PROCEDURAL GENERATION OF GAME-WORLDS

Christopher McCooey
University of Rhode Island, ccmccooey@gmail.com

Follow this and additional works at: https://digitalcommons.uri.edu/theses

Recommended Citation
https://digitalcommons.uri.edu/theses/2033
MASTER OF SCIENCE THESIS

OF

CHRISTOPHER J. MCCOOEY

APPROVED:

Thesis Committee:

Major Professor    Jean-Yves Hervé
Noah Daniels
Guangyu Zhu

Brenton DeBoef

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2021
ABSTRACT

In today’s market, video games are a popular source of entertainment for people. Many successful games require environments that are engaging for players. Procedural generation is a technique that allows the creation of environments and other types of content for video games. What makes a procedural algorithm successful will depend on the type of game being developed. The purpose of this research is to discuss algorithms for generating non-linear game-worlds for 2D platformer video games. During this research, I implemented a platformer game with some very basic goals and mechanics. The player can move and jump around an environment or game-world, while trying to collect coins. Once I finished the game, I implemented two different algorithms for generating game-worlds. The first algorithm did not achieve the results I wanted for my game so I implemented a maze-based algorithm. This algorithm worked better and was customized to create non-linear platformer game-worlds. After completing the algorithm, I needed a way to determine how effective the algorithm was at creating game-worlds. I completed a user-based study where subjects were asked to play my custom platformer game. The game recorded the results from the test subjects. After the study, I did some statistical analysis on the data and determined that the algorithm did a decent job controlling the difficulty of the game-worlds.
ACKNOWLEDGMENTS

There are many people that I would like to thank for helping me through my graduate career here at URI. First and foremost I would like to thank my advisor, Dr. Jean-Yves Hervé. Dr. Hervé has helped guide me through my thesis, on occasion staying quite late to give me feedback on my game, to fix issues I had with my game, and to give feedback about my writing. He was very patient with me throughout my time in the Master’s program and was always willing to help me when he could. I would like to thank Dr. Guangyu Zhu for being on my defense committee as well as helping with my research. Dr. Zhu gave me a lot of guidance performing the statistical analysis on the data collected in the user study. Additionally, I learned from him how to create various charts and graphs using R code, which was a language I had never used before.

I would like to thank Dr. Lutz Hamel for allowing me to participate in his research as an undergraduate here at URI.

For the other members of my committee, Dr. Noah Daniels and Dr. Gretchen Macht, I offer my sincerest thanks for agreeing to take the time to attend my defense.

I’d also like to thank all the people who volunteered to participate in my study.

Lastly, I would like to thank my parents for being so supportive of everything I do; without their assistance, I would never have been able to accomplish all I have.
# TABLE OF CONTENTS

ABSTRACT ......................................................... ii

ACKNOWLEDGMENTS ........................................... iii

TABLE OF CONTENTS ......................................... iv

LIST OF FIGURES ............................................... viii

CHAPTER

1 Introduction ................................................. 1
   1.1 Summary ................................................. 1
   1.2 Procedural generation ................................. 1
   1.3 Justification and Significance of the Study .......... 2
      1.3.1 Significance of platformer games ............... 2
      1.3.2 Research Focus ................................... 3
   List of References ......................................... 5

2 Background .................................................. 6
   2.1 Uses of procedural generation ....................... 6
   2.2 More content for players ............................. 8
   2.3 Decrease the workload on artists and designers .... 8
   2.4 Three degrees of procedural game-world generation 9
   2.5 Existing Procedural Generation Algorithms .......... 10
   2.6 Choices of game genre for the study ............... 12
   List of References ......................................... 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1 Creating game-worlds with a custom editor</td>
<td>33</td>
</tr>
<tr>
<td>5.2.2 Editor Tool capabilities</td>
<td>33</td>
</tr>
<tr>
<td>5.3 Procedural Generation Algorithms</td>
<td>36</td>
</tr>
<tr>
<td>5.3.1 Drunkard Walk Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>5.3.2 Drunkard Walk Implementation</td>
<td>37</td>
</tr>
<tr>
<td>5.3.3 Maze-Based Generation</td>
<td>39</td>
</tr>
<tr>
<td>5.3.4 Generation Parameters</td>
<td>46</td>
</tr>
<tr>
<td>6 Results</td>
<td>48</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>48</td>
</tr>
<tr>
<td>6.2 Game Enjoyment</td>
<td>48</td>
</tr>
<tr>
<td>6.3 Game-world Difficulty</td>
<td>52</td>
</tr>
<tr>
<td>6.3.1 Coins Collected</td>
<td>54</td>
</tr>
<tr>
<td>6.3.2 Win or Lose</td>
<td>54</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>56</td>
</tr>
<tr>
<td>7.1 Lessons from the User Study</td>
<td>56</td>
</tr>
<tr>
<td>7.2 Procedural Generation Algorithm Conclusion</td>
<td>56</td>
</tr>
<tr>
<td>7.3 Future Work</td>
<td>57</td>
</tr>
<tr>
<td>7.3.1 More Obstacles</td>
<td>57</td>
</tr>
<tr>
<td>7.3.2 More intelligent turret placement</td>
<td>58</td>
</tr>
<tr>
<td>7.3.3 Special Powerups for Players</td>
<td>58</td>
</tr>
<tr>
<td>A IRB Proposal</td>
<td>60</td>
</tr>
<tr>
<td>A.1 Project summary</td>
<td>60</td>
</tr>
<tr>
<td>B Glossary</td>
<td>62</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A screenshot of Donkey Kong [1], released in 1981.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>A screenshot from the Maryo Chronicles game [2].</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>A screenshot of procedural generation of caves from Terraria [2].</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>A screenshot of a random event in the randomized Rimworld story-line [3].</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>A screenshot of randomly generated weapons in the Borderlands game [5].</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>An image generated by the Cellular Automata algorithm [15].</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>Examples images of all the different tile types</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>A screenshot of a small game-world</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>A screenshot of the game</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>A screenshot of the game</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>Examples all the questions asked at the beginning of the game</td>
<td>22</td>
</tr>
<tr>
<td>12</td>
<td>Examples all the questions asked after the player completes the game-world</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>A screenshot of the tutorial game-world</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>A diagram illustrating the dataflow of the user-based study</td>
<td>29</td>
</tr>
<tr>
<td>15</td>
<td>A screenshot of the data file viewer program being run in Windows Command Prompt</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>A screenshot of the CSV writer tool user interface</td>
<td>31</td>
</tr>
<tr>
<td>17</td>
<td>An example of a game world created using the editor tool</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>A screenshot of the toolbar in the editor user interface</td>
<td>34</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>An example of the generation algorithm causing the player to get stuck</td>
<td>38</td>
</tr>
<tr>
<td>20</td>
<td>An example of the generation algorithm causing the player to get stuck</td>
<td>39</td>
</tr>
<tr>
<td>21</td>
<td>Examples images of all the different chunk types defined by the maze generation process</td>
<td>41</td>
</tr>
<tr>
<td>22</td>
<td>The initial maze generation for creating the game world</td>
<td>42</td>
</tr>
<tr>
<td>23</td>
<td>Two different vertical passage chunk types with the tiles filled by tile placement phase.</td>
<td>43</td>
</tr>
<tr>
<td>24</td>
<td>Generate a types for each chunk defined by the maze</td>
<td>44</td>
</tr>
<tr>
<td>25</td>
<td>Place coin tiles and change the tile images based on adjacent tiles</td>
<td>45</td>
</tr>
<tr>
<td>26</td>
<td>Choose random position for player and complete the generation</td>
<td>46</td>
</tr>
<tr>
<td>27</td>
<td>The initial parameters to the multi-phase procedural generation algorithm</td>
<td>47</td>
</tr>
<tr>
<td>28</td>
<td>The distribution of game-worlds played for the players.</td>
<td>49</td>
</tr>
<tr>
<td>29</td>
<td>The relationship to the player perceived difficulty and enjoyment responses</td>
<td>50</td>
</tr>
<tr>
<td>30</td>
<td>The relationship to the player perceived difficulty and enjoyment responses</td>
<td>51</td>
</tr>
<tr>
<td>31</td>
<td>The relationship between the enjoyment responses and the chunk dimension parameter of the random generation algorithm</td>
<td>52</td>
</tr>
<tr>
<td>32</td>
<td>The relationship between the difficulty response of the participants and the intended difficulty set by the random generation algorithm</td>
<td>53</td>
</tr>
<tr>
<td>33</td>
<td>The relationship between the difficulty and the percentage of coins collected from 0 to 1</td>
<td>54</td>
</tr>
<tr>
<td>34</td>
<td>The relationship between the intended difficulty of the game-world vs whether or not the player won or lost</td>
<td>55</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>A screenshot of a game-world where a large number of turrets were clustered together</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>A maze generated where some areas are colored different to imply them as areas that are only accessible by collecting powerups</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Summary

