6-9-2020 2:30 PM

A Hybrid Approach to Procedural Dungeon Generation

Mathias Paul Babin, *The University of Western Ontario*

Supervisor: Dr. Michael Katchabaw, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Abstract

This thesis presents a novel approach to the Procedural Content Generation (PCG) of both maze and dungeon environments. The solution we propose in this thesis borrows techniques from both Procedural Content Generation via Machine Learning as well as Constructive PCG methods. The approach we take involves decomposing the problem of level generation into a series of stages which begins with the production of macro-level functional structures and ends with micro-level aesthetic details; specifically, we train a Deep Convolutional Neural Network to produce high-quality mazes, which in turn, are transformed into the rooms of larger dungeon levels using a constructive algorithm. For our dungeon's micro-level details, we use a context-free grammar for the instantiation of interactable puzzle elements, and an n-gram model for decorating our dungeon's entrance rooms. This unique combination of methods successfully generates a large number of visually impressive game levels without compromising on any desirable PCG metrics such as speed, reliability, controllability, expressivity, or believability.

Keywords: Procedural Content Generation, Video Games, Machine Learning, Evolutionary Strategies, Context-Free Grammar, N-gram Model

Lay Summary

This work presents a method for generating video game maze and dungeon levels. We refer to the production of any video game music, graphics, levels, or rules by a computer algorithm as Procedural Content Generation (PCG). Many popular video games today rely on PCG in order to lower development costs through a reduction in the number of artists and level designers needed for projects as well as increase player satisfaction through vastly more substantial replay value. In terms of the system presented in this work, we use a novel combination of PCG methods found within academic literature such as machine learning models as well as human-controllable algorithms inspired by games found in the commercial games industry. The advantage to using a blend of various PCG strategies is that it allows us to carefully select when and where each method is used in order to leverage their respective strengths while simultaneously circumventing their inherent weaknesses; In particular, our aim was to develop a system which can ensure the playability of each newly created game level, as well as maximize how fast the generator can produce levels, how much control the developers have over the generator's final output, how many different levels the generator can produce, and how well the final product can fool a player into believing that it was designed by a fellow human as opposed to a computer. A system which possesses all of these desirable traits is extremely important if we wish to have commercial game developers adopt the approaches we are providing, as they are the ones driving the video game industries' ever-increasing relevance. We view this work as a step towards expanding the possible set of practical methods both game developers and PCG practitioners have at their disposal by demonstrating a novel PCG system which is capable of generating a nearly infinite number of distinct 3D game levels.

Contents

List of Figures

List of Tables

List of Appendices

List of Acronyms

ANN Artificial Neural Network 34

BFS Breadth-First Search 34, 39

BFT Breadth-First Traversal 37

BNF Backus-Naur Form 14

BSP Binary-Space Partition 5

CFG Context-Free Grammar 13, 14, 26, 50, 51, 55

CMA-ES Covariance Matrix Adaptation Evolutionary Strategy 9

DCNN Deep Convolutional Neural Network . . . 2, 6, 9, 10, 13, 17, 26, 30, 31, 33, 36, 38, 44, 52–55

DFT Depth-First Traversal 37, 38

ES Evolutionary Strategies 2, 6, 25, 26, 33, 52, 53, 55

GAN Generative Adversarial Network 6, 9, 10, 25, 26, 31, 33, 44, 53, 54

GE Grammatical Evolution 14

LoZ The Legend of Zelda 6, 8, 11, 21, 25

ML Machine Learning 1, 4, 6, 8, 33, 55

PCA Principal Component Analysis 8, 9

PCG Procedural Content Generation 1–8, 11, 13, 15, 29–32, 44, 51–53

PCGML PCG via Machine Learning 2, 4–8, 15, 16, 25, 29–31, 52, 53, 55

SD standard deviation 45, 49

Slerp spherical linear interpolation 44

SMB Super Mario Bros. 6, 8, 9, 14–16, 25, 28, 44, 55

WFC Wave Function Collapse 12, 13, 54, 55

Chapter 1

Introduction

Over the past several years, the influence of the electronic video game industry has experienced unprecedented growth. In the United States alone, 65% of adults play video games, with 75% of households reporting at least one gamer amongst them [1]. Based on global sales figures [35], we can see that the interest in video games is a global phenomenon. As of 2019, the video game industry generated over $120 billion in global revenue and is projected to reach over 200 billion by 2022. These figures have surpassed the combined revenue of both the film and music industry as early as 2016. With an industry this large, it should come as no surprise that the development of commercial video games has become highly competitive, leading both small- and large-scale studios to search for methods that reduce development time and costs, while simultaneously aiming to improve the game's quality and overall player satisfaction. This has led to the emergence of several new fields of research which focus on the improvement of games and their development.

One of the most popular and well-publicized roles for video games in academic research is to provide an environment for the study of AI models and methods. As a recent example, significant advancements in Machine Learning (ML) have been made by researchers at *OpenAI* and *DeepMind* training computer-based agents to play games such as *DOTA 2* [37], and Go; and while the importance of video games as applications for AI models and methods cannot be overstated, it is equally important to recognize the value these methods can bring to the development of the video games themselves. In this thesis, we will present how methods from various disciplines of Computer Science can be used for the procedural generation of video game dungeon environments, and as such, view this work as one which contributes to the emerging field of Procedural Content Generation (PCG) in games.

1.1 Procedural Content Generation in Video Games

PCG refers to the generation of any game content, both functional and non-essential, through an algorithm that requires limited to no human input [28]. It can be used to generate both functional game content such as levels, rules, and interactable objects, as well as non-essential elements such as music, textures, meshes, and narrative. The latter of these examples are considered to be "non-essential" as the removal of these elements from any popular game genre would still render them as playable in some diminished capacity; however, removing of any of

the game's functionally critical elements, such as the map or world, would compromise fundamental aspects of gameplay, thus leaving it in an unplayable state. In today's commercial game development, PCG is commonly used to lower development costs through a reduction in the number of artists and level designers needed for a project as well as increase player satisfaction through vastly more substantial replay value. Academic research in the field has contributed solutions to many of these problems, but also explores how PCG can generate content which adapts to the player's behaviour [20]. This work will be contributing to the most common form of PCG found in both academic research and modern commercial game development: the generation of novel game environments. In particular, we will be presenting a multi-stage approach to the procedural generation of both maze and dungeon environments. Our solution draws inspiration from methods found in both PCG via Machine Learning (PCGML; [33]) as well as constructive PCG [28], which are two fields that have garnered the attention of both academic researchers and commercial game developers.

In the first of our four stages, we use a Deep Convolutional Neural Network (DCNN) to produce a wide variety of simple mazes trained using Evolutionary Strategies (ES; [26]). ES is an alternative to backpropagation for the training of deep neural networks and is found to be useful when the problem involves non-differentiable elements in the network's architecture or loss function. We use this network to produce a corpus of simple maze levels, which in turn are used for constructing more complex dungeon-like environments in the following three stages, the first of which is an algorithm that builds these larger structures by chaining sections of these mazes together, while the second is a generative grammar which is responsible for instantiating simple puzzles within them. Finally, we populate a dungeon's entrance with decorative objects using a probabilistic model trained on a small corpus of hand-decorated levels. This model was inspired by the n-gram probabilistic language model, which is often used to predict the next word in a sentence; however, in this work, it is used to predict a sequence of characters representing a horizontal strip of decorative tiles.

While it is common for a piece of PCG literature to only focus on one of these four approaches, we deliberately chose to explore a combination of methods in hopes of addressing several of the PCG evaluation metrics outlined in [28]: *speed*, *reliability*, *controllability*, *expressivity*, and *creativity*. In our view, the most critical facet of our problem statement is the requirement for our system to satisfy all of these metrics without compromise; moreover, we believe that this is a difficult task to accomplish without making use of several techniques spanning the lexicon of PCG literature. The following sections of this chapter will introduce these metrics as a precursor to the definition of our problem statement and the motivations behind it; we will conclude this chapter with the novel contributions of this work as well as a brief outline of the contents and structure of the thesis remaining.

1.2 Desirable Properties of a PCG System

Before specifying the exact problem we are attempting to solve with this work, we must first have an understanding of the properties a procedural content generator can possess and how these specific properties manifest themselves while defining most PCG problems. Consider a scenario that calls for content to be generated during gameplay, at its core, this problem is asking for a generator to possess the property of fast generation speeds as a fundamental

aspect of its design. [28] outlines several desirable metrics that are used to evaluate procedural generators which include:

- *Speed*, referring to how fast a generator can produce its content.
- *Reliability*, describing how consistently a generator can produce high-quality content. When referring to the generation of game levels, high-quality content typically refers to environments which are both beatable, and provide a sufficient level of challenge.
- *Controllability*, specifying how much control is allotted to an external algorithm or human designer over the resulting content. This quality is useful in systems that need to adapt to the player's behavior dynamically.
- *Expressivity*, describing the diversity of content produced by the generator.
- *Creativity/Believability*, referring to how well a generator's content resembles that of a human designer; this may refer to the placement of aesthetic elements as well as the clever functional design of interactable gameplay elements such as puzzles.

As a clarification of language, when we refer to these metrics as a property, we are simply indicating that a generator's performance captures that specific metric sufficiently well. Most of the contributions made to this field have tackled problems which allow for trade-offs between these desirable properties to be made; for example, problems that focus on only maximizing reliability and creativity will often lead to solutions which forgo speed as a consequence, often relying on methods such as genetic algorithms to generate high-quality content at the cost of long generation times. This trend of sacrificing one aspect of a generator's performance in order to maximize another's is common amongst PCG research [28], and leads to some important questions which must be answered during the formation of our problem statement, such as which properties are best to try and capture in our solution? Is it possible for a procedural content generator not to make any concessions and simply possess all of these properties at once? The most significant component of this work's problem statement will involve outlining which of these properties must be considered while designing our system as well as the motivations behind why these properties must be captured for our solution to be deemed meaningful in respect to existing PCG literature and commercial applications.

1.3 Problem Statement

The general problem being addressed in this work is the procedural level generation of dungeons and mazes; however, we are also undertaking the task of designing a PCG system which possess all of the desirable attributes outlined in Section 1.2. This means in order to define the entirety of this problem, we must not only state each of these properties as they pertain to both our generator and its resulting levels but also the specific gameplay elements and structural features that constitute the dungeon environments themselves. For the purposes of this thesis, we will specify our dungeons as containing exactly one entrance room stemming into a series of hallways and puzzle rooms which house a modest range of interactable elements including keys, pressure plates, weights, and enemies. For these dungeons to be considered feasible,

there must be at least one valid solution that requires the navigation of all the rooms generated.

With this specification of a dungeon in mind, we state that the purpose of this work is to present a solution to the problem of designing a modifiable procedural level generator for the production of a diverse range of high-quality dungeon environments in an online context.

In terms of PCG problems, the most ambitious aspect of this specific problem is the necessity to capture all of the desirable properties we have discussed so far; specifically, the property of *speed* will be necessary for delivering online content generation which, in the context of PCG literature, does not refer to the production of content in a networked environment but rather to content that is generated during gameplay. The desire for modifiability refers to the *controllability* of the system; *expressivity* is captured in the need for a diverse range of content, and both *reliability* and *creativity* in the high-quality dungeons which are expected from our solution. We acknowledge the subjectivity of what it means for a generated artifact to be of "high-quality", so to both clarify the term as well as place it within the context of this work, we consider any game level which displays an aesthetic consistency in its visual components along with a reasonable degree of non-random construction amongst its interactable game elements as high-quality; of course, any level which does not offer the player an opportunity to beat it successfully, that is, it offers no valid solutions, is considered to be both infeasible and not of sufficient quality.

