Allegheny College
Department of Computer Science

Senior Thesis

Collapsing Through Anxiety: a Novel Approach to 2D Side Scrolling Platformer Level Generation

by

Pedro Neves Souza Carmo

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Douglas Luman**
Co-Supervisor: **Janyl Jumadinova**

22 July 2022

Abstract

By Combining the video game level challenge analysis method known as anxiety curves, rhythm group based metadata, and the wave function collapse algorithm, we propose and develop new method of creating platformer levels that have both large output variety and user control over the level difficulty. This method works by utilizing wave function collapse's ability to generate levels based on the metadata associated with the level's tiles to build sequences of rhythm groups that follow a chosen difficulty value. Thus, enabling the level designer to dictate how difficult each section of a level should be. We hope this method for 2D side-scrolling platformer level generation serves as a base for developers and hobbyists interested developing games in the genre.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Procedural Generation in Video Games

Using generation algorithms is nothing new in gaming, all the way back in 1980 the game Rogue used randomly generated dungeon floors as a way to increase replayability, which, back then, was already a difficult concept to tackle [13] [9]. Although Rogue was a success, pure randomness in generation is often unpredictable and the resulting level might be unplayable or worse: boring. To circumvent these problems and have more control over their output, developers have opted for procedural generation. For many years now, procedural generation algorithms have been an important tool for game developers, these algorithms are often employed in order to present the player with diversity in challenges and exploration. Many games with tremendous success have employed this type of generation: *Minecraft* [26], *Terraria* [35], *Hades* [18], *Dwarf Fortress* [16]… Procedural generation utilizes many techniques for piecing together levels, but their goal generally is to ensure a level is playable, challenging, interesting to explore, and/or has minimal tedious sections.

1.1.1 Level Difficulty

Although the main goal of level generation is to create a fun level the concept of fun is far too broad and difficult to synthesize, other research papers on level generation judge their levels using Csikszentmikalyi's concept of "flow" [39] or by constructing an **anxiety curve** representative of the level and using some metric to decide if the level's curve is fun [40].

1.2 Wave Function Collapse Algorithm

This algorithm belongs to a class of algorithms called constraint based solving algorithms or constraint solvers. These algorithms operate by iteratively assigning values to variables while following a set of rules (the constraints) until the algorithm finds a valid value for all the variables it needs to compute or reaches a fail state in which it has to backtrack or stop running and throw an exception error [15]. The end goal that this algorithm iteratively solves for is a generated image made of combined tiles that must adhere to its neighboring tiles' adjacency rules.

The wave function collapse algorithm is a procedural generation algorithm that has already been applied to video game level generation. It is capable of generating very diverse output from a set of tile images that will be pieced together to form the finished level image [19]. Video games are already using wave function collapse to generate their playable environments, notable examples include *Bad North* [32] and *Townscaper* [43] by Oskar Stålberg, and *Caves of Qud* [17] by Freehold Games which uses the wave function collapse algorithm many times to generate the world map as a whole and then small localized environments [4].

1.2.1 Terminology of video-game level generation:

- **Level:** refers to the environment that the player is put in during gameplay. For example, in *Super Mario Bros.* [31] it would be the ground, platforms, boxes, pipes, background and foreground elements while in *Minecraft* it would be the 3D voxel based world and its elements (trees, water, caves, etc).

- **Tile Sets:** refer to the groups of incremental images that can be put together to make a 2D level. Usually refers to small pixelated sets of images that are used in classic games and indie titles [30].

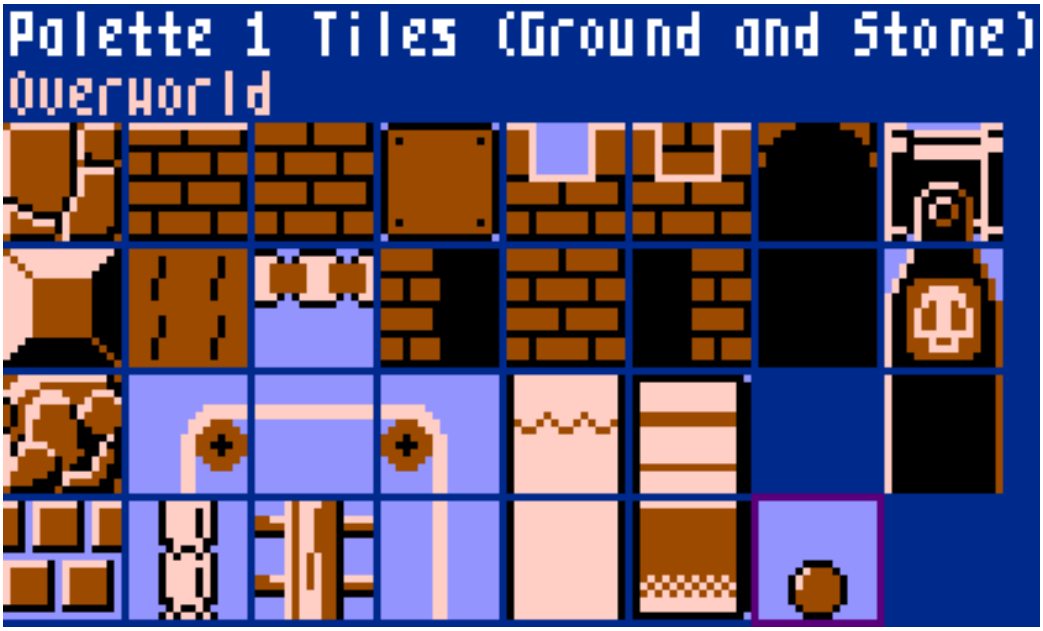

Figure 1: Example Overworld Tile Set from 1985 Mario Game

- **Tile:** can have two meanings.
  - A tile can be a single unit from a tile set, a part of the texture the smallest element in level design that can be combined with other tiles to create the level.

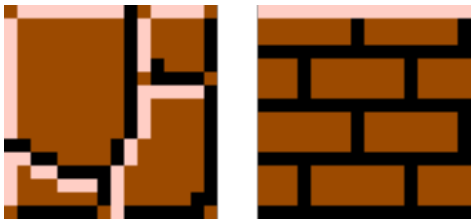

Figure 2: Tiles Present in Level "World 1-1" of Super Mario Bros.

  - **Tile**, in the context of generation, can also refer to a larger group of the smaller individual tiles. This group of tiles will also be called a tile when they behave as a unit to build levels.

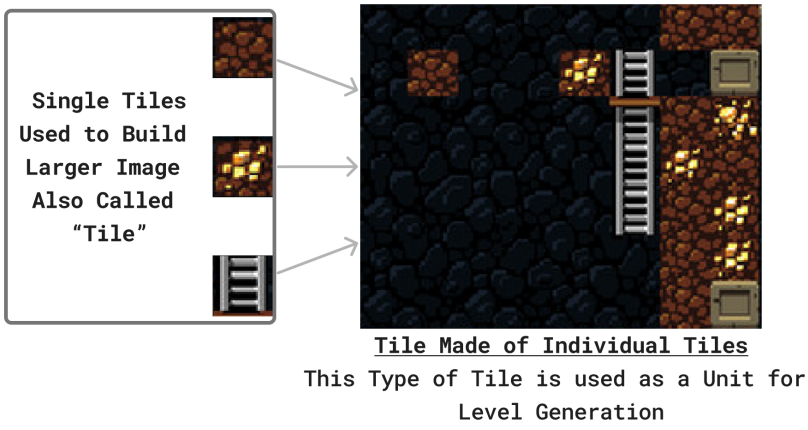

Figure 3: Tiles Used in the Game "Spelunky" to Generate Large Cavernous Levels

- **Cells or Grid Spaces:** are the empty (and usually square) spaces where the tiles will be placed. In wave function collapse the empty cells will have a value that represents how many tiles can fit into a given cell [23].

1.3 Platformer Games

The platformer game genre contains some of the most iconic and well known games of all time, like the aforementioned, "true-platformer", *Super Mario* series [31]. However, the first entry of the genre is attributed to 1980s *Space Panic* [45] which was a type of platformer now referred to as single screen platformers. From today's viewpoint *Space Panic* is an odd platformer since its player character cannot jump, which seems like vital feature present throughout the genre. In the game the player character moves through the level by climbing ladders and walking on platforms while fighting enemies. The most

famous of all single screen platformers, released the year after *Space Panic* is Nintendo's arcade game *Donkey Kong* [29], a game in which the player character (Mario's precursor "Jumpman") starts on the bottom of a level with the goal of reaching the top by running on platforms, jumping over obstacles, and climbing ladders. These games became known as "climbing games" or "ladder genre" games.

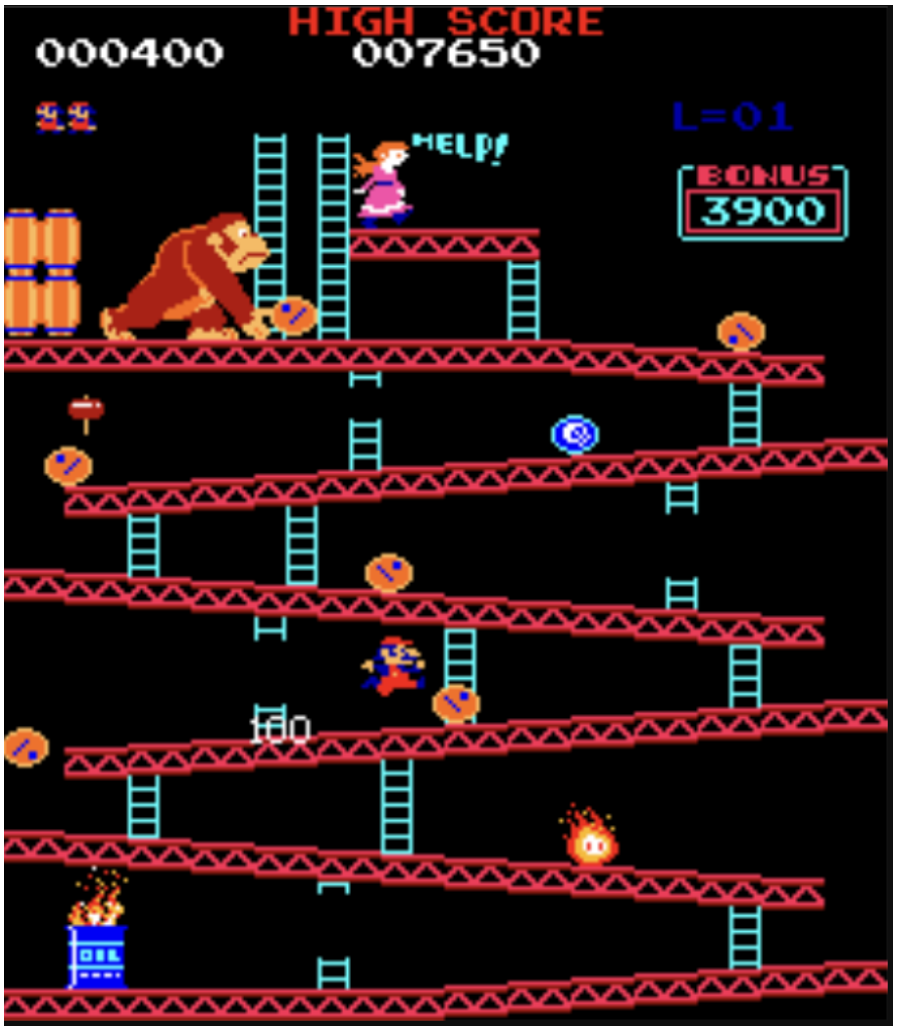

Figure 4: Image of Donkey Kong's first level, "25 m"

The naming convention "platformer game" or "platformer genre game" would only become popular around 1989 [7] from its usage in British re-

view and guide magazines about video games like “Complete Games Guide” [22] and “The Games Machine” [36]. Nowadays, there are many subgenres of platforming games such as 3D platformers, puzzle-platformers, precision platformers, run-and-gun platformers, action-adventure platformers (sometimes called Metroidvanias) but games referred to as just platformers or side-scrolling platformers [5] are the ones that follow the vein of *Super Mario Bros.* or *Sonic the Hedgehog* [37]. The main characteristics of these games are the player’s ability to run and jump to get from the beginning of a level to the end by moving left and right in a 2D side-scrolling level while avoiding enemies, hazards, and pitfalls and also being able to interact with different elements like **?** blocks from the Mario games or speed boosting springs in the Sonic games.

