PROCEDURAL GENERATION OF METROIDVANIA STYLE LEVELS

by

Trevor Stalnaker

2020

TABLE OF CONTENTS

LIST OF TABLES . . . . **vi**
LIST OF FIGURES . . . . **vii**
ABSTRACT . . . . **ix**

Chapter

1 INTRODUCTION **1**

1.1 Motivation 2

2 BACKGROUND **3**

2.1 Metroidvanias 3
2.2 Gating Taxonomy 3
2.3 Graphs 4
2.4 Pygame 5
2.5 Physics Engines 5
2.6 In-House Graphics Package 5
2.7 Other Used Files 6
2.8 Related Works 6

3 APPROACH **10**

3.1 Overview 10
3.2 Model 11

3.2.1 Modeling the Map 11
3.2.2 Modeling the Key Orderings 12

3.3 Implementation 12

3.3.1 Representing Keys and Gates 12

3.3.2 Preprocessing 13

3.3.2.1 Generating an Ordering for the Keys 13
3.3.2.2 Creating Directional Mappings for Gate Types . . . 14

3.3.3 Generating the Underlying Map 16

3.3.3.1 Creating the Lattice 16
3.3.3.2 Verifying that the Map Configuration is Winnable . . 20
3.3.3.3 Placing the Keys 21

3.3.4 Rendering the Map into a Playable Form 23

3.3.4.1 Keys 25
3.3.4.2 Gates 26
3.3.4.3 Barriers, Walls, and Platforms 27
3.3.4.4 The Player 29
3.3.4.5 Creating the Physical Representation of the Map . . 33

3.3.5 Mini Map 39
3.3.6 Drawing, Updating, and Event Handling 40
3.3.7 Other Visualizations 41
3.3.8 Summary of Implementation 42

3.4 Saving and Loading Maps and Templates 43

3.4.1 Pickling and Loading 43
3.4.2 The User Interface 44
3.4.3 Supporting Past Versions 45

3.5 Playing the Generated Level 45

3.5.1 Controls 45
3.5.2 Collecting Keys and Passing Through Gates 46

3.6 Plotting the Underlying Graphing of Maps 47

4 ANALYSIS 48

4.1 The Ability to Win Generated Levels 48
4.2 Enforcing Gate Ordering 48
4.3 Playability of Generated Maps 49
4.4 Level Generation Times 49

4.5 Example Map Generations 52

5 CONCLUSION 58

6 FUTURE WORK 59

6.1 Improved Generation of Maps 59

6.1.1 More Intelligent Generation 59
6.1.2 Improved Dropout Rates 60

6.2 Adding Other Gating Types 60
6.3 Increase Non-Linearity of Key Collection 61
6.4 Improve the Mini Map 61
6.5 Improve the Graphics of the Game 62
6.6 Increase Variability 62
6.7 Human Level Designer Access 63
6.8 User Interface for Designers 63
6.9 User Studies 63

BIBLIOGRAPHY 65

LIST OF TABLES

3.1 Formulae Determining Location of Nodes in the Lattice 19

4.1 Generation Data by Level Size and Key Count 51

LIST OF FIGURES

2.1 Representation of the Gating Taxonomy 4

3.1 Flowchart of Algorithm 11

3.2 Example Key Orderings 14

3.3 Example User Input to Gate Ordering 15

3.4 Abstraction of a Map Lattice 17

3.5 Possible Positions for Node i in the Grid 18

3.6 Finding the Current Explorable Region 22

3.7 Zone for Proper Key Placement 24

3.8 An Example of an Unwinnable Pit in the Map 24

3.9 Anatomy of Gate Types 30

3.10 Example of Corner Generated without Overlap 31

3.11 Anatomy of a Platform 31

3.12 Player's Finite State Machine 32

3.13 Horizontal Movement Physics 34

3.14 Vertical Movement Physics 35

3.15 Plotting Interior Walls on the Map 37

3.16 Plotting Interior Walls on the Map 38

3.17 The Save Interface 45

4.1 Undirected 4x4 Level Demo View and Underlying Graph 52

4.2 Undirected 4x4 Level Demo View and Underlying Graph 53

4.3 Undirected 6x6 Level Demo View and Underlying Graph 53

4.4 Early Rendering of a 4x6 Game Map and Associated Level Demo View 54

4.5 Directed 6x6 Level Demo 55

4.6 Rendered Game View and Underlying Graph 56

4.7 Generated Game View, Level View, and Underlying Graph (5x5) . 56

4.8 Generated Game View, Level View, and Underlying Graph (8x8) . 57

4.9 Generated Game View, Level View, and Underlying Graph (8x12) . 57

ABSTRACT

Video game maps can become dull with repeated play-throughs and hand-crafting a variety of maps can be a tedious and time consuming process. This is especially true for games of the Metroidvania genre, games which focus on exploration. If there was a way to adequately automate the creation of levels, then in theory, the games would have enhanced replay value. Previous researchers have used artificial intelligence and genetic programming techniques to engineer new mappings. But, is it possible to procedurally generate levels using graph theory and without using training examples or simply placing pre-built assets? In this paper we propose a system to model Metroidvania maps as directional graph structures. The system uses an algorithm that crafts graphs meeting all of the constraints necessary for level generation. These generated graphs are verified as winnable with the keys assigned to appropriate nodes. Once the graph has been created and validated it is rendered into a 2-D level using pygame. During the rendering process, the game demo constructs the walls and platforms essential to the game. We were able to procedurally generate Metroidvania levels of varying sizes and gating techniques using this sequence of steps.

Chapter 1
INTRODUCTION

Metroidvanias, named after the classic franchises Metroid and Castlevania, have been a staple of gaming since the late eighties and early nineties [6]. A game in the Metroidvania genre typically provides the player with a non-linear gaming experience. Access to parts of the world are "gated," meaning that access to these areas is not immediately available. Various different gating technologies are collected by the player throughout the course of their play, each allowing them to access and explore more and more new regions of the map. The order in which these gating technologies can be collected by the player is called the gating order or the gate ordering. As a game designer, creating worlds such as these can be a tedious and time consuming process. If the process could be automated, then the number of levels these games contain could be near infinite. A game could provide players with new levels on each play through, preventing the game from becoming stale. Gutierrez-Rodriguez et al. make use of artificial intelligence and genetic algorithms to create novel levels [1, 9], however their work has not been verified through user studies. In this paper, we discuss how we used principles of procedural generation to create new maps. Procedural generation is not a new innovation in the field of gaming. Hendrikx et al. describe the various facets that procedurally designed content can fall into, including textures, sounds, vegetation, maps, environments, puzzles, stories, and leaderboards [3]. None of the previous research has applied these techniques to the creation of gated-style maps required of Metroidvania games. So, while the ideas and principles of procedural generation are nothing new, applying them to create Metroidvania levels is a novel approach.

Contributions made in this thesis:

1. created a gating type taxonomy for Metroidvanias based upon *Castlevania: Symphony of the Night*,
2. devised a system to represent Metroidvania gating orders as directed acyclic graphs,
3. designed and implemented an algorithm for procedurally generating Metroidvania levels as directed planar graphs,
4. created two systems to render the underlying level graph into a playable game representation,
5. evaluated the effectiveness of and the time it takes to procedurally generate Metroidvania-style levels,
6. provided a github repository with the code base necessary for designers to generate their own Metroidvania-style levels.

1.1 Motivation

The creation of levels and maps for video games can be a time consuming and arduous process. This is especially true for games in the Metroidvania genre. These games are known for their large maps and exploration based game play. As map sizes increase, it can also be increasingly difficult to verify that there are no ways for the player to cheat the gating order. Therefore, if a machine were able to procedurally generate maps, it could dramatically decrease the time and effort needed to create and design such a game. Even if the generated maps were edited and improved by a human designer, it is easier to improve on something functional, than to start completely from scratch. For this reason, we developed a program that is capable of Metroidvania-style level generation. As discussed in Section 3.3, the designer has access to many parameters which they can use to tailor the map design to their specific needs. Additionally, a map checker was also created as part of the generation process, which could easily be repurposed to verify that purely human designed maps meet the exploration criteria set by the designer.

Chapter 2

BACKGROUND

2.1 Metroidvanias

Metroidvanias are a genre of video game with a focus on exploration. The genre is named after the two flagships of the genre: Metroid and Castlevania [6]. Typically these games incorporate a platforming element and various different gating techniques. Gating, as a term in Metroidvania, includes but is not limited to literal gates and keys. Sometimes a "gate" can be a ledge too high to reach and the key a grappling hook. In order to progress through the map, the player must collect a series of keys or movement technologies, each of which will allow for further exploration. The player continues to explore and navigate through the maze-like world until they reach the final goal (usually a boss enemy). Gutierrez Rodriguez et al. aptly summarize Metroidvanias as games that:

> "Feature a large interconnected map through which the player can move, having to obtain objects, weapons or abilities to unlock the different locked areas. The map is composed of different areas, each of which is in turn composed of different rooms (including secret rooms). Rooms are where the different enemies, objects, new abilities, are placed [9]."

2.2 Gating Taxonomy

The popular Metroidvania *Castlevania: Symphony of the Night* was chosen for analysis based on its standing as one of the two video games that founded the genre [4]. From this analysis we devised a gating taxonomy. This hierarchy groups gates into categories, thus making conceptualizing them for procedural generation easier. Categories include movement tech, doors, enemies, transport, transformations, and puzzles. While other categories undoubtedly exist, these seem to make up the vast majority. Additionally, some gating techniques could reasonably fall under multiple categories, making some sort of a hybrid gate. Double jump with a keyed gate or a puzzle using enemies would be good examples of this. For simplicity, these combinations are not included in the taxonomy. See Figure 2.1 for the full taxonomy diagram. In this project, we focus on the movement, doors, and transformation categories, however the other categories could also be implemented given adequate time and planning.

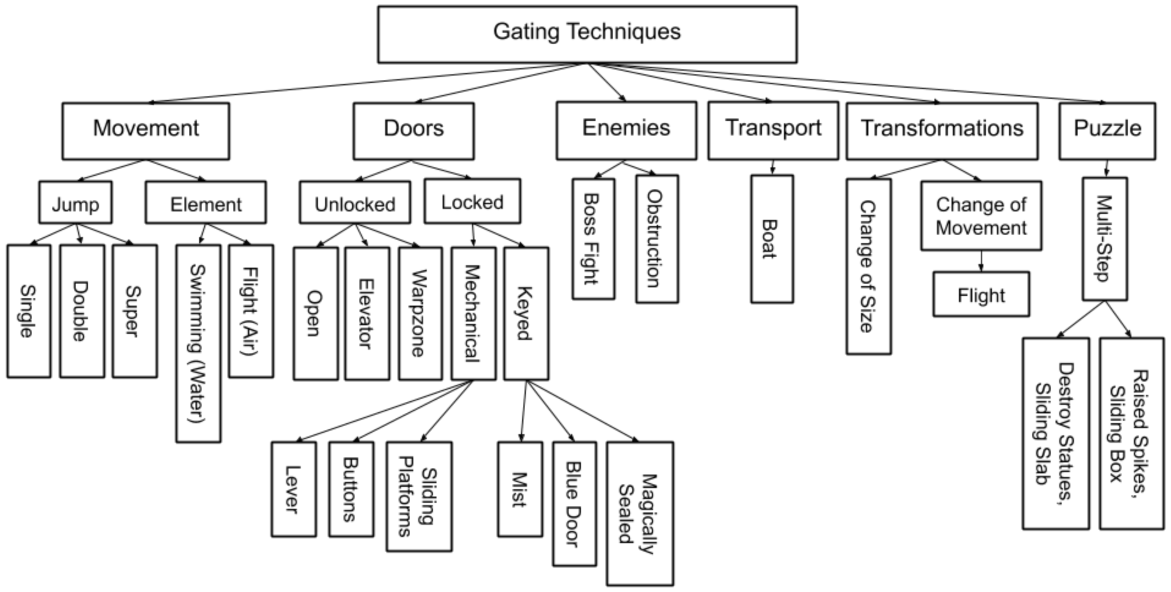


Figure 2.1: Representation of the Gating Taxonomy

2.3 Graphs

Graphs are data structures made up of nodes and edges. Each node can represent a data point and each edge a connection between points. Nodes can be numbered or named and edges can have weights. Graphs also come in a variety of forms. In undirected graphs, the edges have no direction. This means that if there is a connection between node A and node B, then there is also a connection between B and A. In a directed graph, this isn't necessarily the case. Each edge in a directed graph has a direction as well as a weight. Thus A could lead to B, but B might not lead back to A.

Planar graphs are also key to the implementation of this project. A planar graph is a graph that can be visualized in a 2-dimensional plane without any of its edges crossing. Finally, directed acyclic graphs (DAGS), are directed graphs that contain no cycles. In a DAG, sources are nodes that have no incoming edges and sinks nodes that have no outgoing edges. This means they are analogous to the root node and leaf nodes of a tree respectively. To model the graphs in this project, we make use of the networkx Python package [2].

2.4 Pygame

Pygame is a Python package used primarily for video game development. The interactive game demos of this project are built using pygame. In a nutshell, pygame allows a programmer to draw a game to the screen, handle game events, and update the game accordingly [8].

2.5 Physics Engines

Physics engines establish the rules for how objects in a virtual world behave. Forces such as gravity, friction, and velocity are all simulated by the physics engine. The physics for this project are defined within the avatar class. Normally a physics engine would be more generalized, allowing for all objects to be affected by the forces defined. However, since the player is the only object in the demo that needs to be affected by gravity and the like, the physics engine is built into that class. More information on how this is accomplished can be found in Section 3.3.4.4.

2.6 In-House Graphics Package

The user interfaces for saving and loading maps were built using a previously created graphics package. Justin Pusztay, my partner at the time, and I created this graphics package while working on our game Squirrel Simulator [10]. This package builds upon pygame's functionality, allowing game designers to more easily create UI elements. Buttons, scroll boxes, text fields, and more are defined in the package. Minor modifications were required to some of the files for this project, further improving their functionality.

2.7 Other Used Files

In addition to the graphics package, a number of other files used in this project originated from Squirrel Simulator [10]. The first of these is Drawable. This class provides a framework that streamlines the process of interacting with pygame objects. It provides methods for drawing sprites, handling events, updating sprites, and much much more. Another example is the FSM class, which is used to model finite statement machines.

2.8 Related Works

Other scholars have also worked towards automating the generation of levels in the Metroidvania genre. Typically, these other projects aim to automate the creation of more than just the underlying map structure of a level. That is to say that components such as weapons, tools, or even the story could be generated as well. It is generally accepted that video games with few or static level designs can become monotonous on future play-throughs [1]. Additionally, creating more levels to solve this issue proves challenging for a development team. Creating more content can be time consuming, tedious, and expensive [1]. Using procedural generation to produce this new content typically lessens the issue of monotony and is appreciated by players [9]. Overall, the design phase makes or breaks a game [9].

Gutierrez-Rodriguez et al. worked to automate the video game design process using deep evolutionary training [1]. This project was a continuation of their past work and sought to create an AI-assisted design tool for developers capable of designing full games. This tool was specifically designed to aid in the creation of Metroidvania levels [1]. They used evolutionary algorithms to generate the in-game content and neural networks to simulate a designer's thought processes. In the end, they were able to generate 'acceptable' results [1]. However, there is no mention of user studies comparing the generated levels with those of human design.

In another project, Gutierrez-Rodriguez et al, once again furthered their work in the field of video game design automation. In this study, they sought to create a tool capable of producing a complete game, with the exception of art and sound [9]. This would involve an autonomous designer creating mechanics, game rules, game elements, Non Player Character behaviors, and levels [9]. A very ambitious project. The authors discuss some of the difficulties involved in procedural generation. For example, there could be infinitely many designs for the player sprite, but which of those designs best encompasses the character of the main protagonist [9]? Human artists are readily able to craft characters based off of their traits and abilities. Evil characters could be shrouded in black, while heroic characters wear pristine white for example. They then explain how this issue can be expanded to the fuller game design. What sort of experience should the game bring to the player? How do the mechanics, rules, etc effect or bring about that experience [9]? To tackle these issues, they used a combination of neural networks and evolutionary algorithms [9]. The evolutionary algorithms simulated the different ideas that a development team might create during brainstorming [9]. They determined that their resulting proof of concept was capable of creating quality levels, but was also able to provide promising hints to designers [9].