The purpose of this research is to determine the strengths and weaknesses of various procedural generation algorithms for 2D video games and to try to improve existing algorithm or create new ones.

1.2 Procedural generation

A video game should be both challenging and entertaining for players. In today’s gaming market, one important expectation regarding entertainment is that the game should not be the same every time\(^1\).

In a classical development format, game worlds, made up of multiple levels are created by skilled artists. This is a time-consuming task, involving the creation of 2D and 3D graphic content, complex maps, music, and scenarios, for sequences of events triggered by the player’s actions in the game. The production costs and storage limitations, even in this era of cheap, abundant storage, limit the number of such alternative levels offered to the player.

Procedural generation addresses this limitation by generating automatically all or parts of a game world. Procedural generation should be able to produce varied game worlds at a consistent level of difficulty: All game worlds produce for a given difficulty level selected by the player should have comparable degrees of difficulty. In any case, procedural generation should never produce a game world that is not playable: A game world may contain various forms of dead-ends, whether geometrical or logical, and it is part of the challenge for the payer to avoid such “dead-ends,” but it should not be “all dead ends,” or completely unsolvable,

\(^1\)This is in contrast with some legacy console games still played today, in which the challenge consists in beating records for a fixed course.
There are multiple components that are required to go into a game. The disciplines in game development fall into three broad categories: design, art, and programming. Designers create the game worlds, game mechanics (rules of the game, interactions with the environment, user controls, etc.), and overall structure of the game. Programmers make the game function according to the design. Artists create art assets for the games such as images, icons, animations, sounds, and environments. These three kinds of contributions are required to create a successful game. The work of designers and artists both involve a large degree of repetition, and also limits the range of options offered to the players: Any variation requires new content generation by an artist and/or a designer. Procedural content generation addresses both issues by using algorithms to create content for games. Procedural generation takes charge of the repetitive aspects while allowing a nearly infinite variety of game worlds and art content.

1.3 Justification and Significance of the Study
1.3.1 Significance of platformer games

Platformer-type games are a very popular genre of games and have been around for a long time. Platformer games originated in the late 1970s and 1980s. An example of an early platformer game would be “Donkey Kong,” [1] shown in Figure 1, which was released in 1981.

Over 40 years later, platformer games are still a popular genre, with significant demand for new products. A comparatively limited set of art assets allows the creation of a wide range of game-worlds. This makes this class of games a particularly suitable domain for the development and study of procedural generation algorithms.
2D side scroller platformers

A 2D side scroller is a type of video game where the camera is locked on the player. The player can move left, right, and jump. The main challenges of these types of games is navigating through the game world while avoiding obstacles, enemies, and traps.

For example, the game “Super Maryo Chronicles” shown in Figure 2 is a 2D side-scrolling video game where the player travels from left to right while avoiding enemies, traps, and cliffs.

1.3.2 Research Focus

This research will be focused on game world generation for 2D non-linear platformer games. In most platformer games the player starts on one side of the game world. They get to the end by following one path to the other side. A non-
linear platformer game is a game where the player does not follow one specific path to get from Point A to Point B. In a linear platformer game, usually the player must get from one side of the game-world to the other side game-world. They generally follow a path that is close to a straight line while avoiding obstacles. In most cases for linear platformer games, the player will have no need to traverse back to places they already traveled. They simply work there way left to right. Non-linear platformer games are different. The player often will need to backtrack to places they have already been. Additionally, the player is not moving in one direction the entire time. They are often traveling the game-world in all four directions, left, right, up, and down. This type of game often works well for games where the player needs to collect objects to win, rather than travel to a goal location.
List of References


CHAPTER 2

Background

2.1 Uses of procedural generation

Procedural generation is used to generate content for games. Below is a list of content where procedural generation can be used.

Game World In games there is always going to be an environment that the player interacts with. The game world is the environment where the game takes place. Such environments can be randomly generated. A very popular game known as Minecraft [1] uses procedural generation for all the game worlds. Many popular games that exist are 2D platformer games; some of them implement procedural generation. The game Terraria has randomly-generated 2D Gameworlds. [2]

Figure 3. A screenshot of procedural generation of caves from Terraria [2].
Story-lines  Some games involve a story-lines. The game will tell a story to the player as the player progresses through the game. Story-lines can be randomized. The computer game Rimworld [3] has a random story that is told as the game progresses. Figure 4 shows an example of a randomly-generated story-line.

![Figure 4](image)

Figure 4. A screenshot of a random event in the randomized Rimworld storyline [3].