Figure 5: Beginning Section of Super Mario Bros. First Level Showing Player’s Character (Mario), an Enemy, and Intractable **?** Blocks

1.3.1 Use of Procedural Generation in Platformers

Side-scrolling 2D platformer games rarely use algorithmically generated levels since much of the challenge and fun comes from the carefully planned out

layout of platforms, enemies, and hazards that make up the levels. Some platformers do use procedural level generation, a famous example is the very successful game *Spelunky* [27] which can be classified as a roguelike, exploration, adventure platformer. The levels in *Spelunky* are very different than Mario's or Sonic's levels, the game is set in a cave with the player character being an Indiana Jones-esque explorer. Thus, its levels are themed with an underground cave aesthetic with large focus on exploration in the vertical axis as well as the horizontal and the generation is done by creating a grid of empty cells that are algorithmically filled by different types of rooms built randomly from templates [23].

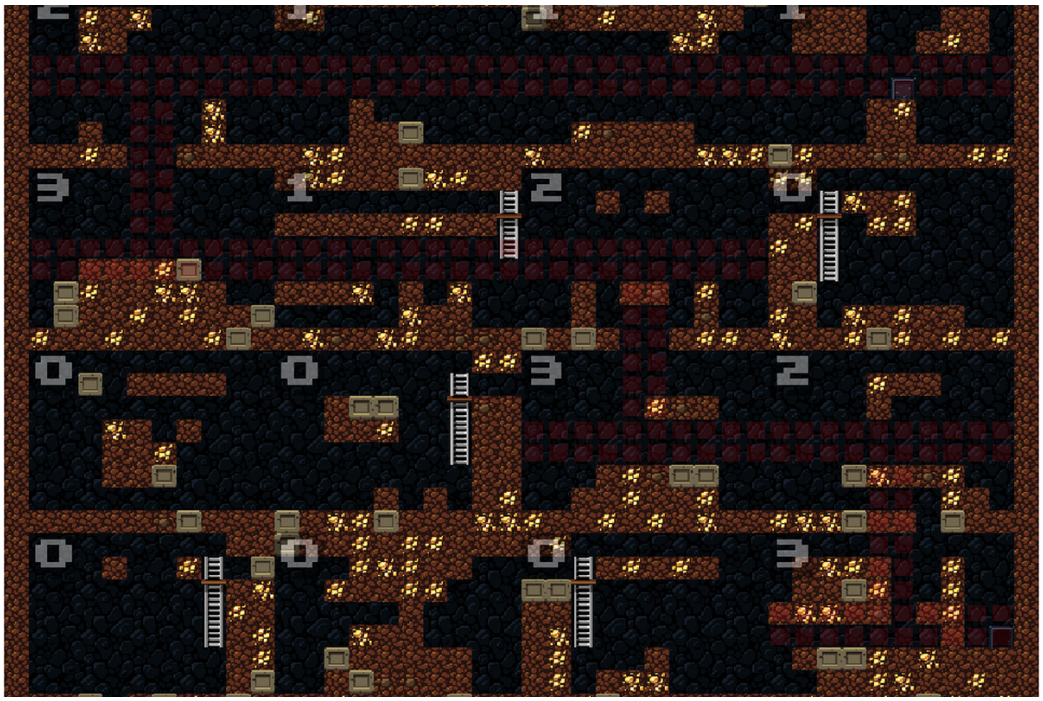

Figure 6: Level Generated with Darius Kazemi's Online Tutorial on Spelunky's Generation Process. The Numbers on the Upper-left Corner of Each Cell Represent the Type of Cell that was Randomly Chosen

The grid nature of *Spelunky's* levels leverage procedural generation very well but the non-grid nature of side-scrolling platformers makes it difficult for them to utilize procedural generation for level building.

1.3.2 Use of Wave Function Collapse in Platformers

The same goes for the use of wave function collapse in platformer games: 2D side-scrolling platformers have not seen it being implemented as a way to generate levels while it has been used in exploration/adventure grid based platformers. The indie game *Bug with a Gun* [33] made by Joseph Parker for ProcJam [20] utilizes wave function collapse to generate platformer levels. Still, *Bug with a Gun's* levels are more similar to Spelunky's rather than 2D scrolling platformers.

1.4 Argument for Wave Function Collapse in Platformers

This research aims to test if the wave function collapse algorithm is a fitting choice for generating 2D side scrolling platformer levels in the vein of *Super Mario Bros.* levels. A consideration can be made for the use of this algorithm since its outputs can be greatly controlled; the algorithm should be capable of level generation that can cater to the demanding expectation of players and the scrutiny of developers while not compromising level variety.

1.4.1 Wave Function Collapse with Anxiety Curves

Besides applying wave function collapse to 2D side scrolling platformers this work's approach includes the use of anxiety curves as a generation parameter. An anxiety curve is an analysis method that describes the difficulty in a video game level as a function of the level's length [41]. As an analysis method this curve is drawn after a level has been generated or even after it has been played by the developer/player as a way to pinpoint the challenging parts of the level. However, in order to use it as an input parameter the user is prompted to choose a difficulty curve that the generation algorithm will map to the level's challenging parts as it generates obstacles for the player. The obvious benefit of this approach is that the user, be it the level designer or the player, can have a lot of control over the difficulty of a level.

This reseach work's experiment will be tested in two main ways:

- Testing if wave function collapse works for 2D side scrolling platformer games by judging the of the algorithm's capacity to generate varied levels while adhering to adjacency rules.

- Testing if the algorithm's generated levels are playable from star to finish, meaning they do not contain *"unjumpable"* gaps for instance.

1.5 Ethical Implications

In order to ensure this generative approach will be put to good use we must strive for the best possible generative outcomes, this means the algorithm should seldom generate impossible levels. A level can be considered impossible if it is not completable at all, as in the case of having a gap too large to be jumped or a wall too high to be climbed by the playable character. The problem of impossible levels has been present in generative titles for a long time, back in the 80s when Rogue was in development its creators had to tweak the enemy placement by playing the game over and over because their initial idea of difficulty made the progress into the dungeon too difficult and luck reliant [13].

2 Related work

2.1 Wave Function Collapse

The idea behind the generation algorithm used in this paper comes mainly from the **W**ave **F**unction **C**ollapse algorithm [19] (referred to as **WFC** from now on) developed by Maxim Gumin in 2016 [24]. WFC is a constraint satisfaction problem solver algorithm, and it is capable of generating textures through combining small pieces of images while following a set of rules that dictate which pieces can be adjacent to one another [25].

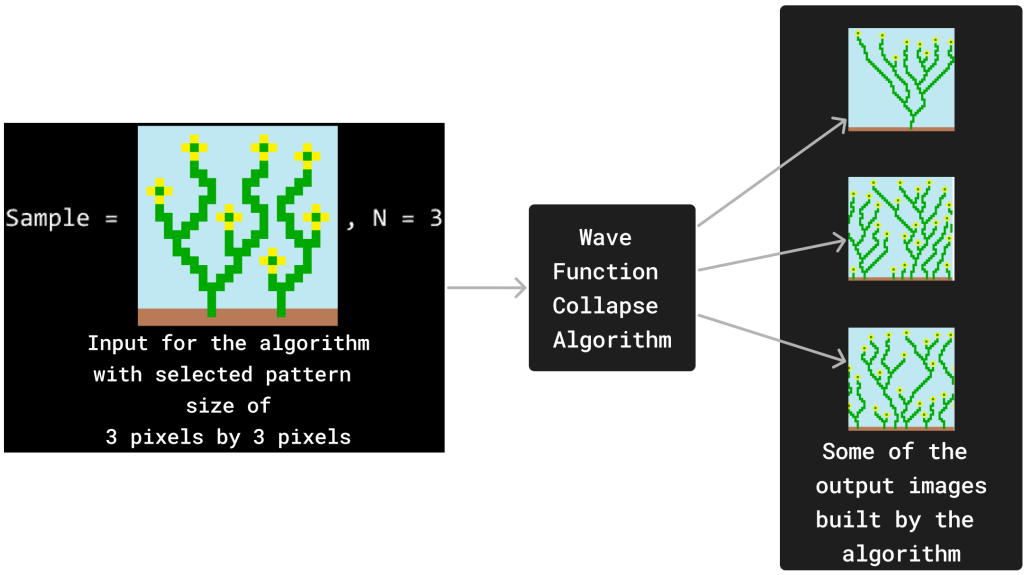

Figure 7: Three Example Outputs from Wave Function Collapse Generation Using Simple Pixelated Image as Input

Gumin's implementation works by taking in an input image and breaking it into N by N patterns that will be used to build the output.

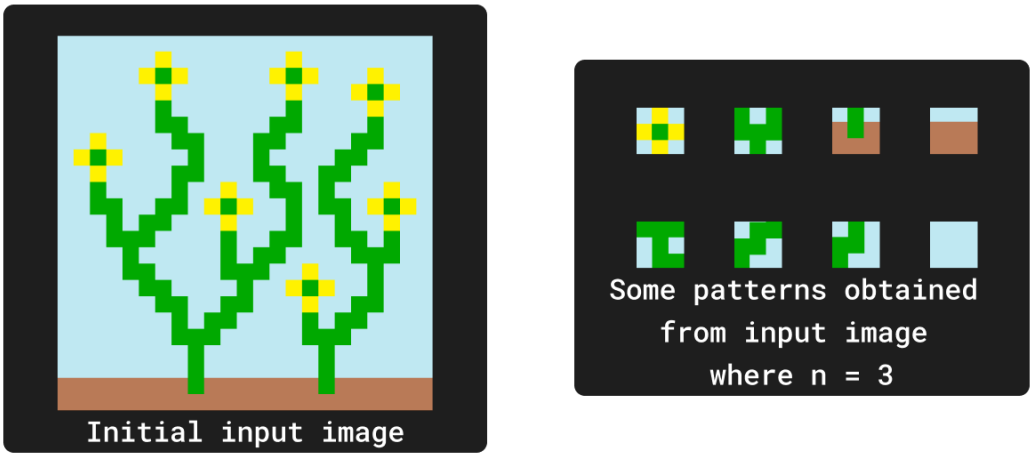

Figure 8: Example Patterns Generated by Algorithm's Preparation Step from the Input Image Before Generation of Output

Next it creates an empty grid for placing the identified patterns into. Then the algorithm will do a first pass by choosing a cell in the grid, placing a random piece, and from that piece it will determine which pieces cannot be adjacent to it and remove them from the possible choices for the next pass.