1.4 Motivations of our Approach

The problem of generating game levels finds itself to be a popular subject amongst academic researchers due to the breadth of valid solutions that span the many disciplines of Computer Science. PCG methodologies can be classified into four major categories, including search-based, solver-based, constructive, and PCGML methods [28, 33, 41]. These categories help organize the wildly varying PCG solutions to the same general problem of procedurally generating game levels; however, what we also see from these different categories is their tendency to display certain generative properties in lieu of others—the alleged trade-offs made between the desirable properties of a generator. Generally speaking, search and solver-based methods produce high-quality content at the expense of slow generation times due to their method of production involving a search through content space for levels which maximize a fitness function in the case of search-based methods, or a set of rules and constraints in the case of solver-based methods [28]. PCGML attempts to alleviate this issue by confining the optimization process to a training phase, which by its end, yields a fully trained model that can instantaneously generate levels. The issue with the levels produced by these ML models is *reliability*, as there is no guarantee that it is actually playable without performing an additional evaluation phase.

The most apparent commonality shared by these three categories is their generate-and-test approach to producing content which involves a two-step loop that first has the system produce a set of candidate levels before assessing them via a quality measure such as a fitness or objective function; conversely, constructive PCG involves producing content within a fixed number

of steps while assuming that any generated artifact will be of sufficient quality. Techniques in constructive PCG have a tendency to rely on highly controllable and predictable modes of generation such as generative grammars, or Binary-Space Partition (BSP) trees. A major drawback to their mode of generation is their lack of flexibility. It is often the case that the constructive method designed to generate levels for a specific game cannot be used for another; for example, Chapter 3 of [28] describes how dungeons can be easily generated using a BSP tree, and while this solution works well for dungeons, it is difficult to imagine how it might be used for any other genre of game.

While academic contributions are still being made to all four of these major categories of PCG, the commercial games industry almost only makes use of constructive PCG methods; specifically, some of the most popular games, which feature procedurally generated levels, use a nearly identical constructive method. The environments of *Spelunky [22]*, *Deadcells [23]*, and the towns of *Minecraft [24]* are all comprised of many small human-authored templates systematically placed by an algorithm. What makes this approach to level generation interesting is that it does not suffer from the aforementioned lack of flexibility as almost any game environment can be represented using a collection of pre-authored segments; however, the issue that arises with the use of these templates is the mechanism for their authorship, which in most cases, is a human designer. In general, the presence of any human-authored content in the resulting artifacts of a generator goes against the purpose of using a PCG system in the first place. This issue aside, an important observation is that this method successfully manages to simultaneously possess all of the desirable qualities we are concerned with as generation times are fast and reliable, content is believable and of high-quality, and a fair degree of controllability can be achieved through the specific template-placing algorithm being used; in fact, one would be hard-pressed to find many modern commercially-released video games which features procedurally generated levels that do not attempt to capture all five of these quality metrics. From this, we draw the conclusion that for any solution we present to the general problem of procedurally generating video game levels to be acceptable within a real-world application, we must strive to develop an approach which does not compromise on *speed*, *reliability*, *controllability*, *expressivity*, or *creativity*; furthermore, we are motivated to present a solution which is not only feasible for commercial use but also broadens the scope of practical PCG approaches which finds itself circumscribed by that of constructive methods by also integrating ideas from PCGML.

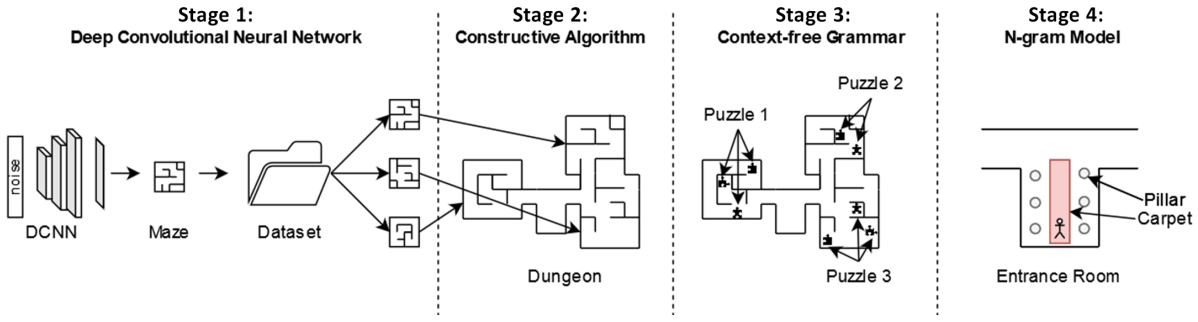

Figure 1.1: An overview of the four-stage approach to dungeon generation presented in this work.

This ultimately leads us to the four-stage approach taken in this work, where a pair of PCGML methods in Stages 1 and 4 bookend a pair of constructive methods in Stages 2 and 3. Figure 1.1 presents a diagram of the specific methods used in each stage, as well as the content they responsible for generating. Stage 1 is responsible for generating small dataset of maze environments using a DCNN. Stage 2 uses a constructive algorithm to build a high-level dungeon topology using the mazes generated in Stage 1. Stage 3 uses a context-free grammar to place puzzles in each of the dungeon's maze rooms. Finally, Stage 4 uses an n-gram model to place decorative elements in the dungeon's entrance. In the later chapters of this thesis, we will further explain how each stage of our solution complements the others such that any inherent weakness in one is supplemented by the next.

1.5 Contributions

While we cite *speed*, *reliability*, *controllability*, *expressivity*, and *creativity* as being crucial to the design of any commercially-viable procedural level generator, we find that many works in modern PCG literature do not seem to concern themselves with this idea; instead, many of these works experiment with the novel application of new models and methods presented in other fields of Computer Science with the most recent example being the application of Generative Adversarial Networks (GANs) as a means of procedural level generation [9, 33, 39]. In this thesis, we make use of [26]'s ES as the training method of choice for our Stage 1 maze generator. To our knowledge, there has been no other works in PCGML which has used this optimization method in the training of their ML models, making this the first work to do so. In Chapters 3, 4, and 5, we discuss how we successfully use this method to overcome most of the issues experienced by GANs in the production of video game environments; in particular, the use of this method allows us to involve game-playing agents directly in the training process of our maze generator. This is important when we are attempting to solve what we view is a major problem faced by many PCGML methods including GANs, which is their process of extracting key structural features from sample environments without any consideration towards the playability of the level itself.

The second contribution we believe our solution presents is a novel combination of constructive and PCGML methods. Constructive approaches are commonly used in commercial game releases, while PCGML methods find themselves to be the primary focus of academic literature alone. As stated in the previous section, one of this work's goals is to broaden the scope of commercially viable PCG methods by demonstrating how the introduction of PCGML methods such as DCNNs and Markov models can enhance the generative algorithms used by these successful game franchises. What we find is that these two approaches to content generation complement each other well and ultimately produce what we consider to be very high-quality dungeon environments.

In order to demonstrate this work's novel approach to 3D dungeon generation, an application was developed using the *Unity* game engine. While we acknowledge that it is customary for most PCG works to present their approaches using a familiar game franchise such as *Super Mario Bros.* (*SMB*; [21]), or *The Legend of Zelda* (*LoZ*; [29]), a minor goal of this application was to test our solution on a more modern 3D environment; however, this is not to say that our solution could not be applied to these classic examples. A definition for this work's dungeon

environments is provided in Chapter 3, with a detailed description of our example game in Section 3.2.

1.6 Remaining Chapters

The purpose of this chapter was to introduce both the problem and solution being presented in this thesis, as well as the motivations behind what makes our problem of procedurally generating game levels without compromising on *speed*, *reliability*, *controllability*, *expressivity*, or *creativity* an interesting one to solve. This chapter also provided a brief overview of the important concepts discussed in modern PCG literature, as well as the most common approach taken to procedural level generation in commercial game titles. Chapter 2 will build on these concepts by analyzing the related works found in both constructive and PCGML. The discussion around these related works will frequently reference the five desirable PCG properties outlined in this introduction, as will Chapter 3, which describes the specifics of our proposed solution and hopefully justifies how each of its individual stages contributes to the success of the system as a whole. Chapter 4 focuses on the implementation details of our system, including specifics on the training process for both of our ML models in Stages 1 and 4, the details of our constructive algorithm in Stage 2, and the production rules which constitute the grammar being used in Stage 3. Chapter 5 will present the methods we use to evaluate our generator, as well as an analysis of the results. Finally, Chapter 6 summarizes the work presented in this thesis and introduces some areas of future improvement.

Chapter 2

Related Works

This chapter contains a review of current PCG research as it pertains to the topics of constructive and PCGML methods. Each of the subsections in this chapter represents one of the four stages of our solution and will present the works which are most related to that particular stage, including any background information necessary to understanding them. We will also discuss any major differences between each of the related works and our own solution.

2.1 Stage 1 - PCG via Machine Learning

Traditionally, ML models have found great success training on large datasets of texts, images, audio, and the like; however, issues arise for researchers trying to apply these models to video game levels as no two games use an identical level representation. A survey of modern PCGML methods is presented in [33], which outlines the most common map representations and training methods found in academic PCGML research. Of the games featured, *Super Mario Bros. (SMB)* is by far the most popular due to the flexibility of its level representation. As a 2D Platformer, *SMB* has the player moving linearly from left to right, allowing its representation to be easily interpreted as a graph, grid, or sequence; By contrast, Action-Adventure games such as *LoZ* are often only interpreted as a grid. Because the topic of this thesis is the generation of dungeon environments similar to those in *LoZ*, we will look for PCGML methods that lend themselves to the production of levels using a grid representation.

2.1.1 Dungeons via Bayes Nets

In [34], a method for generating *LoZ* dungeons using Bayes Nets for the high-level topological structure and Principal Component Analysis (PCA) for generating individual rooms is presented. The first portion of this work trained a Bayes Net on 38 levels comprised of 1031 rooms in order to learn the topological structure of these dungeons. The resulting network learned a variety of high-level features such as the number of rooms in the dungeon or the length of the optimal solution path, as well as low-level features such as room types and door types. A common issue encountered by all PCGML methods, which was addressed in this work, is reliability. In this case, the topology produced by the Bayes Net may infrequently produce levels that do not provide a valid solution due to missing critical room types or objects

such as keys. The authors resolve this issue by looping through all of the non-critical rooms in the dungeon and replacing the one with the highest log-likelihood of being the missing room type. With the high-level structure in place, individual rooms are constructed by simply interpolating between two rooms of the desired type found in the training set. To help decide which two rooms should be chosen, PCA was used to reduce the feature set of a room by a factor of 6, from 120 down to a weight vector of 20, followed by a k-means clustering step to ensure the two rooms selected are already remotely similar. Once again, there is no guarantee that a room generated in this manner will be playable, so another evaluation pass is performed on each of these rooms such that any critical objects such as keys are not blocked from the player. Should any room be deemed unplayable, it is simply discarded, and a new room is generated and re-evaluated in its place.

2.1.2 Generative Adversarial Networks

Both [39] and [9] make use of GANs in order to produce levels for *SMB* and *Doom* [11] respectively. GANs have become a popular machine learning approach for generating photorealistic images [14, 19, 25] and function using two DCNNs: A Generator, which produces content, and a Discriminator, which classifies whether a sample was produced by the Generator or belongs to a dataset of real examples. Typically, the Generator will accept Gaussian noise as input, and through a series of fractionally-strided convolutional layers, it will produce an image of the same dimensionality as those found in the training dataset. The training process of a GAN involves the Generator network passing the samples it produces to the Discriminator network for evaluation, and once received, the Discriminator's role is to determine if the sample is real or fake. The aim of the Discriminator is to minimize the number of misclassifications it makes, and contrary to this, the Generator's goal is to gradually learn how to produce samples which maximizes the Discriminator's number of misclassifications. By the end of the training process, the discriminator network is often discarded, and the Generator network should be able to produce novel content that resembles those found in the original dataset.