Sentient Sketchbook is a project geared towards map design. The created tool is capable of determining a map's play-ability and balance and of providing suggestions for alternative mappings [5]. This tool was used to generate levels for strategy games, like Starcraft, rather than Metroidvanias [5]. Most strategy games consist of a board-like grid of tiles, which the player and NPCs can navigate. Sentient Sketchbook can be used to design low-resolution abstractions of these tiled game maps [5]. Designers are also able to interact with Sentient Sketchbook through a graphical user interface [5]. The tool was tested on industry experts, and proved to provide useful suggestions and feedback to the designers [5].

Players expect more and more out of the games that they play. As Hendrikx et al. put it, “the most popular commercial games get larger, prettier, more atmospheric, and more detailed with each generation” [3]. Procedurally generating content can be used more broadly to help meet these demands. There are six types of game content that can be procedurally generated [3]. These include game bits, game space, game systems, game scenarios, game design, and derived content [3]. Game bits make up the fundamental units of a game. Textures, sound, vegetation, buildings, and behavior are examples of game bits. The game space is comprised of indoor and outdoor maps. The game systems include the environment, roads, and entity behaviors. Puzzles, stories, and levels make up game design. And finally, derived content consists of news and leaderboards [3]. Needless to say, there are different methods and techniques for generating content from the various categories. These include artificial intelligence, genetic programming, and algorithmic approaches.

There are four distinct approaches to game space generation. These include designer-created, random, player-created, and procedural spaces [7]. Very few games however make use of procedural space generation [7]. The main obstacle in procedurally generating spaces for game play isn’t making the content itself but making the content interactive. Games like MojoWorld generate fractal based planets with great detail, but the worlds are devoid of active NPCs and interactive objects [7]. Nitsche et al. decided to bridge the gap with their Charbitat prototype. Charbitat is capable of generating worlds based on the playing styles of its users. That is that the player shapes and changes the game world as they play [7]. The crux of procedurally generating worlds is that the game space might not make sense. The only limits on the procedurally generated worlds are provided in their rule sets. For example, in Minecraft’s newest April Fool’s update, they allowed the player to travel to a near infinite number of procedurally generated dimensions. Some of these were well-formed and interesting, while in others the ground was made entirely of flowerpots. The problem with infinity is that you get everything. This is the main reason some would contend that human-designed worlds are superior. But it has been shown time and time again that procedural generation can be used, often times in conjunction with human designers, to enhance or augment game play.

Chapter 3
APPROACH

3.1 Overview

Generating fully playable, winnable Metroidvania-style levels is the ultimate goal of this project. Generally a level can be considered as one of many areas or maps that comprise a larger, complete game. Metroidvanias are often games that are made up of one or just a handful of levels. Accordingly, the games generated by this process, consist of a single level. Figure 3.1 provides a visualization of the level creation algorithm. At the start of the process, the designer provides a gating order, a horizontal mapping, a vertical mapping, the number of columns n, the number of rows m, the player's start position, the end goal position, and a weighted neutral. All of these inputs are described in more detail in Section 3.3. A DAG is created from the designer's provided ordering input, and a topological ordering of the DAG is generated (Section 3.3.2.1). This manufactures a linear gating order from the designers specifications, which the generated level will have to obey. The mapping inputs are standardized, a process that simply adds convenience for the designer. See Section 3.3.2.2. All of the inputs are then used to produce a graph representing a potential level, as described in Section 3.3.3.1. This graph is verified for winnability, according to the criteria of Metroidvanias and the gating order (Section 3.3.3.2). If the graph does not represent a winnable configuration, then another potential graph is produced using the same inputs. Otherwise, the keys from the linear gate ordering are assigned positions in the graph, with respect to that ordering. This process is described more fully in 3.3.3.3. The final graph representation, the specified room dimensions, and a player object are all used to render the underlying graph into a finalized playable level.

This process is explained in Section 3.3.4. The end result is a playable, winnable game that meets all of the criteria specified by the designer at the start of the algorithm.

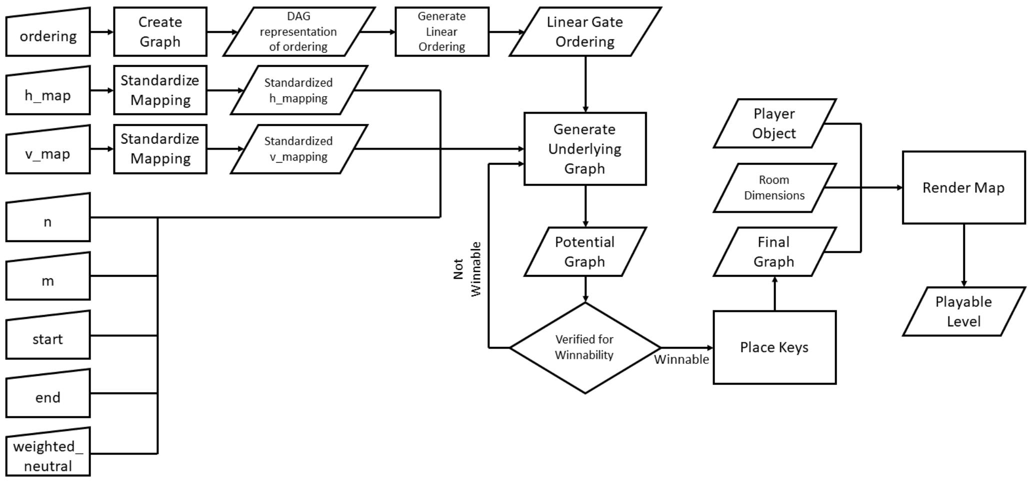


Figure 3.1: Flowchart of Algorithm

3.2 Model

Before unique level mappings can be procedurally generated, we need to have representations that model the map and key orderings.

3.2.1 Modeling the Map

A level map can be represented as a directed graph of nodes and edges. Each node represents a room within the map and each edge represents a gated connection. The weights or labels across these edges signify the particular gating technique required to cross that connection. More specifically, wherever an edge between two nodes exists, there is also one in the opposite direction. These two edges, connecting the same two nodes, most often have the same label or weight, but are not required to have such a pattern. The edge in one direction can be completely different from the one in the opposite direction. The graph model is also planar; there exists a 2-Dimensional representation of the graph structure in which no edges overlap. To create a physically possible map, planarization of the graph is required. Each node in the graph can be connected to no more than 4 nodes and to no less than 1 node, ensuring that every node is reachable. The 4 node maximum constraint mirrors the structure of a room in a typical Metrodivania using 2D graphics. There is an up, down, left, and right. The resulting graph model resembles a lattice of nodes and edges. Figure 3.4 shows an example of a Map Lattice that could be used. The process through which this structure is constructed is described in Section 3.3.3.1 below.

3.2.2 Modeling the Key Orderings

The ordering in which keys can be found is also represented as a graph, more aptly a directed acyclic graph (DAG). The single source of this DAG, the node with no incoming edges, is the first gating technique that the player will have access to. In most cases, this first gating technique is the neutral connection, but the designer can change this as they please. The children of a node represent gating techniques that are reachable once the parent node has been attained. For example, if single jump were a parent node, then double jump could be one of its children nodes. This creates a rigid system that enforces the progression of the gating techniques. This prevents situations where you collect single jump **after** you already had access to double jump. The sinks of the DAG, the nodes with no out going edges, represent all of the potential ending gating techniques. The process through which this model is created and how a gate ordering is generated from it are described more fully in Section 3.3.2.1.

3.3 Implementation

This section will cover in detail the implementation and design responsible for the generation of our Metroidvania style levels.

3.3.1 Representing Keys and Gates

When providing information to the level map generator, keys and gates are synonymous. There is no need to make a distinction between the two at this point, since it follows logically that the red key opens the red gate and the double_jump key allows the player to double jump. These gating techniques, as they will be referred to from this point forward, are represented by strings. For more information on how this gating technique information is input into the generator, see Section 3.3.2.1.

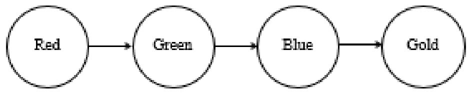
3.3.2 Preprocessing

A short pre-processing step allows designers to provide inputs before levels are successfully generated. That is that the designer provides information on the desired gate ordering.

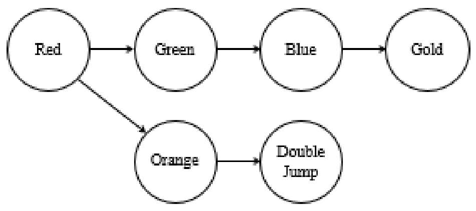
3.3.2.1 Generating an Ordering for the Keys

As was described in Section 3.2.2, the designer can provide a DAG structure to convey all of the viable orderings for their map. Such a DAG could be skinny with only one valid ordering, or it could be bushy, with many many valid orderings. See Figure 3.2 for examples of potential orderings. The typical user or designer cannot be expected to create a DAG such as these to input into the generator, so instead they provide the generator with a dictionary, where keys represent a gating technique and values represent all the new keys that are viable after said key has been found. See Figure 3.3 for an example of just such an input and an example of its output generation. If a particular gating technique can only lead to one other technique, then the designer can simply provide that technique as a string for the value associated with that key. If there is more than one viable next gating technique reachable, the designer can assign a list of such strings to the value.

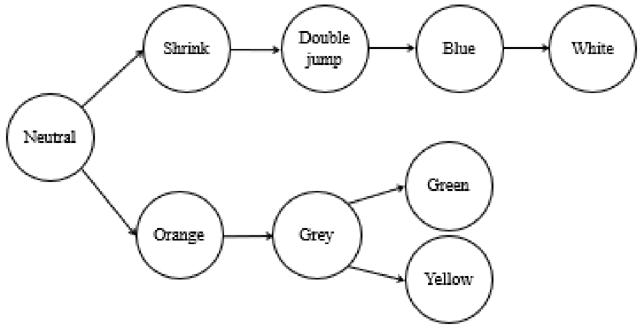
To find the actual linear order of the keys for the particular level, the getGateOrder function in the grapher module is called. This function in turn calls the helper function createGraph, to create a networkx representation of the gating DAG. createGraph is provided with the dictionary created by the designer, called ordering. createGraph iterates through the keys of ordering and adds each gating technique as a node in the graph. Specifically, it adds an edge between the key (parent gating technique) and the value (child gating technique). If the value at a key is a list, then each of the elements in that list is added as described previously. Once this graph is



(a) Simple Linear Ordering



(b) Simple Branched Ordering



(c) More Complex Branched Ordering

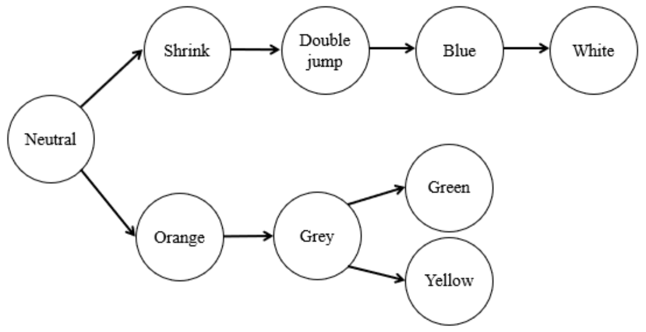
Figure 3.2: Example Key Orderings

constructed, it is returned and control returns back to getGateOrder. With the gating techniques graph now available, getGateOrder finds the one source of the DAG, i.e. the first key that a player should come across (typically the one they start out with by default). Next, while the length of the constructed ordering is less than the number of nodes in the returned gating techniques graph, each node currently in the constructed ordering is looped through and all of its children are saved as potential next keys. One of these potential next keys is selected at random and appended to the end of the constructed ordering. This process continues until all keys have been added. This system provides for variation in the orderings for maps, while obeying the constraints outlined by the designer.

3.3.2.2 Creating Directional Mappings for Gate Types

Along with being able to control the gate ordering, the user can also specify which gates can appear on platforms and which can occur on walls. They can do this by specifying rules for bi-directional gates. The designer can provide a horizontal

```
ordering = {"neutral":["shrink","orange"],"shrink":"double_jump",
            "orange":"grey","double_jump":"blue","blue":"white",
            “grey”:[“green”,”yellow”]}
```



Example Resulting Ordering:
- Neutral
- Shrink
- Double Jump
- Orange
- Grey
- Blue
- Green
- White
- Yellow

Figure 3.3: Example User Input to Gate Ordering

mapping and a vertical mapping. Both of these are lists, which contain strings, which once again represent the various gating techniques. All gates that can be present in walls should be within the horizontal mapping, all of those that can be present in platforms on the vertical mapping. Some gates are yet more complicated. Take for example double jump. The player should be required to have double jump in order to pass up through a double jump gate. But what if they fall off from the top of the platform? The player shouldn't be required to have double jump in order to fall. Such a constraint would seem contrived and inhibit game play. By providing a tuple of two strings, instead of just a single string, we can convey to the generator a setup that allows for different gatings in different directions. ‘(“double_jump”,“neutral”)’ for example would mean that you need to have the double jump movement tech in order to proceed upwards through the gate, but only need the neutral tech to pass downwards. Although designed for this specific case, the functionality is generalized such that any combination of bi-directional gatings is possible (in some cases additional steps may need to be taken in order for the gates to be properly rendered and playable, see Section 3.3.4.3). As you can probably imagine, having some elements of the list as strings and others as tuples could present issues down the line. Luckily the getDirectionalMapping function in the grapher module cleans up the list such that all items are tuples. That is that each string in the list is converted to a tuple where both elements contained within it are that string. This serves mostly as a convenience function allowing designers to specify ‘[“red”]’ instead of ‘[(“red”,“red”)].’ Also, gating techniques can occur in more than one tuple. Take this for example, ‘[(“red”,“red”),(“red”,“blue”)].’ With this setup, in one configuration a player could pass through the red gate with the red key in both directions. In the other configuration, the player could pass through the gate with the red key, but would need the blue key to return to where they came from. This allows for further variation in the design of maps.

3.3.3 Generating the Underlying Map

Once the pre-processing has been completed and the designer has specified the gating criteria, the underlying structure of the map can be generated.

3.3.3.1 Creating the Lattice

Once the gating order and the directional mappings have been set, the actual graph representing the map level can be created. Parts of this graph are created through random generation, but the underlying structure is created in the form of a lattice. This intelligently designed lattice ensures that the graph is planarizable, no node has more than four connections, and that those connections are its ‘proper’ neighbors. What is considered a proper neighbor can be seen in Figure 3.4. To construct the lattice, we use a list to simulate a stack, appropriately called nodeStack. nodeStack contains the nodes which are yet to be processed. Contrary to popular computer science convention, the first node of the graph is labeled with the number 1, thus 1 is the first thing pushed on the stack. As long as the nodeStack is not empty the following continues to happen. The nodeStack is popped and the popped node is saved to a variable i. Using the logic found in Table 3.1 to find the relative location of i within the overall structure of the lattice, the various connections of i are determined. Examples of the different configurations can be found in Figure 3.5. If a potential connection is between i and a node that hasn’t been processed yet and it isn’t randomly dropped out, it is assigned a gating technique and is added to the growing graph. The newly connected node is also pushed onto the nodeStack, so that it too can be processed.