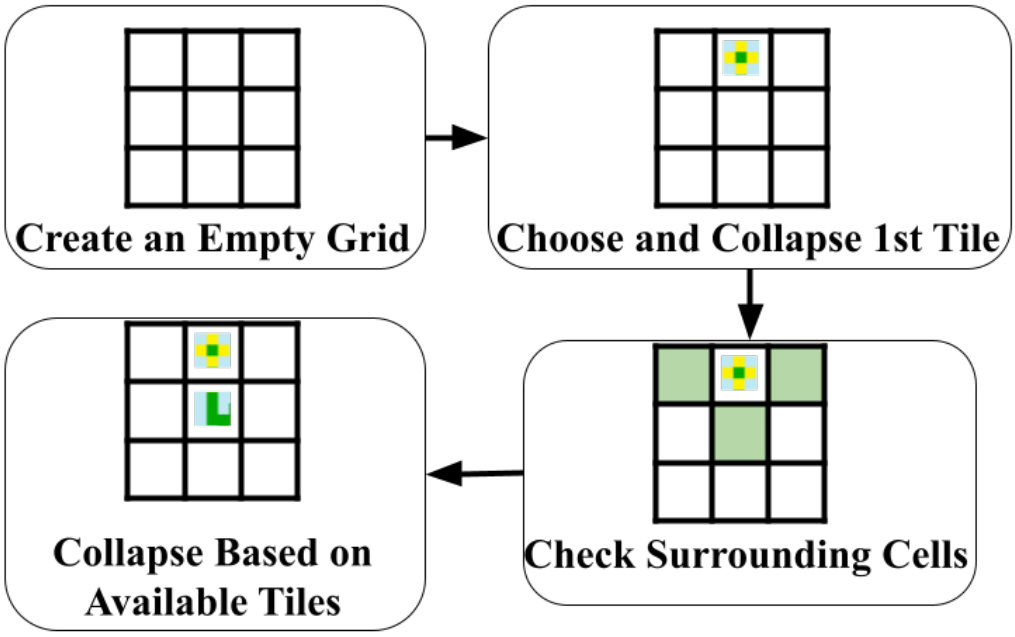

Figure 9: Simplified Example of WFC Algorithm Cell Collapse Process

Then the algorithm will choose the cell that has the smallest amount of choices to chose from and put a random piece there from the remaining possible piece choices.

It repeats this process until the empty grid is filled.

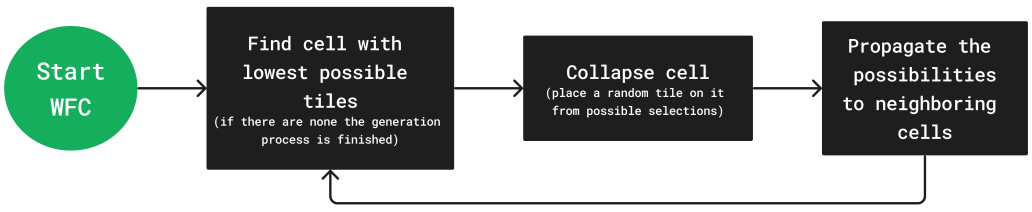

Figure 10: Diagram of WFC Algorithm Generation Process

2.2 Rhythm Groups

Video-game level generation has been approached from many angles. A particular approach for 2D side scrolling platformers is the use of rhythm groups, which consists of the breakdown of a level into paths, cells, and portals then generating new levels by combining different rhythm groups in succession [39]. This approach defines paths as uninterrupted sequences of cells that lead from the beginning to the end of the level, portals as spaces that switch between paths, and cells as one or more sequences of "rhythm groups." The smallest element used in this approach are the rhythm groups which are short, non-overlapping sets of components that make up a small section of the level, in essence these groups are a challenging area that ends in a brief rest area that will lead into another rhythm group and so on until the end of the level. The method of generation proposed in this work does use rhythm groups to a certain extent as the input for the algorithm uses sections from *Super Mario Bros.* levels that are broken down similarly to the definition above.

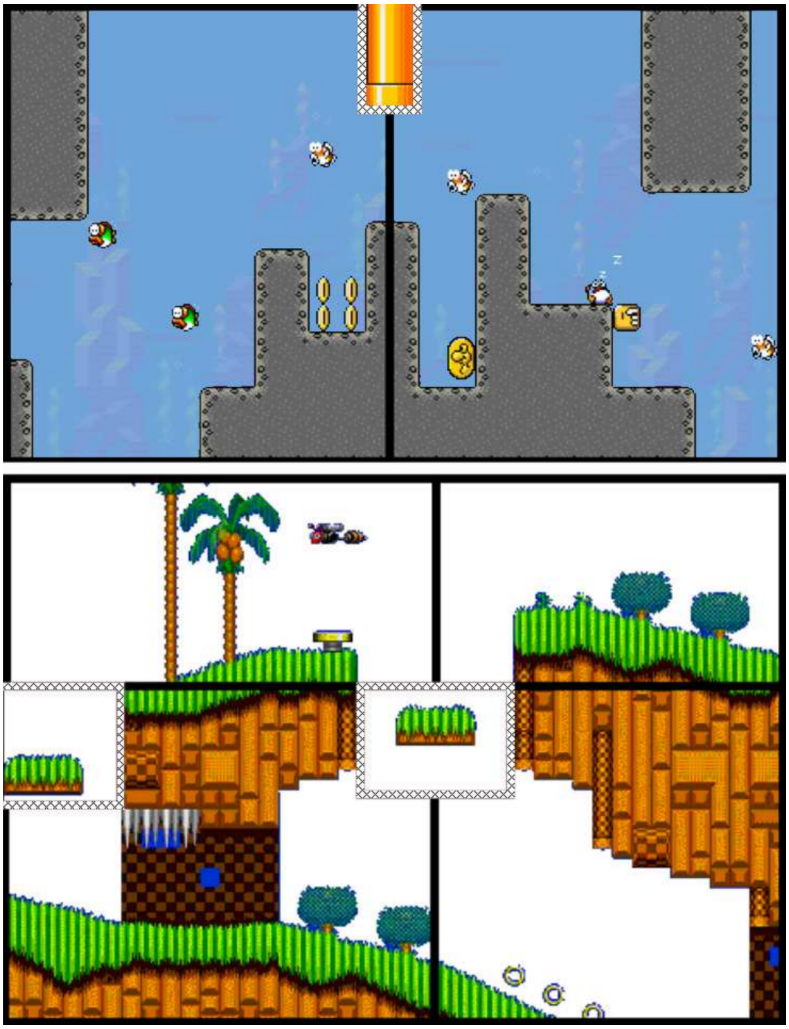

Figure 11: Rhythm Group Examples in a Section of the Platformer Games Super Mario World and Sonic the Hedgehog

The above figure, from *A Framework for Analysis of 2D Platformer Levels* [38], shows the rhythm groups surrounded by black boxes and the portals surrounded by the dashed lines. The pipe and the moving platforms are considered portals because they allow the player character to move from one path of completing the level to another.

2.3 Genetic Algorithms

A method that also uses rhythm groups is the use of genetic algorithms for level generation [42]. Although the system proposed in this paper makes no use of any kind of artificial intelligence algorithm, the advancements in the A.I. field makes their use in level design promising. An implementation of genetic algorithm consists of creating two populations (FI-2Pop algorithm) of level generators then judging their ability to make good levels, individuals deemed fit (those that are capable of generating levels that meet certain specifications set by researchers) are moved into one population while individuals that fail are moved to a different population. The two populations will be allowed to evolve with constant individual migration, which maintains diversity throughout the evolutionary process [40]. All this setup work for making a level generator is one of the reasons this project avoids A.I. algorithms altogether. The amount of work and complexity to apply such advanced algorithms will often be outside the scope of most hobbyists and small developer teams.

2.4 Occupancy-Regulated Extension Generation

An algorithmic level generation method knows as Occupancy-Regulated Extension (ORE) has been proposed as a good alternative to the other mentioned methods [34]. This method has successfully been applied to 2D side-scrolling platformer level generation in a research capacity. The algorithm method uses potential positions the player can occupy to generate level chunks around the player, then it combines the chunks into a full level. The chunks come from a pre-made library of chunks. This method is very promising, but it comes with some drawbacks and challenges. For example, tuning how the locally generated chunks will fit together can be difficult.

2.5 Anxiety curves

Albeit not a generation method, anxiety curves can be very useful in level generation when custom difficulty is needed. Generating on-demand difficulty or having dynamic game difficulty balancing (DGGB) [10] is a desirable feature

in a game, as Jason Robin, director of *Crash Bandicoot* [12], said in *Making Crash Bandicoot* [3]: "We had realized that if a novice player died a lot of times, we could give them an Aku Aku at the start of a round and they had a better chance to progress. And we figured out that if you died a lot when running from the boulder, we could just slow the boulder down a little each time. If you died too much a fruit crate would suddenly become a continue point." Another famous example is the "Difficulty Scale" system found in 2005's *Resident Evil 4* [6] which works behind the scenes to judge player skill and adapt enemy behavior based on player performance or the previously mentioned game *Hades* [18] which has a setting called "God Mode" which will continually increases the player character's endurance after every loss.

These adaptive approaches to difficulty have been present in games for a long time, but they differ from the approach taken in this paper. We plan to utilize anxiety curves as a way to test different difficulties as levels are generated. Anxiety curves are an objective way to analyze a level's difficulty without the player playing it. Meaning a level is taken as is, without a player having to run through it. Thus, the analysis of difficulty lies in judging how the obstacles in the level are placed and then graphing a line that increases and decreases as a function of the difficulty of each obstacle over the length of the level [41]. Anxiety curves by themselves are only useful after a level is generated, however, we plan to use them as an input parameter of the algorithm; the anxiety curve is chosen first, then a level will be generated by having an algorithm pseudorandomly place obstacles that follow the instructions of chosen curve.

2.6 Validation Methods

An important part in determining the quality of the output of a level generation is the feasibility of completing any given level. In a platformer the player will maneuver around hazards utilizing a combination of movement and abilities like jumping, dashing, hopping, etc. To determine whether a single obstacle is *"overcomeable"* the algorithm or model must calculate if the player character can make it through that obstacle. For example, to determine if it is possible to overcome a gap in platforms with a jump the algorithm must take into account how much distance a jump can cover and how much leniency is allowed for the player to time the jump. The other part in determining if the output of the level generation is good is how many

possible levels can be generated and how unique are they. If a procedural generator method only generates the same level or a few very similar levels it is not a good source if variety and likely will not keep players engaged.

3 Method of approach

3.1 Initial Exploration of Wave Function Collapse in GODOT with Add-on

The first approach to generation was through an add-on package, created by Alexey Bond [2], found in the Godot game engine asset library which implemented 2D and limited 3D wave function collapse for generating top-down perspective game levels. However, through some tweaking, the add-on algorithm was able to generate platformer levels. According to the package's documentation the "2d WFC generator is able to infer rules from an example of a valid map." Which means the developers of this package opted for implementing an overlapping wave function collapse architecture.

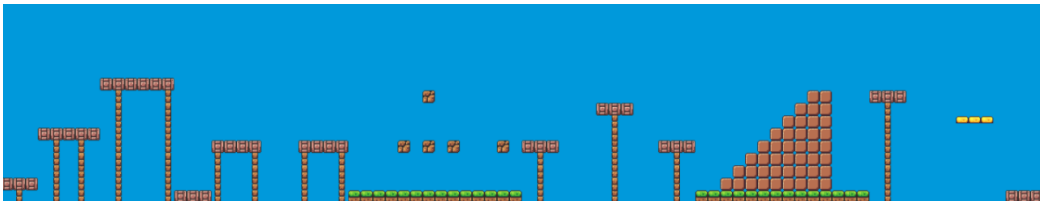

Figure 12: Example of manually created level to be used as input for overlapping WFC algorithm

In order to use the package for platformer level generation we manually built many examples of playable levels, made up of 26 different tiles based off of rhythm groups found in the Mario Bros game [31], for the algorithm to breakdown into cells and build procedurally generated levels. Our levels were created by using visually simple publicly available tile sets and drawing levels with the Aseprite pixel art tool [1].

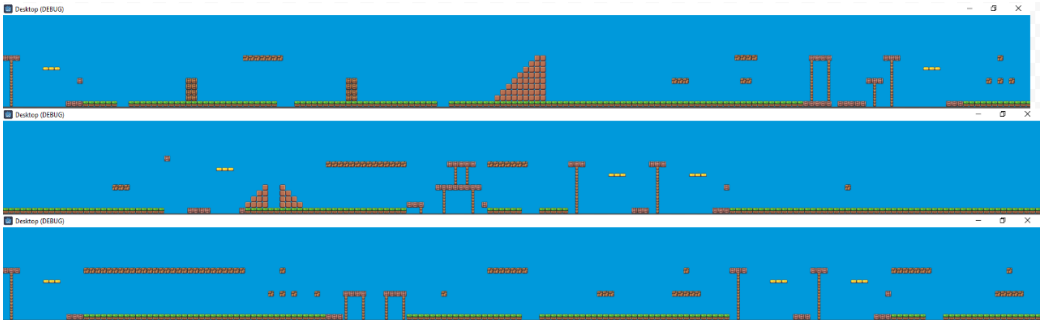

Figure 13: Example of output from overlapping WFC algorithm

The generated levels show the algorithm's ability to combine different parts of the input in new ways by creating tile sequences that were not present in the input. However, this implementation of the WFC algorithm presented two obstacles when generating 2D side scrolling platformer levels.