In the instance of [39], a GAN was trained to produce levels for *SMB* using a training set consisting of 173 images gathered from sliding a 28x14 window left across a single level of the game one tile at a time. The cropped dimensions of the output images from the Generator were 10x28x14, where 28x14 matches the width and height of the window ran across the original level, and 10 represents the number of tile types that can be found in a level. In order to interpret this output as a *SMB* level, the authors take a one-hot encoding for each of the tile positions, ultimately yielding a map of size 28x14. Like [34], there is no guarantee that any of the levels produced by the network will be playable, and to resolve this issue, an A* agent provided by the Mario AI competition framework [13], evaluates the level for both playability and difficulty. Finally, a Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) is used to search for the latent vectors of levels that either maximize or minimize agent-centric performance metrics such as the number of jumps performed as well as some static level properties such as the number of ground tiles present.

Like [39], [9] makes use of GANs to produce levels for *Doom*. The training set used for this work consisted of 1000 levels in WAD format, which store topological data such as the position of floor tiles and the vertexes/connecting vectors which constitute walls. These files were processed into six grayscale images used for training the Discriminator: a FloorMap,

WallMap, HeightMap, ThingsMap, TriggerMap, and RoomMap. The unique aspect of this work's approach involves training and comparing two GANs: an unconditional GAN which receives a vector of Gaussian noise as input, and a conditional GAN whose Generator receives a vector of features extracted from the training set as input. For feature selection, the authors choose features that correlated well visually with their respective levels, and were not going to be heavily influenced by noise generated by the network, ultimately settling on seven features: equivalent diameter, major axis length, minor axis length, solidarity, number of rooms, and two wall distance skewness metrics. The results for both networks show a high degree of structural detail in the image maps produced, yet no results are shown as to how these maps would be recompiled into WAD files and rendered in-game. Due to the noisiness of the resulting image maps, one would expect levels with missing geometry in the floors and walls to be frequent; moreover, unlike [39] and [34], there is no mention of how this unreliability problem could be addressed.

The three works discussed in this subsection are most closely related to the first stage of our solution which uses a DCNN to produce simple maze environments using a grid representation; in fact, our network architecture is most similar to the Generator network found in [39], and like this work, we too produce 3D images which are reduced to 2D maps by taking a one-hot encoding at each tile position of our grid. Where our solution differs the most from these three works is our approach to training. We train our network using a weighted sum of various structural properties of the maze's design and update the network's parameters using an ES. This approach means that we do not need a dataset of pre-existing maze levels in order to train our generator. This becomes especially important when working in the domain of producing video game content as other mediums such as text or images have massive datasets made available to them, datasets for video game levels are not only rare but also game-specific.

2.2 Stage 2 - Constructive Approach to PCG

Spelunky, *Deadcells*, and *Minecraft* are three well-known commercially released games which feature procedurally generated levels. The constructive method used by all four of these games is remarkably similar, as each of them simply connects a series of human-authored level templates together in order to form larger topological structures. The only major difference in their strategies is in the mechanisms which inform the algorithm on what template should be chosen.

2.2.1 Spelunky

Spelunky is a 2D action platformer developed by *Mossmouth*. In a 2013 blog post [16], the level generation algorithm used for each of *Spelunky's* levels was revealed. Generation is split into two phases: a macro-level phase which outlines the placement of specific room types and the player's expected solution path through the level, and a micro-level phase which populates each room with traps, treasures, enemies, and the like. Generation begins by populating a 4x4 grid with rooms belonging to one of four types. Each room type is defined by the location of entrances/exits along its outer walls, which will connect it to the other rooms on the grid. The algorithm follows a series of constraints to place rooms such that there is always a solution path through the level. These rooms and constraints are defined as:

- Room 0: Does not have any guaranteed exits; it will not occur on the solution path.
- Room 1: Guaranteed to contain both a left and a right exit.
- Room 2: Guaranteed to contain a left, right, and bottom exit. It will also contain a top exit if another type 2 room is placed on top of it.
- Room 3: Guaranteed to contain a left, right, and top exit.

With these room types specified, generation begins in the top row of the grid and randomly works its way to the bottom row, making sure to only place room types that align with the entrances and exits of those already placed. By this point, a complete solution path must exist through all of the rooms, and the algorithm can move on to its final step of macro-level construction, in which all unfilled positions on the grid are populated with non-critical rooms of type 0. Next, the micro-level construction of each room is accomplished by randomly choosing from a template depending on the room type; each of these templates contains both static and random elements. Each room is represented as an 8x10 grid and values can be represented by either static elements such as ladders or random ones represented by the probability that certain tile will appear: for example, a "1" for 100% percent chance a block will appear, or "2" meaning only a 50% chance. There is also a chance that a template may contain Obstacle tiles, which instantiate a 5x3 predefined structure the player will need to navigate in order to progress.

2.2.2 Deadcells

Deadcells is a Roguelike-Metroidvania developed by *Motion Twin* which handles its level generation in a very similar way to *Spelunky*, with the only major difference being in how its room templates are selected. Instead of *Spelunky's* approach to choosing room templates based on a set of rules and constraints, *Deadcells* uses a graph unique to each level which informs the PCG algorithm which room styles it should instantiate, what objects a room should contain, and how each of these rooms should be connected. This is a similar approach to [34] which first learned the high-level topological structure to *LoZ* dungeons, then on a lower-level, attempted to generate new rooms by interpolating between two human-authored examples; The only difference is that the developers of this game felt it was sufficient to explicitly declare their level topologies using a graph, then supplied a massive corpus of predesigned room templates which are stochastically placed without any alterations made to their interiors. The only other difference between this approach and *Spelunky's* is the shape and dimension of each of the room templates the algorithm can choose from. *Spelunky's* rooms are all rectangular and placed on a 4x4 grid, while *Deadcells'* can be any irregular polygon and placed based solely on the previous piece's exit location. This could potentially lead to a substantial number of conflicts, which according to [2], is resolved simply by exhaustively searching the corpus of room templates until one does not conflict with the rest of the map.

2.2.3 Minecraft

Minecraft is an open-world sandbox game that is famous for procedurally generating its entire world. Generation begins by constructing the world's base-layer geometry using 3D Perlin

noise. The next step is to add a network of caves to the world using a naive path-walking algorithm which simply tunnels in random directions, occasionally placing special rooms along its path. In the final stage of generation, the algorithm places all of the map's resources, foliage, and villages. Resources and foliage are placed randomly based on a predefined distribution set for each biome type, while villages are constructed using a scheme we have already seen in *Deadcells'*. Like the room templates in *Deadcells*, *Minecraft* uses a collection of preconstructed buildings and roads which pertain stylistically to the biome in which the town will be built; but unlike the graph used in *Deadcells*, *Minecraft* uses a simple generative grammar [30] to capture the familiar structure of a village with the following production rules:

- Town Center $\longrightarrow$ Street
- Street $\longrightarrow$ Street
- Street $\longrightarrow$ House
- Street $\longrightarrow$ Decoration
- House $\longrightarrow$ Animals
- House $\longrightarrow$ Villager

This grammar uses a Town Center as its starting symbol with Streets and Houses as nonterminals, and Decoration, Animals, and Villagers as terminals. The production rules of this grammar not only controls how the village is structured but also ensures there is an appropriate assignment of Villagers and Animals to Houses. One may notice that this grammar is capable of producing infinitely large towns by recursively calling its Street $\longrightarrow$ Street production rule; however, developers have addressed this issue by halting generation once a specific village depth has been reached, which in this case, measures how many pieces beyond the Town Center have been placed.

2.2.4 Wave Function Collapse

The Final constructive method we would like to acknowledge is a constraint satisfaction algorithm called Wave Function Collapse (WFC). [15] describes WFC as "an example driven image generation algorithm recently developed by gamedeveloper Maxim Gumin". Unlike any of the constructive algorithms we have examined thus far, WFC learns a set of constraints from a sample image, and uses them to construct similar images of any size. What makes this algorithm so exciting is its flexibility in representation, as applications range from 2D images to full 3D game environments. A recent example of this algorithm's use in a commercially released title can be seen in the game *Bad North* [3].

The WFC algorithm begins by scanning a sample gameworld such that for each game object encountered, a list of all possible neighbouring objects in each direction is recorded. This list will inform the algorithm on what objects are allowed to be placed next to one another. Next, for each position in either a 2D or 3D grid, the algorithm will attempt to select the object that not only adheres to the constraints of its neighbours, but is also the most likely to appear based on what it has observed in the original sample. The process of choosing a specific object at a given position is called collapsing, as the number of possible objects that could have been placed is collapsed down into only a single option. Once a position in the grid has been collapsed, the algorithm performs a constraint propagation step where the possible objects in all

neighbouring positions of the recently collapsed object are reduced down to only those which conform with the current state of the grid. This step functions much like a Sudoku puzzle, as once a number as been written onto the grid—it has been collapsed—all of the vertical and horizontal grid positions that intersect with it can not longer possibly be that number—you have propagated the constraint. This process of collapsing grid positions and propagating constraints continues until all positions have been collapsed or until a conflict occurs. Should the algorithm encounter a position where no object satisfies all neighbouring constraints, i.e. there is a conflict, one has to decide whether to backtrack to a point in the algorithm where there were no conflicts, or simply restart and try generation from the beginning.

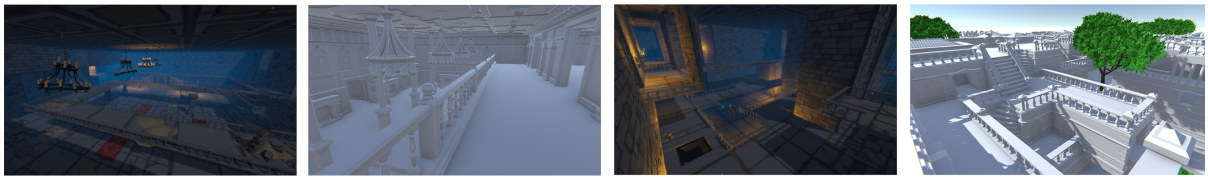

Figure 2.1: A side-by-side comparison of this work's dungeon environments and [18]'s infinite city environments.

While the generative algorithm used in the second stage of our solution does not use WFC, we include it as a topic of discussion because of the striking similarities between our dungeons and the infinite city levels produced using the WFC algorithm in [18]. Figure 2.1 provides a side-by-side comparison of one of our dungeons versus a city from [18]. From these images, some immediate structural similarities between our levels are apparent, more so than any other example we have or will discuss in this work.

And while our dungeons appear to have come from an algorithm like WFC, our actual mode of generation bears a more heavy resemblance to the system used in *Deadcells*, as we too make use of graphs to instruct our algorithm on the types of rooms and connections that should be placed. The only difference in this work is the source of the room templates being placed by the algorithm. In *Spelunky*, *Deadcells*, and *Minecraft*, large databases of content are provided by human artists, while in this work, content is being provided by the DCNN in Stage 1; This means instead of only a finite set of templates, our approach has the potential to access the full distribution of maze levels the network is capable of producing.

2.3 Stage 3 - Level Generation through Grammars

Another interesting approach to constructive level generation involves the use of generative grammars. The works of [27] and [36] both use grammars for the production of 2D Platformer levels. Both of these works use Context-Free Grammar (CFG) whose production rules consist of a combination of terminal and non-terminal symbols. For a CFG, these production rules must contain exactly one non-terminal on the left-hand side and any number of terminals and non-terminals on the right. Assuming we begin with a starting non-terminal symbol, strings can be formed by following the grammar's production rules for replacing non-terminals until none remain. This process yields a string of terminals symbols which, in the context of PCG,

can be interpreted as a 2D Platformer level by mapping predesigned level chunks to each of them.

2.3.1 Context-Free Grammars

Sure Footing [8] is a 2D Platformer which uses this approach to generate infinitely long levels. [36] outlines the details of this game's CFG, as well as the predesigned pieces placed by the system. This works novel contribution comes from a cost penalty assigned to each piece based on the difficulty designers felt the player would experience when encountering it. This cost parameter allows for control over how level generation will unfold as each piece placed by the grammar would consume a finite budget assigned for the level. This system allows gameplay to dynamically adjust to the skill of the player by simply modulating the budget to spike the difficulty of a level if a player is progressing too easily. Unlike the work presented in [27], each of this system's predesigned level chunks can be placed independently of others, meaning that there are no concerns towards the playability of any sequence generated by the grammar.

