many objects, such as having a sprite, colliding with platforms or being destructible.

[1] The original Sonic the Hedgehog games, released for the Sega Mega Drive in the early 1990s.

- `Level` - encapsulates all of the elements needed to define a level: its object layout and starting positions. At runtime, it calls the drawing and updating methods of the player and all applicable objects and measures the distances of the player (and some objects, such as bouncing rings) to solid objects to enable collision detection (see section 4.3, Physics model).

The execution loop of a game using this engine follows a cascading pattern. The main loop, in the `Runner` class, periodically calls two methods on the currently-running screen (which will be a level, most of the time; the Game Over screen is another example), named update and draw. The update method is responsible for updating the game's state and responding to the player's input; the draw method is then called to render that state onto the screen. Both the player and the objects within the level (including platforms) have, in turn, their own update and draw methods, which the level's methods call as part of their execution. A consequence of this design is that the majority of the game's logic is directly or indirectly contained within the various implementations of update.

To allow objects to interact with the player and define their specific behaviours, there is a set of methods, in addition to update, that are defined in the `Object` class, which are called in response to the player colliding with the object on each of its four sides (top, bottom, left and right). Subclasses may override these methods to react to player collisions; for instance, the `Spikes` class reacts to the player colliding with its top side by damaging them.

In the course of detecting collisions and drawing the level, it is necessary to determine which objects intersect various rectangles, such as the player's hitbox and the camera's view rectangle. To speed up the determination of this set of objects, the engine divides the level space in tiles (256 by 256 pixels by default), each tile containing a list of objects that intersect that tile. This structure can then be searched efficiently for objects that intersect any arbitrary rectangle, by looping over the tiles that intersect it, and checking each object within each of those tiles for whether it intersects the rectangle as well.

Due to performance concerns, not all objects are drawn and updated at every step, but only those which are currently visible on the screen; with the exception of moving objects, which are updated (but not drawn) unconditionally every step to prevent them from freezing in place as soon as they go offscreen.

4.2 Physics model

One of the most notorious features of the original Sonic games, which this engine aims to replicate, are their physics, which allow the player to run smoothly across sloped platforms and hills and even along walls and loops. Thus, the collision detection mechanism must be able to measure the player's distances and angles to the ground, walls and ceilings, and continuously modify Sonic's speed along the X and Y axes according to both the detected shape of the ground and the player's input [12].

In SonicKit, this is achieved by measuring distances to solid platforms along a set of sensor lines associated with the player, some vertical and some horizontal; if no platform is found, the value for the corresponding sensor is defined as positive infinity.

These values are then fed into the player object's collision detection methods, which determine, based on them, if it is colliding with walls, ceilings or ground, and, if relevant, at which angle. Whenever the player is running on walls and ceilings, which is detected when the player's angle exceeds certain thresholds, the sensors are replaced with a set of rotated ones, such that the sensors that ordinarily point downwards point towards wherever the ground is. There are, thus, four sets of sensors for ground movement - for floor, ceiling, and left and right walls - as well as a slightly modified set for in-air movement.

To emulate the classic Sonic physics and behaviour as well as possible, the game logic updates itself one step at a time, at a rate of 60 steps per second. This rate is kept independent of the display frame rate, to ensure that the physics and timing function correctly even if the frame rate drops. In addition, when drawing, the engine interpolates the last two states of the physics simulation, specifically the player and the camera's positions, to avoid temporal aliasing[2][2].

[2] A phenomenon where on-screen movements stutter, due to the physics simulation being out of sync with the drawing cycle.

Chapter 5

Level generation

5.1 Concept and motivation

Our generator constructs levels by a sequential three-phase process. First, it generates a representation of the logical structure of the level; then, in the second and third phases it creates an arrangement of actual platforms and objects that matches this structure as closely as possible. This separation of concerns has two benefits: it lets the generator define levels in terms of their logical structure (i.e. possible paths through them); and it allows either phase of the generation to be performed by a human designer instead - for instance, said designer could manually craft a structure and still save the work of implementing that structure, combining the benefits of procedural content generation with a measure of human creativity and refinement.

In light of this, each of the three phases strives to be independent of the others: that is, the algorithms applied in each were designed, as far as possible, to accept any input of the required type, whether or not it was created by the generator itself.

5.2 First phase: structure generation

The level structure is modelled by a directed graph (see Figure 5.1 for an example), where the nodes correspond to so-called "points of interest" (POIs). A POI is, broadly speaking, any point in the level where the player has to overcome a challenging element, can obtain some kind of bonus, or faces a meaningful choice. Examples of POIs include forks and merges[1], locations with hidden or hard-to-reach bonuses, areas with higher than average concentration of enemies, or special enemies - such as bosses - that are substantially harder to defeat than most.

[1] Merges: points where two or more paths converge into one

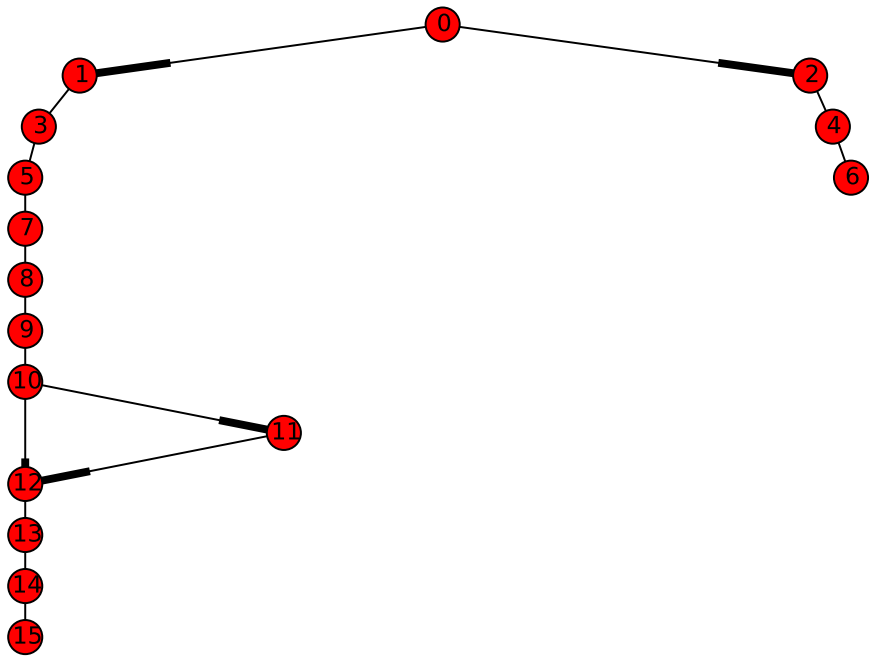

Figure 5.1: An example of a level structure graph created by the generator.