Art Assets  Most games contain art assets. It is possible to randomly generate art assets for some games. For instance, the computer game Starbound [4] chooses random colors for the tiles and enemies in the game.

Collectible items in the game  Some games have items that the player can collect or obtain. For example, the Borderland series [5] has procedurally-generated weapons (see Figure 5).

Enemies and Obstacles  Enemies can be randomized. For example, Starbound [4] has random enemies with random stats and behaviors. [2]
2.2 More content for players

Procedural content generation aims to create a high variety of content for players while also reducing the burden of creating content from artists and designers. Procedural generation allows players to have a wide variety of game worlds to play. It also can often make video games cheaper to make. Without procedural generation, it would become more expensive to pay artists and designers to create game worlds. “By utilizing procedural generation the specific instances of gameplay differ each time the game is played – enemies are in new locations, quests might differ, crucial items are in distinct places, level layouts are different, and things the player encountered in one playthrough may not even appear in the next – and this should give such games significantly more replay value for the player than their handmade equivalents where nothing changes on each new playthrough” [6].

2.3 Decrease the workload on artists and designers

Art and level design have played an important role in making games entertaining to players. “Manual labor has so far ensured that the quality and quantity of game content matched the demands of the playing community, but is facing new scalability challenges due to the exponential growth over the last decade of both
the gamer population and the production” [7]. It can become increasingly expensive for game companies to pay for artists and designers to create content. “As the demand from players for ever more content rises, the video game industry faces the prospect of continually rising costs to pay for the artists and programmers to supply it.” [8]

2.4 Three degrees of procedural game-world generation

1. Generate the game-world before releasing the game: Game worlds are generated before the game is released. A designer takes the game-world and makes modifications to improve it. That game-world is published as part of the game.

2. Generate the game-world before playing the game world: The game will generate the entire game-world before the player starts playing it.

3. Generate the game-worlds dynamically at runtime: The game will continue to generate the game world while the player is playing that game world. As the player progresses through the game-world, the game-world expands or changes. These changes or expansions may be dependent on the state of the player.

Ideally, we should aim for the highest degree of procedural possible for the type of game considered. For example, in the case of a real-time war game, only the first kind of procedural generation is feasible. In the case of linear 2D platformer games, dynamic procedural generation is feasible, and therefore preferable. By contrast, the “don’t generate unplayable game worlds” constraint makes it difficult, if not outright impossible, because the playability criterion is based on physics, to use dynamic procedural generation techniques for nonlinear platformer games.
2.5 Existing Procedural Generation Algorithms

Procedural generation has been used since the 1980s for video games. For instance the, the 80’s games of Rogue and Elite use algorithms to generate dungeons and planets while playing the game [9]. Several algorithms have been used.

**Searched-Based Procedural Content Generation:** This is a generate and test algorithm. These algorithms will generate content and test it meeting some criteria. If the test fails the algorithm will discard what it created and try again. “Search-based procedural content generation is a special case of the generate-and-test approach to PCG where the test function will grade the content using real numbers instead of accept/reject content. Generating new candidate content is contingent upon the fitness value assigned to previously evaluated content instances” [8].

**Markov Chains:** Markov chains are a method for modeling probabilistic transitions between states. For procedural content generation of a 2D game the chains are structured in a 2D array. A study of these chains was used to generate maps for 2D games. “low-level Markov model is used to generate the low-level details of the map (the tiles used to render the map, such as ground, blocks, etc.), and a high-level Markov model is used to generate high-level structures of the map (such as slopes, platforms, etc.)” [10].

**The Puzzle-Dice System:** This technique is used to generate narrative content and puzzles for adventure RPG type games. The algorithm generates puzzles from puzzle components which are designed by people. “The initial basic approach was to separate puzzles into discrete reusable units (puzzle building blocks), which could be recombined and customized by designers making use of designer tools” [11].
**Action-based Level Generation:** This technique involves generating the actions or rhythms the player will make in the game such as jump and move. The game creates the objects based on their actions. “To determine the positions of each lamp, the generator first calculates the player’s position after following a generated rhythm.” [12].

**Drunkard Walk:** Continually have the player explore the blank world and generate the world while moving through it. The reason for using this algorithm is because it is simple. The level is generated by moving through space while obeying the rules of the game physics. The level is generated as the player walks through the level.

**Maze Generation:** There are many different types of maze generation algorithms. Maze generation algorithms are worth looking at because mazes can be used to create game worlds. “Mazes are used in various fields. Besides entertainment purposes, mazes are also used in psychology studies [9, 13] of human and animal behaviour to determine space awareness and also intelligence.” [13].

**Cellular Automata:** This algorithm can be used to generate a map which is can be used to dictate the structure of a game world such as a dungeon. “The representational model of a cellular automaton is a grid of cells and their states. An example set of allowed states would be , such that cells represent either a location where a player can go, [wall, path] or a location where the player cannot go.” [14] This algorithm can be used to generate the structure of a dungeon or cave. Figure 6 shows an example of a randomly-generated structure with Cellular Automata, which is used to define the overall structure of a dungeon or cave.
2.6 Choices of game genre for the study

Game genre

I chose nonlinear 2D platformer games for my research because the procedural generation of game-worlds for platformer games is often more challenging than top-down games. The reason for this is because the physics of the game need to be taken into account when creating the levels. A procedural generation algorithm
for a platformer game should never make a game-world that is impossible for the player to navigate. For instance, there should not be a cliff the player cannot jump over, or an area they cannot reach.

Algorithm

I chose the Drunkard Walk algorithm as starting point for my study because it seemed to be the most appropriate to 2D platformer games. This algorithm chooses a random direction North, South, East, West, and will fill that area. This algorithm would be appropriate because the gameworlds being generated are for non-linear platformer games. The game-worlds are non-linear; therefore, a good algorithm should utilize all four possible directions, rather than a single direction which would be used in a linear game world.

List of References


CHAPTER 3

Design of the Platformer Game

3.1 Game Overview

The goal of this chapter is to discuss the mechanics of the game I created. The platformer game that I created needed to be as simple as possible. The focus of the research is procedural generation so it is essential that the effort is focused on the procedural generation rather than the game mechanics, effects, art, ect. In this game the player will navigate through randomly generated game-worlds and collect coins while avoiding obstacles.

3.2 Controls

The player moves around the environment by moving left, moving right, jumping, and falling. Below is a list of all the inputs using the keyboard. The mouse is not needed to play the game.

Left arrow key  Moves the player to the left;
Right arrow key  Moves the player to the right;
Up arrow key  Causes the player to jump;
Down arrow key  Causes the player to fall through platforms.

3.2.1 Goal of the game

The goal of the game is for players to collect all the coins in the game world while avoiding being shot by enemy turrets.