2.3.2 Grammatical Evolution

[27] presents an approach to the procedural level generation of *SMB* levels using Grammatical Evolution (GE). In this work, a CFG in Backus-Naur Form (BNF) is used by the GE evolve levels which are both playable and aesthetically pleasing. In this grammar, non-terminals require a set of integers specifying the object's x and y position as well as height h, or width w if necessary. The process of generating a level using GE involves producing a variable-length vector of integers and mapping it to a set of production rules from the grammar in a syntactically correct manner. The fitness function used to evaluate these levels is the weighted sum of the difference between the number of chunks placed by the system and a target threshold by the authors, minus the number of conflicting chunks found in the level. The authors' claim that their levels can ensure playability by restricting the height and width of specific platforms and gaps such that the player will always be able to navigate them.

2.3.3 Grammars for Puzzle Generation

In the third stage of our solution, we use a CFG for the production of puzzles. This differs greatly from these two works as their focus was on the generation of an entire level, while ours is only concerned with one small facet; this said, the motivation for using a grammar for puzzles is much the same as that of a 2D Platformer as both are solved by the player performing a specific sequence actions. In our case, we define our grammar's production rules such that puzzle elements will be dispersed amongst multiple rooms which must be visited in a specific order to have the items necessary to progress further.

Finally, we would like to briefly acknowledge [17], which presents an extensive survey of procedurally generated puzzles. Of the 32 works surveyed, only 4 appear to use a grammar-based systems [5, 6, 7, 38]. Of these works, [6] presents a system for procedurally generating quests for Massively-Multiplayer Online Role-Playing Games (MMO-RPGs) using a CFG. A majority of this paper was spent analyzing the domain specifics of MMO-based questing, with the resulting solution being a CFG in BNF. An interesting consideration made in this work is

the selection of production rules based on the state of the game such that the quest and the reward that the players receive is consistent with the scenario in which it is being presented. Ultimately, our own Stage 3 puzzle generation system takes a very similar approach to this, as it too includes a system for selecting production rules such that puzzles reflect the state of our player and the abilities they have at their disposal.

2.4 Stage 4 – Markov Models in PCGML

The final PCGML method discussed in this work is the generation of content through Markov models. Like their namesake, Markov models assume the Markov property wherein future states of a system depend only on their current state and not those which came before it. This property is used to define Markov chains, a system where the probability of transitioning from one state to another depends only on the current state. Markov models have found to be useful in probabilistic language modeling for word prediction and sentence generation. An n-gram is an *n*-token sequence of words used to provide additional context to these predictive models and can be viewed as a Markov model of order *n-1*, while others will refer to them as examples of Markov chains [12]. In the context of PCG, both [4] and [32] make use of Markov models for the production of *SMB* levels. Like generative grammars, Markov models lend themselves well to the production of sequential content, making an excellent fit for 2D Platformers.

2.4.1 Markov Chains

[32]'s approach to generating levels for *SMB* makes use of 2D Markov chains to construct the level in rows, starting in either the top-left or bottom-left corner. Their model learns several 3x3 dependency matrices where the bottom right-hand corner is always the tile being generated, and a combination of neighboring tiles are used for context. The authors of this work decided to train a total of six dependency matrices, each requiring less context than the one before it. During generation, the system will attempt to use the most contextually sensitive matrix possible until it reaches an unknown state; should this occur, the system can fall back to a simpler matrix which hopefully has learned a probability distribution for transitioning out of that state. As we have seen with other PCGML methods, reliability is a concern. Using the same A* agent as [39], 25 levels were evaluated, with only 56% of them being playable in the best case; and while the author's do claim issues with the agent's inability to complete some beatable levels, we can still draw the conclusion that this method cannot guarantee playability.

2.4.2 The N-Gram Model

The authors of [4], have decided to also generate levels for *SMB*, but have opted to use an n-gram model instead of a 2D Markov chain. For training, 15 levels from the original game are processed into a series of one tile wide vertical slices which the system uses for learning *n*-slice long sequences; that is, it obtains the unigram, bigram, and trigram counts of these slices, and uses them for the production of new *SMB* levels. The core idea behind generating a level using the n-gram model is that given n previous slices, the Markov property allows them to obtain probabilities for generating each of the next possible slices in the sequence. With this model,

the size of n dictates how much context is used for determining which slice should be chosen; for example, if n is equal to 1, the system will place slices completely at random. What we should see is an increase in the size of n correlates to an increase in reliability, but a decrease in expressivity as the system will increasingly produce levels that resemble significant portions of those found in the training corpus. For this work, the authors decided on constructing levels with values of n at 1, 2, and 3. Results for trigram-based generation show a remarkable amount of aesthetic consistency and playability. This is likely due to the fact that the early levels in the original *SMB* this method trained on are extremely simple and likely do not need more than 3 slices of content in order to navigate, thus guaranteeing playable levels in this specific case. In general, we suspect that this method's reliability could not be maintained for more complex level structures, and like all of the other PCGML methods we have discussed, it would require a separate evaluation phase in order to judge level playability.

The final stage of our solution has an n-gram model similar to [4]. The only difference being, we use our model to place decorative objects in a pre-existing level as opposed to generating a brand new level from scratch. We train our model using horizontal slices gathered from a small training corpus of hand-decorated entrance rooms, then using a trigram model, place stylistic elements in any level our system generates in the future. By choosing to only place decorative elements, we can effectively ignore any of the reliability issues typically encountered by these models.

Chapter 3

Proposed Solution

The purpose of this work is to provide a solution for the problem of designing a modifiable procedural level generator which is capable of producing a diverse range of high-quality dungeon environments in an online context. In Chapter 1, we stated several desirable properties we wish our level generator to possess, including speed, reliability, controllability, expressivity, and creativity. The goal of this chapter is to provide the rationale behind the choices made in each step of our proposed solution's design, saving many of its implementation details for the next chapter. Since this work will not be generating levels for a familiar game franchise, we will first describe the environments we will be generating in detail as a precursor to the more technical elements of our approach.

3.1 Maze and Dungeon Overview

As it is not obvious, this work views dungeons and mazes as two distinct structures. This section will clarify these two terms as well as provide a high-level overview of the dungeon environments presented in our example game.

3.1.1 Mazes

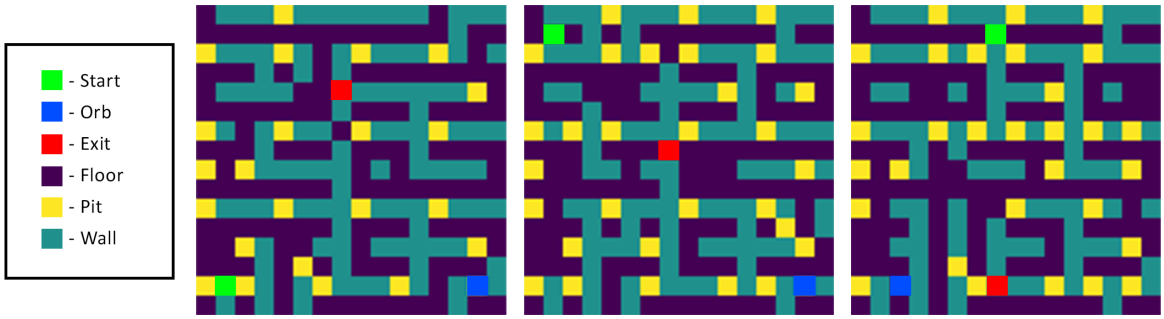

Figure 3.1: Three example mazes generated using our Stage 1 DCNN.

When discussing mazes, we are simply referring to the 2D tile grids produced by the DCNN in the first stage of our solution. These mazes are later processed in Stages 2 and 3 where they

are used as individual room templates in the construction of our larger dungeon structures. Figure 3.1 provides examples of three such mazes, with a further explanation on how these are generated in Subsection 3.3.1, and the technical aspects of their design presented in Section 4.1 of Chapter 4. And while we will not go into the details of how these mazes are designed now, we will discuss how one should navigate them as this is an important concept necessary to understanding how they are utilized when constructing our larger dungeons in Stages 2 and 3. In each of Figure 3.1's mazes, we see a *start*, *orb*, and *exit* tile, and we consider a valid solution to these mazes as one where the player begins on the *start* tile, navigates to the *orb* tile, then finally works their way towards the *exit* tile. During the construction of our dungeons, we will make use of these three critical tiles as well as the solution path which connects them.

3.1.2 Dungeons

We refer to a dungeon as the high-level structure being navigated by the players of our game. Unlike mazes, they make use of a 3D grid representation, and are composed of several smaller rooms including entrances, hallways, and puzzle rooms. Stage 2 of our solution is responsible for interpreting the mazes generated in Stage 1 as certain rooms and populating our dungeon such that they are spread across two main floors. Stage 3 further expands our dungeons by generating additional puzzle chambers that can be found on separate subfloors. The following will describe each of these room types in detail as well as provide a thorough explanation on how a dungeon's floors are defined and populated.

Hallways

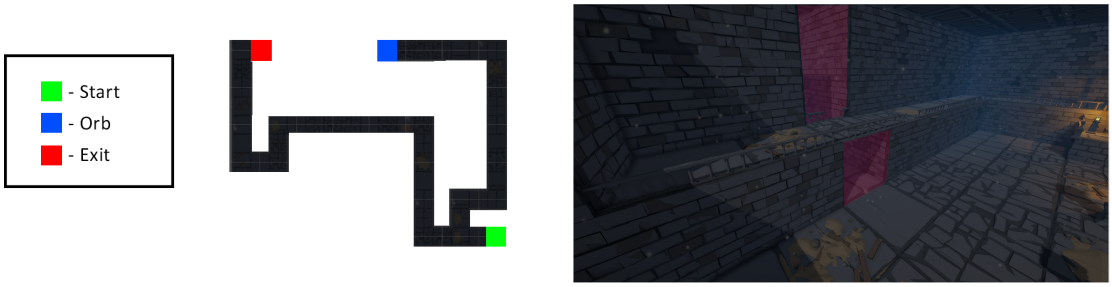

Figure 3.2: An example hallway with corresponding start/orb/exit tiles from their source maze template. The Hallway Padding p of this room is 0.

Hallways are meant to serve as connectors between the dungeon's entrance and puzzle rooms. They are constructed by simply rendering the solution path of a given source maze, resulting in a considerably linear structure; however, one may wish for a more complex room which more closely resembles that of the original maze, in which case, we provide access to a parameter we refer to as *Hallway Padding*. This value renders an additional p neighboring floor tiles away from each one on the solution path. This means for a value of p at 1, only the direct neighbors of each tile on the solution path will be additionally rendered, whereas a p of 16 would render the entire maze (assuming a source maze of size 16x16). This additional

rendering will only run until it is one tile away from conflicting with other preexisting rooms before halting. An example of this *Hallway Padding* parameter *p* at various values can be seen later in Figure 3.13 when we discuss the advantages of including this parameter in our system.

Aside from acting as connectors, hallways can support access to additional hallways and puzzle rooms, providing branching pathways through our dungeons. When deciding where a room should connect to a hallway, we will look to place them in the direction of the most available space according to the layout of the current floor's grid; that is to say, hallways do not lead to higher floors of the dungeon, and only produce access points that face north, south, east, and west.

In Figure 3.2 we see an example of a hallway room with a *Hallway Padding p* of 0, meaning that only the solution path of the source maze is being rendered as indicated by the linear route of grey floor tiles connecting the *start*, *orb*, and *exit* tiles. The right-most image shows how this hallway is connected to the first floor of the dungeon's entrance.

Puzzle Rooms and Chambers

Figure 3.3: An example puzzle room containing two puzzle chambers associated with the *orb*, and *exit* tiles of the source maze as depicted by the 4x5 rectangular rooms.