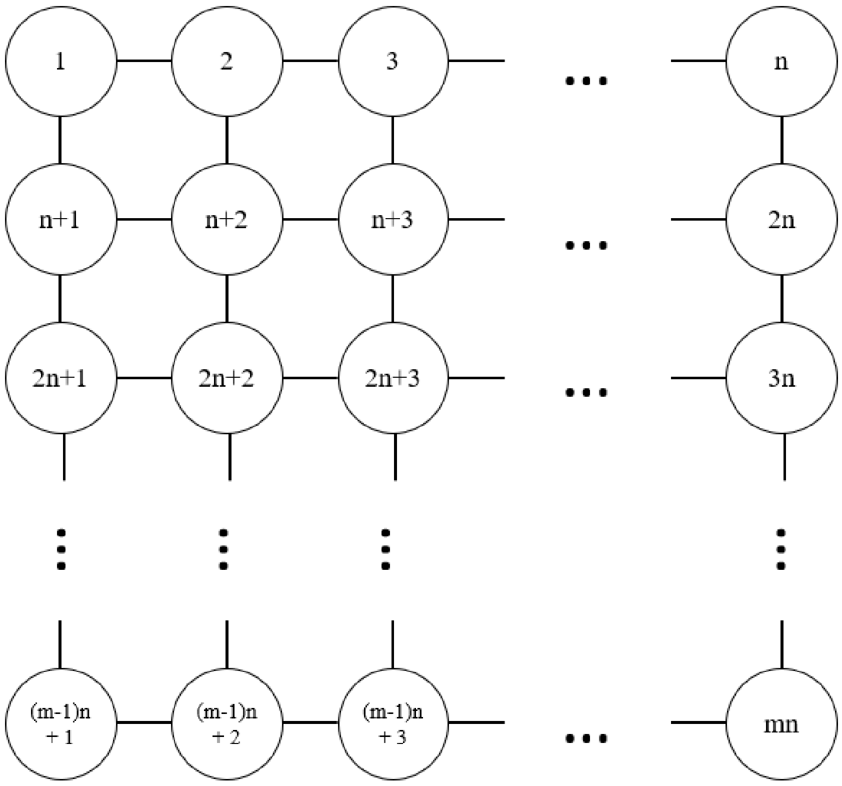
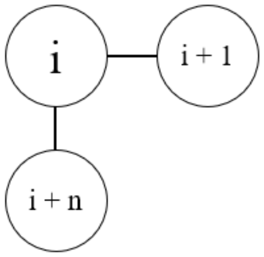
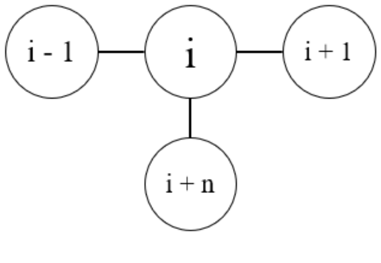


Figure 3.4: Abstraction of a Map Lattice

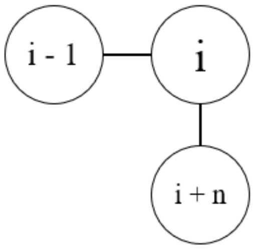
The detConnection function is responsible for the logic behind which connections are added and of what gate variety. The connection being considered is between the node i and another node. If this other node has already been processed, meaning that all of its connections have already been determined, then there is no need to re-evaluate the connection. Therefore no additional connection is added. If the other node has not yet been processed, then the available gates are pulled from the mapping. Next, the connection is randomly assigned one of these available gating techniques. This assignment is not truly random, since the designer can also provide a weighted neutral as a parameter. This weighted neutral affects how many of the nodes are connected via neutral edges. This allows for the map world to be more open and less restrictive. There is no need for all rooms, or nearly all rooms, to be connected by locked doors



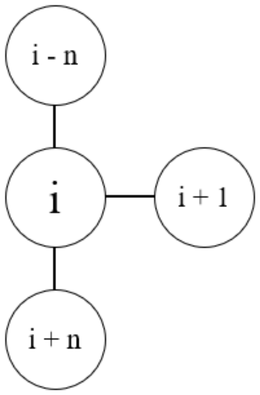
(a) Top Left Corner



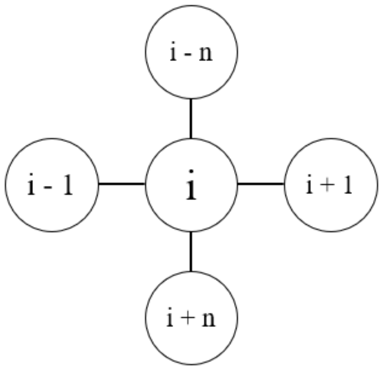
(b) Top Edge



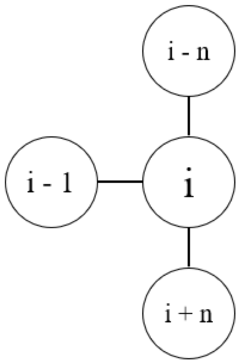
(c) Top Right Corner



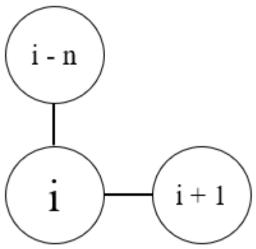
(d) Left Edge



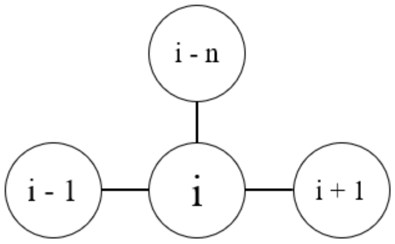
(e) Middle



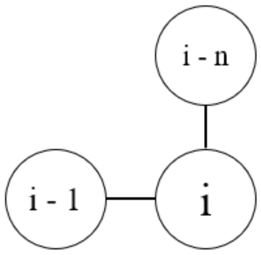
(f) Right Edge



(g) Bottom Left Corner



(h) Bottom Edge



(i) Bottom Right Corner

Figure 3.5: Possible Positions for Node i in the Grid

Position in the Grid	Criteria for Being in that Position
Top Left Corner	i == 1
Top Edge	2 <= i <= (n-1)
Top Right Corner	i == n
Left Edge	i % n == 1 and i != 1 and i != (m-1)n + 1
Middle	i % n != 0 and i % n != 1 and n <i <(m-1)n + 1
Right Edge	i % n ==0 and i != n and i != mn
Bottom Left Corner	i == (m-1)n + 1
Bottom Edge	(m-1)n + 2 <= i <= (m-1)n + (n-1)
Bottom Right Edge	i == mn
Where i is the position in the grid of the node under consideration, m is the number of rows, and n is the number of columns.	

Table 3.1: Formulae Determining Location of Nodes in the Lattice

or double jump platforms. The name weightedNeutral is a bit deceiving however. In reality this value really represents how many of the connections will be of the first gating technique. Typically this first technique is the neutral connection. The random assignment is overridden if either i or the other node is the endNode. For this special case, the connection is set to the final gating technique. This ensures that the end node is not reachable until after the last key has been collected.

Once the type of the connection has been determined, detConnection determines if the connection should exist at all. Not every room should be connected to all of its neighbors. Such a map would be rather bland and uninteresting. A helper function r returns a boolean true or false representing if the connection should be added to the overall graph. This r is simple for the time being, but could potentially be made more complex and interesting in the future. Currently, a random value between 0 and 1 is compared with the dropout rate, which is defined at the top of the file. The designer actually has no control over this dropout rate because slight changes to it can result in slowed runtimes or worse graph generations. If the random value is greater than the dropout, then the connection is added to the graph. Now knowing that the connection should be incorporated into the graph, detConnection considers the various different options for bidirectional graphings. As discussed previously, a single gating tech can have multiple different bidirectional configurations. It is at this point that one of those configurations is randomly selected. Which gating tech will go from left/up to right/down depends on the relationship between i and the other node. Finally, the node that was under consideration is pushed onto the nodeStack, so that it too can be processed eventually.

3.3.3.2 Verifying that the Map Configuration is Winnable

The viableMap function is crucial to the map generation in its current form. Because of the random nature of how connections are created, it is essential that the maps be checked to determine that they actually meet the criteria of a Metroidvania. In Section 6.1.1 we discuss how this process could be improved using a more intelligently designed algorithm, rather than just randomness and checking. This function has further applications outside the realm of procedurally generating maps however. With slight modification, it could potentially also be used to verify that human designed levels conform to the constraints chosen by the designer. This would drastically decrease the time needed to test large levels and rapidly prototype.

First, viableMap checks that the end node, the ultimate goal of the player, is actually contained within the graph representing the map level. If the end node is not in the graph, then logically the player can't reach it, and the map is unwinnable. In this case the function halts and returns False.

Next, the continuing expansion of new regions during exploration is checked. Essentially if the currently explorable region (the region reachable with the current keys) doesn't grow when another key is added, then the level does not meet the gating constraints. That is to say that some key or area would be reachable before it should be. It could be possible to get double jump before triple jump for example. The explorable regions are found using the findExplorable function. In essence, this function iterates through the graph along all edges that have weights that are keys that the player has acquired. See Figure 3.6 for an example of this process.

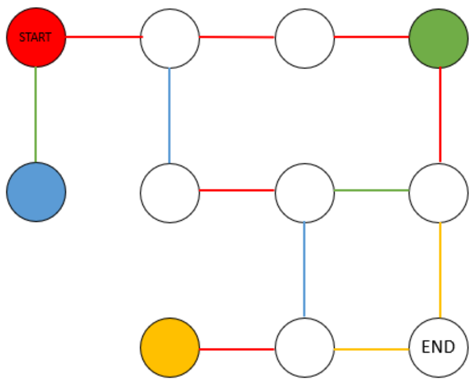
Once viableMap determines that there are enough distinct explorable zones for each key to be set in, the keys are placed. The process through which the keys are placed is described in Section 3.3.3.3.

Lastly, viableMap checks that there is at least one primary connection between the startNode and another node. A primary connection is simply a connection that uses the first gating technique, typically the neutral connection. This ensures that at least one other node is reachable from the start.

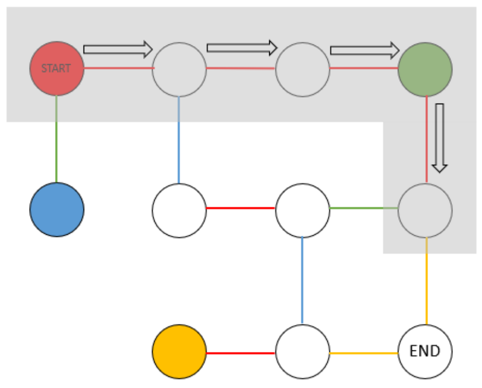
Currently, viable map also creates a potential map, which it then determines the validity of using the above conditions. It does this with a call to the createGraph function described in Section 3.3.3.1. The function generateViableMap, which is called by both of the demo classes, is used to actually create and return a fully fledged viable map. This is done by calling viableMap until a map is returned. This means that many maps are actually created that don't meet the constraints laid down by the designer, but they are ignored, and generateViableMap keeps searching until it finds a suitable mapping. As discussed in the future work section, this system can be improved. This configuration lends itself to a slower runtime and also introduces the halting problem. Given certain parameters the generateViableMap function may never halt. If the constraints given to the function are faulty, there could potentially be no viable mappings and thus the end loop condition will never be met. However, within reasonable limits, this system seems to work well.

3.3.3.3 Placing the Keys

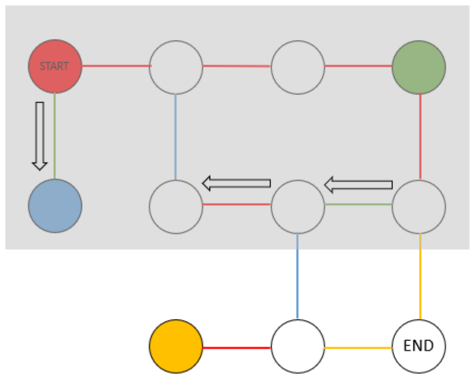
Each new key must be placed such that they can only be acquired one at a time. There are no situations in the levels generated that two keys can be collected interchangeably. As has been mentioned multiple times previous, this enforces the gating order. To find the area in which a key can be placed, we look at the area



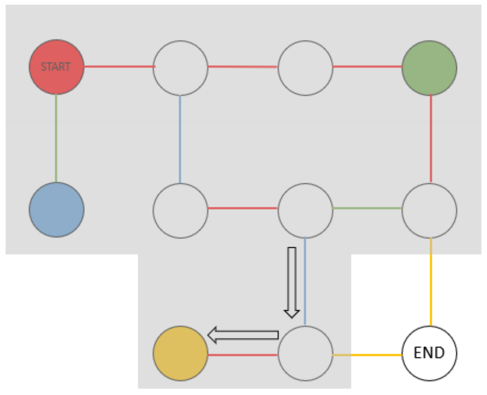
(a) Current Keys: []



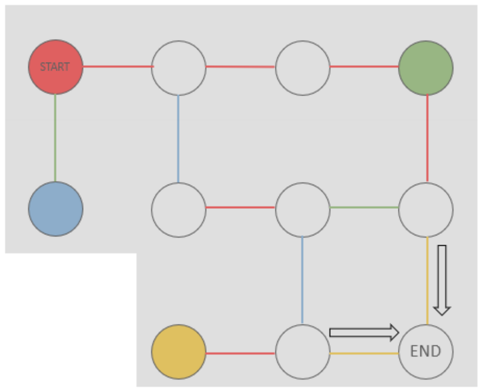
(b) Current Keys: [Red]



(c) Current Keys: [Red, Green]



(d) Current Keys: [Red, Green, Blue]



(e) Keys: [Red, Green, Blue, Orange]

Figure 3.6: Finding the Current Explorable Region

explorable with the current keys the player has. We then compare this to the explorable region with the addition of the next key in the sequence. The difference between these two region sets represents the area in which the newest key can be placed. See Figure 3.7. A node is chosen at random from this difference set and is set as the key location. Now the function must check that the key is reachable from all of the previous nodes. This prevents a situation in which a player could fall into an unwinnable pit, a problem introduced with the addition of bidirectional gating. See Figure 3.8 for an example of this. To do this, we need to check for paths between each node in the difference set and the node that contains the last placed key. If there exists a path between all of these nodes and the key location, then paths must exist for all previous nodes as well. We can prove this transitively. All previous nodes had paths to the last key location. Therefore if the last key location has a path to the new key location, all the previous nodes do as well. Due to complications with the bidirectional nature of the graph and some restraints inherent to networkx, it's best to create a temporary graph to check for paths rather than use the full structure. This is because if we use the dijkstra's pathfinding method for example, a path can technically exist between two nodes even if our constraints say that the player can't pass along that edge yet. So even after applying weights to the edges, if no better path existed, that technically incorrect path was returned. Thus, by creating a graph that only contains the edges that are viable, we can avoid that issue. If at any point a path does not exist, then the viableMap function will return False, and the entire process should repeat. If there is always a path from every node to the key location, the location of that key is officially set, and the process continues for the rest of the keys in the sequence.

3.3.4 Rendering the Map into a Playable Form

After the underlying graph has been created through the process described in Section 3.3.3, a playable level can be created using the pygame package [8].

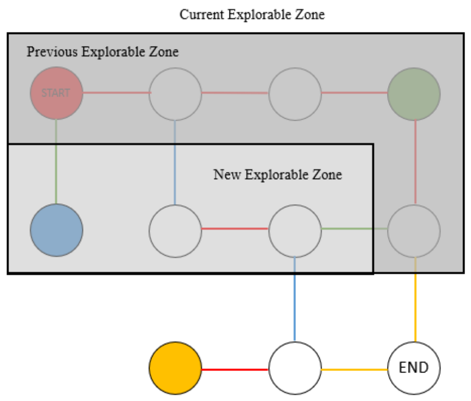


Figure 3.7: Zone for Proper Key Placement
The next key, in this case blue, can be placed in any rooms in the New Explorable Zone

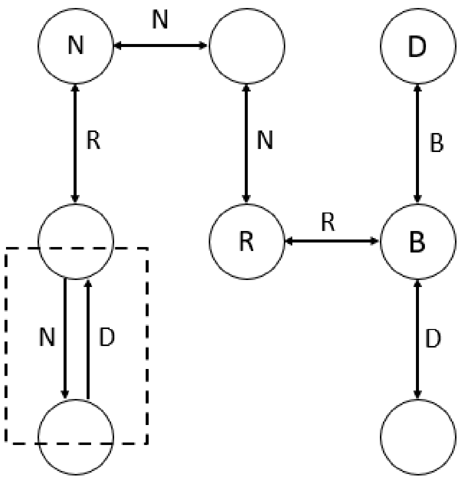


Figure 3.8: An Example of an Unwinnable Pit in the Map
It's possible for the player to reach the bottom node before having the D key. This results in a situation where the player is unable to make it back up to the rest of the map

3.3.4.1 Keys

The physical keys that appear in the game world are instances of the Key class. This class inherits from the Drawable class, for more information see Section 2.7. Keys are initialized with a position in the world, a color, and a gating type. Although it may at first seem that the color of a key and its gating type could be stored within a single attribute, it is actually simpler for the general case to keep the two values distinct. For the simple keys, such as 'red' or 'green,' having these attributes joined would suffice, but for more complex keys like double_jump, this would present a new issue. Logically the double_ump key cannot be of the color double_jump. Therefore it stands to reason that these two fields should be distinct.