3.2.2 How the player loses

The player has a number of hit points at the beginning of each level. If the player’s hit points reach zero they lose. The player’s hit points are restored to 100
at the beginning of a new level.

3.2.3 Obstacles

There are several obstacles that the player must deal with in order to successfully collect all the coins in the game. Below is a list of all the obstacles.

Turrets The game worlds can contain any number turrets which attack the player. These turrets will attack players by periodically shooting projectiles towards their position. Whenever the player collides with the projectiles, the player will lose hitpoints. The turrets will only shoot at the player when they are in range of the turrets. The turrets do not shoot at players if the linear distance between the turret and the player is large enough. The player must constantly avoid the projectiles in order to avoid losing hitpoints, and ultimately lose the game world.

Spikes Some game worlds will have spikes. Whenever the player collides with these spikes their hitpoints instantly drop to zero and they ultimately lose. The spikes are typically inside pits. The player can avoid the spikes by simply not falling into pits.

Walls and platforms Players cannot go through walls. Players can stand on walls and platforms. Platforms are similar to walls except that players can jump through and fall through them.

3.3 Game Worlds

Game worlds are all based on a grid of tiles. Every single tile in the game has two properties, a type and an image. The type determines the role of the tile in the game, while the image just determines the visual aspect of the tile. It is possible to have two tiles that have the same type but have different images. For example, a wall tile could be a metal block, a dirt block, a crate, or a dirt block
with grass. The wall behaves the same way in the game but they have different images. Below is a list of all the tile types in the game and a description on their role in the game. Visual examples of each tile type can be shown in Figure 7.

3.3.1 Tile Types

(a) Empty Space
(b) Wall
(c) Platform
(d) Coin
(e) Spikes
(f) Turret
(g) Sign

Figure 7. Examples images of all the different tile types

**Empty Space** This type is an empty space in the game world. It is possible to have an image in an empty space tile. In this particular case, the player will just walk right through the image.

**Wall** Players can safely stand on a wall. The wall blocks the player. The player cannot go through a wall from any direction. Walls block projectiles shot by turrets.

**Platform** Players can safely stand on a tile. The player can choose to fall through a platform if they hold the down arrow key. Players can jump on to platforms by jumping through. Platforms do not block projectiles shot by turrets.
**Coin**  The coin tile represents a coin that the player collect. When the player collides with the coin the coin gets collected.

**Spikes**  The player instantly loses when the player collides with spikes.

**Turret**  The turret tile is the location where a turret will be. The turret will shoot projectiles at players.

**Sign**  The sign has no physics and behaves the same as the empty space. Whenever the player is in the proximity of a sign, a message will be shown to the player on the screen. The message for each sign can be customized. The purpose of having the sign in the game is to support having a tutorial game world.

The figures 8, 9, and 10 show what the game looks like to players.

![A screenshot of a small game-world](image)

Figure 8. A screenshot of a small game-world
Figure 9. A screenshot of the game

Figure 10. A screenshot of the game
CHAPTER 4

Design of the User-based Study

4.1 Research Objective

The purpose of this chapter is to discuss the strategy for determining the success of the algorithm. Most of this information will be about the design of the user-based study.

4.1.1 Metrics for Procedural Generation

Determining and measuring the success of the procedural generation algorithm for game-worlds is not an easy task and is very subjective. There are many ways to define success for procedural generation. Some examples are included in the list below.

Enjoyment A successful algorithm generates game-worlds that players enjoy playing. If the game-worlds cause the player experience to be compromised then it is likely not a good algorithm.

Diversity In some games it might be important for the algorithm to generate a large variety of game-worlds that do not all seem overly similar.

Difficulty Many video games are structured in a way where the player is presented with easy game-worlds and they gradually get more difficult as they play more. It might be important for the algorithm to have a difficulty input and be able to generate a game-world with the intended difficulty. The difficulty of game-worlds must be consistent. If two game-worlds are generated at medium difficulty, then it is expected that both game-worlds will be medium-level in difficulty for players.
Performance How long does it take for the algorithm to generate a game-world? Sometimes run-time performance is an important metric.

Game World Validity A successful algorithm needs to be able to generate game-worlds that players are able to win. Generating an impossible game-world could be a problematic scenario.

4.1.2 Metrics for this research

For this research the goal is to write an algorithm that creates game-worlds that are enjoyable and are at the intended difficulty. How do we decide if the algorithm created a game-world that is easy or hard? The only way to do determine the enjoyment of the game-worlds is to have people test the game, and record feedback from them. We can determine how difficult the game-world by getting feedback from the player and by measuring their performance during their play through. Players simply play the game, answer a few survey questions, and send a data file that includes both their answers and their performance. All of this is accomplished in the implementation of the game.

4.2 Data Collection

4.2.1 Survey Questions

Participants will be asked some very simple survey questions at the beginning of the game and at the end of each game-world. The questions are shown below.

4.2.2 What participants do in the study

Users simply run the application to begin playing the game. Once the game starts the users are presented with three questions shown above in Figure 11. The are presented one at a time in the order shown in the figure. Once the participant answers all questions, they can immediately begin playing the game. The first game-world that is shown is the tutorial game-world. The tutorial is a special
Figure 11. Examples all the questions asked at the beginning of the game game-world that is not randomly generated. It is a game-world that I designed in the level editor tool. More information about the tutorial game-world and design decisions made when creating it can be found in Chapter 5 Section 5.2.1.
Figure 12. Examples all the questions asked after the player completes the game-world.

Once participants complete the tutorial or any other game-world, the participant is presented with the three questions shown above in Figure 12. The participants are
presented with a new game-world to play. The difficulty parameter used to generate
the game-world is completely random; therefore, participants will be unaware of
how difficult the game-world was intended to be. At this point, the game will
present participants with randomly generated game-worlds over and over again.
There is no limit to how many game-worlds the player can play. Participants are
predicted to play at least two game-worlds; however, they can play as many as
they want. This is to give participants more freedom as well as allows me to have
control over how long the play session should be. There is no way to predict how
long it will take for a participant to complete a game-world. This is why it is
better to not have a hard-limit on the number of game-worlds that participants
must play.

4.3 Tutorial Game-world

The tutorial game-world is designed to teach the player the mechanics and
objectives in the game. I wanted a tutorial game-world where a person who has
never played the game before could just sit down and learn the controls without
me having to say anything to them. Figure 13 shows the tutorial game-world that
all participants in the study will play.