3.1.1 Obstacle 1: Need for too many groups of tiles

This algorithm works perfectly fine if the goal is just level variety but in order to achieve a different level of difficulty in the output we must feed the algorithm with different tiles, tiles that represent the new desired difficulty. Thus, in order to have fast generation of levels of different difficulties we would need to group tiles by their level of difficulty and feed them to the algorithm as bundles of tiles for the algorithm to breakdown. These bundles would be made up by grouping tiles of different difficulties so that their overall difficulty matched a certain desired criterion.

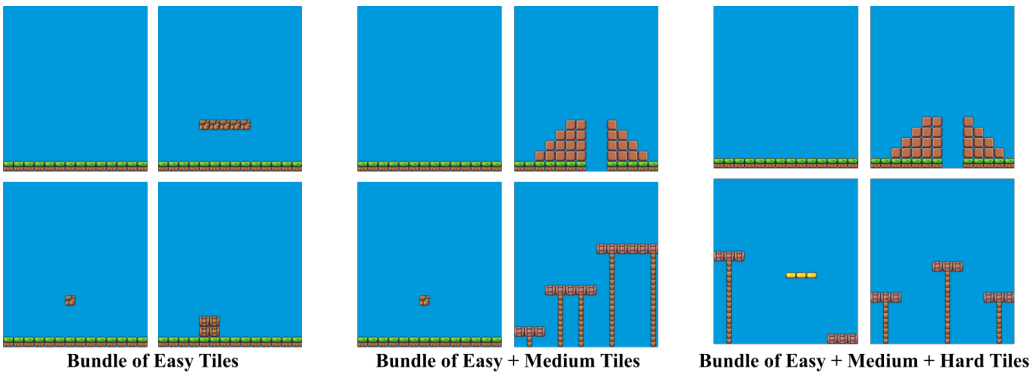

Figure 14: Some possible tile bundles based on difficulty

It would follow that a bundle containing only easy tiles would create an easy level because there is nothing in the input parameters capable of generating a hard level. But if we want many different difficulties we need to create many different bundles that will be selected according to the users choice.

3.1.2 Obstacle 2: Lack of Output Control

Since the only way to alter the difficulty of the output is by giving it new groups of tiles we would end up consuming a lot of time experimenting on how grouping different tiles in bundles would affect the generated levels. And even if we pursued this approach there is the issue that the algorithm does not guarantee that a certain tile will necessarily be used in the generation. This aspect of overlapping WFC does give it a high variability of outputs but without the certainty that a certain tile will show in the output we cannot determine that the output will follow the difficulty set by the user.

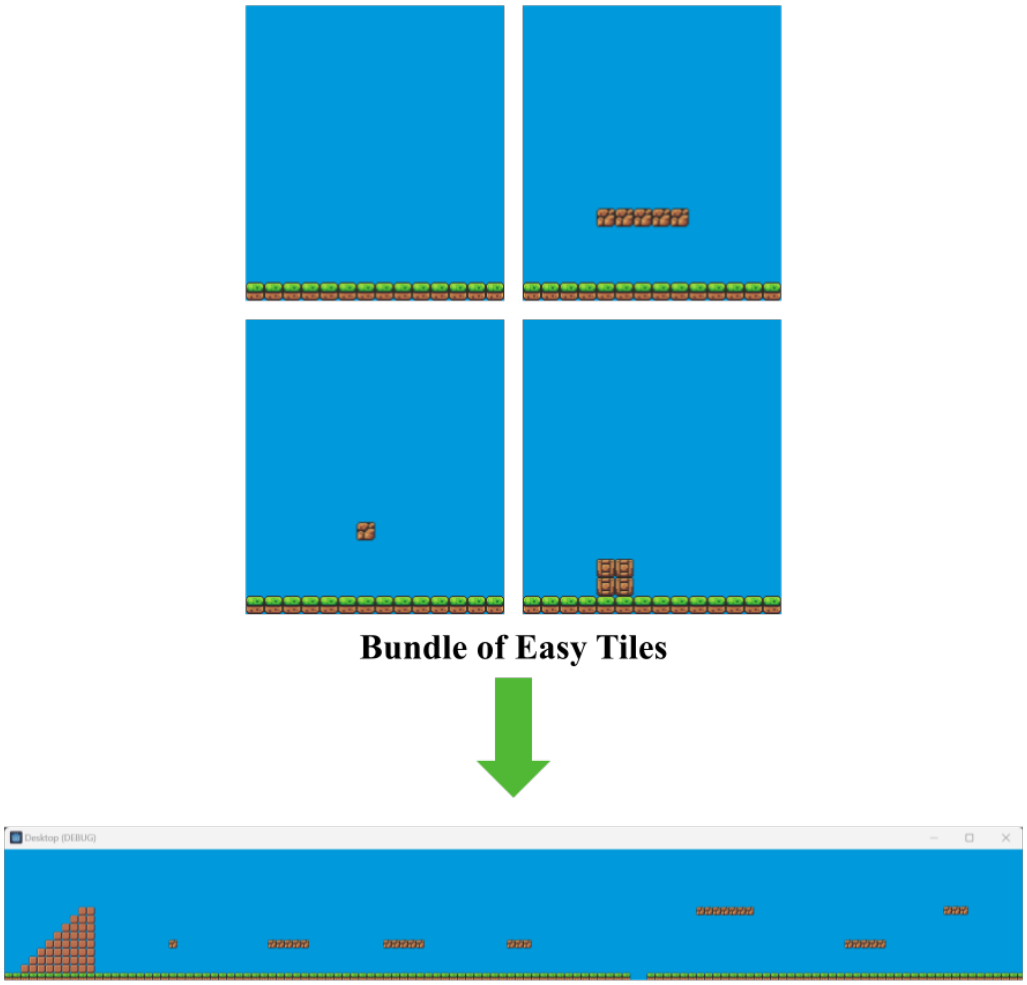

Figure 15: Some possible outcomes of generation with easy tile bundles

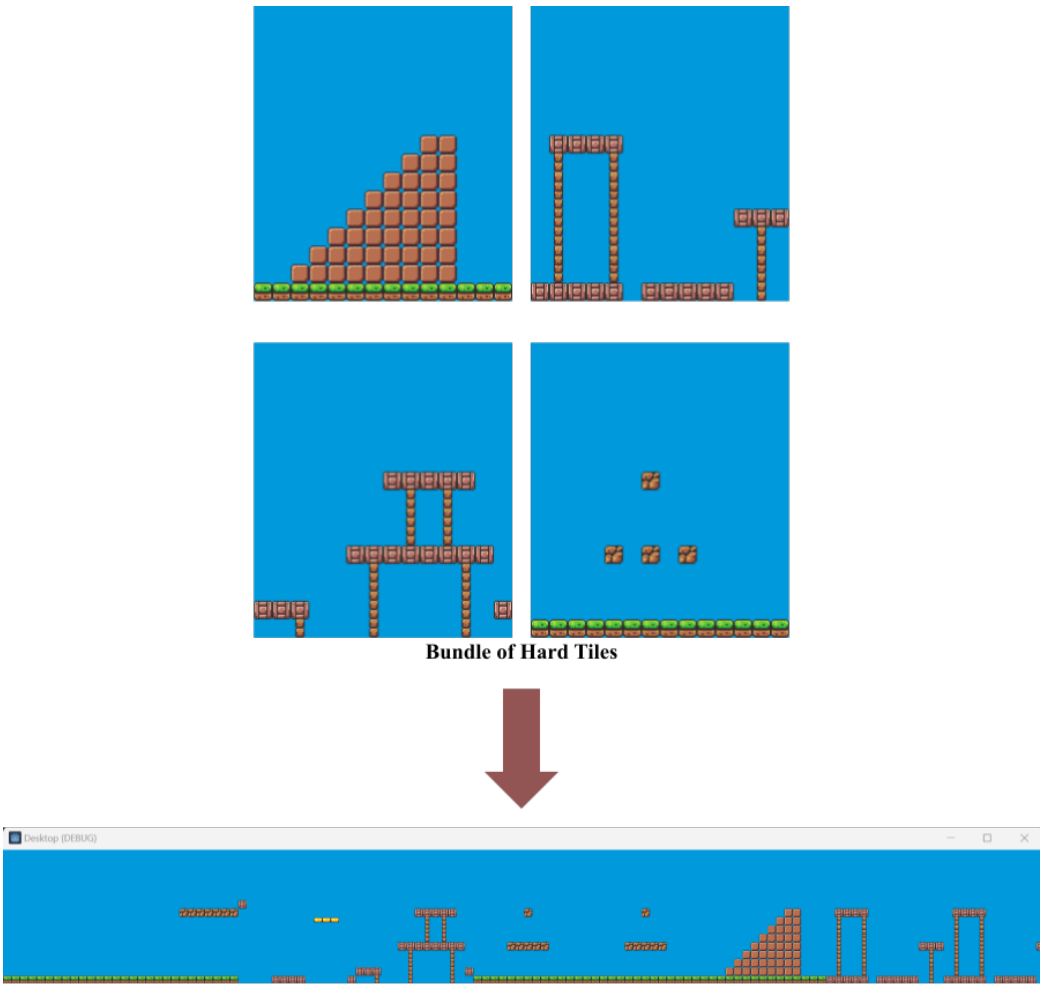

Figure 16: Some possible outcomes of generation with hard tile bundles

Thus, the algorithm, even if fed a very difficult bundle made up of mostly very challenging tiles, might still generate a very easy level.

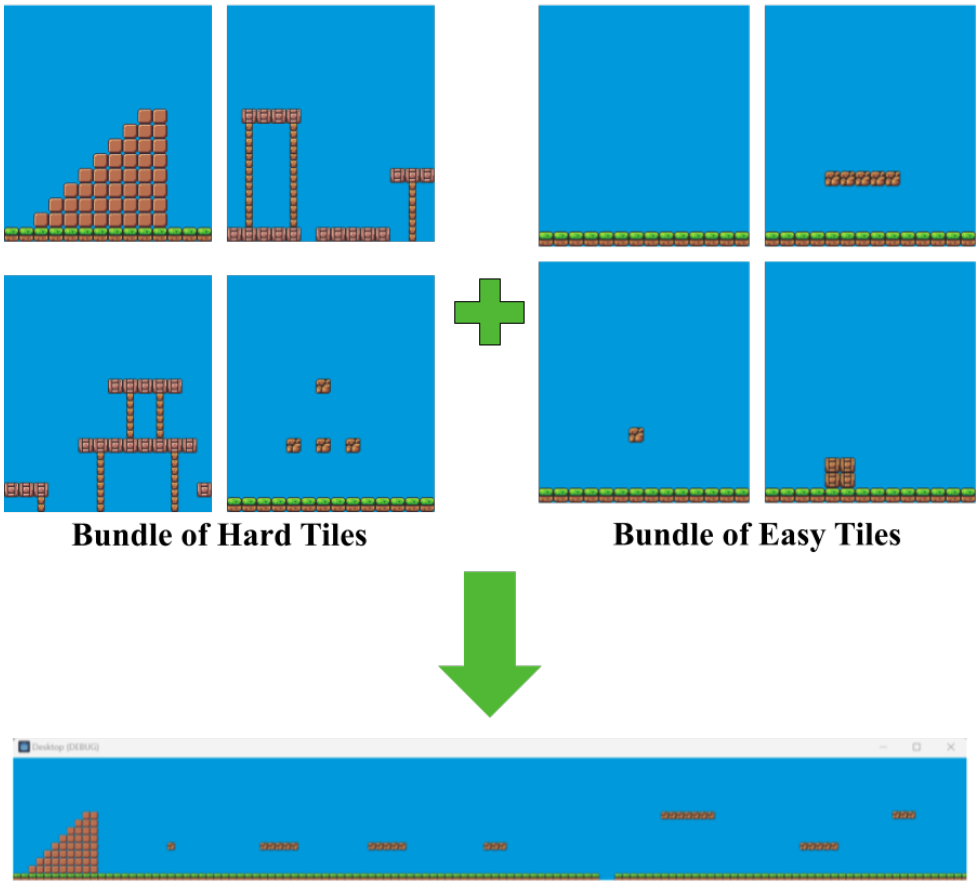

Figure 17: Some possible outcomes of generation with easy + hard tile bundles

Since altering this already existing package's implementation of WFC did not fit the scope of our researched, we've chosen to design a WFC algorithm from scratch that would fit better with platformer level generation.