Forks and merges are a particularly important type of POI, because they are the points from which the player can enter and exit alternative paths through the level; hence, the difficulty of these segments corresponds to the difficulty of reaching those paths. They are represented in the graph as normal nodes with an out-degree or in-degree, respectively, greater than 1.

The edges of the graph, in turn, represent the parts of the level that allow the player to travel between one POI and another. To give the level an adequate rhythm, and avoid inducing player errors, these sections should be low-difficulty, with few obstacles, so that they effectively serve as safe rest areas between challenging zones [10].

Using this representation, the overall size of the level can be controlled by the number of nodes in the graph. This is a more relevant metric for gameplay purposes than the physical length or number of platforms, since sections of the level outside of POIs should not require significant effort to go through.

In the actual implementation of this concept, the different types of node are represented by objects of different classes. The generator defines four types of node:

1. `StartPoint` and `EndPoint` - mark the initial and final points in the level, respectively.

2. `ChallengeArea` - marks an area where the player should have to overcome obstacles, enemies, or both to progress.

3. `InterstitialNode` - marks an intermediate area whose only purpose is to connect other areas in the level. These are not created during this first phase of the level's generation.

4. `BonusArea` - marks an area where the player can collect bonus items and power-ups.

Using these, and a set of corresponding weights (used to choose which type to use when creating new nodes) the generator produces a structure graph by constructing all paths in parallel; while it operates, each path being generated is represented by its most recently added node so far (the "tip" of that path). The algorithm is as follows:

1. Add a StartPoint to the graph. Create one path containing that point.

2. Until the desired number of nodes has been reached or exceeded:

   (a) With probability P_{split}, if the current number of paths is below the limit, split a random path into N paths, where N is a randomly-chosen integer between 1 and the branching factor.

   (b) With probability $P_{deadend}$, if there is currently more than one path, delete a path from the list at random.

   (c) For each path:

      i. With probability P_{merge}, choose the tip of another random path as the path's next node, if there is more than one path. The path is removed (as it has been merged onto another one).

      ii. Otherwise, generate a new node and add it to the path, by changing its entry in the path list to point to that node.

      iii. Add an edge from the previous tip of the path to the chosen successor.

3. Generate an end node, and add edges from the tips of all remaining paths towards it.

4. Return the graph and its starting node.

5.3 Interlude: a first attempt at layout generation

After creating the structure-generation algorithm, we turned towards the problem of generating the level from that structure. The first approach we experimented with was to follow one random path through the graph at a time, until the level was complete, building platforms and objects for each yet-unvisited node; to line up platforms, each node was associated with an "insertion point", which corresponded to the position of the ground at the right end of the last platform built for that node. Building a single path with this system was simple, but splits and merges turned out to be more problematic, for several reasons:

- There was no reliable way to position the alternative paths such that they did not end up overlapping with existing ones.

- Since paths were, by the nature of this mechanism, free-form, it was difficult to map paths to a well-defined area. This sometimes caused two paths to occupy very close positions on the map, creating a degenerate layout (see Figure 5.2).

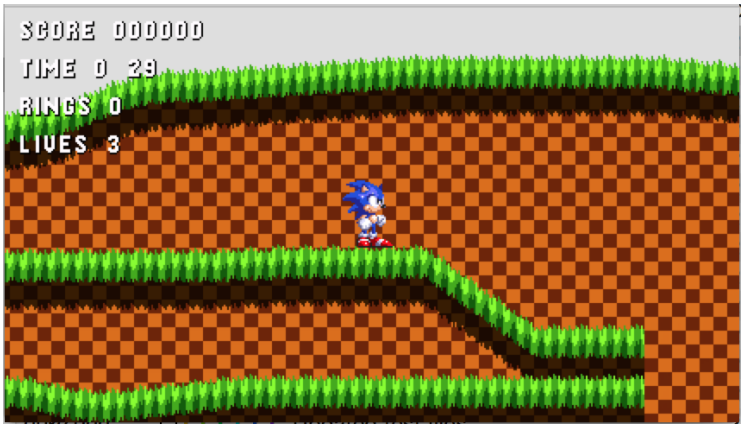

Figure 5.2: Example of a degenerate level layout

- Most importantly, merges were rather problematic; even assuming all paths were horizontal and left-to-right, the general problem of linking together two arbitrary points on the map would be rather complex to solve, particularly with merges between non-adjacent paths (which the structure generator could create), due to the need to avoid overlaps.

The solution

While trying to find solutions to the aforementioned problems, it occurred to us that if the level's structure could be somehow lined up with a grid, creating a layout would be potentially much easier. At first, we considered generating the initial structure graph along a grid, but this would remove the advantage of being able to construct and reason about the level's structure in abstract terms. This then led to the idea of generating the structure using the already developed method, and then aligning its nodes and edges with a grid, before finally constructing the platforms in each cell of the grid; essentially, solving the layout problem by introducing another layer of abstraction.

5.4 Second phase: layout generation

Once the paths through the level are defined, it is necessary to determine what space in the map each of them will occupy. The next step of the generator is thus to allocate parts of the map to each node, in such a way that the connectivity constraints described by the structure graph from the previous phase can be followed.

The generator accomplishes this by dividing the map into a grid of blocks, following one path through the graph at a time, and allocating a sequence of adjacent[2] unallocated blocks corresponding to each node in that path, if that node does not already have a block; the direction of advance is

[2]Where this text uses the word "adjacent", it is referring to 4-adjacency unless stated otherwise.

chosen at random from the available ones, with a weighted probability of choosing each available direction; these weights are parameters to the layout generator. The process ends when every edge in the graph - and thus, every path - has been visited.