For the time being, keys are represented by simple square sprites. In future versions of the game, it is more than possible that the representations of keys could become more interesting and complex. This is an idea that is discussed Section 6.5. However, designing individual key sprites for each gating type was outside the scope of this project, which sought to generate the underlying level structure. More appealing keys would not have improved or detracted from this original goal. All the keys share the same dimensions. Each is a 15 pixel by 15 pixel square of the provided color. If the color provided is None, then the square's surface will be blitted (drawn to the screen) black.

Keys contain two other attributes, which are important to game play. The first of these is a flag called 'collected.' As the name would imply, this Boolean flag denotes if the given key has been collected by the player. True meaning the key has been collected and False meaning that it hasn't. This flag is set to True by the collect method. The programmer can get the value of this flag using the collected method, which returns an intuitive Boolean value. The other important attribute stored within an instance of a key object is the key type. The programmer can access this value using the getType method. Additionally, there is a getColor method that returns the key's color. The color returned is the same color that was provided during the initialization of the key. Throughout this project rgb values are used to represent colors, but this is just by convention and is not actually enforced in the code. However, if rgb values are not used, errors could be created, which would otherwise be absent.

3.3.4.2 Gates

Gates, not to be confused with walls or platforms, which will be discussed later, are the regions of the map which temporarily impede the progress of the player. That is, the player cannot pass through a gate if they do not possess the required key, movement tech, or other gating tech. Although the game allows for bidirectional gating, this functionality is actually achieved by using two distinct instances of the Gate class and placing them beside each other. Of course that describes the typical case, not cases like double_jump.

The Gate class inherits from the Drawable class, which is discussed in Section 2.7. Upon its initialization, the programmer provides a position, color, and gate type. Optional parameters include direction, size, and pass through. The position represents where the upper left-hand corner of the final gate will be drawn onto the screen. The color represents the physical rgb value the gate will be colored, whereas the gate type provides the actual type of the gate. For a further discussion about the differences between color and gate type see the Section 3.3.4.1. The direction parameter determines the orientation of the gate. 0 is the default and denotes that this is a vertical gate, i.e. one that might appear in a wall. A direction of 1, then, denotes a horizontal gate that one might find in a ceiling or floor component. This is accomplished by flipping the height and the width of the gate. This height and width is provided through the size parameter. The size is provided as a tuple that contains the width and then the height in pixels. The default size of gates is 10 pixels wide by 40 pixels tall. The final attribute of an instance of the Gate class is the passThrough tuple. This tuple contains four True-False values, one for each of the cardinal directions. Here these directions are up, down, left, and right, represented in that order. All of these values are False by default, meaning that the player cannot pass through the gate in that direction without the proper key. If one of these values is True, then the player can pass through the gate in that direction. A good example is a jumping platform. The player is able to jump through the platform, but does not fall through it. This is because the tuple value representing up is True and the tuple value representing down is False. This passThrough tuple will be discussed more in later sections. Besides the init method, there is one other method in the Gate class. The getType method returns the gate type.

3.3.4.3 Barriers, Walls, and Platforms

The physical surfaces in the game demo are created from Gates. Walls, vertical barriers, and platforms, horizontal barriers, both inherit from the Barrier class. The Barrier class establishes both walls and platforms as multi-sprite objects. This means that each barrier consists of multiple smaller objects. In this case, the smaller objects are gates. The Barrier class provides functionality for drawing, getting components, updating, and making the internal objects picklable. Walls and platforms differ only in their initialization. The various components of a wall or platform are composed during the initialization. It is essential that the initializations be distinct because, for example, a double jump gate will look different if it occurs on a wall, rather than a platform.

Let's first consider how Wall objects are created. As with all Drawable objects, the Wall is provided with a position, which represents the coordinates where its top left corner will be drawn to the screen. The connection type and color of the gate are also provided. These are distinct for the reasons discussed in the above sections. The size of the wall is also provided, allowing the designer to make walls larger or smaller. Finally, the designer can specify a standard unit. This unit allows all wall and platform sizes to be relative and comparable to that unit. For the sake of my work, this standard unit is 1.5 times the height of the player's sprite. This means that when the height of the player changes, so too do the dimensions of the level. The neutral wall color, in this case referring to walls that the player can never pass through, is set to (120,120,120), a shade of grey. The height for gates is set to the standard unit, allowing the player to pass through them without the gates being too snug.

There are four different types of walls: exterior walls, double jump walls, shrink walls, and standard walls. As is probably apparent, all novel gating types that involve a non-standard gate will most likely also require a unique wall generation. Exterior walls are denoted with a connection type of 0. Counterintuitively, exterior walls are just Gate objects, with a height equal to the height of the entire wall. The color is set to the neutral grey and the connectionType is set to 99, denoting that the player will never have a key enabling them to pass through. Double jump walls are composed of 3 components. These include an optional top portion of the wall, the bottom segment of the wall, and a platform for the player to land on. A graphical representation of this layout can be seen in Figure 3.9d. The shrink walls are created in a similar fashion. These walls consist of just two components, including the top portion of the wall, and the lower bottom piece of the floor. This bottom piece is essential for all walls, so that the walls are flush with the platforms. In fact there is slight overlap between the bottom of walls and the right edge of platforms, preventing an awkward design like what can be seen in Figure 3.10. An example of a shrink gate and the math behind its creation can be seen in Figure 3.9c.

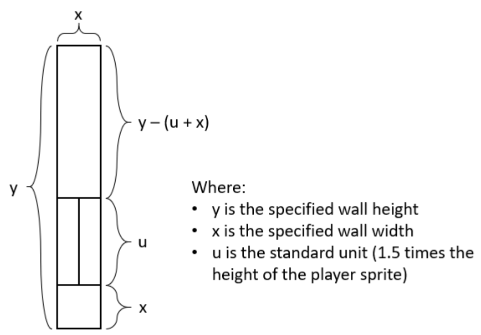
The final wall is the easiest to generalize. This wall represents all the other gating techniques by default. There are two variants of this general wall. The first variant consists of an upper wall segment, a left gate and a right gate, if they are not neutral connections, and a floor piece. See Figure 3.9a, for an example of this layout. The second variant represents the same type of gates, but with a single jump added into the mix. This would need to be adjusted if the player was not able to single jump at the start. These walls consist of an upper wall component, left and right gates, if they're not neutral, a bottom segment, and a platform for the player to land on. See Figure 3.9b for a visualization.

The creation of platforms is similar to walls, but currently with less variation. But a designer could easily add more conditions to create more variety in the types of platforms. Exterior platforms are created in the same way as exterior walls, but in the horizontal direction. All other platform types are created with a left and right sub-part, which are both of a standardized size. These sub-parts are the landings on which the player can walk. Platforms that do not make use of a neutral connection or double jump contain a gate / gates between the left and right sub-parts. All platforms also spawn additional platforms that allow the player to jump up to the gated entrance. The distance between these platforms depends on whether the main platform represents double or single jump. Double jump platforms create lower platforms with greater distance between them. The player can also pass through these sub-platforms in the upwards direction. This is because the passThrough parameter of gates is set to (True, False, False, False) for the sub-platforms. See Figure 3.11 for examples of platforms and their underlying math.

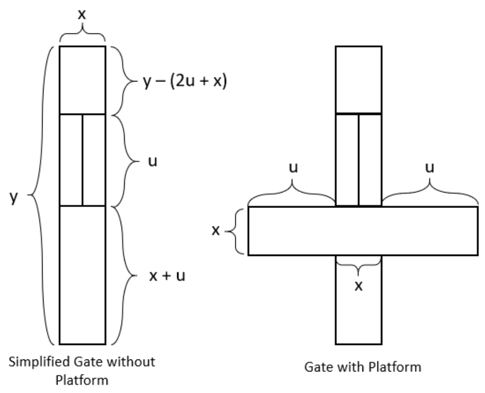
Bidirectional gates, whether part of a wall or a platform, make use of the passThrough parameter as well. For example, on a wall, if the gate on the left is red and the one on the right is blue, then the player should be able to move from left to right with just the red key. To accomplish this, the passThrough parameter for the blue gate is set to allow the player to pass through on the left side. The opposite is done for the red gate.

3.3.4.4 The Player

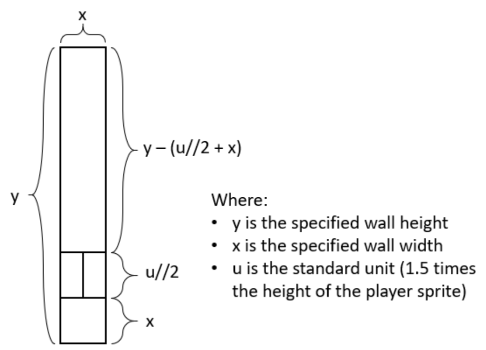
The player is represented and modeled by the Avatar class. This class is responsible for handling player input and updating the player's sprite accordingly, as well as, managing all collisions between the player and gate and key objects. The player has a position, which initially is within the starting room, a velocity, which is initially zero, a list of keys, initially empty, and an internal finite state machine managing the state of the player. The player's different states include, standing, jumping, falling, and walking. A graphic of this state machine can be seen in Figure 3.12. The Avatar class provides methods for getting the player's keys, determining if a player has a given key, giving the player a new key, moving the player, and updating the player.


x
y
y – (u + x)
u
x
Where:
• y is the specified wall height
• x is the specified wall width
• u is the standard unit (1.5 times the height of the player sprite)

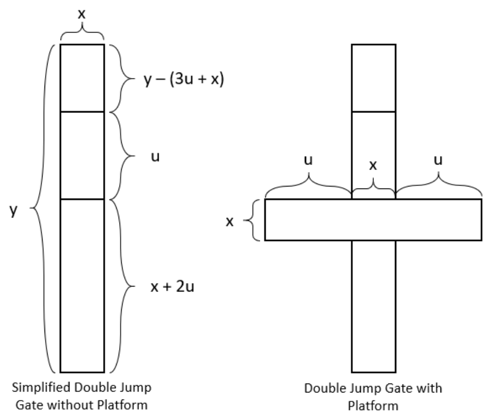
(a) Standard Gate Variation One



(b) Standard Gate Variation Two


x
y
y – (u//2 + x)
u//2
x
Where:
• y is the specified wall height
• x is the specified wall width
• u is the standard unit (1.5 times the height of the player sprite)

(c) Shrinking Gate


x
y
y – (3u + x)
u
x + 2u
u
x
u
x
Simplified Double Jump Gate without Platform
Double Jump Gate with Platform

(d) Double Jump Gate

Figure 3.9: Anatomy of Gate Types

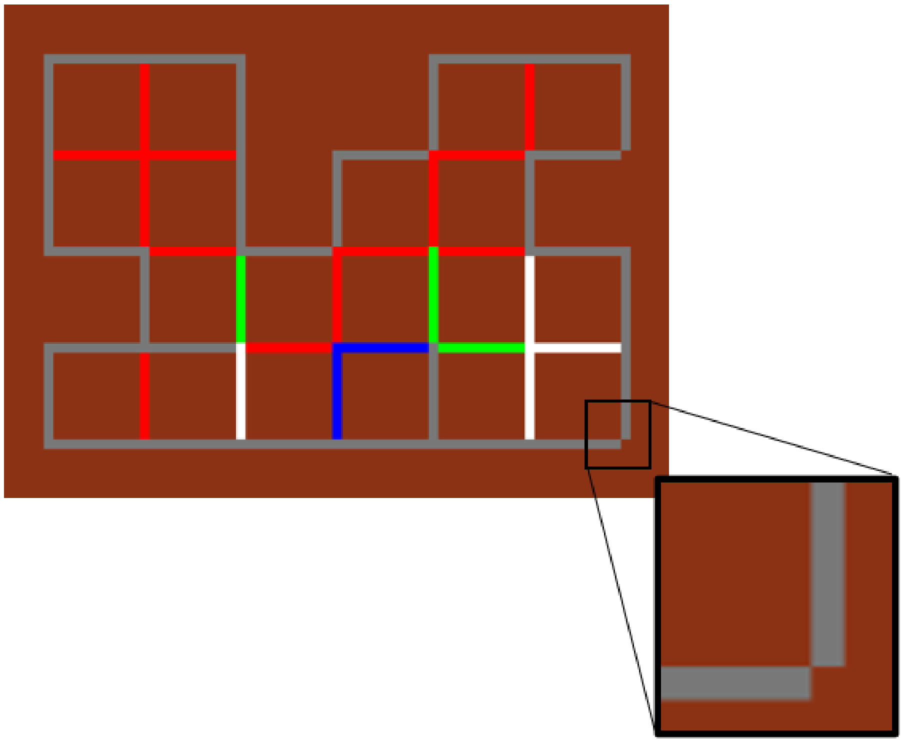


Figure 3.10: Example of Corner Generated without Overlap

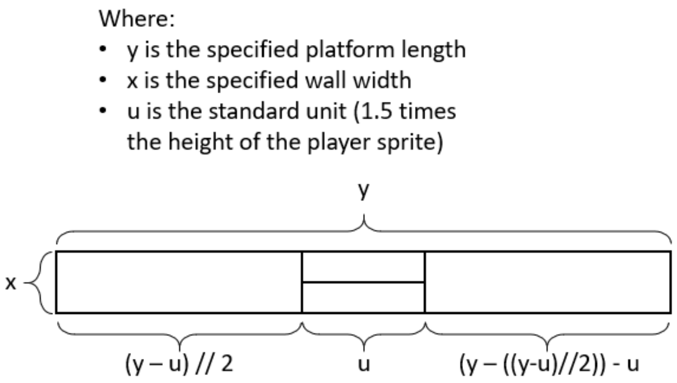

Where:
• y is the specified platform length
• x is the specified wall width
• u is the standard unit (1.5 times the height of the player sprite)
y
x
(y – u) // 2
u
(y – ((y-u)//2)) - u

Figure 3.11: Anatomy of a Platform

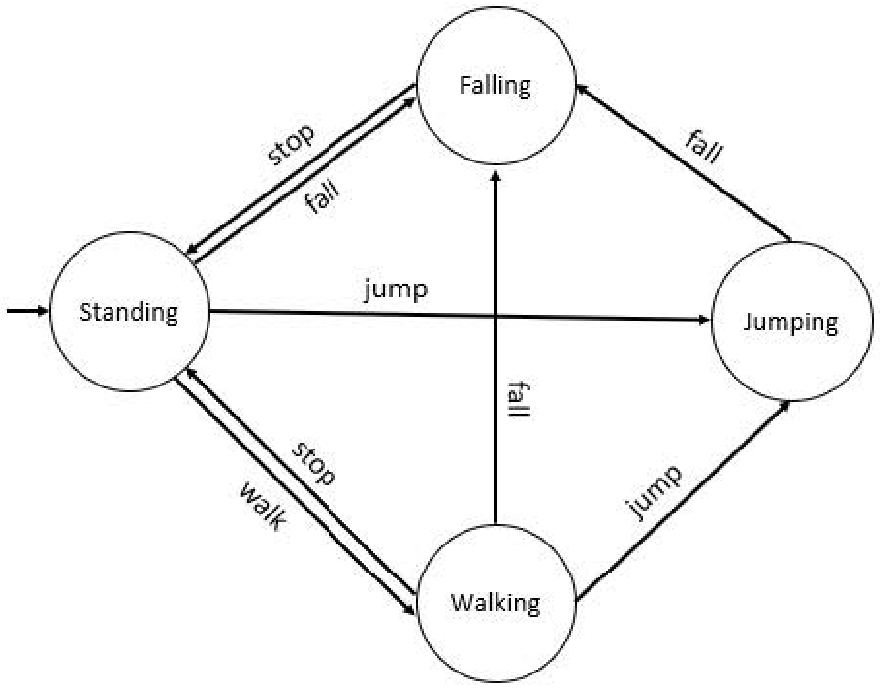


Figure 3.12: Player's Finite State Machine

When the space key is pressed, if the player is standing or walking, the player switches states to jumping. This also sets the boolean flag onGround to False and increments the jump count by one. If the space key is pressed and the player is currently jumping, if the player has double jump and the jump count is 1 or less, the player can jump again. This resets the jump time and increments the jump counter. When the s key is pressed, if the player is at full size and they have the shrink key, the player shrinks. If the s key is pressed and the player is shrunk and they possess the shrink key, the player grows. The scaling factor is two. Other movement events consist of the arrow keys moving the player left and right.