3.2 Tiling Wave Function Collapse in Python

We've chosen to create a generation algorithm based on tilling WFC because this type of WFC is reliant on metadata to determine the tiles' adjacency rules instead of inferring them from examples. Tilling WFC comes with different upsides and drawbacks than overlapping WFC or Markov WFC, but its heavy reliance on metadata is desirable for the level generation task. The algorithm and visualization of generation was made using Python 3 [14] and the Pygame [8] package.

3.2.1 Design of Tile Set and Rules

By using tilling WFC we must give the algorithm single tiles rather than complete levels as input because every tile will have specific data that describes how it will fit with the other tiles. We created tiles by selecting sections from the *Super Mario Bros.* overworld levels which contained one or many rhythm groups each composed of a mix of enemies and jumps. To create the adjacency rules associated with each tile we looked at the height of the platforms on the left and right edges of the tile, and to create a value related to the anxiety curve we judged how difficult the obstacles in the tile were to overcome.

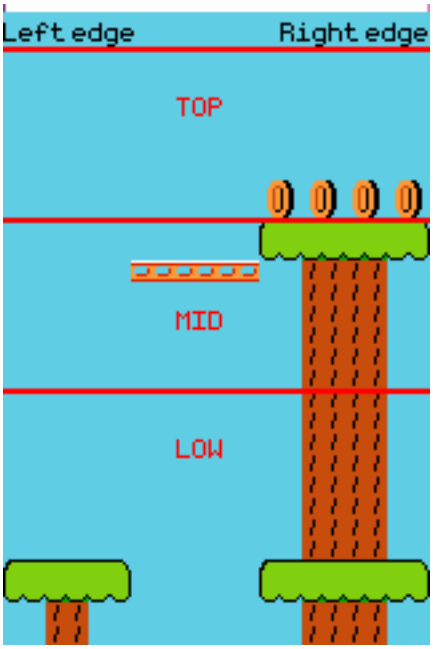

Figure 18: Height Based Adjacency Rules

The tile above illustrates how the adjacency rules behind each tile came to be. Each level in Mario used in the creation of the tile set was split into three heights based on the player character's ability to reach platforms placed on those heights. In *Super Mario Bros.* the player character is able to jump onto a platform placed four blocks above the starting point of the jump even without prior momentum. So, when considering that every level has a minimum starting height of two blocks every tile that has platforms with heights below 6 blocks are given the label `LOW`, platforms in the range 7 to 10 are `MID`, and above 10 are `TOP`. This rule set limits the generator to only place corresponding labels next to one another, which avoids the generation of scenarios with unreachable areas.

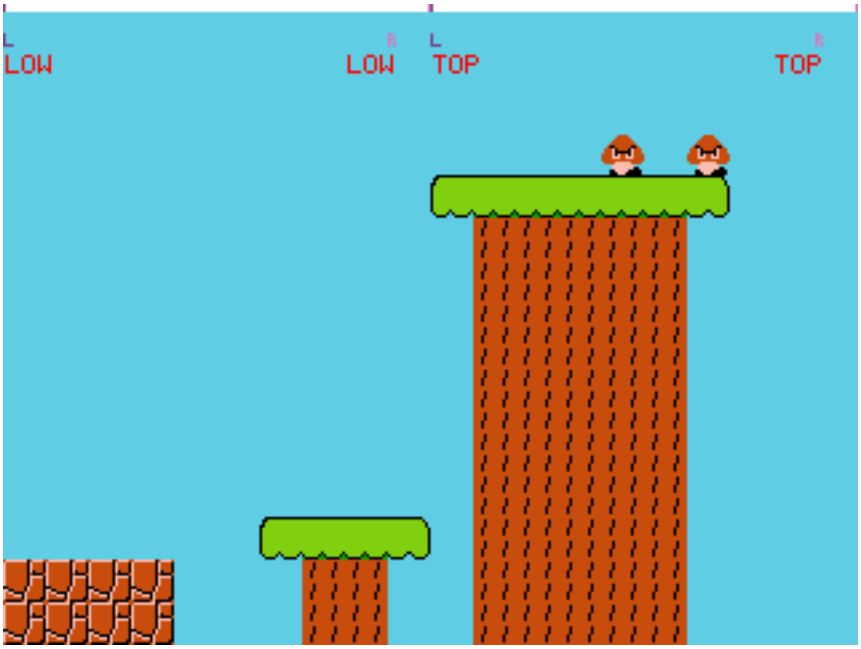

Figure 19: Unreachable Platform Example

The figure shows an example of two tiles that, if generated next to one another, would create a platform that is unreachable. The situation above cannot happen once the adjacency rules described previously have been implemented into the algorithm since the `LOW` label and the `TOP` label cannot be placed next to each other.

Initially, a tile's difficulty was judged by the jumping gaps that had to be overcome. This difficulty value is used for focusing the algorithm's generation to cater to a specific desired level of challenge:

- `EASY`: No Gaps
- `MEDIUM`: Combined Gap Width Smaller than 4 Blocks
- `HARD`: Combined Gap Width Larger than 4 Blocks

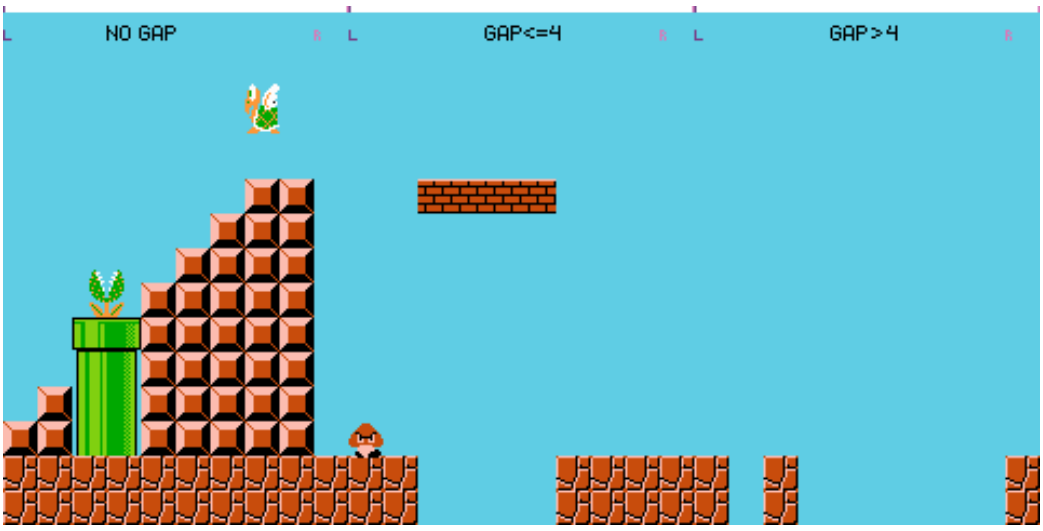

Figure 20: Difficulty Determined by Gap Size

Using this method of judging tiles, the three tiles depicted have the associated difficulty values of `EASY`, `MEDIUM`, `HARD` respectively.

With one out of three possible difficulties and two out of three adjacency rules associated with every tile we have reached twenty-seven different combinations of adjacency and difficulty rules possible for a given tile. However, since our tiles are made up of unedited sections from *Super Mario Bros.* not all configurations show up. In fact, after adding rules to every tile we find that, out of 185 total tiles, 50% are considered `EASY`, 46% are considered `MEDIUM`, and only 4% are `HARD`. With this imbalance of distribution 12 distinct combinations never show up:

Table 1: Combination of Rules Present in First Generation Tile set

Tile Left Edge	Tile Right Edge	EASY	MEDIUM	HARD
LOW	LOW	✓	✓	✓
LOW	MID	✓	✓	✗
LOW	TOP	✓	✓	✗

Tile Left Edge	Tile Right Edge	EASY	MEDIUM	HARD
MID	MID	✓	✓	✗
MID	LOW	✓	✓	✗
MID	TOP	✗	✗	✗
TOP	TOP	✗	✓	✗
TOP	LOW	✓	✓	✗
TOP	MID	✗	✓	✗

This creates a problem in generating levels with the WFC algorithm because if generation reaches the point where it must place a tile that does not exist in the set (a LOW,MID,HARD tile for example) it will break the algorithm. In order to address this issue we can either re-define the rule set or make up for the lacking tiles by adding custom tiles that will appear when the algorithm calls for one of the missing combinations.

First we addressed the rule set, instead of judging difficulty using only width of gaps we created a new rule set which counts the amount of gaps in a tile as well as the amount of enemies present in it as obstacles. The resulting number of obstacles will place the tile within one of three categories:

- EASY: 0 or 1 obstacles
- MEDIUM: 2 or 3 obstacles
- HARD: 4, 5 or 6 obstacles

With this rule set, the 185 tiles are now divided more evenly with 16.7% EASY tiles, 64% MEDIUM tiles, and 18.3% HARD tiles. An overwhelming majority is still present, but now there are more tiles in the HARD category which makes up for some previously missing combinations.

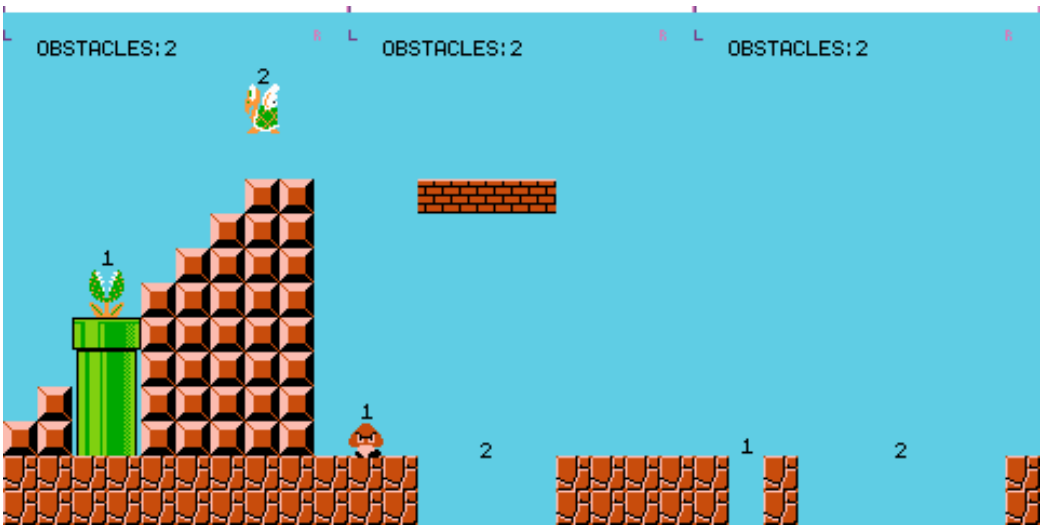

Figure 21: Difficulty Determined by Gap Size

The previously `EASY`, `MEDIUM`, `HARD` tile now became `MEDIUM`,`MEDIUM`,`MEDIUM`. And the possible combination table changed to:

Table 2: Combination of Rules Present Generation Tile set After Improving Difficulty Rules

Tile Left Edge	Tile Right Edge	EASY	MEDIUM	HARD
LOW	LOW	✓	✓	✓
LOW	MID	✓	✓	✓
LOW	TOP	✓	✓	✗
MID	MID	✓	✓	✓
MID	LOW	✗	✓	✓
MID	TOP	✗	✗	✗
TOP	TOP	✗	✓	✗
TOP	LOW	✓	✓	✗
TOP	MID	✗	✓	✓

Now there are 9 combinations still missing. Since there are no tiles present in the original game that fall into the needed combinations we manually add 9 tiles that fulfil these requirements.