The tutorial game-world was specifically designed to only be winnable if the
participant learns all the controls of the game. In the game, there is a tile-type
called the sign tile. The sign displays text information to the player. The reason
the sign was even added to the game was strictly for usage in the tutorial. The sign
tile is not ever created in the random game-world generation. More information
about the sign tile can be found in Chapter 3 Section 3.3.1. The tutorial game-
world was designed to ensure that the players were familiar with all the controls
and game mechanics after completing it. The tutorial game-world has several areas
with a specific mechanic or control that the player is intended to learn. There are
Figure 13. A screenshot of the tutorial game-world

4.4 Data Instances

Each data instance contains information about a single play-through of the game. The data files are automatically written to the disk whenever the game is exited. The data files are in a custom format that is unreadable to people. This was done to ensure that people who play the game do not manipulate the data file. Each data file contains information about the overall play-through and information about each game-world. I was very careful about what data I chose to record to the file. I needed to keep the data anonymous so I made sure not to record anything that could be used to identify the individual that played the game. Below is a detailed list of all the data recorded to the file and the justification for recorded that data.

Beginning of game Questions The answers to the questions asked to test par-
participants at the beginning of the game are written to the data file. The amount of experience a person has with video games could have a large influence on their performance in the game and enjoyment thereof. The questions asked are shown in Figure 11.

**Monitor Information** The bit depth and device pixel ratio might affect the display by making everything really small. This could affect the player’s overall performance.

**Operating System** The OS will either be Windows, macOS, or Linux. This information is recorded to the data file because the windowing of the game will be different on different operating systems. This is unlikely to affect the results of the study, but is included anyway to be safe.
Number of game-worlds played  The number of game-worlds played must be recorded. Participants in the study have the ability to decide how many game-worlds they want to play. One reason for this is that each game-world generated is random; therefore, there is no way to predict how long it will take a participant to finish the game-world. The tutorial is always the first game-world.

For each level the following information is recorded.

Post Game-world Questions  The answers to the questions asked to test participants at the end of each game-world are written to the file. I wanted to know how enjoyable they found each game-world, as well as how difficult they perceived each game-world to be. The questions asked are shown in Figure 12.

Generation Parameters  The parameters that were used to generate the random game-world are recorded to the file. This allows me to regenerate and view any of the game-worlds that a participant played through. More details about the game-world generation and its parameters are discussed in Chapter 5 Section 5.3.4.

Player Performance  Answers to survey questions are not the most accurate way of determining how well a player performed during each game-world. People will have different interpretations on what they consider to be easy or hard. Therefore, looking at the factual data such as the number of coins they collected makes for a better data analysis. There are several different parameters are written to the file that are an indicator of player performance:

- Number of coins collected,
- Number of coins in the level,
• Number of turret shots fired,
• Number of jumps made,
• Number of seconds before the player won or lost,
• Number of seconds before the player started moving,
• Remaining hit-points when the player won or lost,
• A 0 or 1 indicating whether the player won or lost.

A lot of the data recorded to the file may not be that important in drawing conclusions about the procedural generation algorithms; however, it is safer to record too much information rather than not enough information. The main purpose behind the player performance variables is that they can be used to try and predict what happened when the player played the game-world. For example, if there was a high number of turret shots fired and the player lost the game very quickly, than it is reasonable to assume the player immediately got bombarded with turret shots and lost.

4.5 Data Analysis

The study concludes by having a collection of data files. Each data file represents one play-through by a single participant. For example, if there are 20 data files then that would mean 20 people played the game. A statistical analysis on this data must be done to get an understanding on how the procedural generation algorithm did in creating enjoyable game-worlds at the intended difficulty. In order to complete the statistical analysis, all the data must be combined in CSV files so that graphs and charts can be constructed. The data that goes into the CSV file can be data taken directly from the DAT file or data that is computed from data in the DAT file. Figure 14 shows the entire process from having participants play the game to the final data analysis.
4.5.1 Record Data Reader Tool

The data files that are created when participants play the game are in a custom binary format that is unreadable to any text editor or known programs. This was done intentionally so that participants cannot manipulate the data that is collected. The Record Data Reader Tool shown in figure 15 is a tool I wrote in C++ to allow me to view the record data files. Obviously, participants in the study are not given access to this tool.

The Record Data Reader tool is very simple, it just prints all the information in the DAT file to the command line. It prints the data in the following order: Header information, level 1, level 2, level 3, ... level N.
4.5.2 CSV Writer Tool

It is important to be able to automate the process of creating a CSV file from DAT files. The CSV file data must be recreated if ever I need to add more computed data to the CSV or if new data gets added to the data set. This is
something that happens often enough where automation of the CSV generation becomes necessary.

![Figure 16. A screenshot of the CSV writer tool user interface](image)

The CSV writer tool shown in Figure 16 is a user interface that I created which is used to accomplish this task. This allows me to instantly import all of the data files and immediately know what the CSV file will look like. It has built in settings that allow me to customize what goes into the CSV file. For instance, the tool allows me to put the tutorial game-world data and the randomly generated game-world data into two separate CSV files. The tutorial is not a randomly generated...
game-world and is intended to teach the players the game; therefore, the tutorial should not be included into the statistical analysis of anything that has to do with the random game-world generation.
CHAPTER 5
Implementation

5.1 Abstract

In this chapter, I will discuss the procedural generation algorithms implemented and how they work. There are two algorithms to discuss. The first algorithm is the classical Drunkard Walk algorithm that is often used for dungeon generation. However, even after several modifications and “tweaks,” it proved not to be suitable for nonlinear platformer games. Based on the drawbacks and limitations of the Drunkard Walk, I developed and implemented a second algorithm, which performed significantly better and was adopted for the user study implementation.

5.2 Creating and working with Game Worlds
5.2.1 Creating game-worlds with a custom editor

The level editor is a component in the game that allows the creation and modification of game worlds. Before any procedural generation could be implemented, I needed to have game worlds that could be used to test the mechanics of the game. Game worlds in the editor start out as an grid of empty tiles with the perimeter tiles being set to wall tiles. Tiles can be modified by changing both the tile type and tile image. Figure 17 shows a game-world that was created using the editor tool.

5.2.2 Editor Tool capabilities

The editor has a variety of tools that allow customization of game-worlds. Figure 18 displays the toolbar for all the tools in the game. Below is a list of all the tools and what they do.

**Save** Saves the game-world to a file on the disk.
Figure 17. An example of a game world created using the editor tool

Figure 18. A screenshot of the toolbar in the editor user interface

**Load** Loads an existing game-world file from the disk. The old game-world gets wiped out and changes are lost unless the user saves the changes.

**Pan** Changes the cursor to the pan tool. The pan tool will cause the mouse to
pan the area of the game world that is on screen.

**Change Block** Changes the cursor to the block tool. Left clicking will change the tile type and image while this tool is active.

**Delete Block** Changes the cursor to the delete tool. Left clicking will change the tile type to an empty space and any image that is used for that tile will be removed.