Along with the block allocation table, the generator also keeps track of which pairs of adjacent blocks are connected (which will be useful information in the following phase of generation). Between each pair of nodes, the generator also allocates space for an interstitial node, as if it were in the graph; this serves to space out the more interesting parts of the level.

Splits are handled naturally as a result of this; the generator will automatically extend the layout in a new direction upon reaching a split node. To handle merges, it finds a shortest path through the unused grid from the source node to the merge node using a breadth-first search, and adds additional interstitial nodes along that path.

The combination of the above methods essentially transposes the original graph onto the grid, while preserving its topology. However, since only 4-adjacent neighbours are considered as candidates for the layout's expansion, this algorithm cannot handle graphs containing at least one node with a degree greater than 4. Extending this would require either using 8-adjacency instead of 4-adjacency - which would in turn require adaptations to the level implementation functions, with the increased number of possible exits out of a block; or using the interstitial nodes in a more sophisticated manner, although the latter would result in the graph's topology being slightly changed at each 5+-degree node.

More precisely, the algorithm is as follows:

1. Assign the block (0, 0) to the initial node of the graph.
2. Let there be an *adjacency map*, containing, for each block, the set of adjacent blocks that are connected to it.
3. While there are unvisited edges:
   (a) Let the current node be the initial node.
   (b) Until the current node is a dead end (has no out-edges):
      i. Choose a random unvisited edge from the current node, if one exists; otherwise choose a random visited one
      ii. If the chosen edge has not been visited:
         - If the destination node has already been assigned a block, perform a breadth-first search to find a shortest path between the source and destination. Add interstitial nodes to the grid and add each consecutive pair of nodes to the adjacency map, including the extremities.
         - Otherwise, choose a random direction (that does not lead to an occupied block), and add an interstitial node along it, as well as an adjacency between the current node's block and the one belonging to the interstitial node. Then, repeat this for the actual destination node.

iii. Let the current node be the destination of the chosen edge.

4. Return the mapping from blocks to nodes, and the adjacency map.

Figure 5.3: An example of a layout generated for the structure graph in Figure 5.1. The red node is the StartPoint, blue nodes are InterstitialPoints, green nodes are BonusAreas, lighter orange ones are ChallengeAreas and the darker orange node at the top-right is the EndNode.

This algorithm is not guaranteed to succeed in any one run; the first part may fail if all four adjacent blocks (to which the layout could be extended) are already allocated; the BFS may fail if the path between the two nodes is blocked, or they are farther away than the iteration limit set for the search (30 in our implementation). To ensure that the layout generation succeeds, while avoiding the relaxation of any constraints, the solution used is to retry the generation if one of these conditions occurs; as long as the algorithm can handle the graph at all, it will eventually succeed, usually in less than 10 attempts. An iteration limit can be specified to prevent impossible-to-layout graphs from hanging the program forever.

5.5 Third phase: level implementation

With the layout obtained in the previous phase, it is at last possible to construct the output level. Each type of node in the layout, including the interstitial nodes inserted by the layout generator, is assigned a function (called an "implementation function") that is responsible for filling in with platforms and other objects the area of the level corresponding to the block allocated to that node - this requires defining the physical size of blocks (which arbitrarily defaults to 256x256). This mapping from node types to functions is called an “implementation table”. Due to the need to construct many low-level details of the level, the implementation functions form the bulk of the level generation code, even though they are, strictly speaking, not part of the generator itself.

Each set of implementation functions represents a particular "theme" of levels, in the same way that all levels in a zone in a classic *Sonic* game follow a similar theme. This is, of course, not a hard rule; since each block is constructed separately from the others, it is entirely possible to mix-and-match functions from different level themes.

These functions take as parameters the rectangular area within which they should place objects, a set of boolean values indicating whether the block being implemented is connected to each of its possible neighbours - and has a neighbour in these directions at all -, and, most importantly, the lower and upper limits of the exits towards adjacent connected blocks (along the edge of the area shared with the other block). Their return value is a 4-tuple containing these same limits, but taking into consideration the objects inserted by the function.

These “exits” indicate the zone through which the player may pass to reach the adjacent block, and are used by the implementation functions to position platforms and navigation aids such that the level is traversable, as well as to align the platforms in each block so that they form a continuous surface. If an exit towards a neighbouring block is not defined, the implementation function may choose one by any method - most often at random -, so long as its coordinates are within the area's limits. The function doing this should then return the exit coordinates it chose - as well as any that were passed as arguments to it - as part of the tuple it returns; by this process, the level generator builds up a record of the locations of exits from one block to another, which are passed as needed to the implementations of subsequent blocks. As a consequence, this method does not depend on blocks being implemented in any particular order.

5.5.1 Implementation functions used

For the purposes of this work, a basic set of implementation functions has been developed, one for each type of node described so far, including interstitial nodes. These functions all share a common backbone, which generates a chunk of ground for the player to run on, taking into account any exits. The height of the ground, relative to the bottom of the block it occupies, is generated (except for the case where the ground has a hole) by the following sigmoid function, where L and R are the heights at the left and right ends of it, and x_i and x_f area the X coordinates of the left and right edges of the block, respectively:

$$h(x) = L + \frac{(R-L)(erf(\frac{4}{x_f - x_i}(x - x_i) - 2) + 1)}{erf(2) + 1} \tag{5.1}$$

If L or R are not yet known, they are determined by choosing a normally-distributed value with mean equal to the vertical midpoint of the block and standard deviation 32.

This backbone function also adds springs and holes on the ground as necessary to allow vertical movement between blocks, as well as decorative elements and sets of rings along the floor.

Bonus areas (see Figure 5.4) are implemented by generating a normal chunk of ground, then placing a set of bonuses atop it; these bonuses are randomly chosen from four possible options: ring, shield or life monitors, or ring clusters, whose shapes are also constructed at random.

Figure 5.4: Example implementation of a bonus area

Challenge areas (see Figure 5.5) are implemented both by replacing the ground with a set of moving platforms (allowing the possibility of the player falling through, and possibly losing a life, if there is no platform below), with probability p_{moving}, and adding enemies with probability $p_{enemies}$. If there is room in the block and there is no other block directly above it (whether connected to it or not), a loop can be created instead of either moving platforms or enemies, with probability p_{loop}. (These probabilities are part of the generator's controllable parameters; see Section 5.7).