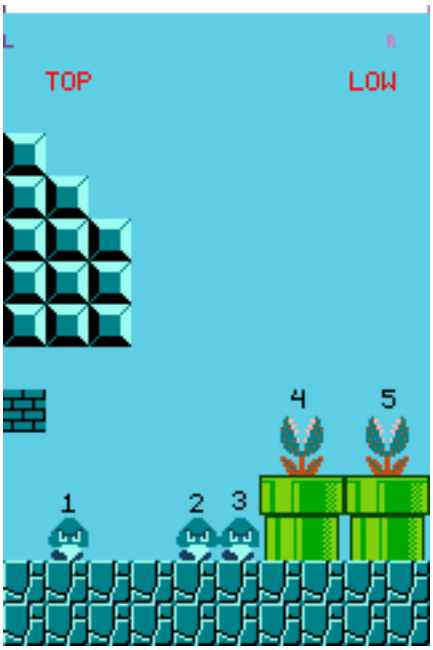

Figure 22: Example of Tile Added to Make up for Missing Combinations

The tile above was made to represent the combination of `TOP`,`LOW`,`HARD`. All the 9 tiles added on top of the tiles that came from the original game can be differentiated after generation by their color (they use the underground tile set which features a blue and black color palette), this difference is only visual and does not affect generation in any way.

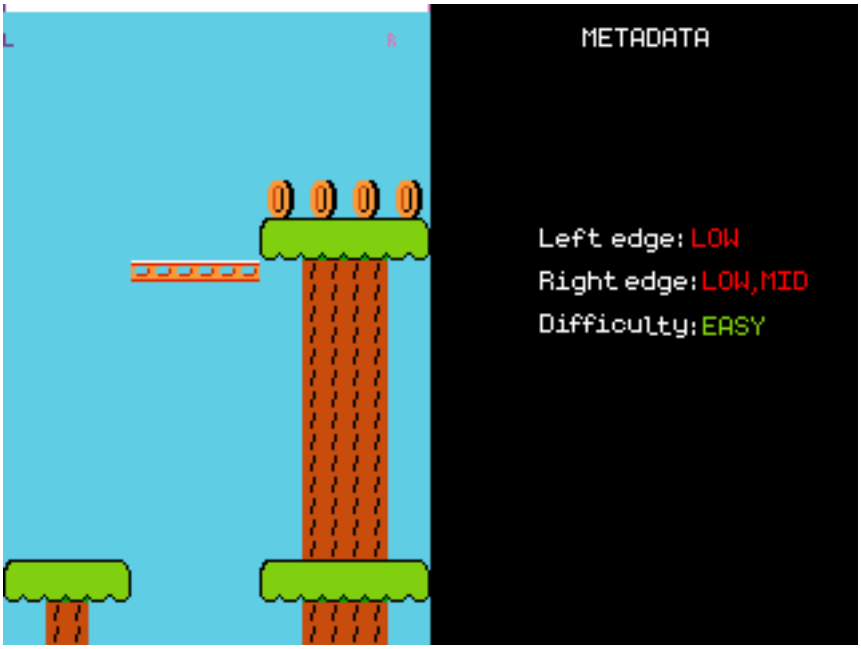

Figure 23: Single Tile and Data Associated with It

This image shows all data associated with a tile. It is worth to note that some tiles may have two adjacency rules in a single edge, the tile above has both `LOW`, and `MID` adjacency in its right edge. The implementation for this is very simple: we simply duplicate the tile and add a new entry in the adjacency dictionary for the duplicate. The above tile has two entries: (`LOW`,`LOW`) and (`LOW`,`MID`) that refer to the same tile, so it can show up in generation in either of those cases. The created tile in figure 22 also has the same property on its left edge, however since we aim to create levels with more usage of the original tiles we did not treat it as a duplicate thus it will only show when the (`TOP`, `LOW`) configuration is needed and not when the (`LOW`, `LOW`) configuration is needed.

In the interest of adding more tiles that fill the lacking previously missing configuration as well as showing a different workaround for solving the problem of missing configurations we added 3 more tiles (one for each difficulty) that follow all adjacency rules simultaneously. These have all three edge rules present on them on both the left and right edge, thus they can be put next to any other tile.

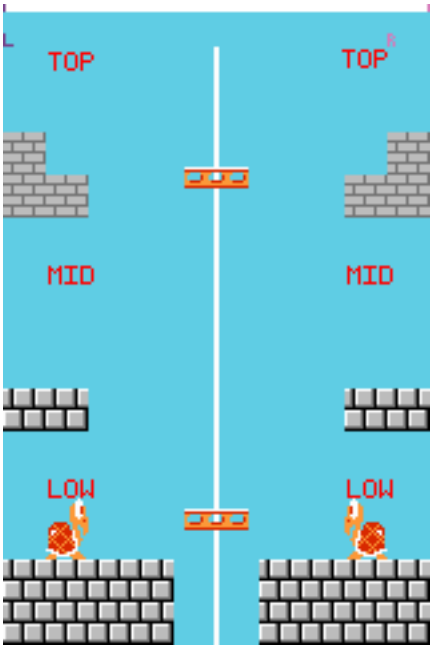

Figure 24: Example of "All Connector" Tile for Medium Difficulty

These three "all connecting" tiles could, in theory, take over the space that would otherwise be filled with another tile. We'll only add three because adding too many of them could end up lowering variety of generated levels. With all new tiles and the ones made from the game we have a total of 197 possible tiles.

3.2.2 Generation, Propagation, and Collapse

The algorithm starts by building an empty level of stipulated length, for the experiment we choose a level made up of 10 tiles with 160 pixel width and 240 pixel height. Each of the 10 generation tiles has dimensions of 10 by 15 blocks thus making a 1600 pixels wide and 240 pixels tall level, these values come from the 16 by 16 tile set's tiles that make up the 8bit sprite art style used in *Super Mario Bros.* for the blocks, enemies, and obstacles.

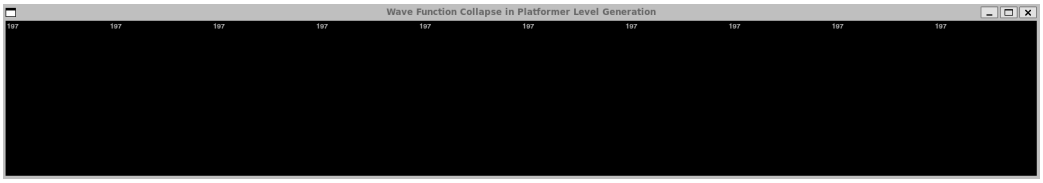

Figure 25: 10 Empty Cells that Compose a Level

Once the empty level is generated the algorithm displays the number of possibilities each tile can collapse into at the top left of each tile, the initial number is 197 because that is the number of original tiles, 185, the number of filler tiles, 9, and the number of "all connecting" tiles, 3.

Figure 26: First Cell Collapse with Labels for Adjacency Rule Identification

The algorithm picks the cell with the lowest value of possibilities to collapse, for the first collapse all values are the same, so the algorithm chooses at random. Before the first collapse the algorithm checks the anxiety curve values that each cell should have and as it collapses the first tile it ensures that it has the correct difficulty value. After the first collapse the algorithm will check the edges that of the collapsed cell and cull the possibilities of the surrounding cells to contain only tiles that have corresponding edges. Since the anxiety curve is set before the level is generation the possible values for each cell technically are not 197 at the start, but rather the maximum number of tiles of that kind. However, we chose to only read the anxiety curve after the first tile is chosen because that way the first tile chosen can be in any position, whereas if we let the algorithm choose the first tile only based off of the anxiety curve culled populations of tiles it would diminish the variety of levels since for a given anxiety curve the first tile collapse would always be in the same position.

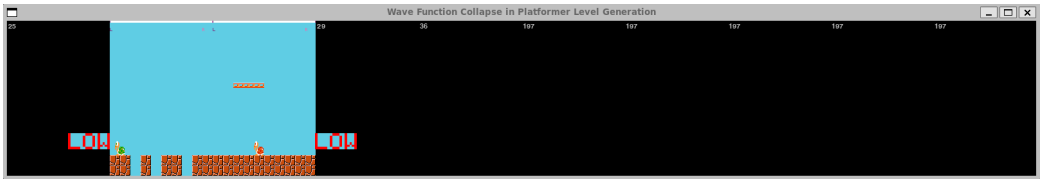

Figure 27: Second Cell Collapse with Labels for Adjacency Rule Identification

Once the next cell collapses the algorithm repeats the process and propagates the collapse result to neighboring cells and picks the cell with the lowest number of possibilities to collapse.

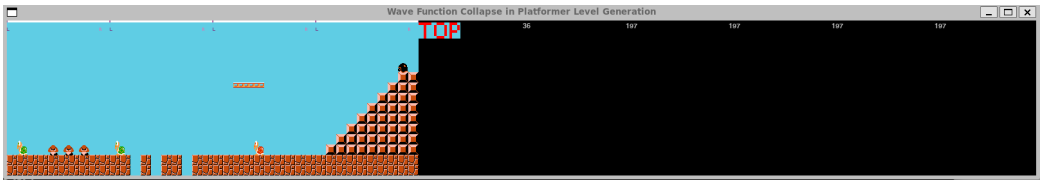

Figure 28: Third and Fourth Cells Collapse with Label for Adjacency Rule Identification

The process repeats until all cells have collapse and the level is deemed finished. In the example above the left-most cell collapsed in accordance to the adjacency rules of its neighbor and the fourth cell that collapsed, in the fourth position, also followed the adjacency rule of its neighbor but changed the level landscape from the low section to the top section.

3.2.3 Generation, Propagation, and Anxiety Curves

The propagation step of the first collapse has a special property that does not happen in the following propagation steps. As the first collapse happens the algorithm checks which value each cell should have for the level to follow the anxiety curve given to the algorithm and removes all tiles that do not have that value from the respective cells.

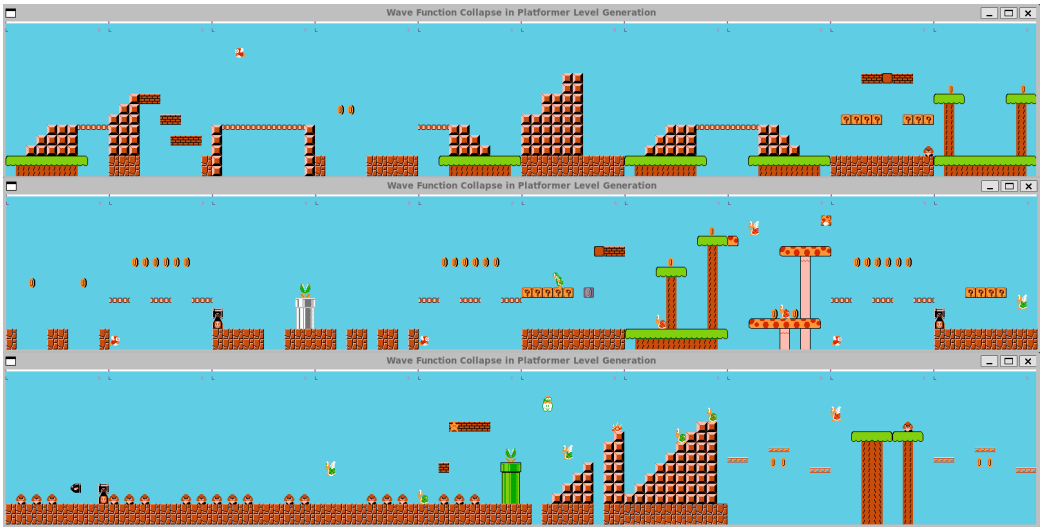

Figure 29: `EASY`,`MEDIUM`,`HARD` Generated Levels Respectively

The three levels depicted above have their anxiety curve set to a single value all through the level. The tiles may change the level verticality at random as different tiles with different adjacency rules are collapsed but their difficult always follows the pre-set difficulty value.