The update method is actually responsible for moving the player around the world. If the left arrow is down, then the velocity is set to the negated maximum velocity. If the right arrow key is down, then the velocity is set to the maximum velocity. If neither of the keys is down, then the velocity is set to zero. This means that as soon as the keys are released, the player should come to a complete stop. If the player's state is standing or walking and they are no longer on the ground, their state is changed to falling. If the player is jumping and there is still time on the jump timer, the player's y velocity is set to the negated max velocity and the jump timer is decremented according to the ticks, otherwise the player's state is changed to falling, the jump timer is reset, and the jump count is reset to zero. If the player's state is falling and they are still not on the ground, their y velocity is set to the maximum velocity, otherwise they transition to the standing state. The player's new position is calculated using the velocities determined previously. See formula:

$$newPosition = currentPosition + (velocity * ticks)$$

The onGround flag is set to False. It is safer to assume that the player is always falling than it is to assume that the player has reached the ground. This prevents the player from achieving further jumps by falling off of a high platform and waiting to jump until they have nearly reached the ground. In other words, this safe guards against players reaching areas they shouldn't have access to.

Finally, the update method checks for collisions between the player and the various components of the walls and platforms. If the player does not possess the necessary key and collides with a barrier sprite, it is determined which direction the player approached the barrier from. The player is pushed back the way they came until they are flush with the outer edge of the barrier. That is unless the barrier specifies that the player can pass through in the given direction. A diagram of this behavior can be found in Figures 3.13 and 3.14.

3.3.4.5 Creating the Physical Representation of the Map

The prepareMap method combines all of the pieces described in the other Sections of 3.3.4 and the underlying graph of the map to create the final playable level. It is in this function that the standard unit u is determined. As previously mentioned this standard unit is 1.5 times the height of the player. List containers are also created to hold the walls, platforms, and physical representations of the keys. The room height

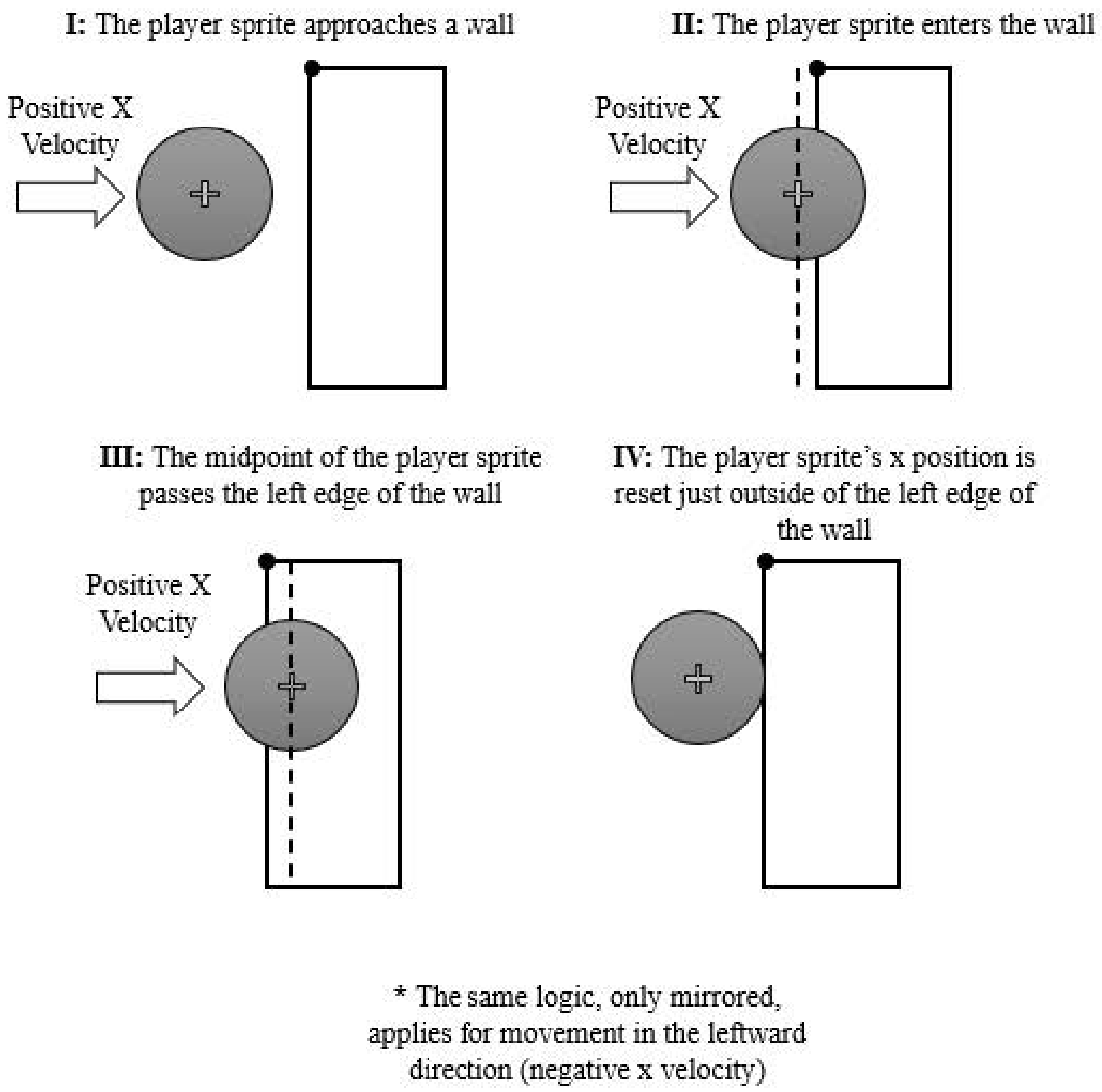

I: The player sprite approaches a wall
Positive X Velocity
II: The player sprite enters the wall
Positive X Velocity
III: The midpoint of the player sprite passes the left edge of the wall
Positive X Velocity
IV: The player sprite's x position is reset just outside of the left edge of the wall
* The same logic, only mirrored, applies for movement in the leftward direction (negative x velocity)

Figure 3.13: Horizontal Movement Physics

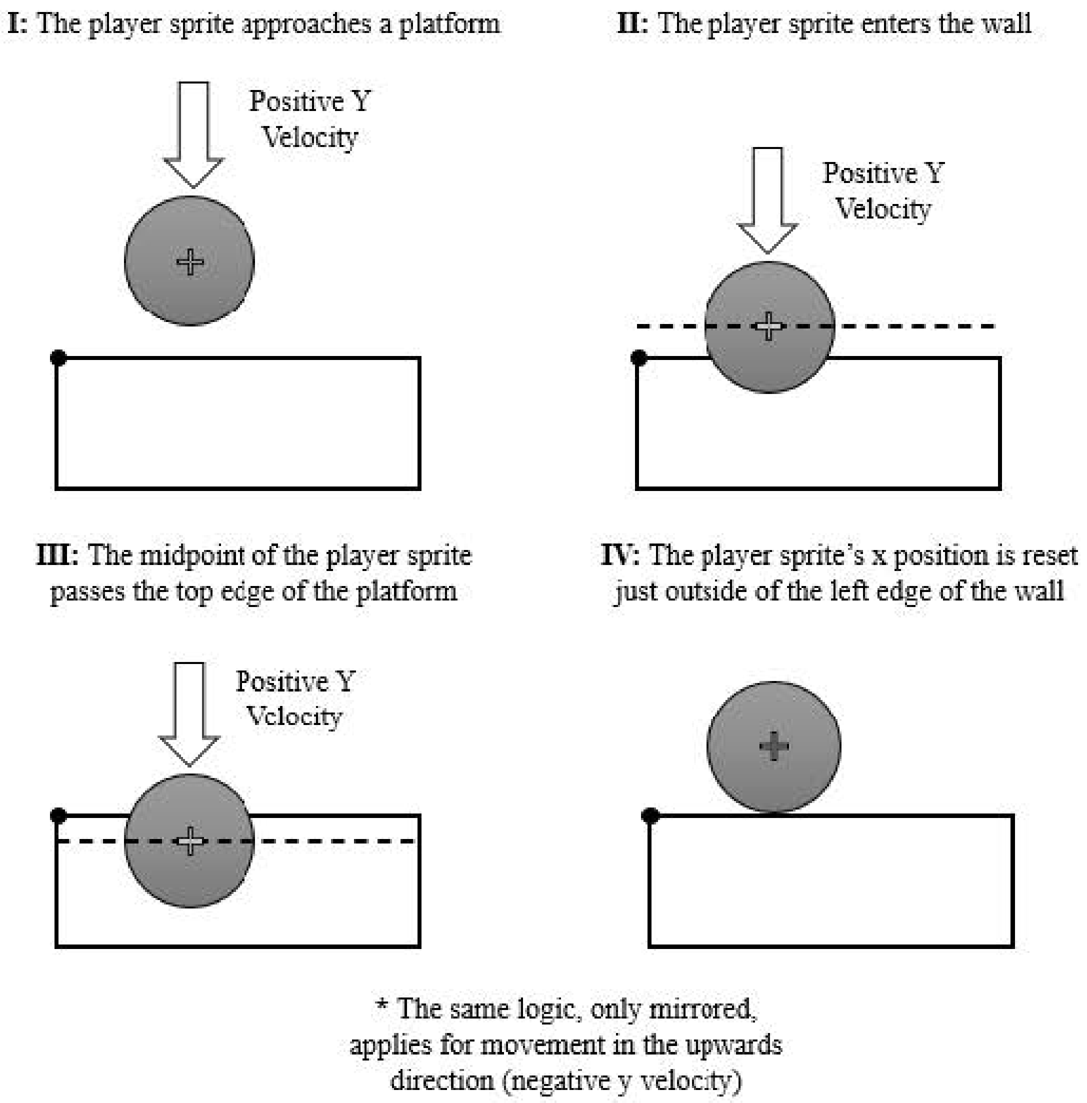

I: The player sprite approaches a platform
Positive Y Velocity
II: The player sprite enters the wall
Positive Y Velocity
III: The midpoint of the player sprite passes the top edge of the platform
Positive Y Velocity
IV: The player sprite's x position is reset just outside of the left edge of the wall
* The same logic, only mirrored, applies for movement in the upwards direction (negative y velocity)

Figure 3.14: Vertical Movement Physics

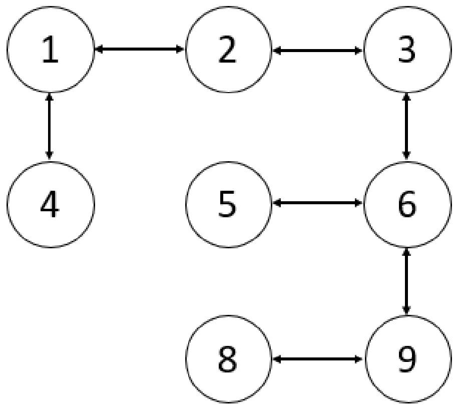
and room width are also set, and these two values are combined into the roomSize tuple. The width of the barriers is set to be 1/4u or .25 standard units. This means that the wall size is (barrierWidth, roomHeight + barrierWidth). As mentioned previously, the border width needs to be added to the height so that visuals like those in Figure 3.10 don't occur. The platform size is set to (barrierWidth, roomWidth + barrierWidth). Finally, the start coordinates for the generation are set. Currently the coordinates (100,100) are set as the start.

Next, the top corners for each room in the mxn grid are found and added to a list called topCorners. This list will be used to easily place the walls and floors/ceilings of rooms later on. Then all of the distinct rooms in the graph are found and added to a list called rooms. The edges of the graph are then split into two distinct categories, forward edges and backwards edges. In this case, for a room i, a forward edge connects i with the room to the right or i to the room below. A backwards edge connects in the opposite direction. If the graph being used to generate the level predates the dual-gating, then the backwards edges are the same as the forwards edges. This enables some backwards compatibility.

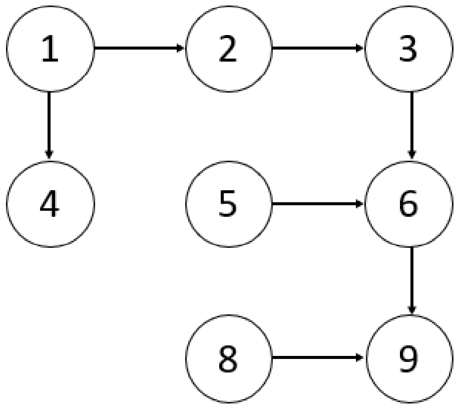
Once all of the edges have been found, we iterate through the forward edges to add the interior walls and platforms to the map. To do this, we get the forward edge gating type and the backward edge gating type and determine the relationship between the rooms they connect. If the first node is r and the second node is r+1, then the edge represents a wall. If the second node is r + the number of columns, then the edge represents a platform. See Figure 3.15 for a pictorial description. This simple system suffices to add all interior barriers, meaning that all gating types are added here.

After all of the interior edges have been added, the exterior edges are put in. While this might at first seem simpler, it actually takes quite a bit more logic to find all such edges. There are four distinct cases, each with two subcases. The logic for these is described more fully in Figure 3.16. Once this is done the map is fully enclosed, with the tops, bottoms, and sides of rooms fully added.

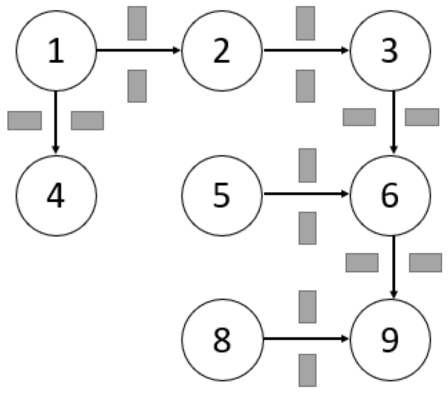
With all of the rooms created and ready to be drawn to the screen, next the



(a) Simplified Underlying Graph of Node Connections

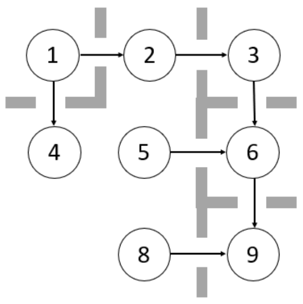


(b) Graph showing only edges in the forward* direction

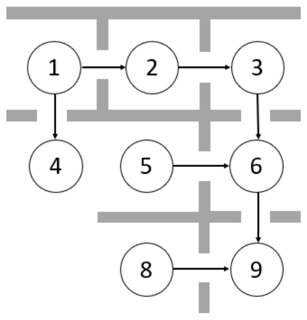


(c) Map with all interior barriers generated

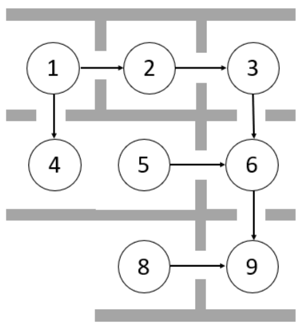
Figure 3.15: Plotting Interior Walls on the Map
Iterate through the forward edges*, adding walls/platforms (with gates) between the nodes/rooms they connect.
*A forward edge is defined as an edge from X and Y, where Y is greater than X



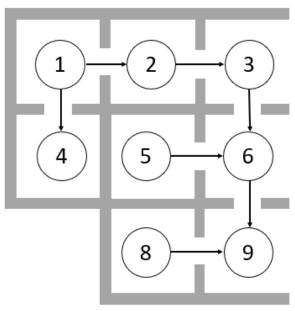
(a) Map with all interior barriers generated



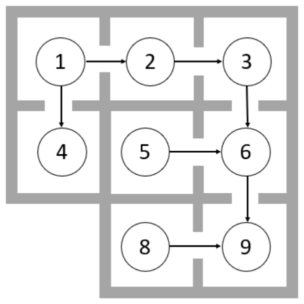
(b) Add barriers to the tops of rooms, using the following rules, where r is the current room and n is the number of columns: $r \leq n$ or (r-n, r) is not an edge in the graph



(c) Add barriers to the bottoms of rooms, using the following rules where r is the current room, n is the number of columns, and m is the number of rows: r m*n or (r, r+n) is not an edge in the graph



(d) Add barriers to the lefts of rooms, using the following rules where r is the current room, and n is the number of columns: r % n = 1 or (r-1, r) is not an edge in the graph



(e) Add barriers to the rights of rooms, using the following rules where r is the current room, and n is the number of columns: r % n = 0 or (r, r+1) is not an edge in the graph

Figure 3.16: Plotting Interior Walls on the Map

physical key objects are placed. Each key is placed in the center of its respective room, such that it is easily within reach of the player. The neutral gate, however, does not have a physical key.