**Rotate Block** Changes the cursor to the rotate tool. Left clicking will rotate the block clockwise by 90 degrees.

**Change Player Position** Changes the cursor to the player position tool. Left clicking will change the designate the position where the player appears when the user enters the game-world. There is only one player start position allowed; therefore, the previous start position is no longer used.

**Change Sign Text** Changes the cursor to the sign text tool. Left clicking on a tile that contains a sign displays a window that allows the user to enter text. Once the window is closed, the text the user entered will be assigned to that sign tile. If the sign tile is deleted or changed, the text that was used for that sign will be lost.

**Expand Placement Tool** Modifies the change block and delete block tools such that they edit a larger area. The default is $1 \times 1$. This button changes it to $2 \times 2$, $3 \times 3$, etc.

**Shrink Placement Tool** This button modifies the change block and delete block tools such that they edit a smaller area. For example if the user clicks it and the size is $3 \times 3$ it will become $2 \times 2$. If the tool’s dimensions are $1 \times 1$, then this button does nothing.
Add Column This expands the game world by adding a column of tiles to the right side. The tile type and image is copied from the tile directly to the left of the new tile.

Add Row This expands the game world by adding a row of tiles to the right side. The tile type and image is copied from the tile directly above of the new tile.

Remove Column The last column of the game-world on the right side is deleted, which decreases the width of the game-world.

Remove Row The last row of the game-world on the bottom side is deleted, which decreases the height of the game-world.

Circular Shift Left Shifts all the tiles by one to the left. The first column becomes the last column.

Circular Shift Right Shifts all the tiles by one to the right. The last column becomes the first column.

Circular Shift Up Shifts all the tiles by one upward. The first row becomes the last row.

Circular Shift Down Shifts all the tiles by one downward. The last row becomes the first row.

5.3 Procedural Generation Algorithms
5.3.1 Drunkard Walk Algorithm

This is the algorithm that I originally planned on using to generate game worlds for my non-linear platformer game. The basic idea consists in using an AI to control a virtual player, and using the displacements of that virtual player to generate the world. Basically, it is like having the player walk and jump around,
and using these actions to generate the tiles. The algorithm works by following the following procedure:

1. **[Initial creation]** Create a game-world where the entire grid is an empty space surrounded by wall tiles on the perimeter. Place the player on top of one of the tiles at the bottom of the game world.

2. **[Determine possible directions]** Look around the player and determine what actions are possible. Possible actions can include move left, move right, jump, or fall. The action that is determined has some degree of randomness and is also based on what the player had done in the previous iteration. For instance, switching directions should not be a common occurrence or the player would end up moving left and right too much without traversing the whole game world.

3. **[Create tiles]** Create tiles in the proximity of the player. The tiles that are chosen are based on the existing tiles near the player and based on some type of randomness.

### 5.3.2 Drunkard Walk Implementation

The main problem with this algorithm is that, due to limited “look-ahead,” the tile creation phase can generate wall cells that completely block the player’s progression. Figure 19 shows an example of such a situation. Once the AI-controlled “player” used by the generator is stuck, the algorithm can no longer generate tiles, and therefore stops work properly altogether.

I attempted to overcome this problem by making the following changes to the tile creation step:

- Do not create tiles too close to the AI-controlled player;
Figure 19. An example of the generation algorithm causing the player to get stuck

- Only create walls close to the AI-controlled player’s position if the wall is below the player;

- Create multiple tiles at once in specific patterns;

- Delete a tile if the AI-controlled player is stationary for too long.

All of the above changes helped somewhat, but never really solved the problem. The AI-controlled player would still get stuck in walls as shown in Figure 20. Another issue with this algorithm that was never addressed is writing it in a way where most of the game world gets filled with tiles. Randomly moving the player around the game world does not guarantee that the player will explore all areas of the game world. This creates a problem because it will result in several regions of the game world having no generated tiles. After carefully considering all the issues I decided that coming up with a completely new algorithm would have a better chance of being successful than a further-amended Drunkard Walk algorithm. This algorithm is too dependent on the physics of the game and has no way of controlling the overall structure of the game world.
5.3.3 Maze-Based Generation

This is a new algorithm that I developed for generating game worlds. The approach was very different than the Drunkard Walk. Instead of randomly moving around the game world, this algorithm proceeds in a “coarse-to-fine” manner by generating first a high-level blueprint of the game world and then filling in the details, that is, instances of the tiles shown in Figure 7 of Chapter 3, which are the finest level of detail in the game. To insure that the game-world produced is playable, the coarse blueprint must verify some basic navigability conditions. This is where I had the idea of using mazes for that blueprint generation. There are many algorithms for maze generation that will construct a maze where all areas are accessible. This algorithm has three completely separate phases that all play a different role in generating the game.
**Structure generation**

The first phase is the game world structure generation. This phase aims to define the high-level structure of the entire game. I decided to use a maze to define the high level structure of the game. I needed a maze generation algorithm that would fill every square of an empty grid and generate a maze such that every square could be accessed. This is important because every dead-end and location in the maze must be accessible. Inaccessible areas in the maze would result in game worlds with inaccessible areas. This would cause coins to appear in places that are inaccessible; therefore, making game worlds impossible to complete.

The elementary objects manipulated by the structure generation algorithm are chunks, which correspond to an $N \times N$ grid of tiles. Usually the value of $N$ is small. In the case of this algorithm I always used $N = 8$; however, the chunk size is a parameter that can be changed before generating the level.

There are 15 different possible chunk types, shown in Figure 21, for the maze. Different types of chunks correspond to navigability possibilities to adjacent chunks. For example, the chunk in Figure 21(a) allows displacements to adjacent chunks in all four directions, whereas the chunk in Figure 21(f) only allows displacements to the chunk on it immediate left and to the one immediately below it. It is important to keep in mind that this list of possible displacements is only defined at the coarse, chunk level: At the finer, tile generation, level of the algorithm, it will be possible to generate within a chunk of Type (a) a tile preventing motion to some directions, but such tiles should not form a “wall” preventing all moves in a given direction, thus invalidating the type of the chunk they belong to.

The maze is generated using the Depth First Search algorithm, which proceeds by attributing neighbors to chunks along the following steps:
1. Choose the initial chunk, mark it as visited and push it to the stack;

2. While the stack is not empty;
(a) Pop a chunk from the stack and make it a current chunk;

(b) If the current chunk has any neighbors which have not been visited;

(c) Push the current chunk to the stack;

(d) Choose one of the unvisited neighbors;

(e) Remove the wall between the current chunk and the chosen chunk;

(f) Mark the chosen chunk as visited and push it to the stack.

3. Each chunk’s type is determined by the adjacent chunks that have been attributed to it as neighbors. For example, a chunk that is only connected to the chunks on its left and that below it would be of Type (f) according to the numbering of Figure 21.