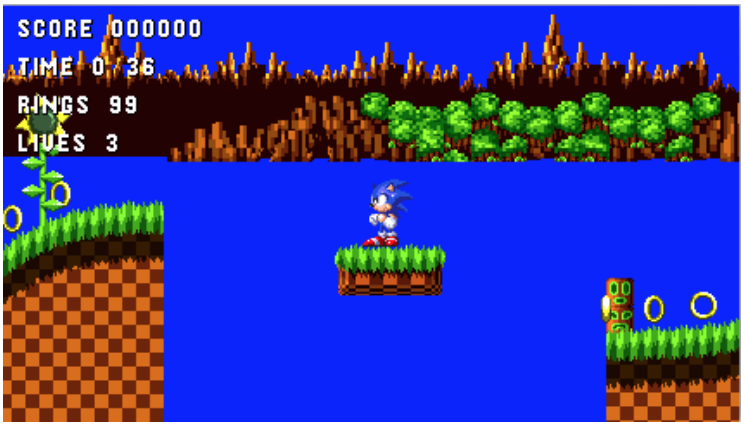

Figure 5.5: Example implementation of a challenge area

5.6 Boss generation

An important element of many platform game levels, which previous level generation efforts have not touched upon, is the boss. Given that these often iconic battles each feature unique elements and mechanics, the question of how to generate a varied set of bosses with a controllable difficulty level arises.

In the classic Sonic games, many bosses feature Dr. Eggman (the games' main antagonist) piloting a hovering vehicle named "Eggpod". Usually, the Eggpod is equipped with a weapon of

some kind to attack the player (see Figures 5.6, 5.7 and 5.8 for examples); or the area where the boss fight takes place contains elements that help or hinder the player in attacking the boss; or, in some cases, both. With a few exceptions, attacking the Eggpod is the only way of dealing damage to the boss.

Furthermore, the boss frequently moves within the arena following a regular pattern; this pattern, combined with any weapons and environmental features of the area, establishes the boss's difficulty level.

Based on these observations, we can see that it is possible to logically represent a boss as a combination of a base machine, which can be attacked to deal damage to it, and several elements, which each increase the boss's difficulty in some manner. These elements are divided in "weapons" - which are separate harmful objects that stay aligned with the boss's position - and "movement patterns", which are function-like objects that, when added to the boss, each affect its speed in a different way.

In the generator, the following weapons are available:

- Wrecking ball (see Figure 5.6) - a checkered ball that follows a circular path around the Eggpod, remaining attached to it by a chain of (non-collectible) rings. Its rotation speed and chain radius are configurable.

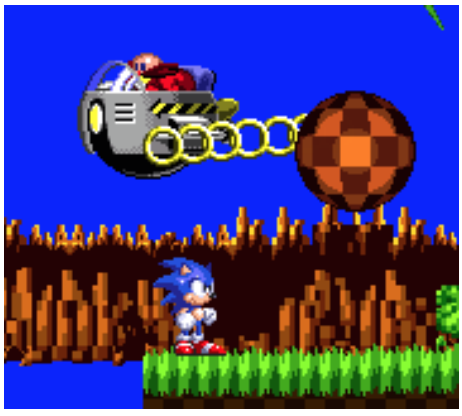

Figure 5.6: Wrecking ball in-game

- Drill (see Figure 5.7) - a static triangular drill attached to the front of the Eggpod.

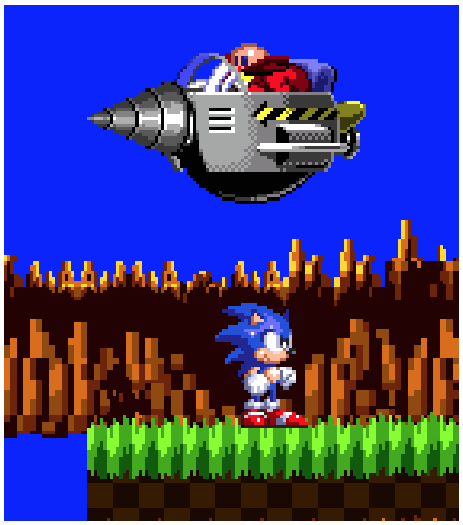

Figure 5.7: Drill in-game

- Hammer (see Figure 5.8) - a hammer also attached to the front of the Eggpod that periodically slams down on the area beneath it.

Figure 5.8: Hammer in-game

The following movement patterns are provided:

- Hovering left to right across the boss area
- Bouncing up and down atop a given Y coordinate (the upper limit of the boss's movement is defined by gravity)

Given this, and a set difficulty level (provided as a non-negative integer), the boss generation process consists of creating a boss object, then attaching a number of randomly-chosen elements

from the above set equal to the difficulty level, up to the number available. If the difficulty level exceeds the number of available elements, then, for each additional difficulty point, the generator increases the speed of a randomly-chosen element that was already added by a fixed factor.

Note that unlike other objects in the game, boss objects do not support being serialised to, nor loaded from level files, and must thus be created programatically, as the existing support for level serialisation does not support objects containing references to other objects, which the boss object uses.

Due to this mechanism, the difficulty level 0 corresponds to a trivial boss (which neither attacks nor moves) and level 5 is, as is readily apparent, nearly impossible to beat. For this reason, the generator has been restricted to generating bosses only up to difficulty 4. However, these numbers only take into account the boss's own mechanics; the geometry of the location where the boss is placed (at the last block of a level, the one corresponding to its end node) and the boss's positioning within it can affect the difficulty of the battle. For instance, the battle taking place on a slope or atop a narrow platform can substantially increase the difficulty.

5.7 Summary of controllable parameters

NOTE: when making a choice between several possibilities according to a set of weights, the probability of choosing each option is equal to its weight divided by the sum of the weights of all options.