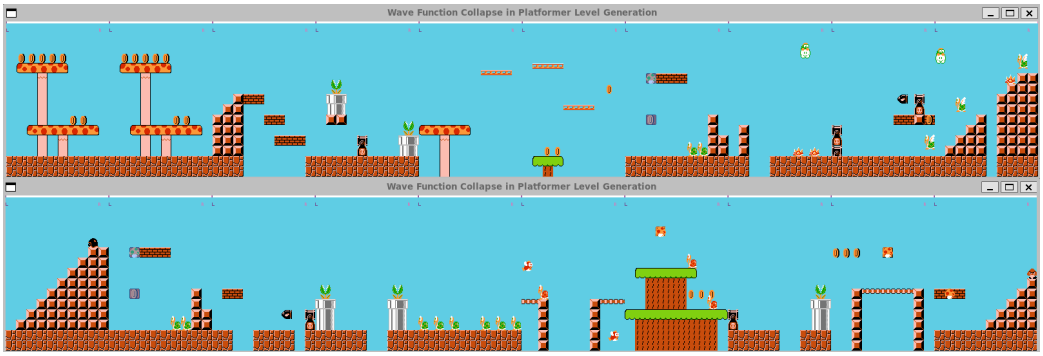

Figure 30: "Rising", and "Peak" Anxiety Curve Based Generated Levels Respectively

The top level depicted in figure 30 shows a level that follows the anxiety curve with rising values: `[EASY,EASY,EASY,MEDIUM,MEDIUM,MEDIUM,MEDIUM,HARD,HARD,HARD]`

which result in a level that gradually gets more difficult as the player progresses through it. The bottom level in figure 30 shows a level with a difficulty that rises, then peaks in the middle of the level, then declines as the level comes to a close, it follows an anxiety curve with `[EASY,MEDIUM,MEDIUM,MEDIUM,HARD,HARD,MEDIUM,MEDIUM,MEDIUM,EASY]` values.

The cell collapse examples shown in figures 26,27,28 were also generated using an anxiety curve. The desired values of the first four tiles were set to `[HARD, HARD, EASY, EASY]`. As the images show the first tile collapse (position 3) had only one enemy, which fits the description of an `EASY` tile, the second (position 2) had 3 gaps and 1 enemy, which totals 4 obstacles and thus fits the requirement for `HARD`, the third (position 1) collapse 4 enemies which also fits the requirement for `HARD` tile, and the fourth to collapse (in position 4) had a single enemy which is also fits the anxiety curve value `EASY`.

4 Experiments

Platformer levels, like any video game levels, are subjective, and their quality is highly dependent on the player's opinion of the game, their view of the game genre, their skill level, and their preference in general. In order to properly judge the output of the algorithm we will attempt to not rely on the subjectivity of whether a level is "good" or "bad" and instead focus on whether the generated levels are varied and possible to complete when taking into account the player character's abilities.

4.1 Generation rounds

As addressed in the previous chapter, the set of tiles used for generation changed as the algorithm was in development. Thus, the different generation rounds ended up with slightly different outputs due to the differences in those tile sets, we have separated the different generated levels by the configuration of the generation algorithm and the tile set used. The experiment were programmed in Python [14] using OpenCV [44] and NumPy [11] for the testing tool.

Table 3: Generation Rounds and Their Respective Generation Parameters

Round	Tile set	Anxiety Curve	Weights
1	Only original Mario tiles	None	None
2	Only original Mario tiles	Alternating AC	None
3*	Only original Mario tiles	Rising AC	None

Round	Tile set	Anxiety Curve	Weights
4*	Only original Mario tiles	Valley AC	None
5*	Only original Mario tiles	Peak AC	None
6	Original Mario tiles and extra tiles	None	None
7	Original Mario tiles and extra tiles	Alternating AC	None
8	Original Mario tiles and extra tiles	Rising AC	None
9	Original Mario tiles and extra tiles	Valley AC	None
10	Original Mario tiles and extra tiles	Peak AC	None
11	Original Mario tiles and extra tiles with priority on tiles less likely to cause unplayable levels	Valley AC	Weighted

Rounds 1 to 5 use the 185 tiles that are found in *Super Mario Bros.* and have the 12 missing configurations of adjacency and difficulty rules. While most generation rounds have 1000 generated levels each, generation rounds 3, 4, 5 are marked with a * because they suffered from the problem of the missing tile configurations described in the last chapter; these rounds have a much smaller amount of levels generated since, at some point in generation, the algorithm encountered the need to place a tile configuration that did not exist in the input tile set and stopped generating. From generation 6 onwards the algorithm utilizes the full 197 tile set and does not have any trouble

generating levels. It is interesting to evaluate the levels only containing the 185 original tiles in comparison to levels that utilized the extended set of tiles because there might be a difference in their playability, and they also serve as a baseline to check if the extra tiles made to cover for the missing configurations could cause problematic levels that are not completable.

4.2 Experimental Design and Evaluation

4.2.1 Level Variety Analysis

The first way to analyze the algorithms output objectively is to check if the output is varied, or if all tiles from the tile set are being utilized in the output, and how often tiles are repeating themselves. This analysis matters because we must be able to create levels that are similar to the levels in the actual *Super Mario Bros.* game and the game seldom repeats its rhythm groups within the same level.

To find out how often tiles repeat in generation round we will gather every level and run through each tile, if it is the first instance we add it to a dictionary and if it is already in the dictionary of tiles we increase its occurrence by one.

Table 4: Evaluation of Results of Generation Rounds for Variety Metrics

Level Generation Round	Total Tiles	Most Frequent Tile Occurrences	Least Frequent Tile Occurrences	Mean	Standard Deviation
1	185	93	24	54.054	13.763
2	78	107	22	56.179	15.747
3	185	159	2	18.648	26.436
4	104	88	1	13.077	17.913
5	159	26	1	4.277	4.117
6	197	93	23	50.761	12.244
7	158	185	16	63.291	53.091
8	197	134	16	50.761	29.512
9	754	399	43	133.333	67.016

Level Generation Round	Total Tiles	Most Frequent Tile Occurrences	Least Frequent Tile Occurrences	Mean	Standard Deviation
10	197	124	28	50.761	17.043
11	753	458	16	133.333	70.348

The standard deviation above represents how close each tile in the set is to being used equally. A value of 1 would mean that all tiles have been used exactly the same which could indicate that the levels utilize the tile set really evenly or that the levels just shuffle the tiles in a repeating sequence or worse: that every few levels are the exact same. The opposite, a very high standard deviation value, means that some tiles are heavily over-represented while others are heavily under-represented. Another piece of the data above that is interesting to observe is that the total number of tiles for each generation changes, which is expected since in round 2, for example, the algorithm was capable of making 1000 levels, but it did not utilize hard tiles since the anxiety curve only contained `EASY` and `MEDIUM` values. However, the total tiles used in generation did affect the standard deviation to a certain degree; and, as expected, with larger samples the standard deviation decreases.

4.2.2 Playability of Generated Levels

One of the main benefits of procedural generation is the ability to have near infinite distinct levels. In our case, utilizing only the tiles made up from rhythm groups found on the original Mario game we find that we can have 185^{185} different levels and this number only increases once we add the extra tiles included for better generation; though depending on the anxiety curve used, the number of tiles picked from the set may be smaller than the total. Furthermore, the amount of playable levels among the total is very hard to estimate because there are too many configurations to check. A more efficient method is to analyze the level after we have already generated it and discard it if we find it to not be playable.

To analyze if a level can be completed we look at the gaps because they are the only obstacle that can be generated in a way such that the player character cannot overcome them. To test whether these gaps can be jumped over we find a parabola that describes the player's character jump path and

set its starting origin at the point in which the player jumps from, then we test if the landing point is inside the parabola or not.

For the 1985 Mario game, the player's jump path can be described with the following equation [21]:

$$result = (-Y + a) - ((\frac{82}{5041}) * (X - b)^2) + ((\frac{164}{71}) * (X - b)) \qquad (1)$$

The Y and X values for the equation represent the landing point while the a and b values represent the shift of the origin (or the starting position of the jump) of the inverted parabola that describes the player character's jump. If the *result* from the equation is positive the landing point is inside the parabola and thus the jump is doable, if it is negative the jump is not possible.

There is a special case in which the equation will not be able to judge a jump correctly.

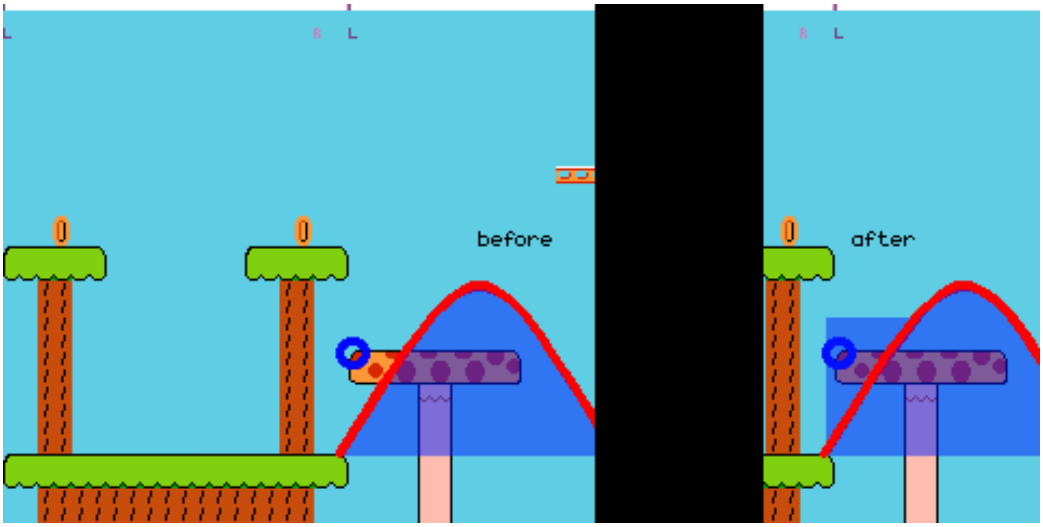

Figure 31: Case in Which the Platform is Reachable Outside the Jump Path

The image shows Mario's highest and widest jump path as the red inverted parabola, the testing tool's checking are as the blue area, and the landing point as the blue circle at the corner of the mushroom platform.

This same scenario can be found in different tile combinations as well, and, in order to account for it, we check all the platforms before the vertex of the jump path and if there is any platform below the maximum jump value (without prior momentum since we cannot guarantee the player is at max speed) the jump is considered doable. The image show's the algorithm checking area before and after accounting for this scenario.

Table 5: Evaluation of Results of Generation Rounds for Playability Metrics

Level Generation Round	Total Levels Available for Generation	Playable Total	Unplayable %
1	1000	813	18.700%
2	1000	770	23.000%
3*	345	314	8.985%
4*	136	119	12.500%
5*	68	62	8.824%
6	1000	834	16.600%
7	1000	856	14.400%
8	1000	861	13.900%
9	1000	877	**12.300%**
10	1000	836	16.400%
11	1000	878	12.200%

The results shown above indicate that some levels do contain impossible jumps. With an average of 85.654% playable levels through the generation groups the algorithm can be considered very successful at generating 2D side-scrolling platformer levels. However, the WFC algorithm is very versatile because it can be programmed with any rule set that fits the problem; the approach and implementation taken in this research is only one of many possible ways to use WFC for platformer levels.

Testing Tool Accuracy

Although the results would indicate that, on average, 14.246% of levels are unplayable the actual number of unplayable levels is smaller because the level

testing tool is only accurate at judging the path to level completion that is closest to the ground.