Figure 22 shows an example of final output of the maze generation phase.

Figure 22. The initial maze generation for creating the game world
**Tile placement**

A tile type must now be associated to each of the $N \times N$ tiles that make up a chunk. The tile placement phase is only aware of what chunk type is being used. Each chunk-type follows a different set of rules on how tiles are placed. For example, the vertical passage chunk needs tiles to be placed in a way where the player can travel through the tile by exiting from the top or the bottom. The tile needs to be set up where the player cannot escape on the left or right side. This is accomplished by setting the first and last column of the chunk to wall tiles. The middle area is filled with platform tiles. The platform tiles are separated by 1 or 2 empty spaces which is determined by the maximum jumping height of the player. Other than this rule there is some randomness to the horizontal layout of these platforms. The two different images in Figure 23 illustrate the results of the tile placement algorithm on vertical passage chunk shown in Figure 21(n).

![Figure 23. Two different vertical passage chunk types with the tiles filled by tile placement phase.](image)

**Coin placement**

After all the tile types have been defined, it becomes possible to generate coins
Figure 24. Generate a types for each chunk defined by the maze

in the game-world so that all coins are reachable. This means that all coins must be placed in an empty space above a wall or a platform. Each chunk has a fixed number of coins that are placed throughout the level.

**Image filling**

Once this phase is reached, all of the tile types for the entire game-world have been defined and all of the coins have been placed. However, a decision on what images to represent each tile must be made. As described earlier, each tile type can be represented with more than one image. A wall could be a block of dirt or a block of dirt with some grass on it. A decision about what image should be used must be made for each tile. The goal is to choose a tile that is ascetically appropriate. For types such as coins, there is only one image so no decision is made. For types such as a wall, there are many different possible images. For walls, the image that is chosen is based on the type and image of the tile above and to the left. The
algorithm runs from left to right, top to bottom. Here is an example decision. For a wall choose to use the dirt with grass image if the tile above it is an empty space, otherwise, if the tile above it is another wall type, then choose the dirt image without the grass. There are many more decisions similar to that during this phase of the algorithm. Figure 25 shows what the game-world looks like after choosing an image for each tile.

Figure 25. Place coin tiles and change the tile images based on adjacent tiles

**Player placement**

The final phase of this algorithm is the simplest and consists in choosing a starting location for the player. The location chosen is mostly random with a few rules. The player can only be placed above a wall tile or a platform tile. The player should not be placed on top of a coin or turret tile. Other than those simple rules, there was no need to have any more complex decision making for player placement. This is because the game is a non-linear platform game and because all areas of the
game-world are accessible as a result of the previous procedural generation phases. Figure 26 displays the game-world after all generation phases are complete.

![Figure 26](image)

Figure 26. Choose random position for player and complete the generation

### 5.3.4 Generation Parameters

The random generation algorithm has parameters that influence the game-world that is generated. Each parameter serves a different purpose. Figure 27 shows the interface that is used to begin the procedural generation process. The purpose of each these parameters is described below.

**Chunk Size**  The size of each chunk in the maze generation algorithm. By default this value is always eight.

**Width in Chunks**  The number of chunk columns

**Height in Chunks**  The number of chunk rows
Difficulty The difficulty setting for the random generation algorithm. The valid numbers are 1 through 10 where 1 is the easiest and 10 is the hardest. The value 1 should generate a very easy game-world while a 10 should generate a game-world that is almost impossible to win.

Algorithm The algorithm value will always be 3. Values 1 and 2 are the algorithms that were unsuccessful which is described in section 5.3.1.

Seed The seed for the algorithm. The seed can be used to produce the same game-world over and over again. Leaving this field blank will cause the seed to be a pseudo-random value computed from the operating system clock.
CHAPTER 6

Results

6.1 Summary

A total of 14 subjects participated in the study. Each participant played a different number of game-worlds (some subjects played only three game-worlds while others played more than 30 game-worlds), for a total of around 170 game-worlds played. The pie chart in Figure 28 shows how many game-worlds were played by each subject. The numbers in the pie chart represent the ID number of the participant. The analysis of the data involves creating contingency tables with the various fields of the data. For example, looking at the relationship between the reported enjoyment variable vs. the dimensions of the game-world or any other field would give insight on to what influenced subject enjoyment responses the most.

6.2 Game Enjoyment

One of the objectives of the research is to determine how effective the procedural generation algorithm was at creating game-worlds that would be enjoyable to players. In this analysis, I compared the subjects’ responses to the questions shown in Figure 12. I did this by comparing various objective components of the data with the subject’s responses about game-world enjoyment. Figure 29 shows that players enjoyed game-worlds when the difficulty was reasonable. In Figure 29 it is obvious that most subjects didn’t enjoy a game-world that they perceived as impossible to complete. This suggests that players do not enjoy playing a game-world where they feel they have no chance of winning.

I compared the enjoyment variable to the relative difficulty. This is the question that asked the participant if they thought the game-world was too easy, just
Figure 28. The distribution of game-worlds played for the players.

right, or too hard. Figure 30 illustrates similar results to Figure 29. Players enjoy games that are not too easy and not too hard. They prefer the difficulty to be “just right.”

Another variable that enjoyment was compared to was the chunk dimension variable. The chunk dimensions is a parameter to the random generation algorithm of the game-worlds. This parameter is described in more detail in Chapter 5 Section 5.3.4. The mosaic chart shown in Figure 31 presents the results of this
Figure 29. The relationship to the player perceived difficulty and enjoyment responses

The chunk dimensions appeared to have no effect on the enjoyment responses from the study.

The intended difficulty parameter is different from the difficulty parameter. The intended difficulty is not a subject response: it is a parameter to the random generation algorithm which is described in more detail in the previous chapter 5 section 5.3.4. I wanted to know how this parameter affects how enjoyable the
Figure 30. The relationship to the player perceived difficulty and enjoyment responses.

Players perceived the game-worlds to be. It turns out that participants enjoyed the game-worlds generated with parameters less than 9. When the intended difficulty was maxed out at 10, the game-worlds were not enjoyable.
Figure 31. The relationship between the enjoyment responses and the chunk dimension parameter of the random generation algorithm

6.3 Game-world Difficulty

The other objective of this research was to determine the consistency of the game-world generation algorithms. The expectation is that an easy game-world is easy for players and a hard game-world is hard for players. The most obvious comparison is comparing the difficulty parameter of the generation algorithm to the player response for the game difficulty. The Figure 32 shows that the algorithm
did a fairly decent job in making a game-world easy or hard. The mosaic shows that game-worlds that were intended to be easy were considered to be easy based on player responses. The same concept holds true for very hard game-worlds.