Now that the level has been completely generated, a player object is created and placed within the start room. The walls and platforms are broken into their components, so that the player object can check for collisions later. Backups of the physical keys are created and stored so the player can reload the level. The finishing block is created and added to the ending room. The mini-map, described in Section 3.3.5, is created.

3.3.5 Mini Map

The mini-map is specific to the game demo and serves as a navigational guide for the player. Additionally, the mini-map provides the user with an idea as to the overall structure and shape of the map, probably more so than even the networkx plot. This mini-map does not, however, provide information pertaining to all the connections between rooms, or even if such connections exist. For information such as this, the level demo, described in Section 3.3.7, is ideal.

The physical mini-map can be thought of as a different representation of the underlying graph structure. The purpose of this representation is to detail the basic shape and layout of rooms. Additional information provided includes the players current position, which is obviously completely independent of the underlying structure of the level. The mini-map is created in much the same way as the actual physical level. The top corners of each room are determined and saved and then all distinct rooms are found. Then for each room, a rectangle (actually a pygame Surface) is positioned using the top corner for the respective room. All of these rectangles are colored white, except for the start and the end, which are green and blue respectively. In implementation the mini-map is nothing more than a list of mySurfaces, where each surface represents a room.

The player's current position is calculated and the square representing the player is moved by the updateMiniMap method. The position of the player on the mini-map is an approximation as to which room the player is in. Given some configurations, the map can be out of sync by a few pixels. This error seems to increase as the size of the level increases, becoming more apparent as the player nears the bottom right corner. The player's position on the mini-map is calculated using the following process. The x position for player's position on the mini-map is found by finding the x-coordinate for the center of the in-game player, offsetting it by the start coordinate's x value, and scaling it by the ratio of the mini-room width vs the full scale width of the in-game rooms. This value is then offset by the mini-map's start coordinates. The same process is done for the y coordinate, but using heights and their associated values. In a sense, the mini-map is treated as a scaled down version of the larger game world, and the player's position is calculated relative using the ratio between the two.

3.3.6 Drawing, Updating, and Event Handling

Because nearly all of the components used to build the platformer demo inherit from the Drawable class, or have internal components that inherit from Drawable, they all contain draw methods. This means that to render the map on the screen, draw is called on the finish block, all walls, all platforms, the keys, and the player. If the mini-map should be shown, then every MySurface comprising it and the pointer's draw methods are called as well. If the player has reached the end and won the game, then the end game text is displayed on the screen. The orbs representing the collected keys are drawn to the screen using pygame's built in draw.circle function.

The LevelTester class allows the player to handle events, if the game has not yet been won. If the game has already been completed then the LevelTester sets all of the player's movement keys to False, effectively stopping the player's movement. Without this, the player would continue to move forward until the pygame window was closed or the game was restarted. Other events handled in the main include listening for various key presses and responding accordingly. These key presses are described more in Section 3.5.1.

The LevelTester updates the game state, first by updating the camera offset. This offset is used to keep the player in the center of the screen when possible and allows for the world size to be much larger than the physical limitations of the screen. Next, the player object is updated. The player's update method is described in Section 3.3.4.4. Then the game determines if the player is colliding with one of the physical in-game keys. If this is the case, then the key is collected and given to the player. The list of physical keys is reset, removing any keys that have since been collected. The LevelTester then checks if the player has reached the final room and won the game. If this condition is true, then the won flag is set to True. Finally, the mini-map is updated. This update is described in Section 3.3.5.

3.3.7 Other Visualizations

Another, older, version of the game demo exists. This game however, doesn't have any sort of applied physics and only strictly adheres to the gating rules. The levelDemo, aptly named since its main purpose was to demonstrate a level's structure, was developed in parallel with the gameDemo, keeping it functional and compatible with the new map save files. With that being said, the fancier features, like the save and load GUIs, were not adapted or applied to this older tester. However, the idle shell or the terminal can still be used to load or save levels. More precisely the levelDemo displays map templates, since the physical maps actually have more features that aren't relevant at this stage, such as different wall variants.

In the levelDemo, the map is represented by rows and columns of black squares, each a stand in for a room. These squares are connected with lines of varying color. These lines represent the connections between the rooms. The bottom / right lines represent forward connections, while the top / left lines represent backward connections. Rooms that contain keys are colored the same as their respective key. So the room containing the red key would be red, rather than the standard black. Unlike in the finalized platformer version of the game, the end node is not represented any differently. The keys that the player has collected are represented as orbs in the bottom left corner of the screen in the same fashion as they are in the other demo.

Logic within the Player class, not to be confused with the Avatar class discussed at length previously, allows the grey square representing the player to navigate the level, obeying all gating rules. The left, right, up, and down arrow keys are used to move the player through the web-like structure.

The end result of this demo is a fully playable level with a bird's eye view of the entire map.

3.3.8 Summary of Implementation

A fully playable level is created through the following steps. First, the designer supplies the generator with rules that dictate an ordering for the keys. This is then used to generate a linear ordering adhering to the specifications. The designer also provides a list of which gates can be found on the horizontal and vertical edges. This list can also contain tuples that define the bi-directional behavior for edges. For example, the double jump / neutral connection. The designer can also provide a custom start node and a custom end node.

Next, the underlying graph for the level is created obeying the constraints of the linear key ordering. The structure of this graph resembles a grid or a lattice. The connections for being neighboring nodes are determined at random and once the entire graph has been generated, the map is tested for viability. If the map is invalid, then another map is generated. Once a valid map is created, it is returned.

Then, the physical game world is created based off of the underlying level graph. This world consists of platforms and walls, which are both made up of gates. There are multiple variations for the different gating types, allowing for further uniqueness in the generated product. After all of the rooms have been created, then the keys are added to their designated rooms.

Once the level has been crafted, the human player can interact with it. The player can pick up keys and pass through gates. They explore the maze-like structure of the level until they collect all keys and eventually reach the end goal.

3.4 Saving and Loading Maps and Templates

Designers and players have the ability to save and load levels as both maps and templates.

3.4.1 Pickling and Loading

There are two ways in which mapping information can be saved. The first involves saving a map template which can be used to generate unique maps that have the same underlying structure, but a different physical representation. The second is a true save, that stores all of the features of the map. These features include the underlying map template as well as all of the walls, platforms, and gates, which would be unique on each generation. Both varieties of saving make use of Python's built in pickle module and two data storage classes. Now we will take a more detailed look at each of the respective save types.

Map templates are the simpler save format and are stored as .mapdat files. The underlying structure of the pickled object is an instance of the MapData class. The MapData class contains the following instance variables: a networkx graph g, keys, gates, m, n, the end node, the ordering, the start node, the weighted neutral, the horizontal mapping, and the vertical mapping.

Physical map levels are more complex than templates and are stored as .mapfile files. The underlying structure of the pickled object is an instance of the GeneratedMap class. The following instance variables are stored within a GeneratedMap object: templateData MapData object, finishing block, walls, platforms, physical representations of the keys, the player's start position, and the dimensions for rooms. Because the platforms, walls, gates, and other Drawable objects contain pygame Surfaces, it is impossible to pickle and then reload them without modification. Doing so causes the image surfaces associated with objects to become corrupted and ultimately causes the game to crash. Therefore, each and every one of these surfaces needs to be converted into a string representation before the pickling process. After the pickling process the Surface must be converted from a string back to a pygame.Surface object. To accomplish this in a clean and concise manner, makePickleSafe and undoPickleSafe methods were added to the Drawable class, which all of the level's assets inherit from. The last hurdle was adding the makePickleSafe and undoPickleSafe methods to the multi-sprite Barrier objects that would then call the Drawable method on each contained sprite.

All saves can be found in the saves folder. Map templates are saved in the templates subfolder and physical maps are saved in the maps subfolder.

3.4.2 The User Interface

The graphical user interfaces for saving and loading are built on the graphics package discussed in Section 2.6. Both interfaces are nearly identical in their layout, and are therefore both instances of the same menu class. There are two notable differences between the saving and loading interfaces. The biggest of which is that users are not able to type in the input field on loads, but they are on saves. Since all of the saved maps are prominently displayed in the selector field, there seemed to be no reason for the user to be able to manually type which map they wanted to load. The only other difference is the text displayed on the confirmation button. This text either reads 'load' or 'save.' See Figure 3.17.

The GUIs for loading and saving maps use the following classes from the graphics package: TextInput, Button, and ScrollSelector. For more information on the graphics package used in this project, see Section 2.6.

The ScrollSelectors in both of the interfaces are populated using Python's built-in glob module. The glob module provides a means of navigating and extracting information from the file system. In this instance, it is used to get all of the file names that are located within a particular directory.

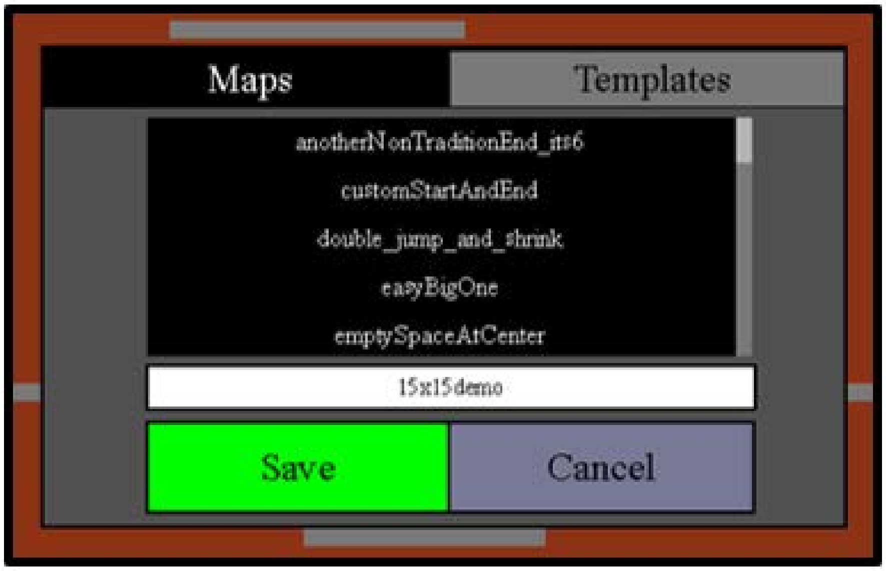


Figure 3.17: The Save Interface

3.4.3 Supporting Past Versions

The loading functionality also supports most backwards compatibility. That is to say that older maps, in which many newer features were not yet available, are still fully playable in the latest version of the platformer. Examples of features that original maps would not support include: varying the percentage of neutral connections, user specified start position, user specified end position, and dual-directional gating, to name a few.

3.5 Playing the Generated Level

Once the game has been generated and rendered, it can be played.

3.5.1 Controls

The controls for the platforming game demo are similar to those of other such games in the genre. The left and right arrow keys are used to move the player left and right across the screen respectively. When the spacebar is pressed, as is the convention, the player jumps. If the player has unlocked the double jump movement tech, then they can press the spacebar a second time, while mid-jump, to further increase their player's altitude. The player cannot jump while falling, as this could result in behavior similar to the double-jump movement tech. Once the player has acquired the shrinking gate tech, pressing the 's' key on the keyboard will allow the player to shrink. Hitting the 's' key again reverts the player back to their normal, default height.

The player also has the ability to see the overall structure and shape of the map, as well as their current position in it. This information is conveyed through the use of a mini-map, which is described in Section 3.3.5. The mini-map is invisible by default so that it does not clutter up the screen and inhibit the player from enjoying or playing the game. To show the mini-map, the player must press the 'm' key. Pressing the 'm' key a second time will cause the mini-map to become invisible once more.

Other controls include the saving and loading functionality. By convention, hitting CTRL + s, allows the player to save their current game map (note, this does not save the player's progress in the game). Also by convention, the player can load a saved map by hitting CTRL + o. For more information on saving and loading maps, see Section 3.4.

It is also possible for the player to display a graph of the underlying map structure. This is used mostly for testing or visualizing the map as a whole. Hitting CTRL + p (p being for plot) will cause a matplotlib window containing the graph structure to appear. Before continuing the game, this window must be closed. For more information on the graph plot, refer to Section 3.6.

3.5.2 Collecting Keys and Passing Through Gates

The objective of Metroidvanias, including this demo platformer, is to collect keys to explore new regions until the end goal, room, or boss is finally reached. In this case, the end goal is simply the last explorable room, which contains a large golden rectangle. In order to reach this final room, the player must first collect all of the keys in the game and use them to skillfully navigate through the maze-like structure of the mapping. The player always starts off with the first key or gating tech by default, lest the game would become unwinnable. In a majority of the generated levels, the neutral key is this first default key, but that is only by convention. Any other key, including the various colors, double jump, or shrink, could just as easily be this first, default gating tech. All other keys beyond this first one must actually be collected by the player. To do this the player must search through the various rooms of the map. The player must have the required gating tech to pass between rooms. That is to say that if the gate requires the red key, the player must have already collected the red key in order to pass through and enter the next room. As discussed in Section 3.3.2.2, gates can also be directional, meaning that it is possible to pass through a gate in one direction and impossible to pass back through it in the opposite direction. Each key that is found leads to the next key, until the final key has been discovered.

3.6 Plotting the Underlying Graphing of Maps

The underlying gating structure of a mapping can be plotted for the designers or players convenience. The functionality of this feature is built on top of matplotlib and networkx. To make the graph more visually pleasing and decipherable, the nodes of the graph are color coded. That is to say that the node that contains the red key should be colored red in the plot. To prevent a crash, this logic does not follow for nodes that contain movement tech or other non-traditional gate type keys. So, for example, the node that holds the double-jump movement tech is just colored grey. This is done as a precautionary measure because 'double-jump' obviously is not a viable color to plot. But this could be easily remedied in the future by creating a dictionary mapping such movement techs to representative colors. The above logic is implemented in and described by a color_map. This color_map is then fed into networkx's draw function. Next, the edge labels are found using networkx's get_edge_attributes function. The edge labels are then drawn onto the final plot by calling draw_network_edge_labels. Finally the plot is actually displayed and printed to the screen.

Chapter 4

ANALYSIS

4.1 The Ability to Win Generated Levels

By the nature of their creation, all the levels generated are winnable. As discussed in Section 3.3.3.1, the createLattice function enforces rigid constraints on the connections that can exist between rooms. These constraints result in underlying graphs that are not only planarizable, that is that no edges cross, but also meet the more grid-like pattern seen in Figure 3.4. But just because the underlying graph model can be used to generate a feasible dungeon or maze doesn't mean that that map is winnable. In fact, due to the random nature of how the gate types are selected, many maps generated by the createLattice function are not winnable. However, these faulty maps never make it to the full game rendering process. Section 3.3.3.2 details the process through which these generated graphings are verified for winning routes by the viableMap function. The generateViableMap function uses both createLattice and viableMap in concert until a viable, winnable map has been created. Once such a map is found, it is returned and any of a variety of visualizations can be built atop it. So, because all unwinnable maps are less than viable, they are scrapped, thus only winnable maps are presented to the player.

4.2 Enforcing Gate Ordering

A linear gate ordering is generated from the directed acyclic graph that the designer provides to the program. The process through which this information is input and processed into the final ordering can be found in Sections 3.2.2 and 3.3.2.1. The manner in which the keys are placed ensures that the gate ordering is enforced. No key is accessible before the previous key has already been acquired. More details about this implementation can be found in Section 3.3.3.3.

4.3 Playability of Generated Maps

Although no formal user studies have been conducted to date, a handful of my peers from the Computer Science department took an interest in our work. Once the game demo had reached a playable point, many of them took the opportunity to test the game. Obviously nothing conclusive can be drawn from these playthroughs, but their reactions are still worth mentioning. For many, the graphics or the physics were the worst aspects of the game experience. Considering both of these functionalities are incredibly bare-bones and serve only to demonstrate the underlying idea, that was to be expected. However, there were no complaints about the levels or how they were generated. On the contrary, one of my colleagues even commented that with improved graphics, he could easily see this project being something available for purchase on the app store. For this reason, it seems prudent to move forward with user studies in the near future.