Unlike hallways, puzzle rooms represent the entirety of the source mazes produced in Stage 1. These are the only rooms in the dungeon which contain interactable elements such as puzzles, enemies, and rewards that are held in at most three separate puzzle chambers and whose entrances correspond to the *start*, *orb*, and *exit* tiles of their source maze. As previously stated, puzzle rooms occupy the dungeon's main floors while puzzle chambers are found in subfloors either above or below them. In Figure 3.3, we see an example of a puzzle room containing two rectangular puzzle chambers: the first above the *orb* tile, and the second above the *exit* tile. And while it is not seen in this example, it is possible to have a third puzzle chamber placed above the *start* tile as well, but in this case, the third stage of our system generated a puzzle which only requires two chambers.

The right-most image of this figure depicts the *exit* tile puzzle chamber which is shown to be a large rectangular room containing a single reward for the player. For this application, we arbitrarily limit these chambers to a size of 4x5 as to not have them overlap with one another should their entrance tiles be placed too close together. We acknowledge that generating small rectangular rooms is a simplistic approach to the problem of constructing these chambers, however for the purposes of housing a small set of interactable objects, a simple solution such as this does suffice.

The final difference between these puzzle rooms and those of entrances and hallways, is the number of possible rooms that they can be connected to is limited to one: they must be at the end of a hallway or entrance room. Once again, this decision only impacts our specific scenario, and is not a restriction of the methods we are proposing. It would certainly be possible to implement these structures to support multiple entrance and exits points just has hallways do, but from a game design perspective, we view the role of a hallway as that which connects several rooms together, and having puzzle rooms possibly serve the same function as hallways would diminish their usefulness and identity within our game.

Entrances

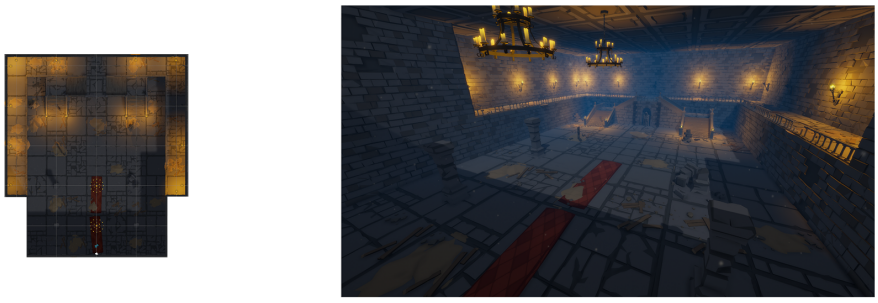

Figure 3.4: A example entrance room with no adjoining hallways or puzzle rooms.

A dungeon's entrance room serves as a starting location for the player, providing access to the dungeon's two main floors via a central staircase. In our game, we limit a dungeon to only have a single entrance room which all other rooms (hallways, and puzzles) are connected; they are also the only rooms that do not require a source maze in order to be generated, instead, we construct our entrances by simply rendering a rectangle of odd width and height such that there is a guaranteed centre line that the staircase object can be placed on. This can be seen in the right-most image of Figure 3.4. Once this first level is instantiated, we render a balcony which covers the entire north-edge of the room and runs a random length down both the west and east-edges as seen in the overhead view of the example in Figure 3.4.

In our example game, we choose to construct all of our entrance rooms on the first floor of the dungeon, with its staircase acting as the only point of access to the dungeon's second floor, and from here, we can connect a single room to the west and east-edges of each level, for a total of four possible connections. Like many of the decisions made when designing our dungeons, this could be revised to support any number of rooms, but as an initial proof of concept, we view four rooms spread across two floors sufficiently demonstrates the capabilities of our solution.

Puzzle Floors and Subfloors

During the prior introduction of our dungeons' rooms types we briefly alluded to how they are placed on various floors and subfloors. We will now conclude this section by clarifying the difference between floors and subfloors, and examine how they are spatially organized within our dungeons.

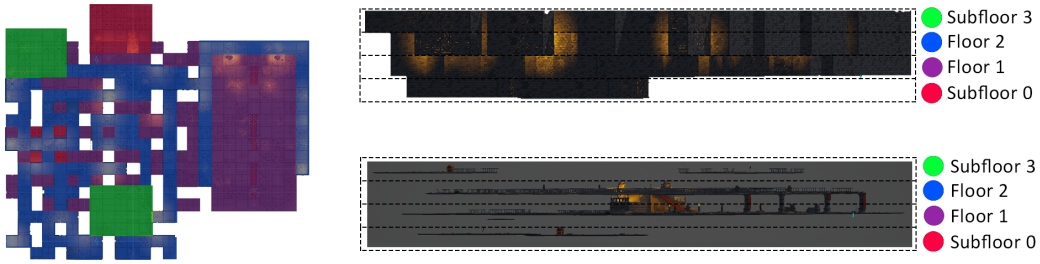

Figure 3.5: A dungeon featuring two stacked puzzle rooms, both on the west side of the entrance room. The pair of right-most images present a profile shot of the dungeon, where the top image is rendered with walls, and the bottom without.

Figure 3.5 provides an example dungeon featuring two puzzle rooms stacked on-top of one another, both connected to the west-edge of the entrance room. In this diagram, we see that floor tiles have been coloured to represented their respective floor or subfloor number: puzzle chambers on Subfloor 0 (red) are associated with the puzzle room on Floor 1 (Purple), and puzzle chambers on Subfloor 3 (green) are associated with the puzzle room on Floor 2 (blue). As a general rule, the dungeon's entrance will always be placed on the dungeon's first floor, and its balcony will always be on the second floor. Rooms such as hallways, and puzzle rooms can be placed on either of these two main floors. This leaves subfloors above (Subfloor 3), and below (Subfloor 0) for puzzles chambers. Whenever a puzzle room requires that a chamber be placed, it simply places it below itself if it is on the first floor, or above itself if it is on the second floor. From Figure 3.5, we can see how these floors and subfloors are organized by looking at the two profile shots of the dungeon.

Our motivation for using subfloors to house these puzzle chambers is twofold: the first reason was to introduce large spaces into our dungeons without compromising the layout of the source mazes used to generate their parent rooms. An additional benefit to isolating these chambers on different levels from their parents is the control we gain over when the player can access them. This feature is especially important when we are designing puzzles with sequential solutions; a topic we will discuss in-depth in the following section which focuses on the player's goals and possible interactions with our example game.

3.2 Example Game Description

This section outlines the goals, rules, and major mechanics of our example game. Here, we will be presenting both the major components and interactions within our game, as well as an exhaustive list of all of our game's interactable elements.

3.2.1 Gameplay Objectives and Mechanics

The inspiration for our example game's dungeons draws heavily from the *LoZ* franchise. As in these games, our dungeons contain enemies to combat, puzzles to solve, and treasures to collect. In order to successfully complete our dungeons, players will have to follow a pattern

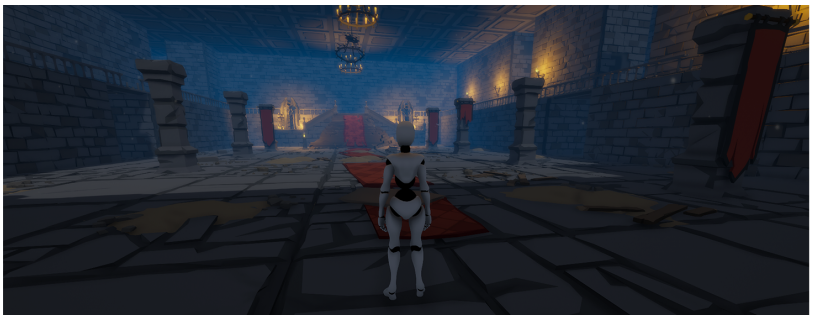

Figure 3.6: The player starting in the entrance room of a newly generated dungeon.

of exploring rooms in search of helpful items, solve puzzles, defeat enemies that impede their progress, and collect upgrades that allow them to interact with new sections of the dungeon. Depending on the length of the dungeon, the player may be engaged in a loop of this pattern for several iterations until no puzzle remains unsolved, at which point, most action-adventure style games would force the player to confront a final boss before collecting their ultimate prize; unfortunately, we have omitted any form of a boss chamber within our own game as its inclusion would only serve as a redundant application of our Stage 2 constructive algorithm's ability to place rooms such as the hallways and puzzle rooms we have already discussed.

As seen in Figure 3.6, when a player first begins one of our dungeons, they will be placed at the front of the dungeon's sole entrance room with the option to enter any of its adjoining hallways or puzzle rooms. The player's goal at this point is to explore the dungeon until they encounter a puzzle which they are equipped to solve; should the player lack the necessary upgrades to solve this room's puzzle completely, they will be forced to continue exploring until they encounter a puzzle which can be solved in their current state. Players initially start with no abilities, but can obtain two upgrades throughout their exploration of the dungeon: shoot, and magnet. The shoot powerup allows the player to combat enemies, while the magnet powerup allows the player to push and pull metallic objects such as weights. Players obtain these upgrades with each puzzle they solve, giving them the ability to progress further into the dungeon until they reach their final reward at the end of the last puzzle. We intentionally leave this final reward undefined as this item would presumably be critical to the game's narrative as one being searched for in the protagonist's overarching quest, in which our game has none.

3.2.2 Solving Puzzles

In Section 3.1, we discussed how our game's puzzles are relegated to puzzle rooms, and how many of their interactable elements are stored within separate puzzle chambers. Because we wish for the solution to these puzzles to be sequential, we restrict access to some of these chambers until others have been completed first. In essence, our puzzles simply require the player to access all of a puzzle room's chambers in a specific order. To accomplish this, we define a set of special elevator tiles we will refer to as *lifts*, most of which are inaccessible without the possession of special items such as keys, or weights which the player must find in earlier chambers. Once a lift is activated, it will carry the player to its accompanying puzzle chamber on either Subfloor 0 or 3 depending on the floor number of its parent puzzle room—

Figure 3.7: The player in front of each of our game's lift types: Normal, Enemy, Lock/Key, and Plate. Additional figures examining our game's lifts and locking mechanisms are provided in B.1, and B.2 of Appendix B.

either 1 or 2 respectively. In image *a* of Figure 3.7, we see the player standing in front of a chamber whose lift has no locking mechanism; we refer to this style of lift as a normal lift, and because these do not have any items or upgrades in order to use them, we typically reserve their use for the beginning of a puzzle when the player should not yet have any powerups in their possession.

Unlike normal lifts, lock/key, plate, and enemy lifts possess a locking mechanism which must be opened before they can be used. The lift depicted in image *c* of 3.7 requires the player to obtain a key from a previous chamber in order to remove its lock. Keys can be collected regardles of the player's state, i.e., their collection is not predicated on whether or not the player has obtained any form of upgrade, while the lifts shown in images *b*, and *d* do require upgrades in order to interact with the objects necessary to unlock them. First in *b*, enemy lifts require the player to destroy all of the enemies guarding it before it will be activated; the ability to fight these enemies requires the shoot upgrade and as a consequence, puzzles involving these lifts cannot be solved until the player has discovered this powerup as a reward at the end of a different puzzle room. Likewise, in *d* we see a plate lift, which requires the player to drag weighted metal spheres onto pressure plates using the magnet upgrade before it will be activated.

3.2.3 Battling Enemies

Battling enemies is a common feature is most action-adventure games. In our game, we include three enemy types: Tower, Shield Tower, and a basic Cube enemy. The player can only combat these enemies once they have obtained the shoot upgrade, at which time, they will be able to lock onto enemies and shoot at them using small orange projectiles. In image *c* of Figure 3.8, we see the player engaging a Cube enemy. The orange circle encompassing this enemy serves as a visual indication that it being targeted by the player for attack. This style of enemy is used to guard chambers, and as previously discussed, can be destroyed in order to unlock a certain style of lift.