Structure generation

- Probability of splitting paths (p_{split})
- Probability of prematurely terminating paths ($p_{deadend}$)
- Probability of merging one path onto another (p_{merge})
- Maximum number of simultaneous paths (max_{paths})
- Maximum branching factor (b)
- Weights for the each type of node to be used (w_{bonus}, $w_{challenge}$)
- Target number of nodes (l)

Layout generation

- Weights for the up, down, left and right directions (w_{up}, w_{down}, w_{left}, w_{right})

Level implementation

(All of these, except the first, are specific to the implementation functions used for this work)

- Block width and height
- Probability of adding rings along the ground in interstitial nodes (p_{rings})
- Probability of including a boss at the end of the level (p_{boss})
- Boss difficulty level (d_{boss})
- Weights for choosing shield, ring, 1-up monitors, or a ring cluster, as a reward in a bonus area ($w_{shieldmon}$, $w_{ringmon}$, $w_{lifemon}$ for the monitors)
- Maximum number of bonuses in a bonus area
- Probabilities of having moving platforms, enemies, and loops in a challenge area ($p_{enemies}$, p_{moving}, p_{loop})

Chapter 6

Adaptation mechanism

In order to achieve the goal, mentioned in Chapter 1, of personalising the game experience based on the player's playstyle, a method of estimating and parameterising that playstyle based on some of the player's actions during the game has been developed, as well as a set of formulas for adjusting the level generator's parameters according to the playstyle parameters.

6.1 Gameplay measurement

To compute the player's playstyle parameters, it is first necessary to collect data from a play session to feed into the formulas that yield these parameters. This data consists mostly of a set of counters that measure the number of times a particular event occurs during gameplay. A separate logging module has been included in the game engine itself for this purpose; the engine has been tweaked to make calls to that module whenever one of the events supported by the module occurs. These events (and, where used, their counters, which appear in the formulas in Section 6.2) are:

- Jumps - N_{jumps}
- Rolls (when the player begins rolling on the ground) - N_{rolls}
- Monitor hits - counted separately per monitor type
- Killed enemies - counted separately per enemy type
- Damage taken - counted separately by source type - total is counted as N_{hits}
- Landing on moving platforms - $N_{landings}$
- Collecting bouncing rings - $R_{collected}$
- Spawning bouncing rings (total number of rings spawned is counted) - $R_{spawned}$
- Deaths due to taking damage or falling off the level - counted separately - N_{deaths}

- Spindashes - for each spindash[1], both the time spent charging and the number of taps used are recorded - t_i and $N_{charges_i}$ respectively, for the spindash i

In addition, the time taken to complete the level and the player's speed in each frame ($gspeed_i$) where it is on the ground are also recorded.

6.2 Playstyle analysis

The player's playstyle is represented as a set of parameters, encapsulated in a "player profile" object:

- Rush tendency - the player's tendency to go through levels quickly.
- Exploration tendency - the player's tendency to perform actions that do not directly advance them towards the end of the level.
- Ring valuation - measures how much the player cares about maintaining their ring count.
- Skill - measures the player's ability to avoid taking damage and dying.
- Monitor preference - measures how many monitors of each type the player tends to collect.

The first four of these measures range from 0 to 1; monitor preference is counted per-monitor-type and has no limit.

For each of these parameters, there is a formula to calculate it based on the aforementioned gameplay measurements. Updating the player profile consists of recomputing each of the parameters using their respective formulas, then blending the new values with the existing ones according to a learning rate ranging from 0 to 1 (the higher this rate is, the faster the profile will adjust to the gameplay measurements; but too high a value will cause it to quickly "forget" past tendencies). The updating formula is as follows, where C is the final value, B is the new value, A the old one, and r is the learning factor:

$$C = (1-r)A + rB \tag{6.1}$$

The formulas for each of the first four parameters are, respectively, as follows, where T is the time taken to complete the level, in minutes, $N_{gspeeds}$ the number of ground speeds recorded, $N_spindashes$ the number of spindashes recorded:

$$R = \text{clamp}\left(\frac{\frac{\sum_{i=1}^{N_{gspeeds}} |gspeed_i|}{N_{gspeeds}} + \left(N_{spindashes} + N_{rolls} - N_{landings}\right)/T + \frac{\sum_{i=1}^{N_spindashes} N_{charges_i}}{N_{spindashes}}}{3}, 10\right) \tag{6.2}$$

[1] A move whereby Sonic spins in a stationary spot on the ground, charging up speed with each tap of the jump button and then dashing forward with that speed.

Equation 6.2 correlates rush tendency positively with the speed at which the player runs, as well the number of spindashes and rolls used during the level, since these moves are often used to gain speed. Landing on moving platforms counts negatively, as a "rushy" player will tend to try and jump over them to save time.

$$E = \text{clamp}\left(\frac{\frac{\sum_{i=1}^{N_{gspeeds}}[G_{speed}<0]}{N_{gspeeds}} + \frac{N_{jumps}}{20T} + \frac{N_{landings}}{4T}}{3}, 1\right) \tag{6.3}$$

Equation 6.3 follows the notion that exploring players will, on average, tend to not move forward all the time, and go back to earlier sections, as well as jump more in order to access different paths. They also are more likely to use moving platforms, since they allow them to stay safely on upper paths or reach them.

$$V = \begin{cases} V_{old} & \text{if } R_{spawned} = 0 \\ \frac{R_{collected}}{R_{spawned}} & \text{if } R_{spawned} > 0 \end{cases} \tag{6.4}$$

Equation 6.4 aims to measure how much players care about their ring count by observing how many of the rings they lose they recapture - actually measuring the ones they *try* to grab, successfully or not, would be too error-prone to be practical.

$$S = 1 - \text{clamp}\left(\frac{N_{deaths} + N_{hits}}{2T}, 5\right) \tag{6.5}$$

Equation 6.5 measures skill as being negatively correlated with the average combined number of hits taken and deaths per minute of game time.

Monitor preference values for each monitor type are equal to the number of monitors of that type collected during the level.

clamp(a,b) clamps a to the range $[0,b]$ then divides the result by b, yielding a value in the range $[0,1]$.

Note that all of these parameters are in some way orthogonal to each other; for instance, it is possible for a player to be “rushy”, by tending to run very quickly and spindash often, but also exploring (if they spend a significant amount of time running left - note that this assumes that levels primarily flow left-to-right, which depends on the parameters used in the layout generation; the values used in our experiments were set to have this result) - the first terms in the rush and exploration tendencies formulas are the mean absolute speed on the ground and the proportion of time spent running left on the ground, relative to the total time spent on the ground.