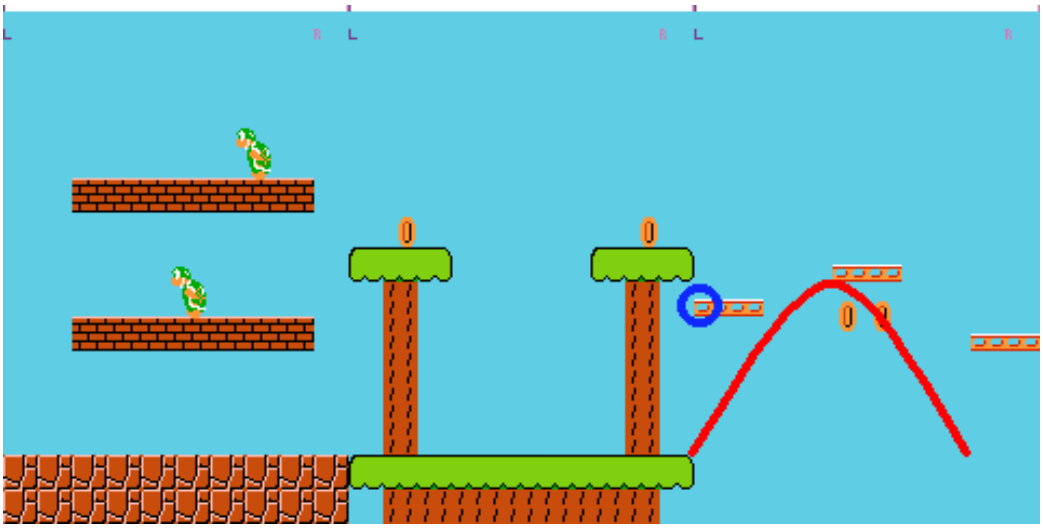

Figure 32: Example of Level Test Failure

Since the testing tool only checks for the lowest path to complete the level it cannot account for the player taking an alternate path to completing the level. In the case depicted in figure 32, the player could jump onto the brick platforms, then into the green platforms, then on the floating platforms. The checker will correctly flag the red marked jump as impossible but incorrectly flag the level as impossible to complete.

5 Conclusion

5.1 Summary of Results

The overall goal of the work was to test if wave function collapse is a good candidate for generating side scrolling 2D platformer levels. In working towards this goal we found early on, through the experimentation in Godot, that generating levels that were both interesting to play and completable using overlapping WFC alone was challenging. So we decided to implement a more metadata reliant version of WFC, tilling WFC, and also utilize anxiety curves as part of the input which allowed for better control over the generated levels which led to better generation. The levels generated were playable 85.654% of the time on average, however, the actual percentage might be higher since there is an issue with the testing tool which can produce false negatives that flag playable levels as unplayable. The levels were also highly varied, with repeated tiles only showing up in 0.05% of levels; meaning that playing through 32 levels, the amount of levels in the original Mario game for example, a player would only find a repeated tile in the same level twice through the entire game. Although the end result still produced some unplayable levels the majority of levels were both playable and unique which does show that tilling WFC can be a good choice for development of 2D side scrolling platformer games.

5.2 Future Work

5.2.1 Multiple Uses of WFC for a Single Level

Generation with this tool could be expanded by having more utilization of different WFC algorithms. An example would be to use WFC to generate

levels without any enemies then using that level as input into another WFC that solves for possible enemy configurations in the level.

5.2.2 ORE into WFC

ORE (Occupancy-Regulated Extension) is a method of generation that assembles levels based on combining levels chunks from a library of select chunks while using the player's possible positions to make level sections then combining them into a full level [34]. This generation method could be used to make the tiles that are collapsed by WFC, thus creating a level generator that create new unique tiles that then are combined into new unique full levels.

5.2.3 Different Tiles

Using the approach to generation proposed in this research in different 2D platformer games like *Donkey Kong Country* [28] for instance could drastically change the quality of the output levels. Though all 2D side scrolling platformer games share similarities they have unique mechanics that might not fit into the WFC generation method well, or experimenting with tile sets from different games might lead to interesting findings about the common elements of the games in the genre. There could even be a way to combine different games' tile sets and create levels that use both styles together.

5.2.4 Different or Additional Sets of Rules

Since WFC is very customizable a future work could take the same approach and test different rule sets performance or output quality. Instead of using left edge and right edge rules based on the height of platforms it could use adjacency rules based on aesthetics in order to create a more visually cohesive level.

5.3 Future Ethical Implications and Recommendations

The generation algorithm creates levels successfully however, the testing tool is still defective; since the testing tool is unreliable any potential developer

that would like to utilize the code associated with this research should either make their own level testing tool or manually look at the levels deemed impossible by the current testing tool and access whether the tool judged the level correctly. Still, although the testing tool is not fully accurate it only produces false negatives, thus it cannot deem an unplayable level as playable. Considering it will mark some playable levels as unplayable, and the average playability of levels was above 80%, and levels take an average of 2.3 seconds to be generated, it is completely reasonable to use this method in game development/testing since in the case of finding an unplayable level the developer can simply delete it and generate a new one because even if the delete level is playable but wrongfully marked as unplayable the replacement level will be created so fast a player would never be able to notice.

The approach to generation described in this paper is also only a single method that can be added upon, expanded, and changed. In the case a developer chooses to follow this method for their game or experiment it is very likely that they will need a new way to test whether their levels are valid (they would also have to define validity to fit their application's needs). Therefore, the unreliable testing tool used in this paper should not cause issues to a future application that utilizes this or a similar method.

5.4 Conclusions

Though the only way to deem a level generator truly successful is for it to generate fun levels, fun is too subjective to test. Building a fully featured game using the novel approach to 2D side scrolling level generation proposed in this research would be a way to *"feel"* how fun the levels are, but ultimately it is still down to preference. Regardless, the tilling WFC with the rules and inputs exemplified in this work was successful in making playable and varied 2D side scrolling levels.

6 References

[1] Igara Studio S. A. 2024. Aseprite. Retrieved from https://www.aseprite.org/

[2] Bond Alexey. 2023. WFC (wave function collapse) and generic constraint satisfaction problem solver implementation for godot 4. Retrieved from https://godotengine.org/asset-library/asset/1951

[3] Jason Robin Andy Gavin. 2022. Making crash bandicoot – part 6. Retrieved from https://all-things-andy-gavin.com/2011/02/07/making-crash-bandicoot-part-6/

[4] Brian Bucklew. 2022. Tile-based map generation using wave function collapse in 'caves of qud'. Retrieved from https://www.youtube.com/watch?v=AdCgi9E90jw

[5] Red Bull. 2017. The evolution of platform games in 9 steps. Retrieved from https://www.redbull.com/in-en/evolution-of-platformers

[6] Capcom. 2005. Resident evil 4. Retrieved from https://www.residentevil.com/re4/en-us/

[7] Shuhei Yoshida Chris Kohler. 2016. Chapter 3: The play control of power fantasies: Nintendo, super mario and shigeru miyamoto. *Power-Up: How Japanese Video Games Gave the World an Extra Life* (2016).

[8] Pygame Community. 2023. Pygame. Retrieved from https://pypi.org/project/pygame/

[9] Michael Cook. 2020. Procedural generation and information games. Retrieved from https://ieee-cog.org/2020/papers/paper_87.pdf

[10] G. Andrade; G. Ramalho; H. Santana; V. Corruble. 2010. Challenge-sensitive action selection: An application to game balancing. *IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (2010).

[11] NumPy Steering Council. 2024. NumPy. Retrieved from `https://pypi.org/project/numpy/`

[12] Naughty Dog. 1996. Crash bandicoot. Retrieved from `https://www.imdb.com/title/tt0295914/`

[13] Epyx. 2009. Rogue. Retrieved from `https://web.archive.org/web/20120815191124/http://www.edge-online.com/features/making-rogue`

[14] Python Software Foundation. 2023. Python. Retrieved from `https://www.python.org/`

[15] Toby Walsh Francesca Rossi Peter van Beek. 2006. Chapter 14 - finite domain constraint programming systems. *Handbook of Constraint Programming* (2006), 495–526.

[16] Bar 12 Games. 2006. Dwarf fortress. Retrieved from `https://www.bay12games.com/dwarves/`

[17] Freehold Games. 2024. Caves of qud. Retrieved from `https://www.cavesofqud.com/`

[18] Super Giant Games. 2020. Hades. Retrieved from `https://www.supergiantgames.com/games/hades/`

[19] Maxim Gumin. 2022. WaveFunctionCollapse. Retrieved from `https://github.com/mxgmn/WaveFunctionCollapse`

[20] itch.io. ProcJam make something that makes something. Retrieved from `https://www.procjam.com/`

[21] Jdaster64. 2013. A complete guide to SMB's physics engine. Retrieved from `https://web.archive.org/web/20130807122227/http://i276.photobucket.com/albums/kk21/jdaster64/smb_playerphysics.png`

[22] Gordon Houghton Julian Rignall. 1989. The ultimate consoles megabook! Complete games guide. *Complete Guide to Consoles* (1989), 46–47.

[23] Darius Kazemi. 2014. Spelunky generator lessons. Retrieved from `https://tinysubversions.com/spelunkyGen/`

[24] Paul Merrel. Model synthesis. Retrieved from `https://paulmerrell.org/model-synthesis/`

[25] Paul Merrel. Model-synthesis. Retrieved from https://github.com/merrell42/model-synthesis

[26] Mojang. 2011. Minecraft. Retrieved from https://www.minecraft.net/en-us

[27] Mossmouth. 2008. Spelunky. Retrieved from https://spelunkyworld.com/index.html

[28] Rareware Nintendo. 1994. Donkey kong country. Retrieved from https://www.mariowiki.com/Donkey_Kong_Country

[29] Nintendo. 1981. Donkey kong. Retrieved from https://www.nintendo.com/us/store/products/arcade-archives-donkey-kong-switch/

[30] Nintendo. 1985. Super mario bros. - miscellaneous - tileset. Retrieved from https://www.spriters-resource.com/fullview/52571/

[31] Nintendo. 1985. Super mario bros. Retrieved from https://mario.nintendo.com/history/

[32] Plausible Concept Oskar Stålberg. 2018. Bad north. Retrieved from https://www.badnorth.com/

[33] Joseph Parker. 2018. Bug with a gun. Retrieved from https://selfsame.itch.io/bug-with-a-gun

[34] Michael Mateas Peter Mawhorter. 2010. Procedural level generation using occupancy-regulated extension. *IEEE Xplore* (2010).

[35] Re-Logic. 2011. Terraria. Retrieved from https://terraria.org/

[36] Dominic Handy Robin Hogg. 1989. Japanese console dawn 16-bit the second coming. *The Games Machine Computer Leisure Entretainment* (1989), 24–25.

[37] SEGA. 1991. Sonic the hedgehog. Retrieved from https://www.sonicthehedgehog.com/

[38] Whitehead Jim Smith Gillian Cha Mee. 2008. A framework for analysis of 2D platformer levels. (2008).

[39] Whitehead Jim Smith Gillian Treanor Mike. 2009. Rhythm-based level generation for 2D platformers. (2009).

[40] DiPaola Steve Sorenson Nathan Pasquier Philippe. 2011. A generic approach to challenge modeling for the procedural creation of video game levels. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 229–224.

[41] Pasquier Philippe Sorenson Nathan. 2010. The evolution of fun: Automatic level design through challenge modeling. (2010).

[42] Pasquier Philippe Sorenson Nathan. 2010. Towards a generic framework for automated video game level creation. *Applications of Evolutionary Computation* (2010), 131–140.

[43] Oskar Stålberg. 2021. Sonic the hedgehog. Retrieved from `https://www.townscapergame.com/`

[44] OpenCV Team. 2023. OpenCV. Retrieved from `https://pypi.org/project/opencv-python/`

[45] Universal. 1980. Space panic. Retrieved from `https://www.arcade-museum.com/Videogame/space-panic`