4.4 Level Generation Times

Most level configurations can be generated within a fraction of a second. One hundred timed trials were run for each of thirty test cases. Level size and the number of keys were the two variables tested. The start position and weighted neutral were held constant. The end node was always set to the node at position m*n. The gating technologies, gating order, and horizontal and vertical mappings were identical across trials. All gating technologies occurred both horizontally and vertically. All gates were bidirectional such that the same key could be used in both directions. Perfectly square level sizes from 3x3 to 8x8 were tested. Within each of these size categories, the number of keys from three to seven was examined. When generating winnable levels, many unwinnable level configurations are also created. This is due to the random nature of the connection types in the lattice creation described in Section 3.3.3.1. The average number of these potential, less than viable maps was calculated for each of the thirty test cases. The average generation time for each of the test cases was also determined. The end results of these trials can be found in Table 4.1.

The collected data suggests that the number of keys in a level has a greater impact on the average number of potential map generations, and consequently the generation time, than does the size of the level. It appears that as the number of keys increases, so too does the number of potential maps generated before a viable map is found. This is especially true with smaller map sizes. A level with size 3x3 and seven keys is the worst configuration among the test cases. Whereas all of the other twenty-nine cases averaged generation times less than a second, this configuration had an average generation time of over six seconds. This makes sense, considering that a 3x3 map contains only nine rooms, one of which is the ending room, leaving only eight rooms for potential key placement. Seven keys must be placed among the eight rooms in such a way that the gating order specified by the designer is not violated. Adding connections between rooms at random, as the current algorithm does, means that more potential maps will need to be tested in cases such as these.

The results in Table 4.1 clearly show that the level generation process is swift enough to provide designers with Metroidvania-style levels without a long wait. The longest wait among the trials was 6.46 seconds, which is still significantly less time than it would take for a human to design and create a comparable level. The generation time can yet be improved further by limiting the number of potential maps that need to be created. This is described more in Section 6.1.1.

Level Size	Number of Keys	Average Number of Potential Maps Generated	Average Generation Time in Seconds
3x3	3	4.94	0.001737027
	4	16.5	0.004733682
	5	123.84	0.03570178
	6	1269.45	0.329310718
	7	26506.77	6.464433384
4x4	3	5.5	0.003650911
	4	8.01	0.007024148
	5	24.38	0.015109587
	6	110.83	0.055219085
	7	655.81	0.286802242
5x5	3	6.02	0.006556003
	4	7.3	0.010442557
	5	18.37	0.021231167
	6	47.54	0.043254747
	7	156.65	0.11885674
6x6	3	5.31	0.010233276
	4	8.34	0.01736299
	5	13.1	0.02476155
	6	33.56	0.050190401
	7	80.01	0.101225777
7x7	3	5.9	0.014234133
	4	9.42	0.026998692
	5	16.35	0.044138813
	6	22.98	0.055986459
	7	43.19	0.091679959
8x8	3	5.87	0.024189534
	4	7.86	0.042017314
	5	12.7	0.065249419
	6	18.86	0.08191386
	7	43.85	0.138554451