Once the player profile is updated, the counters and other information stored by the logging module are reset before a new level based on this profile begins; hence, each update of the profile is based on the data from the play-through of a single level.

6.3 Parameter adaptation

The task of translating the player's estimated playstyle into a level style deemed appropriate for that playstyle is accomplished by defining some parameters of the generator as functions of the player profile's parameters described in the previous section. These formulas have been chosen following an intuitive notion of what kind of level different types of players would prefer; for instance, a player with a high "rush tendency" will likely prefer a level with less vertical movement and a stronger left-to-right orientation; a player with a high "exploration tendency" will want a more complex structure, with additional paths through the level and more platform hopping.

The formulas chosen for the generator's parameters are as follows, where R, E, V and S are the four player profile parameters defined in the previous section:

$$max_{paths} = 4, b = 2, l = 16 \tag{6.6}$$

$$w_{bonus} = 2, w_{challenge} = 3 \tag{6.7}$$

$$p_{split} = 0.3E \tag{6.8}$$

$$p_{merge} = 0.15E \tag{6.9}$$

$$p_{deadend} = 0.5E \tag{6.10}$$

$$w_{left} = 1 + 3E - R \tag{6.11}$$

$$w_{right} = 20 + 80R \tag{6.12}$$

$$w_{up} = 3 + 2E - 2R \tag{6.13}$$

$$w_{down} = 1 + 4R \tag{6.14}$$

$$w_{ringmon} = MW_{ringmon} + 5V \tag{6.15}$$

$$w_{lifemon} = MW_{lifemon} + 5(1 - S) \tag{6.16}$$

$$w_{shieldmon} = MW_{shieldmon} \tag{6.17}$$

$$p_{rings} = V \tag{6.18}$$

$$p_{boss} = 1 \tag{6.19}$$

$$d_{boss} = round(1 + 3S) \tag{6.20}$$

$$p_{enemies} = S \tag{6.21}$$

$$p_{moving} = \frac{E + S}{2} \tag{6.22}$$

$$p_{loop} = 1 - 0.85S \tag{6.23}$$

These formulas, put together, make up a theoretical model of what kind of levels are best for particular kinds of players. In Chapter 7, we examine how well this model matches up with real-world player preferences.

Chapter 7

Testing

To determine the effectiveness of the generator in creating fun Sonic levels, as well as in adapting its output to the player's preferences, a round of play-testing was conducted in the week from 25 to 29 May, using 7 volunteers as test subjects.

7.1 Testing method

As the goal of the testing was to assess the performance of the generator's adaptation mechanism, the main part of the test consisted of playing a series of N levels in a row, then asking the test subject which of these they liked best, identified by its 1-based index in the sequence I. If the adaptation algorithm is being effective, this choice should tend to fall upon the last few levels of the series; the stronger the correlation, the more likely it is that the generator is doing well. This does have the shortcoming that, due to the randomness involved in the generation, it is possible for a particular feature or group of features in a level to stand out to a player and thus obscure the results of the changes in the generation parameters.

This method was preferred over asking players to rate each level directly (in a fixed scale, e.g. 1-5) because without a baseline to compare levels to, there is no way to give an accurate score; furthermore, players usually either like the level or not, leading to scores that are biased towards the extremes. A relative evaluation avoids these issues.

In addition, each test subject was asked whether they preferred Mario-style or Sonic-style platformers (all testers reported being at least familiar with both), to gain some additional insight on their preferences and what effect they may have on the generator's behaviour.

7.2 Data gathered and other observations

Preference (Mario/Sonic)	I	N	$K = \frac{I-1}{N-1}$
Sonic	2.5	5	0.375
Sonic	2	4	0.333
Sonic	3	3	1
Mario	3	3	1
Mario	5	5	1
Mario	3	5	0.5
Mario	5	5	1

Table 7.1: The responses of the 7 test subjects to the test. I is the index of the player's preferred level, and N is the number of levels played in a row.

- Due to some glitches in the engine, as well as flaws in the generator, such as the generation of impassable terrain or bosses, it was frequently impossible for the tester to complete a run of 5 levels on the first try. The data in Table 7.1 represent the final, successful run; although in three occasions the test was ended prematurely as the difficulty of the game (addressed further in the following section) exhausted the test subjects. For this reason, the metric used to judge the effectiveness of the generator was not the absolute index I of the preferred level, but the ratio $\frac{I-1}{N-1}$, which takes into account the number of levels played.

- $K = \frac{I-1}{N-1}$ is used instead of $\frac{I}{N}$ because the latter does not accurately represent the relative position of the level I within the sequence of N levels; for instance, in a sequence of 5 levels, the midway point is at level 3, not 2.5. It also represents the effectiveness of the algorithm under study more accurately: a player who preferred the first level - indicating the algorithm had failed to accomplish its purpose - would yield a value of 0 using this formula.

- One player reported preferring levels 2 and 3 equally; the index for that player was determined to be the middle value between these, 2.5.

7.3 Analysis of the results

From the data, it appears that the generator's player profiling mechanism is at least somewhat effective, as the players' preferred levels were, with two exceptions out of 7, at or beyond the halfway point of the series they played. This effect is strongest for players who reported preferring Mario-style platformers (the mean K value for them is 0.875, versus 0.569 for the Sonic-preferring testers).

However, the test also revealed several flaws that need to be corrected in any future development of this work - most visibly, the difficulty of the generated levels, which the test subjects universally deemed excessive, mainly due to two factors:

- The overabundance of bottomless pits[1] where, below moving platforms as well as at the boundaries of the level, causing frequent deaths which, in turn, implied a return to the start of the level.
- The difficulty of bosses, which were often in such a position that they could not easily be attacked without taking damage or risking falling into a bottomless pit - in some cases a result of the boss being placed on too narrow a platform; in addition, the bouncing and wrecking-ball mechanics added substantially more difficulty than others, and their combination made it overly difficult to find an opening to safely attack the boss.

In combination, these two factors make for levels that are very unforgiving of mistakes; this may explain why the generator seems to adapt to Mario-preferring players better, as classic Mario games tend to be less forgiving than Sonic games due to the lack of the ring mechanic (which makes one unlikely to die from enemy hits) and higher amount of bottomless pits.