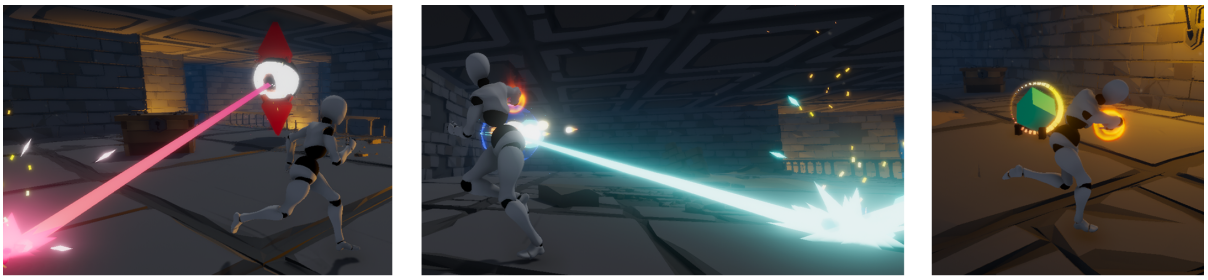

Figure 3.8: The player engaging our game's three enemy types: Tower, Shield Tower, and a basic Cube enemy. An additional figure examining our game's enemy types is provided in B.3 of Appendix B.

The remaining two enemy types are Towers and Shield Towers as seen in images *a* and *b* of Figure 3.8 respectively. Tower enemies shoot a laser from an eye that tracks the players movement. Upon destroying a Tower, the player is rewarded with weight object which can be moved using the magnet upgrade. A variation of the Tower enemy is the Shield Tower enemy, which features a protective barrier that makes it invulnerable to attack, and as such, cannot be destroyed in order to obtain a weight. We include this additional enemy type simply as a means of injecting some variety into our pool of potential enemies, and from a design perspective, this enemy serves as a persistent obstacle the player must be careful to avoid due to its indestructibility.

3.3 PCG Approaches to Mazes and Dungeons

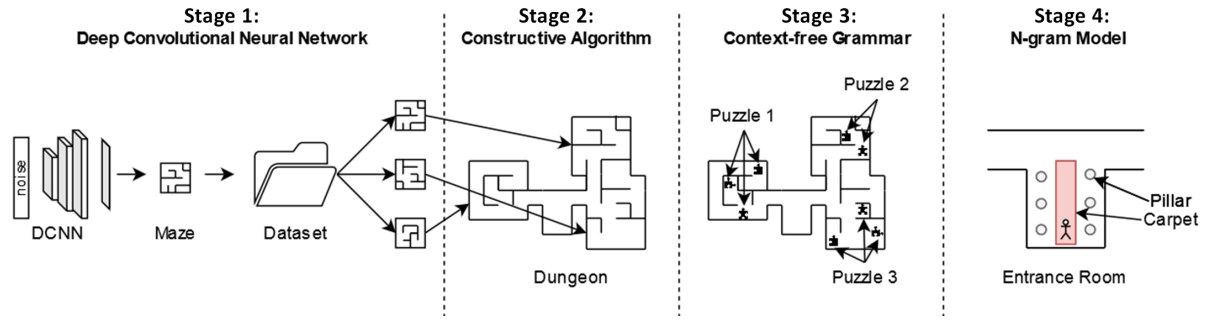

Figure 3.9: A reproduction of Figure 1.1 from Chapter 1: An overview of the four-stage approach to dungeon generation presented in this work.

With our example gameplay scenario outlined, we now provide the conceptual details of how we plan to generate our dungeons, reserving its technical details for Chapter 4. We provide a reproduction of Figure 1.1 from Chapter 1 in Figure 3.9 to serve as a visual aid during the discussion of our four-stage approach.

3.3.1 Stage 1 - DCNN for Maze Generation

While a majority of our related works discussed the generation of 2D Platformers, specifically *SMB*, we saw in the production of *LoZ* dungeons, how dungeon levels could be represented using a grid [34]; furthermore, we saw how GANs are a popular choice for the production of any game level which uses this representation, making them a logical starting point for the generation of our environments. Most PCGML methods excel at quickly producing a wide-variety of high-quality content, yet struggle with properties such as reliability, and controllability. We have already seen an excellent example of controllability in [36] which presented an approach to dynamically controlling the difficulty of 2D Platformer levels by assigning a cost penalty to the pieces being placed by a generative grammar. Luckily, GANs can also address this issue of controllability by mapping their output to points in latent space. We saw in [39] how an ES was used to search latent space for levels which maximized a certain criteria set by the authors, and in theory, one could use this same approach to gather a set of latent vectors which produce levels with any number of properties we wish to reproduce during gameplay. This can be accomplished by either interpolating between two latent vectors such that their individual properties are mixed together or by adding a small amount of Gaussian noise to a single vector in order to obtain a random level with properties similar to the original. The solution in this work does not take this approach, instead, leaving it as an area of future improvement.

The approach taken in this work saves randomly generated mazes along with their optimal solution paths into a finite dataset. As we will discuss shortly, this decision was originally made to help the runtime of our Stage 2 constructive algorithm; however, it also comes with the consequence of hampering the expressivity of the maze generator in this stage as we are now only storing a finite set of mazes instead of having direct access to its entire distribution. This trade-off between expressivity and speed extends into the decision of how many mazes should be stored in our dataset. A large number of samples would yield greater expressivity at the cost of longer execution times, and conversely, a smaller dataset would produce faster runtimes with less variation in the types of levels being produced. Ultimately, we choose to use an extremely small dataset of only 16 sample mazes as we choose to prioritize short generation speeds over a wider range of content. Fortunately, results presented in Chapter 5 show even with an extremely small amount of maze templates, the expressive range [31] of the entire system is remarkably high.

The last important aspect of our maze generator is how we train it. All of the PCGML methods examined in this work learn level representation from an existing dataset; however, we have discussed in Chapter 2 how these datasets are both rare and game-specific, and from an industry perspective, hiring level designers to create a corpus of levels for the purpose of training a level generator is a mostly redundant exercise. Now, turning our attention to the process of training a GAN, we find that the discriminator is evaluating these levels as if they are images, looking at whether or not one resembles those found in the training set. While this approach could prove logical for evaluating a level's aesthetic properties, but in regards to its structural features, it would be more useful if the discriminator could assess the level by actually playing it. In order to accomplish this, we would likely need to replace the discriminator's objective function with one that involves a gameplaying agent. The problem with this theorized approach is that many of its elements may be non-differentiable, a major problem if we are planning to train the network using backpropagation. From this, we arrive

at a solution wherein we abandon the notions of both a training dataset and a discriminator network (and thus the training process of GANs entirely), instead opting for an ES as a means of network optimization. Originally presented in [26], ES is an optimization method that can train a network's parameters using a fitness function regardless of whether or not it is differentiable, making it the training method of choice for our maze generator.

3.3.2 Stage 2 - Constructive Algorithm for Dungeon Generation

With our method for generating our mazes in place, we can now discuss possible approaches to expanding these rooms into larger dungeon structures. In Chapter 2, we discussed how our second stage of generation was heavily influenced by the one used in *Deadcells*, in that we use a graph-informed algorithm to connect a series of pre-authored room templates together. At the time, we also mentioned that the only significant difference in these approaches was the room templates sources; *Deadcells* used human-designed rooms, while ours are generated by the DCNN in Stage 1. We argued in that chapter as well as this one that our approach has the advantage of direct access to the full distribution of mazes produced by this generator, and while this is true, we also alluded to the fact that storing a finite set of levels may actually be beneficial for this stage of generation. Once again, the motivation for this decision stems from the desire to maintain reliability. Our logic is as follows, given that each graph has atleast one valid configuration of rooms, a finite set of rooms can be exhaustively searched in a fixed amount of time such that, in the worst case, the only map generated by that specific graph is its one valid configuration; however, issues arise when we substitute the finite set for an unknown distribution of rooms from our DCNN, as we can no longer guarantee that a valid combination of random mazes will yield a valid solution within a fixed number of steps; in other words, performing an exhaustive search across a finite set is much easier than over the entire distribution of a DCNN's output. With all of this considered, we trust that there are satisfactory solutions to this issue, and will discuss this topic further in the future improvements subsection of Chapter 6.

3.3.3 Stage 3 - Context-Free Puzzle Grammars

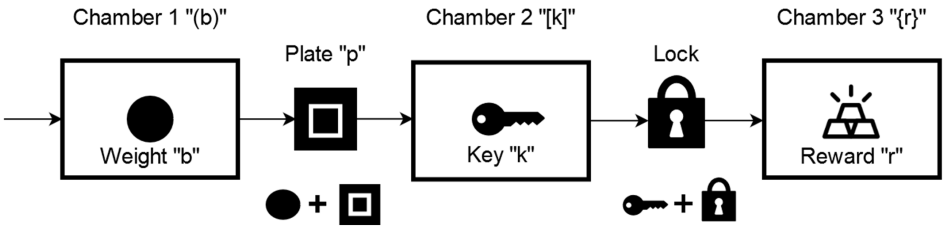

Figure 3.10: A puzzle represented by the string "(b)p[k]r". Weight *b*, key *k*, and reward *r* are contained in puzzle chambers 1, 2, and 3 respectively. Pressure plate *p* is combined with weight *b* to unlock Chamber 2, and key *k* is combined with the lock on Chamber 3's lift.

For the generation of our interactable puzzle elements, we choose to use a CFG due to their flexibility and reliability. Given only a small number of production rules, grammars allow for

the formation of a relatively large range of puzzles. For our game, we use a grammar comprised of 20 production rules which is capable of generating 99 unique puzzle strings—a complete list of these rules is provided in Chapter 4, Section 4.3. In general, we limit our puzzles to a maximum of three steps, with each interactable puzzle element isolated to its own unlockable chamber. As discussed in Section 3.1 of this chapter, this limitation of having a maximum of 3 puzzle chambers per puzzle room is a result of an implementation detail to align each of them with the *start*, *orb*, and *exit* tiles of their respective source maze; however, it is important to recognize that this a self-imposed limitation of our specific scenario and not a consequence of using a grammar, as a different set of production rules could be capable of producing puzzles of any length.

Figure 3.10 provides an example of a puzzle represented by the string "(b)p[k]r". This string describes a puzzle with a unique three step solution: First, the player starts by finding an unlocked chamber which contains a movable weight "b" as described by the substring "(b)". Next, "p[k]" describes a pressure plate "p" which unlocks a chamber containing a key "k" using the weight from step 1. Finally, the player unlocks the last chamber "r" using their recently obtained key from step 2 and receiving their reward "r" for this segment of the dungeon. As shown through this example, we denote interactable objects such as keys, weights, and pressure plates using lowercase terminal symbols and use well-formed parathesis to signify the beginning and end of a chamber as well as its locking mechanism. In general, we wrote all of our production rules such that any puzzle string generated would follow no more than four replacements. This means that all of our rules are simply variations of the same generic template:

- S $\longrightarrow$ a(t)S'
- S' $\longrightarrow$ b(B)c(C)
- B $\longrightarrow$ t
- C $\longrightarrow$ t

Here, terminals "a", "b", and "c" represent objects that are bound to their respective chamber's locking mechanism, such as pressure plates or enemy guards. We use "t" to denote any set of objects that can be obtained within a chamber, including weights, keys, or treasures.

Although [17] presented many possible approaches to procedural puzzle generation, only a hand-full were stated to be useful in an online context, and amongst these works, the amount of time taken for their systems to arrive at a solution appears questionable by this work's standards for an online generator. For example, [40] presents an approach involving the use of a genetic algorithm to generate puzzle levels for a game called *KGoldRunner* [10] and states that their solution can "find a dynamically solvable level in a matter of seconds [...], and a good level can be found in a few minutes." For this work, we will only consider online generation times as those who are no longer than several seconds as any more time than this would begin to negatively impact the player's experience. We choose to use a context-free grammar for this portion of our work because it is a fast, reliable, and controllable method of generating puzzles. We can guarantee speed and reliability due to the fixed number of replacements in our generic four-step production rule template, and achieve a reasonable degree of controllability through grouping our rules based on the player's state. In this game, the player has access to two major abilities: shoot, which can damage enemies, and magnet, which can push and pull magnetic

objects such as weights. By simply restricting which production rules can be chosen during generation, we can guarantee that the solution to our puzzles will only require power-ups that the player already has access to.