Table 4.1: Generation Data by Level Size and Key Count

4.5 Example Map Generations

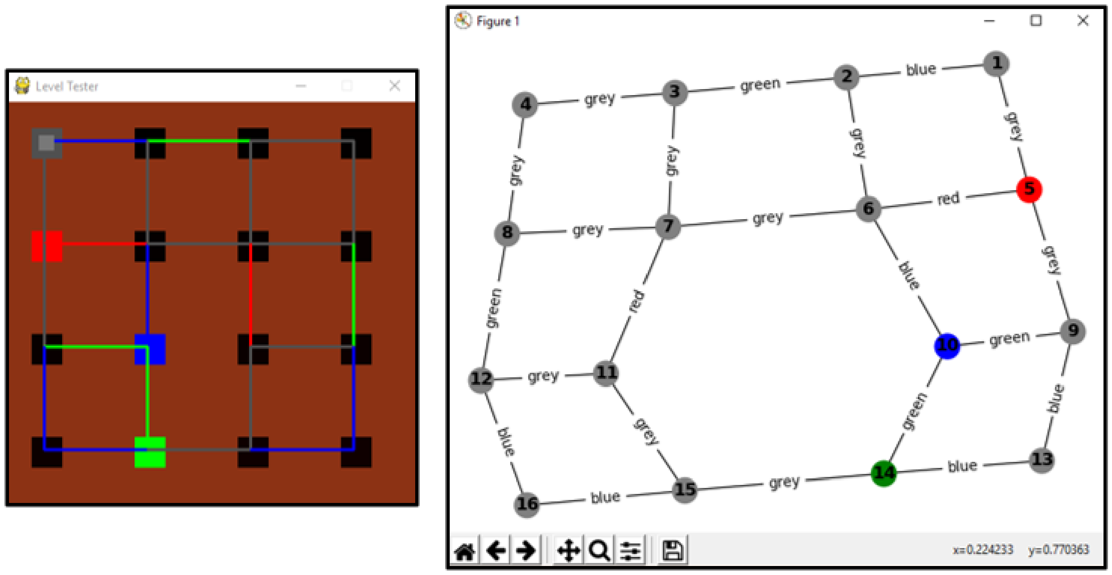


Figure 4.1: Undirected 4x4 Level Demo View and Underlying Graph

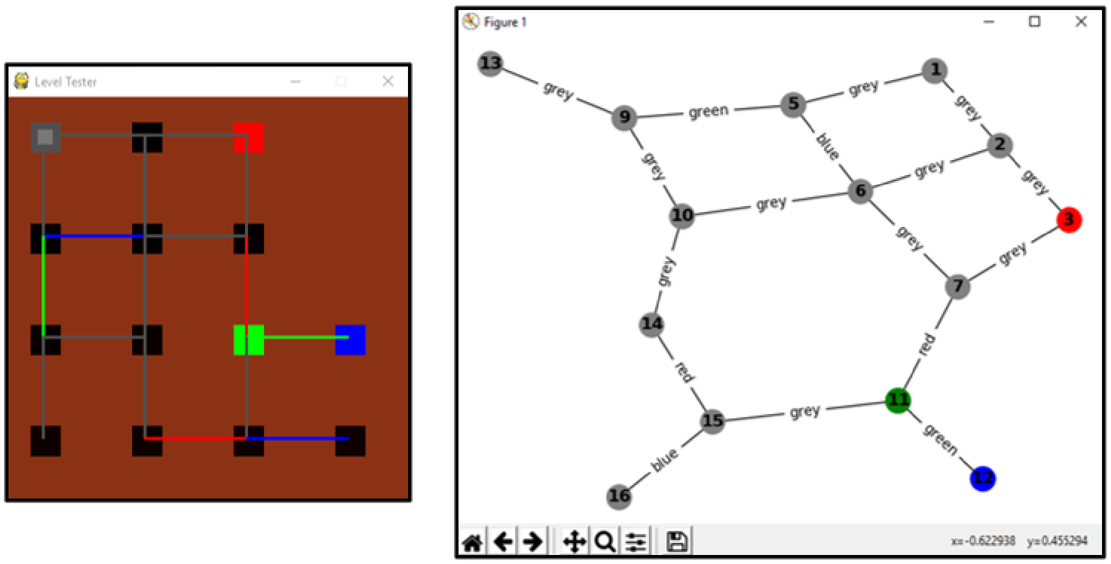


Figure 4.2: Undirected 4x4 Level Demo View and Underlying Graph

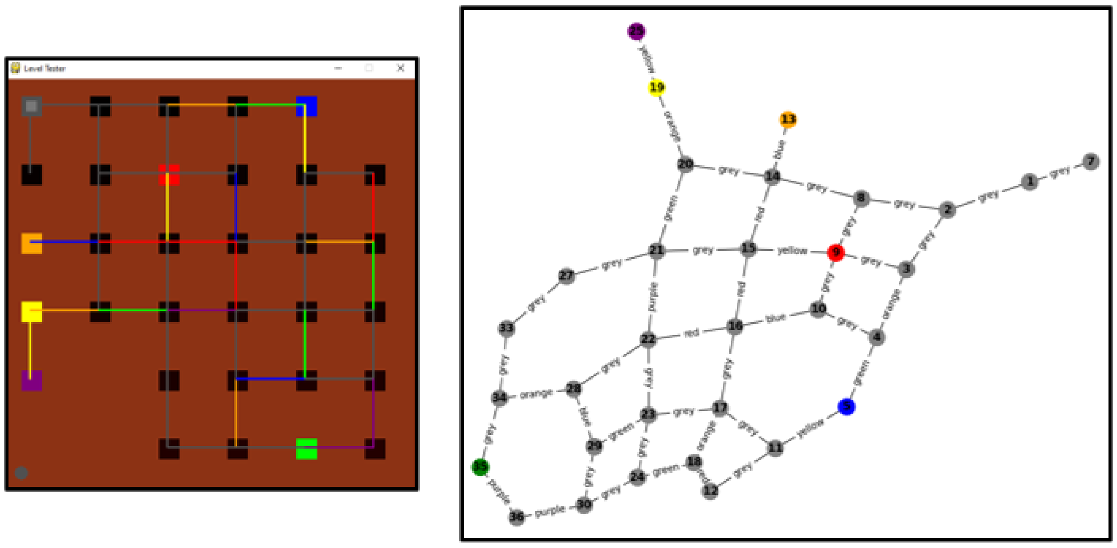


Figure 4.3: Undirected 6x6 Level Demo View and Underlying Graph

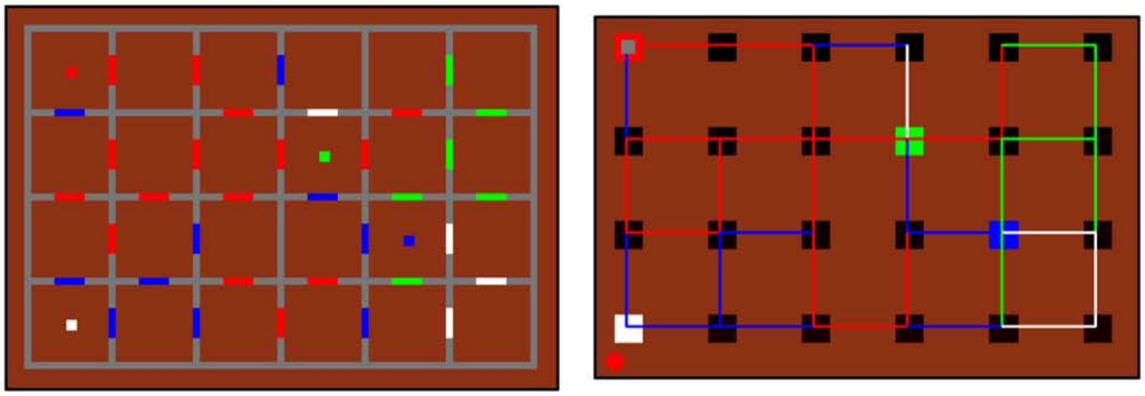


Figure 4.4: Early Rendering of a 4x6 Game Map and Associated Level Demo View

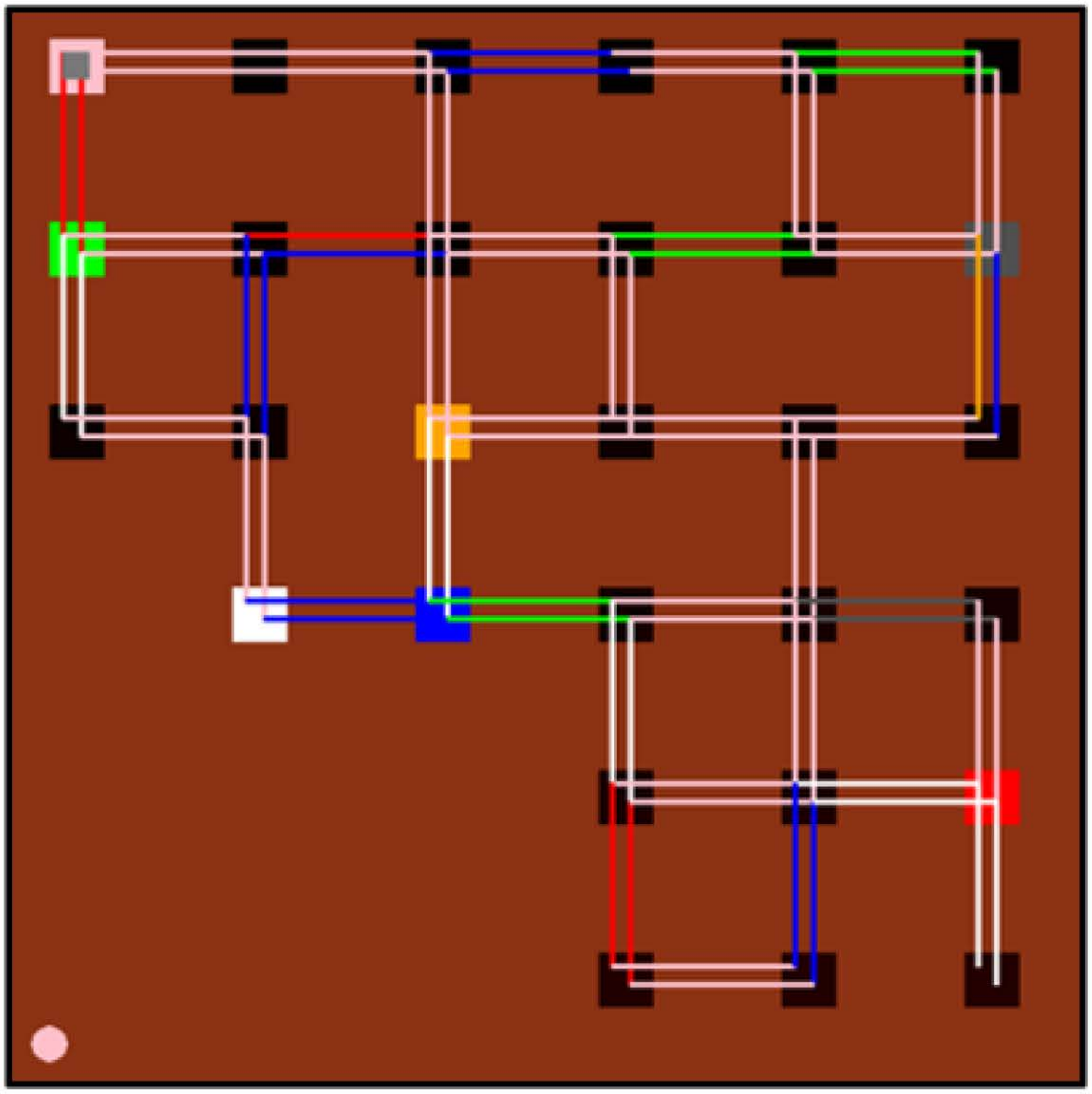


Figure 4.5: Directed 6x6 Level Demo

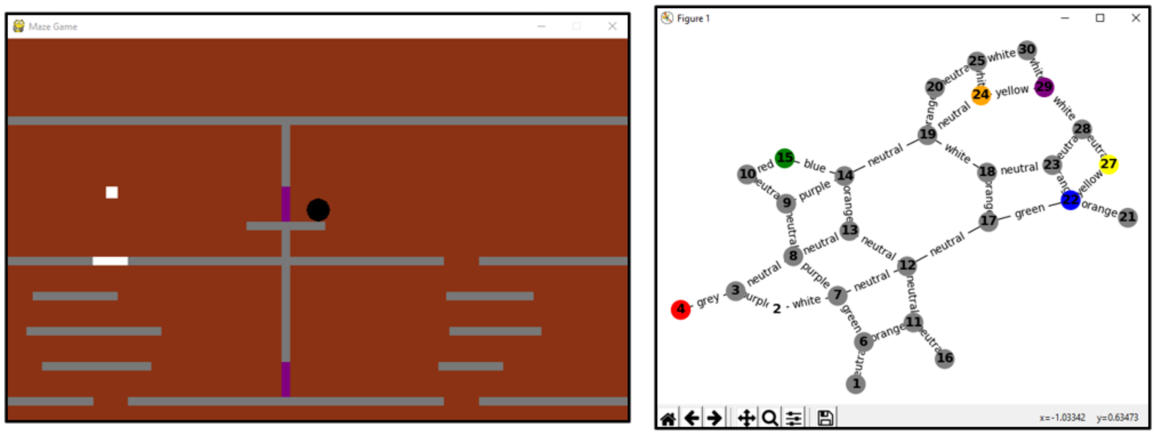


Figure 4.6: Rendered Game View and Underlying Graph

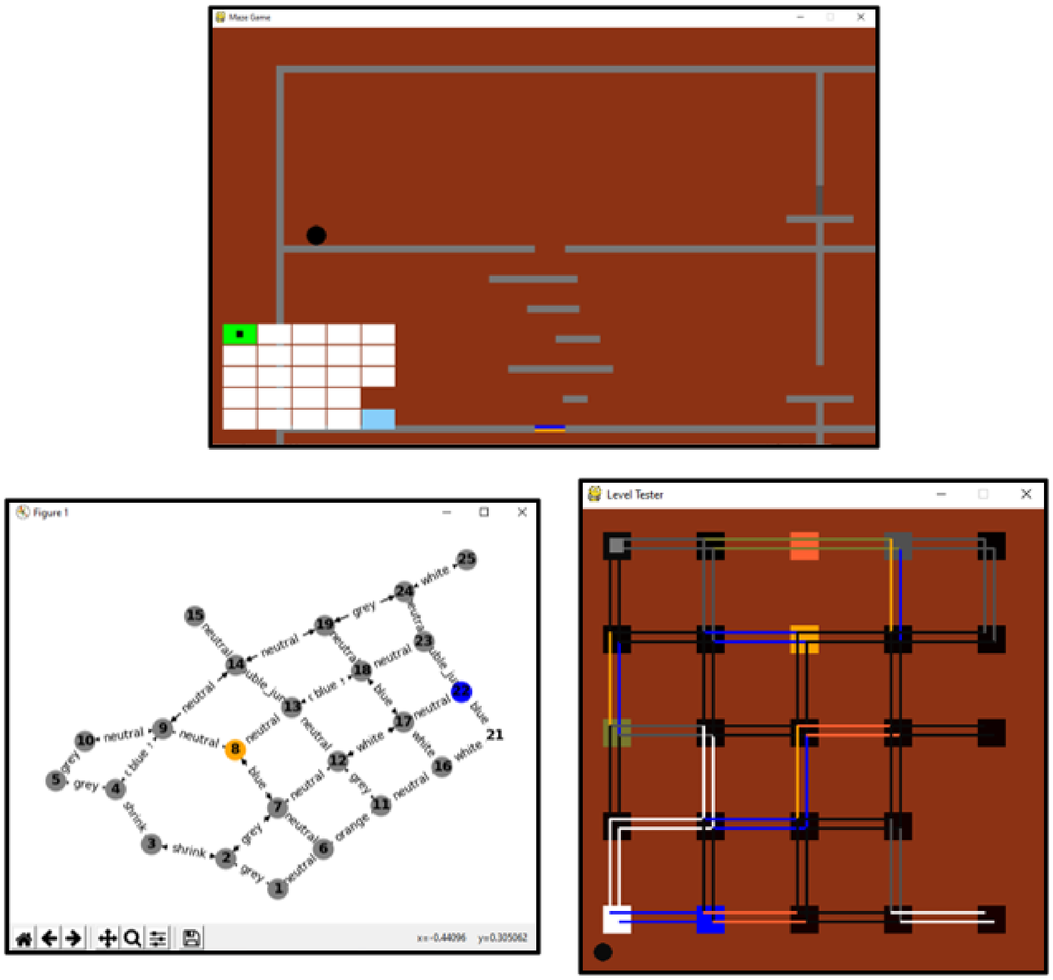


Figure 4.7: Generated Game View, Level View, and Underlying Graph (5x5)

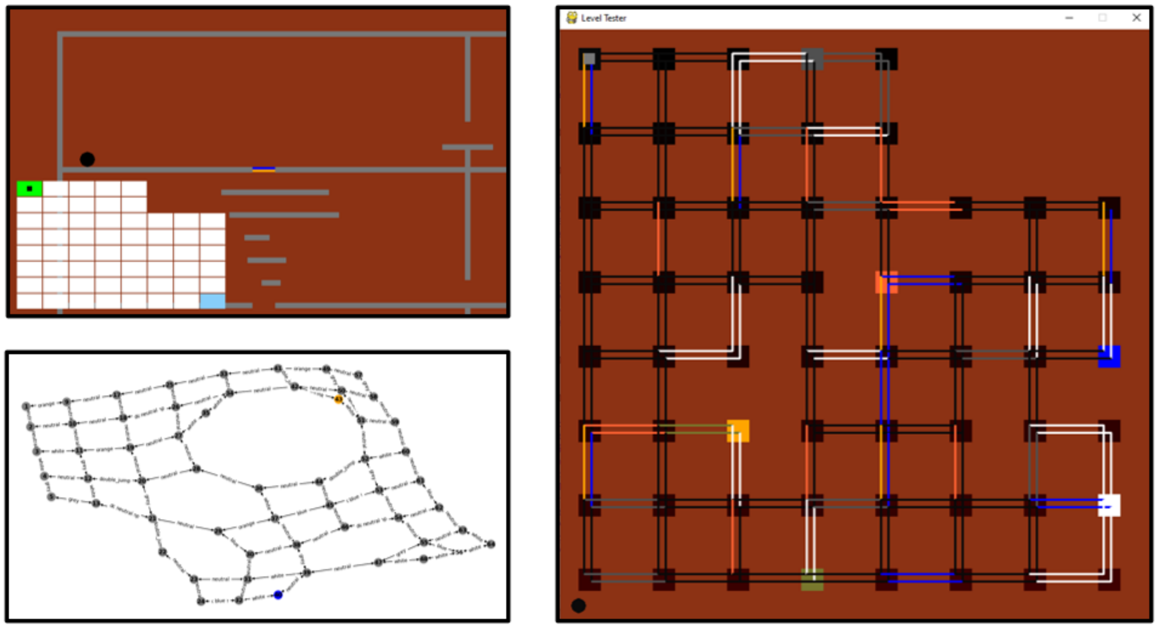


Figure 4.8: Generated Game View, Level View, and Underlying Graph (8x8)

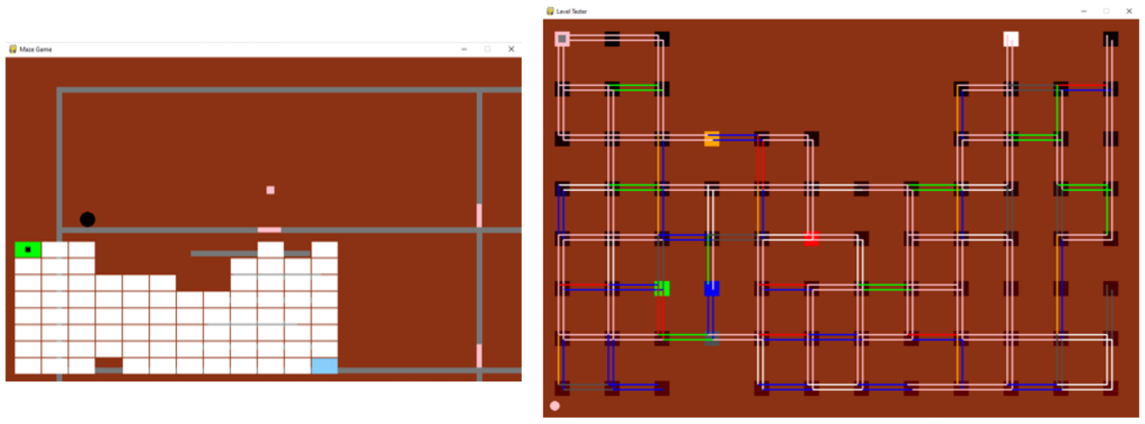


Figure 4.9: Generated Game View, Level View, and Underlying Graph (8x12)

Chapter 5

CONCLUSION

A gating taxonomy was created using the popular Metroidvania *Castlevania: Symphony of the Night* as a reference due to its position as a keystone in the genre [4]. This taxonomy provides invaluable insight into the key components that make up the gating of Metroidvanias, and to our knowledge is the first of its kind. Using this taxonomy we were able to determine which gating techniques should be included in our final product, and which could be added in the future.

As has been shown, it is possible to generate original Metroidvania-style levels using simple principles of graph theory and other mathematical processes. Full maps of varying size can be created with customizable start and end positions. These generated maps can be rendered into a variety of playable forms. The mappings can also be saved and loaded by the player for future use and reference. Additionally, the level designer can specify a key ordering, which is strictly enforced in the generated final product. This ordering doesn't have to be linear in nature, but can resemble a directed acyclic graph of possible orderings. But although the maps can be generated conclusively, it is still debatable whether the generated maps are on par with those that are of human design. Human experience user studies will be necessary to reach a conclusion on this front, and is the next step in this research.

Chapter 6
FUTURE WORK

6.1 Improved Generation of Maps

By employing more intelligently designed algorithms, instead of relying on randomness and checking, we can increase the speed by which levels are generated, without losing variety.

6.1.1 More Intelligent Generation

Currently, the gating techniques associated with the edges between nodes are randomly assigned. While it's not pure randomness, and there is a bit of thoughtful design to the assignment, a better method intelligently assigning values to the edges surely exists. Such an improvement would not necessarily increase map quality, but should decrease the amount of time needed to generate levels. Theoretically, designers could also more readily create larger maps. We were able to generate maps of 15 rows by 15 columns using the current implementation, but it took a few minutes to process. There probably isn't a need for maps larger than a 15 x 15, but the added ability to scale would still be ideal.

A loss of variety is one of the main issues with intelligently selecting placements of the gating techniques. With the random solution, any number of different mappings is possible. These mappings are then verified to meet the strict level constraints. If the gate placement process isn't designed with proper thought and insight, then some of the more obscure or unique combinations for mappings could be lost. Additionally, placing the gates proves to be a more difficult challenge than just verifying that they are correct.

Designs and rough outlines for such an algorithm were drafted towards the end of this project. The proposed algorithm closely resembled the verifying step, but in reverse. We were unable to create a fully functional version of this algorithm within the time constraints of this project. Given other bugs and issues that were of a higher priority, improving the generation of the maps was tabled as future work.

6.1.2 Improved Dropout Rates

The generation of sparse maps is a rather rare event. It seems that this is a side effect of the way in which the maps are randomly generated and then verified for correctness. It stands to reason that sparser maps have a lower probability of being winnable, and are thus less likely to occur in the final generations. The same logic applies for maps that are incredibly dense with connections. If one connection is out of place, then the whole map is trash. Therefore, if the more intelligent gate assignment described above was implemented, it would also follow that a richer variety of maps could be created with regard to sparseness.

6.2 Adding Other Gating Types

As this project neared its conclusion, multiple distinct gating types became supported by the game demo. By its nature, the underlying networkx structure already supports any of a variety of gating types. This is due to the fact that the gate type is stored along edges as a string. Therefore, for the data structure, there is no discernible difference between 'red' and 'double_jump.' Representing these classifications is where the true work lies. It is the onus of the programmer / designer to specify how each distinct gate type should be displayed to the player. For example, shrink gates need to be represented with smaller entrances and exits than typical gates. Automating this process is impossible, since the number of gate types is limited solely by the designers imagination.

Enemies and mobs were one of the interesting categories of gates we would have liked to have included. These gates would have made use of NPC characters. Having to kill a boss before moving to the next room would be an example of such a gate. The rules for these gates would need to be considered, and may actually require a few tweaks to the underlying graph structure as well. How many connections should be closed in a boss room? Puzzles would be another interesting gate type to include. The nontrivial nature of these gates would also require some thought to properly implement.

Given more time, combination gates would be the next gating type supported by the demo. Rather than being a distinct category of gate, this would represent a combination of two or more distinct gates. For example, green double jump or red shrink. Undoubtedly an addition like this would have required slight modifications to how gates are represented in the underlying networkx graph and how the graphs are checked for ability to be completed.

6.3 Increase Non-Linearity of Key Collection

Designers can provide a directed acyclic graph structure containing all of the constraints for key collection. Currently, this structure is then flattened into a linear progression of gating techniques that obeys the rules specified by the designers. The process for this flattening is discussed in section 3.3.2.1. This means that at any point in the game there is only one key that the player can collect. This doesn't have to be the case, and in many fully-fledged Metroidvanias it isn't. Small adjustments to how the keys are placed and how the map is generated and verified could allow for this improvement. It should be possible to respect the limitations imposed by the designer while also adding some variation in game play. For example, that the player could acquire either the red key or the single jump key, within the same explorable region. When one of the keys is collected, the explorable zone would also expand as usual revealing more keys.

6.4 Improve the Mini Map

The mini-map was created as a convenience tool for the player and demo tester. Its aesthetic design and use took a backseat to its functionality. That is to say that the mini-map works, but it might not work in the most ideal of ways. For example, the size of rooms on the mini map is not relative to the size of the overall map. This means that with a world size of 15 x 15, when the mini-map is displayed, it covers the entirety of the screen. As a result, it is near impossible for the player to navigate the world while the mini map is being displayed. This should be corrected in part by confining the size of the mini map and scaling the sizes of the room representations accordingly. Even when the map size is reasonable, it can still be hard for the player to move around the world while the mini-map is on screen. Adding transparency to this layer, allowing the player to see both the mini-map and what's underneath it could be beneficial. This functionality was ultimately relegated to future work, since the mini-map itself is a tangent to the actual project at hand.

6.5 Improve the Graphics of the Game

This improvement would be purely cosmetic and promises to add little to the project. The aim would be making the game demo look and feel more like an actual Metroidvania game. Improvements would include, but are not limited to, textures for surfaces like walls and platforms, an actual sprite with animations, key sprites, and a more visually appealing end room. Ultimately these additions would be nice, but would only be justified if we were moving towards user studies.

6.6 Increase Variability

The generated maps currently resemble a grid. Every room has the exact same dimensions. Allowing rooms to have varied widths and heights would be a minor addition, but one that would make the maps feel more unique. In order to ensure that rooms still lined up properly, that is that there exists a gate between two rooms, each row would have an associated height and each column an associated width. This would result in the same sort of grid structure, but one which isn't so obviously generated as such.

6.7 Human Level Designer Access

It is currently near impossible for a designer to make manual edits to a generated map. If this project is to serve as a real world tool allowing game developers to easily create levels for their games, then an ability to manipulate the generations is essential. Such a system would most likely focus on editing the representations displayed by the game demo. Using this generation as opposed to the underlying networkx template would allow the generator to do more work for the developer. A click and drag graphical user interface would work well for this feature. Obviously, functionality such as this is a project in its own right and was thus left as future work.

6.8 User Interface for Designers

There are various toggles and parameters which the developer can set when generating custom maps. It is easy to forget how, where, or why to include some of these arguments. It therefore seems apt to have an intuitive user interface that would allow designers to customize their settings. Fields for start room, end room, weighted neutral, and others could be displayed with helpful tool tips. A visual representation of the gate-key directed acyclic graph could be displayed letting the designer see how the gating rules will be enforced. The UI could also serve to prevent poorly chosen configurations, those which would hinder or prevent map generation. Ultimately, while this feature would be nice for a final product being sent to market, for the scope of this project, it fell into the bells and whistles category.

6.9 User Studies

Finally, there is no point in procedurally generating Metroidvania style levels if the end results are inadequate. While levels can be generated, the question still remains, are they comparable to human designed levels. The only way to answer this is by conducting user studies. Since the goal of this project was to determine if Metroidvania levels could be procedurally generated, it falls outside the original scope to ask if the generated levels are on par with those that are human designed. However, since we now know that levels can be generated, this seems the next logical step.

BIBLIOGRAPHY

[1] Álvaro Gutiérrez-Rodríguez, Carlos Cotta, and Antonio J Fernández-Leiva. Deep evolutionary training of a videogame designer. *EVO* 2019*, page 6.

[2] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[3] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013.

[4] Alex Huhtala and Paul Davies. Castlevania: Symphony of the night. *Computer and Video Games*, 192:76–77, Nov 1997.

[5] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: computer-assisted game level authoring. 2013.

[6] Toni Minkkinen. Basics of platform games. 2016.

[7] Michael Nitsche, Calvin Ashmore, Will Hankinson, Robert Fitzpatrick, John Kelly, and Kurt Margenau. Designing procedural game spaces: A case study. *Proceedings of FuturePlay*, 2006, 2006.

[8] Pygame. Pygame documentation. https://www.pygame.org/docs/.

[9] Alvaro Gutiérrez Rodrıguez, Carlos Cotta, and Antonio J Fernández Leiva. An evolutionary approach to metroidvania videogame design.

[10] Justin Pusztay Trevor Stalnaker. Squirrel simulator. https://github.com/pusztayj/squirrelSimulator.