Additionally, several bugs were uncovered or highlighted during the course of the testing, including:

- A crash in the calculation of the time bonus when the player took 10 minutes or longer to complete a level.
- A crash in the function for calculating a player's "rush tendency", which occurred when the player had not spindashed at all during a level.
- An issue that caused the keyboard input handler to occasionally crash (this was eventually worked around; its cause remains unknown).
- A glitch in the position of path swappers[2] near loops, which caused players to fall through the terrain if they jumped at the end of one.

[1] Bottomless pits: gaps at the bottom of a level, which kill the player character if it falls into them.

[2] Objects that cause the player character to switch between two collision layers; this makes it possible to implement loop-de-loops by dividing them into two halves, one of which is placed in a lower layer than the other.

Chapter 8

Conclusions

8.1 Goals accomplished

Of the three main goals stated at the beginning of this document, the one that has been best accomplished is the second, the use of richer structural and gameplay elements in level generation. As intended, the generator reliably produces levels with a well-defined and visible structure, with multiple paths that branch off and rejoin each other; furthermore, it is able to create observably different types of structure, depending on the parameters it is given. As far as gameplay elements go, while there is only a limited selection of enemies and items, the generator makes up for this with its ability to generate unique bosses for each level; combined with the unpredictable environments where the bosses are located, this makes for very varied boss fights (even if they're sometimes extremely difficult).

Regarding the assessment of the player's preferences (first goal), the tests performed, as described in Chapter 7, appear to indicate that the means of assessment and parameter adaptation developed is mildly effective at least, though a larger sample is needed to confirm this. Hence, we cannot conclusively say whether the hypothesis stated in Section 1.3 is true or not, although intuitively it should be.

As for the third and last goal, the assembling of a set of levels to create a nearly-complete game, this ended up being a lower priority than the above two goals; hence, this aspect of the generator was not significantly developed, although the testing process involved a similar concept and did make difficulty-affecting adjustments to some parameters.

8.2 Future work

For the most part, the results of our work are satisfactory, but there is much room for improvement. The implementation developed during this project, while it is an effective proof of concept of the level generation method we conceived, is rudimentary and is missing numerous gameplay elements: more enemies, items, and structural features are needed. The lack of many different structural features should bear special attention, as it is also one of the causes of the generated

levels' excessive difficulty and takes away from the "Sonic" feel that the levels were intended to have.

The placement of bonuses items needs to be significantly improved; the current implementation just scatters bonus items in the player's way, not offering any real reward for the player's exploration or skill. One of the simpler ways to improve on this is to introduce the possibility of bonuses, or even entrances to entire paths, being "hidden" in some fashion.

Furthermore, this version of the generator only creates levels of one style (that of the original Green Hill Zone from *Sonic the Hedgehog*); to fully accomplish the construction of a full game, it is necessary to include ways of generating more varied environments – preferably in a way that does not require manually writing a full set of implementation functions from scratch, as that negates a significant part of the benefits of using PCG in the first place.

Finally, the playstyle evaluation mechanism provided is, like the level generator itself, rather rudimentary; it likely can be substantially improved by using more diverse and sophisticated gameplay metrics, comparative testing of different parameter adjustment formulas, and broader testing. Having a better-developed generator, by itself, would also help, as there would be a greater number of possible events to measure and analyse and a wider variety of elements that could be adjusted to reflect the results of that analysis.

References

[1] Diaz Furlong Hector Adrian and Solis Gonzalez Cosio Ana Luisa. "An approach to level design using procedural content generation and difficulty curves". In: *IEEE Conference on Computatonal Intelligence and Games, CIG*. 2013. ISBN: 9781467353113. DOI: 10.1109/CIG.2013.6633640.

[2] Glenn Fiedler. *Fix your Timestep*. URL: http://gafferongames.com/game-physics/fix-your-timestep/ (visited on 2015-06-14).

[3] Robin Hunicke, Marc LeBlanc, and Robert Zubek. "MDA: A Formal Approach to Game Design and Game Research". In: *Workshop on Challenges in Game AI* (2004), pp. 1–4. ISSN: 03772217. DOI: 10.1.1.79.4561.

[4] Manuel Kerssemakers et al. "A procedural procedural level generator generator". In: *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*. 2012, pp. 335–341. ISBN: 9781467311922. DOI: 10.1109/CIG.2012.6374174.

[5] Peter Mawhorter and Michael Mateas. "Procedural level generation using occupancy-regulated extension". In: *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. Ieee, 2010-08, pp. 351–358. ISBN: 978-1-4244-6295-7. DOI: 10.1109/ITW.2010.5593333.

[6] Fausto Mourato, Fernando Birra, and Manuel Próspero dos Santos. "Using Graph-Based Analysis to Enhance Automatic Level Generation for Platform Videogames". In: *International Journal of Creative Interfaces and Computer Graphics* 4.June (2013), pp. 49–70. ISSN: 1947-3117. DOI: 10.4018/ijcicg.2013010104.

[7] Nelson Oliveira. "Generating Entertaining Platform Game Levels". MSc. Thesis. Faculdade de Engenharia da Universidade do Porto, 2014.

[8] Noor Shaker, GN Yannakakis, and Julian Togelius. "Towards Automatic Personalized Content Generation for Platform Games." In: *Proceedings of Artificial Intelligence and Interactive Digital Entertainment* (2010).

[9] Noor Shaker et al. "The 2010 Mario AI Championship: Level generation track". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.4 (2011), pp. 332–347.

[10] Gillian Smith, M Cha, and Jim Whitehead. “A framework for analysis of 2D platformer levels”. In: *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*. 2008, pp. 75–80.

[11] Gillian Smith et al. “Rhythm-based level generation for 2D platformers”. In: *Proceedings of the 4th International Conference on Foundations of Digital Games - FDG ’09*. New York, New York, USA: ACM Press, 2009, p. 175. ISBN: 9781605584379. DOI: 10.1145/1536513.1536548.

[12] *Sonic Physics Guide*. URL: http://info.sonicretro.org/Sonic%5C_Physics%5C_Guide (visited on 2015-06-14).