3.3.4 Stage 4 - Markov Models for Dungeon Decoration

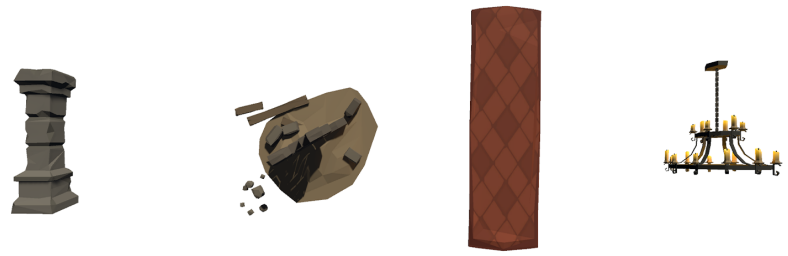

Figure 3.11: Four examples of the decorative elements placed in Stage 4 of our solution: pillars, rubble, carpets, and chandeliers; only pillars, rubble, and carpets are placed directly by Stage 4's Markov model, chandeliers are simply placed directly above carpets.

The final stage of our solution uses a 3rd-order Markov model to generate decorative elements for our dungeon entrance rooms. We chose to only decorate entrances as these rooms will provide the player with their first impressions of our levels. The model presented in this section could certainly be extended to other rooms of the dungeon and will be a topic of future improvement. The model we use in this stage could be viewed as a trigram model trained on horizontal slices of hand-decorated entrances in a method similar to [4]. In order for this model to not impact level playability, we only allow it to place objects which do not impact the player's navigation of the level, such as pillars, rubble, carpets, and chandeliers. Figure 3.11 provides examples of each of these objects. Objects such as pillars and rubble are designed such that they do not take up an entire tile as to not impede the player's ability to traverse the level.

A major difference between our use of Markov models, and those found in our related works, is the context-sensitive nature in which new slices are being placed. For example, [4] generated *SMB* levels using an n-gram model to generate vertical slices of tiles learned from the original game. In this case, generation always started with an empty level and began to populate it from left to right. And while our approach is very similar to this, we begin generation with a map that is already built, making the model responsible for choosing decorative slices that not only fit within the context of the previous two (a trigram) but must also conform to the undecorated level topology already presented. This issue is resolved by first calculating the probability distribution of all potential slices regardless of the context in which they will be placed, and then with all of these potential slices found, we construct a second distribution which only includes the slices which conform to the current topology in which they would be placed.

Like the Markov models discussed in [4], our model decorates our dungeon's entrances by stacking rows of tiles starting at the bottom of the room. Using a similar naming convention

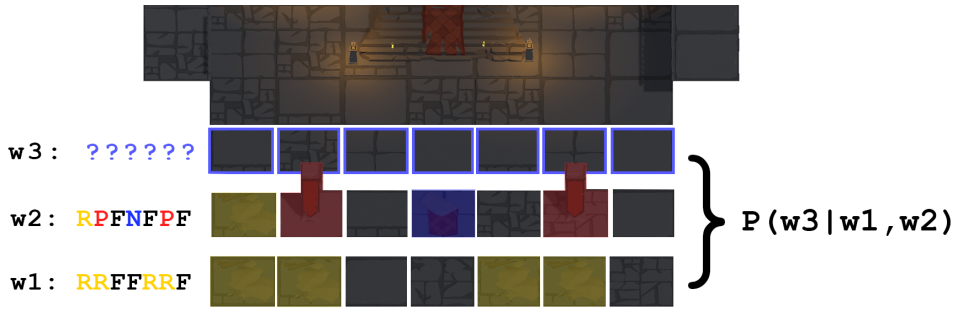

Figure 3.12: An example of how a horizontal slice *w3* is chosen based on the previous slices *w1*, and *w2* using the conditional probability *P(w3 |w1,w2)*. Rubble tiles *R* are highlighted in yellow; pillar tiles *P*, red; carpet tiles *N*, blue; while floor tiles *F* remain unhighlighted.

as [4], we will refer to these rows as horizontal slices. In Figure 3.12, we show how the third horizontal slice *w3* is chosen using the conditional probability *P(w3 |w1,w2)*. In this example, empty floor tiles are represented by symbol *F*, pillars by *P*, rubble by *R*, and carpets/chandeliers sharing symbol *N*. For convenience, the length of the horizontal slices used in this example is 7 to match the width of the room. In addition to a thorough explanation of how this conditional probability is calculated in Section 4.4 of Chapter 4, Section 5.4 of Chapter 5 will show experimental results for varying horizontal slice lengths.

We view many of the methods found in PCGML as those which excel at generating content that structurally resemble those found in a training set and yet understandably struggle at capturing the playability aspect of levels, making them excellent candidates for the generation of purely aesthetic content. Of these models, we selected the n-gram model because of its overall simplicity. As our results in Chapter 5 will show, we obtain very visually pleasing results using a model that takes only several seconds to train.

3.4 Desirable Properties in our Solution

We will conclude this chapter with a summary of the methods used and the desirable PCG qualities possessed by each. We would like to draw attention to how each stage of our solution complements the others such that any inherent weakness in one is supplemented by the next.

3.4.1 Speed

Speed is an attribute that must be possessed by all stages of our solution such that a bottleneck in generation times does not occur. In our discussion of our Stage 3 grammar, we stated that any generation time longer than several seconds would not be considered acceptable as any amount of time spent beyond that would negatively impact the player's experience with the game. We acknowledge that this statement is both ambiguous in terms of duration as well as ignores any consideration towards system hardware specs; however, any further precision given to an acceptable amount of generation time will still be purely qualitative, and any minor discrepancies in performance observed from variances in modern personal computer hardware

should be negligible. We say that our system successfully possesses this quality by choosing PCG methods that avoid a generate-and-test approach to content generation. Typically we would require methods from PCGML to undergo a testing phase to check for content playability, however, we strategically chose to use these methods in scenarios that do not require these secondary evaluation phases and thus do not inhibit our generation times.

During the first stage of generation, we use our DCNN to produce maze rooms, storing them in a small dataset for use by the constructive algorithm in Stage 2. This stage is obviously performed offline, and therefore will have no impact on the system's speed. As we discussed early in this chapter, the decision behind this finite dataset allows for an exhaustive search to be performed by our constructive algorithm in Stage 2. Because this search is guaranteed to halt after a finite number of steps, and having only observed relatively short execution times during the gathering of our results, we would say this stage too possesses the quality of speed. Similarly to this algorithm, Stage 3 is a constructive method with a finite number of steps. This stage of our solution uses a context-free grammar to produce puzzles and guarantees its fast production times by limiting its number of non-terminal replacements to a maximum of four. Our final stage is another PCGML method that does not require the use of a time-consuming generation/evaluation loop. In this stage, we use an n-gram model to place decorative elements in our dungeons' entrance rooms, and because these elements are only decorative, we do not need to worry about running any form of evaluation phase to test whether or not they will impact the level's playability.

3.4.2 Reliability

Speed is a quality that often suffers in the pursuit of reliability, as many common methods for ensuring the playability of a level require a generate-and-test loop to search for sufficient content in an indeterminate amount of time. With the exception of Stage 4, we require all of our other stages of generation to be reliable as each will impact the level's topology and interactable elements. During the production of the dataset in Stage 1, we do perform an evaluation phase that tests each maze generated by the DCNN for playability, discarding any level which does not contain a valid solution path. The reliability of this stage carries into Stage 2 which is careful to stitch these mazes together into larger structures without compromising their integrity as playable structures. In short, we accomplish this by including the entirety of the selected maze's solution into the hallway or puzzle room it is generating, and therefore, maintain its playability.

Stage 3 guarantees valid puzzle solutions through the production rule template outlined in Section 3.3.3 and by carefully considering the terminal symbols we place in each rule. During the definition of a new production rule, we are careful to only select terminal symbols that represent objects that lead to exactly one valid solution. As an example, we are careful to only ever add as many weights to a puzzle as is necessary such that the final chamber of our solution cannot be accessed prematurely.

Finally, because the last stage of our system is introduced for purely aesthetic reasons, it does not impact the player's ability to complete the dungeon, and as a result, we are not necessarily concerned with any failures it experiences; that said, we still want for our system to perform consistently well as to not impact believability. To achieve this, Chapter 5 outlines various experimental parameters such as slice length L and the size of the contextual window

n which appear to produce visually appealing entrances on a reliable basis.

3.4.3 Controllability

While not as important as a quality like speed or reliability, we would like as much controllability over as many of our stages as possible. Between all of the methods found in PCG, PCGML methods are the toughest to control, while constructive are amongst the simplest. During our discussion of GANs in Section 3.3, we stated that the fact that a possible control mechanism for this model may involve the use of latent vectors to control the specific outputs we receive from the system, and this is indeed the case. During this discussion, we also mentioned that the DCNN in the first stage of this solution does not take this approach, and instead, simply stores randomly generated levels. One reason why generating content based on a combination of latent vectors does not make sense for this specific work, is the lack of distinguishable structural features being produced in our mazes. Unlike the faces shown in [25], the mazes produced in this work are only of size 16x16, and as such, are too small to contain many identifiable structural characteristics. This makes it extremely unlikely that the interpolation between the latent vectors of two mazes would yield many interesting results.

Fortunately, our other PCGML model in Stage 4 has more opportunities for interaction. As described above, our n-gram model has two adjustable parameters: slice length *L*, and the size of the contextual window *n*. Adjusting either of these two parameters radically changes the results of the system, yet disappointingly, in ways which do not always add to the overall aesthetic benefit of the level. For example, all of the hand-decorated levels in our training corpus feature a series of carpet tiles running down the center of the room. As the size enough for *L* decreases, multiple horizontal slices per row begin to appear, each misplacing carpet tiles outside of the room's centerline. Although we could technically define these parameters as control mechanisms, there appears to be very few scenarios in which we would ever adjust their values. Experimental results for changes made to these parameters will be presented in Figure 5.5 of Chapter 5.

Figure 3.13: Example mazes generated using the same control graph, with Hallway Padding value *p* at value of 0, 1, 2, and 5 (from left to right).

Unlike these PCGML methods, our constructive methods give us excellent control mechanisms that vary based on the level being generated or the player's progression through the dungeon. First, designers have control over the navigational difficulty of hallways through the Hallway Padding parameter, which for sufficiently large values of *p*, can make the dungeons more closely resemble sprawling labyrinth structures as seen in Figure 3.13 with the dungeon generated using a *p* value of 5.

Finally, the generation of our dungeon topologies is controlled by a graph. This graph dictates how many rooms appear in the level, what type of room is being placed, and how these rooms are connected to one another. In addition to this, each maze room can accept a list of production rules used for generating its puzzles as well as the reward the player receives for completing it. This means that designers can define a progression order to their dungeons as a whole, making sections of the dungeon unsolvable until the player obtains the reward at the end of another puzzle room.

3.4.4 Expressivity

Expressivity is a measure that is best captured by the structural variance of our mazes and dungeons in Stages 1 and 2 and not necessarily by puzzles and decorative elements in Stages 3 and 4. This is because the solution to many of our dungeons' puzzles require the navigation of the maze room in which they are held, and thus most of the game's challenge and interest is expressed by the complexity of the room's solution path.

Our decision to limit our pool of generated mazes down to a small finite set, should greatly impact the expressivity of our constructive algorithm as it will often choose the same mazes repeatedly in order to construct our dungeons. Luckily, results in Chapter 5 show how the expressive range of our Stage 2 dungeon structures is remarkably diverse, despite only having access to such a small pool of mazes. Expressive range [31] is the most common metric used by PCG practitioners to capture the expressivity of a system. This metric plots two variables of a level on a heatmap to visualize how diverse a set of randomly generated levels can be across those two variables. Chapter 5 presents the expressive range for both our maze generator in Stage 1, as well as the dungeons produced in Stage 2.