![Difficult vs Intended Difficulty](image)

Figure 32. The relationship between the difficulty response of the participants and the intended difficulty set by the random generation algorithm

A subject’s responses are not the only way to determine how difficult the game actually was for them. The overall performance of the player for that game-world is an effective metric for determining the difficulty. For instance, if the player
lost and failed to collect any coins, then they performed poorly. If they won the game-world then the player performed well. It is expected that players perform poorly in game-worlds that are intended to be hard or impossible. Likewise, it is expected that most players win when the game-world is intended to be easy.

6.3.1 Coins Collected

The objective of the game is to collect all the coins; therefore, the percentage of the coins they collected is an effective metric for performance. The histogram shown in Figure 33 shows that players that collected most of the coins when the game-world was easy and collected hardly any coins when the game-world was impossible.

![Coins Collected Percent vs Difficulty](image)

Figure 33. The relationship between the difficulty and the percentage of coins collected from 0 to 1

6.3.2 Win or Lose

The most obvious way to measure the performance is whether or not the player won. It is expected that players often win when the game-world is easy and often
lose when the game-world is hard. The Figure 34 shows that players often won the easy game-worlds and almost always lost the impossible game-worlds. The amount of wins goes down as the game-worlds intended difficulty becomes harder.

Figure 34. The relationship between the intended difficulty of the game-world vs whether or not the player won or lost
CHAPTER 7

Conclusion

7.1 Lessons from the User Study

One of the biggest lessons from the user study was that the enjoyment of the game was highly influenced by the difficulty. It is important to get the difficulty setting to be appropriate for players. Creating game-worlds that are too easy will not challenge players making it less enjoyable. Making game-worlds that are way too hard is even worse than too easy. Players overwhelmingly considered impossible games to not be enjoyable.

7.2 Procedural Generation Algorithm Conclusion

What worked poorly

The biggest influence on the game difficulty and enjoyment turned out to be the turrets. For this reason, the logic used to decide where turrets should be placed should have been more intelligent. The way it was coded was that turret could be placed in an empty square adjacent to any random wall tile. There was no other logic to it. This often resulted in a large number of turret to be all clustered together, which often caused the player to be bombarded and lose very quickly, making the game not enjoyable. This problem is shown in Figure 35. This problem became more apparent when the difficulty was high, because the number of turrets that were placed in the game-world was determined by the initial difficulty of the game-world.

What worked well

The maze generation algorithm worked well because it was a way to define the overall structure of the game-world without deciding the type of each individual tile. Mazes have the property of having each area accessible as well as having
7.3 Future Work

7.3.1 More Obstacles

The game was very simplistic in that there were only a few obstacles the player had to avoid. This limited the diversity that could be created for the game-worlds. If I had more time, I would have implemented other types of obstacles in the game, such as elevators, tunnels, teleporters, and enemies. Adding these features would have allowed the random generation algorithm, especially the tile placement phase, to have more options on generating the game-worlds.
7.3.2 More intelligent turret placement

The tile placement algorithm should have been more intelligent about the placement of turrets. If I had more time I would have come up with decisions on turret placement that was based on more than just randomness. This would be a way to avoid the large clustering of turrets that appeared in higher difficulties.

7.3.3 Special Powerups for Players

If I had more time, I would have added to the game powerups that affect the player’s mobility. This would affect the procedural generation algorithm, because I could have made certain areas of the game-worlds only accessible when the player obtains the necessary powerup. Figure 36 shows an example of a maze that generated by the maze generation algorithm.

![Figure 36. A maze generated where some areas are colored different to imply them as areas that are only accessible by collecting powerups](image)
In Figure 36 above, some areas are colored differently to represent areas that are only accessible after obtaining a specific powerup: Green area indicate areas only accessible if the player has a high-jump powerup, blue area are only accessible if the player has crawling ability, and the yellow areas are only accessible if the player has both the high-jump and crawling powerups. This is an example of how powerups could be used to generate more interesting game-worlds, and ultimately create a more engaging and enjoyable experience for players.
APPENDIX A

IRB Proposal

A.1 Project summary

The proposed research is an investigation into the effectiveness of a procedural generation algorithm for a non-linear 2D platformer video game. The study will determine how effective the algorithm is based on two criteria. The first criterion is how enjoyable the game world was, which is determined by asking the user to answer a survey question at the end of each game world. The second criterion is how effective the algorithm is at generating game worlds at the intended difficulty level. The game worlds that players will play through will be generated for a specific difficulty value. The player does not know this value. After the game world is completed the user will be asked how difficult it was. The player did when they completed the level will be recorded. This includes time taken by the player to complete the game, whether or not they won, or how close they were to losing. Participants will be recruited mostly from classes taught by the Department of Computer Science & Statistics. Graduate students and faculty or staff in the department will also be approached to participate. The subject will sit down at a computer to play the game. The subject will then begin the game and will be given a 30-second tutorial to familiarize themselves with the controls and the rules of the game. During the experiment, the player will play through a series of randomly generated game worlds. Additionally, game statistics on how the performed is collected.
FWA: 00003132
IRB: 00000599
DATE: March 22, 2021

TO: Jean-Yves Hervé, PHD
FROM: University of Rhode Island IRB

STUDY TITLE: Player Evaluation of Platform Game Generation Algorithms
IRB REFERENCE #: 1716764-1
LOCAL REFERENCE #: IRB2021-148
SUBMISSION TYPE: New Project

ACTION: DETERMINATION OF EXEMPT STATUS
EFFECTIVE DATE: March 21, 2021

REVIEW CATEGORY: Exempt 2i

Thank you for your submission of materials for this research study. The University of Rhode Island IRB has determined this project falls into the EXEMPT REVIEW category according to federal regulations 45 CFR 46. Per URI IRB policy, the project has been reviewed by either the IRB Chair or the IRB Administrator. Approval is valid for the duration of the project.

No changes to procedures involving human subjects may be made without prior IRB review and approval. You must promptly notify the Office of Research Integrity of any problems that occur during the course of your work using Appendix S - Event Reporting.

If you have any general questions, please contact us by email at researchintegrity@etal.uri.edu. For study related questions, please contact us via project mail through IRBNet. Please include your study title and reference number in all correspondence with this office.

Matthew Delmonico, Ph.D., MPH
IRB Chair
APPENDIX B

Glossary

**Game-World**  An environment that the player plays through.

**Chunk**  An N\times N region of tiles that make up the grid-based game-world. A chunk is always square and N is usually between 6 and 10.

**DAT file**  A file on the disk that represents a single data instance in the user-study. One file represents the game data from single participant playing the game.

**Tile**  A single tile in the grid-based game-world.
BIBLIOGRAPHY