[13] Nathan Sorenson and Philippe Pasquier. “The evolution of fun: Automatic level design through challenge modeling”. In: *Proceedings of the First International Conference on Computational Creativity (ICCCX)*. 2010, pp. 258–267. ISBN: 9789899600126.

[14] Jim Whitehead. “Toward proccedural decorative ornamentation in games”. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames ’10*. New York, New York, USA: ACM Press, 2010, pp. 1–4. ISBN: 9781450300230. DOI: 10.1145/1814256.1814265.

Appendix A

Examples of generated levels

In this appendix, we present the full maps for a sample of generated levels. Each level is accompanied by the structure and layout parameters used to create it (see Section 5.7). The red lines in the maps indicate the movement patterns of the moving platforms.

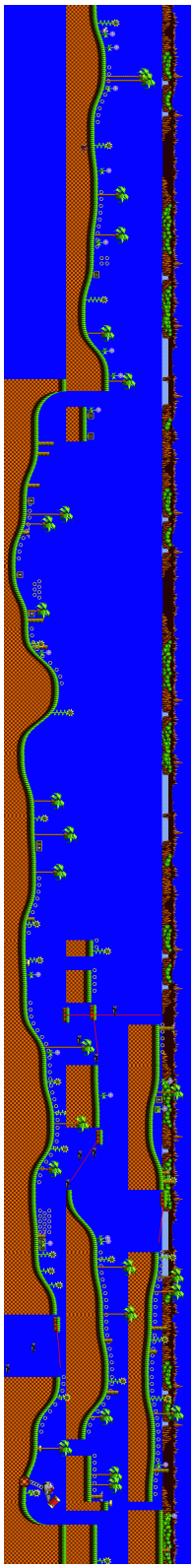

Figure A.1: Example of a left-to-right level: $p_{split} = 0.25$, $p_{deadend} = 0.1$, $p_{merge} = 0.15$, $max_{paths} = 3$, $b = 2$, $w_{bonus} = 2$, $w_{challenge} = 4$, $l = 15$, $w_{up} = 5$, $w_{down} = 3$, $w_{left} = 3$, $w_{right} = 100$

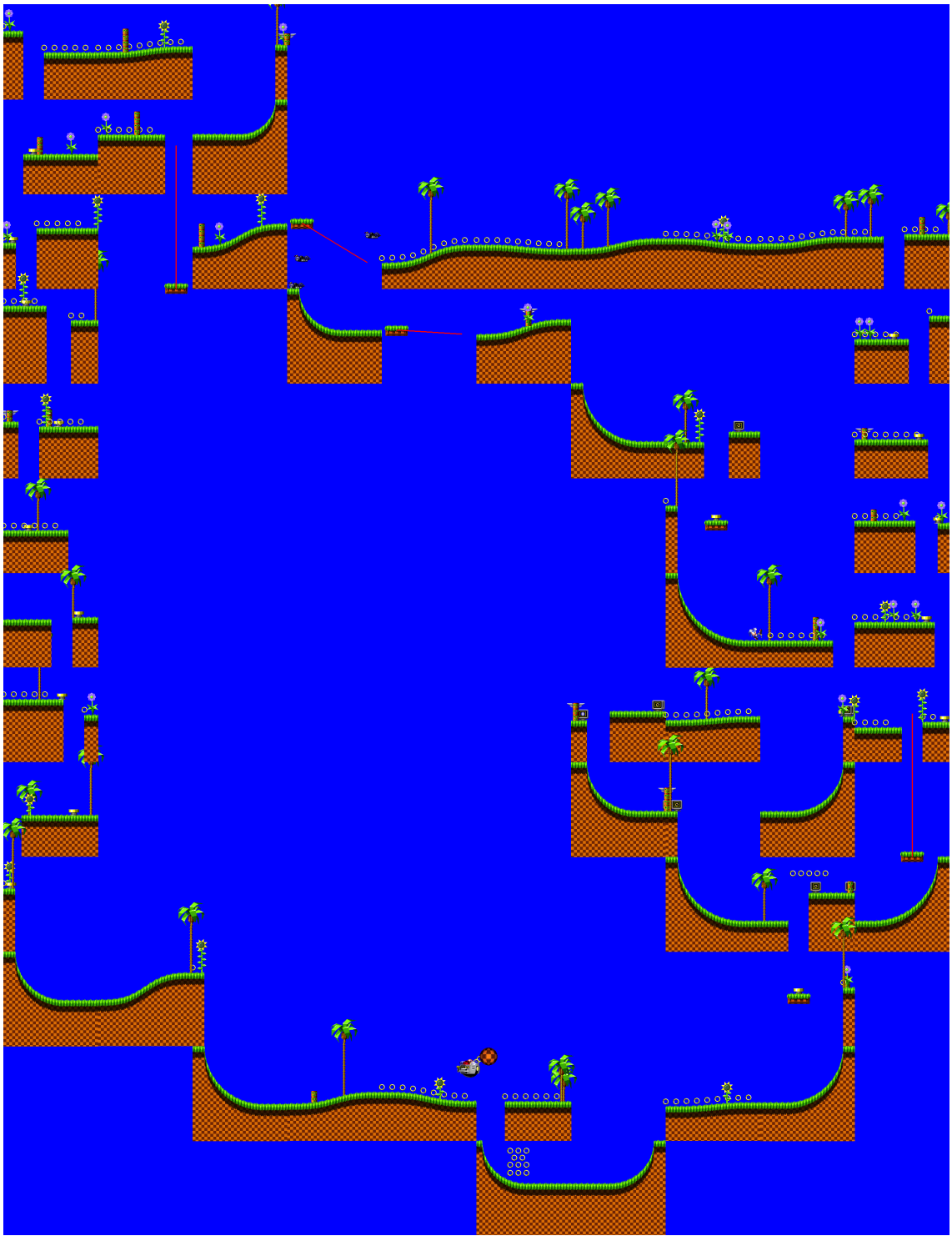

Figure A.2: Example of a directionless level: $p_{split} = 0.25$, $p_{deadend} = 0$, $p_{merge} = 0.1$, $max_{paths} = 3$, $b = 2$, $w_{bonus} = 2$, $w_{challenge} = 4$, $l = 15$, $w_{up} = 1$, $w_{down} = 1$, $w_{left} = 1$, $w_{right} = 1$