3.4.5 Creativity/Believability

Creativity/Believability is a metric that describes how well a system's content resembles that of a human designer's; that is to say, it is a metric that speaks to the overall level of quality in the resulting content. Of the quality metrics we have discussed so far in this work, creativity is mentioned the least; this is because it is a metric that is qualitative and difficult to design methods of evaluation for. Commercial titles such as those discussed in Chapter 2, all feature some form of human-authored content. Specifically, in the case of *Deadcells*, the game's lead designer admitted that human-authored room templates were included to ease the rising skepticism of their PCG system from fans of Metroidvania style games, who favor the meticulously designed level typically featured in this genre. With such an apparent appetite for believable levels, we feel an obligation to address it in some regard, especially because it is a trait that is not often considered by many other academic works in the field. Our inclusion of a final decorative pass in Stage 4 is an attempt to raise the believability of our entire solution by delivering a significant improvement to the overall aesthetic appeal of our entrances, as these will be the player's first impression of our dungeons. Finally, an aspect of training our maze structures in Stage 1, includes a penalty to wall tiles placed without any adjoining neighbors. This penalty was introduced to promote more coherent wall structures that would eventually constitute the entirety of our dungeon's final topology.

Chapter 4

Methods and Implementation

This chapter focuses on the implementation details of our system, including specifics on the training process for both of our ML models in Stages 1 and 4, the details of our constructive algorithm in Stage 2, as well as the production rules and symbols which constitute the context-free grammar being used in Stage 3.

4.1 Stage 1 – Maze Production

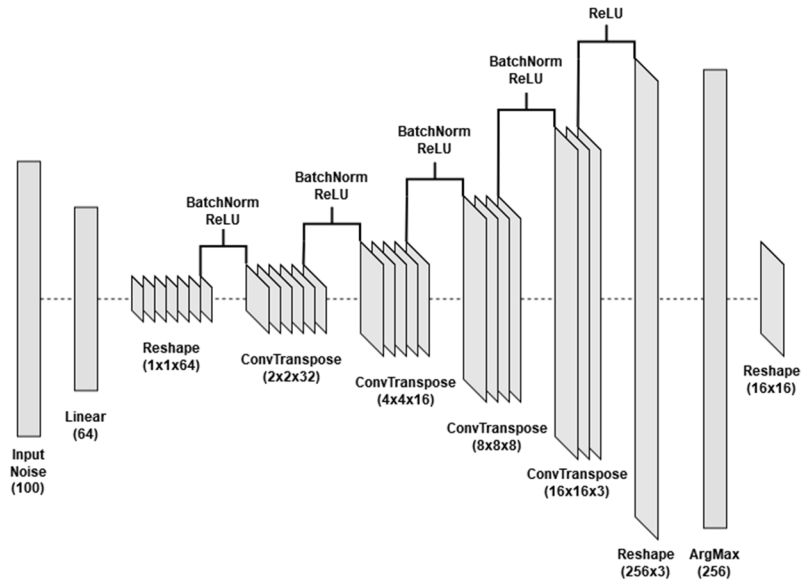

Figure 4.1: The network architecture used for producing mazes.

Production of mazes in this stage involves training a DCNN using an ES. Figure 4.1 provides the network's architecture, which is inspired by the generator network found in GANs. The first layer of this network accepts a vector of 100 random samples gathered from a distribution of Gaussian noise with a mean of 0 and a variance of 1. Next, a linear transformation is performed down to a layer size of 64. This step is for the convenience of the fractionally-strided convolutional layers which double in width and height but halve in depth. We will take

this time to acknowledge that the parameters in this first linear transformation are likely not learning any relevant information, and future works could likely begin with a noise vector of size 64. The last of our convolutional layers is of size 16x16x3, with 16x16 representing the width and height of our mazes, and 3 representing the possible tiles in our mazes: walls, floors, and pits. We reshape our maze and perform an argmax operation to determine the type of tile in each position of our final maze. Again, it is likely that we are performing a redundant reshape in the third last operation of this network, and the network could have simply performed the argmax operation on the 16x16x3 convolutional layer. Finally, we include a ReLU activation function and batch normalization after each convolutional layer.

This network is trained using the ES presented in [26]. It describes as a black-box optimization method that optimizes a function $f(w)$ with respect to w, the parameters of a network. The authors refer to it as a black-box method because the algorithm does not need to know the specifics of $f(w)$ in order to optimize it, only that it accepts the parameters of the network as input and returns a reward r. The idea is, by finding the parameters w which optimize this function, we have effectively trained an Artificial Neural Network (ANN) to perform whatever task is being performed by $f(w)$. The specifics of this process involves duplicating the parameters of a network n times, with each being adjusted by a small amount of Gaussian noise. Next, each of the n variants of the network is evaluated using $f(w)$ with their rewards standardized to have a Gaussian distribution. Finally, the parameters of the network are updated by taking a weighted sum of the n variants such that each of these is weighted proportionally to their reward r. This means that the update which is performed on the original parameters w is most heavily influenced by the variants which performed best in terms of $f(w)$.

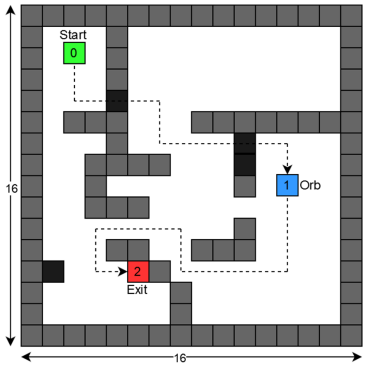

Figure 4.2: A modification of the scenario presented in [28] where an agent's goal is to first search the maze for an orb tile (2) before navigating to an alter/exit tile (3).

The function $f(w)$ in our solution generates a maze using input parameters w and evaluates it using a modification of a scenario presented in Chapter 8 of [28]. Figure 4.2 presents the modified version of this scenario used within this work. In the original, mazes contained a randomly placed start tile (0), orb/midpoint tile (1), and an altar/exit tile (2) as well as floor and wall tiles. The goal of the game was for a Breadth-First Search (BFS) agent starting on tile 0 to first navigate to the orb tile, then find a way to deliver it to the altar tile. In our modified version, we added an additional pit tile, which can be jumped across so long as there is a floor tile waiting on the other side.

To evaluate these levels more consistently, we define a set of possible positions for the start, orb, and exit tiles and create 26 combinations of them to be used for training—a list of these combinations is provided in A.1 of Appendix A. What is interesting about this approach is that these three tiles are being placed post maze generation, and therefore, the network has no idea where they may be. This means that our network learns to produce mazes which are general enough to support multiple configurations of these three tiles. The decision to train using 26 of these configurations could be viewed as a medium between having so few combinations that a likely collapse to the generator's expressivity would occur due to the predictability of these tiles' placement, and a number so large that the placement of these tiles would move towards being complete randomization; in which case, we suspect the production of trivially simple mazes due to the generator compensating for the unpredictability of these tiles' locations.

Next, we ensure a more faithful representation of the generator's performance by testing a total of 96 levels and averaging their results. We generate 32 mazes using the network and test each of them with the same 3 randomly selected start/orb/exit combinations. The idea is that we do not want to evaluate a generator based solely on a single maze, instead choosing 32 as a decent representation of its performance, however, we must also guarantee that we are not judging these mazes only on a single set of start/orb/exit positions as we expect our mazes to be general enough to support multiple combinations of these; hence why we choose test each of our 32 mazes against 3 randomly selected start/orb/exit positions.

The reward function being used to evaluate a maze is the weighted sum of three metrics: non-linearity, wall isolation, and number of pit tiles. We imagine our ideal mazes as those with a modest number of pit tiles, walls which form purposeful structures, and whose solution path is complex enough to force the player into taking a non-linear route through the maze. Ultimately, the reward we return from this $f(w)$ evaluation function is the average of the 96 rewards we calculate using the following weighted sum:

$$\textit{Reward} = 0.80\textit{*Non-Linearity} - \textit{Wall Isolation} - \textit{Number of Pits}$$

Here, non-linearity is a measure that reflects both the length of the optimal solution path as well as how non-linear it is. This metric simply sums the turn angles taken by the agent during its traversal of the level. This means that longer solution paths will have more angles contributing to the sum, and maps with non-linear solutions will provide higher degree turns, resulting in a higher score. Figure 4.3 provides an example of two sample paths, the first with an extremely linear solution with a non-linearity value of 0 and the second with a far less linear path with a value of 225.

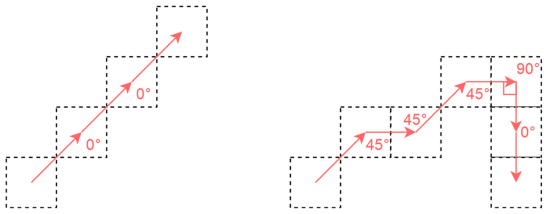

Figure 4.3: Two example solution paths with vectors indicating the direction of the agent's path. Non-linearity is measured by the sum of the angles between these vectors.

In the reward function, we attempt to maximize this non-linearity measure but found that scaling it by 0.8 allowed for the other two metrics to have more of a presence in the resulting mazes. The first metric we are trying to minimize is wall isolation: a count of the number of wall tiles with no neighbors to the north, south, east, or west of it. We penalize the reward function for each occurrence of these isolated walls with the intention of rewarding the generator for producing mazes with coherent, well-formed wall structures.

Finally, we penalize the generator for each pit tile it places in hopes of reducing the amount to a handful per maze. Of course, if this training process ran for enough iterations, we would expect to see almost zero pits placed; however, we end training before this occurs. If this became a concern, we could simply replace this metric with the distance to a target number of pits instead. Each of these three metrics is normalized to a value between 0 and 1 as to give each an equal representation in the reward function. To do this, we simply divide each of them by their maximum possible value. This is easy for walls and pits because we know that our grid contains 256 possible positions, but it is more difficult for non-linearity. Our solution to this problem was to hand-design levels with extremely non-linear paths and recorded the maximum non-linearity scores amongst them. As it turns out, our trained generator found solutions with more complex solution paths than our human-designed levels as the average reward was above a theoretical maximum of 0.8 (0.8*1 maximum non-linearity – 0 isolated walls – 0 pits).

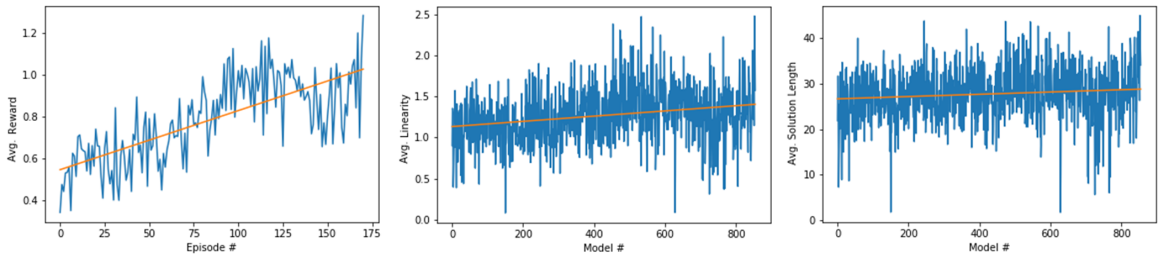

Figure 4.4: Training results from left to right report the average reward of all five network variants, the average non-linearity (labelled as Avg. Linearity) of each variant, and the average solution length of each variant.

Figure 4.4 presents results for this training process, which ran for 175 iterations with a population size of five network variants, a noise standard deviation σ of 0.1, and a learning rate α of 0.01. These results show a dramatic increase in the average reward per iteration with slight positive trends for the networks' average non-linearity and solution lengths.

The dataset of source maze levels used in the next stage of generation is comprised of only 16 maps. We organize these maps by their solution length and store the 16x16 maze as well as a list of coordinates for its optimal solution path.

4.2 Stage 2 – Dungeon Production

The graph-informed constructive algorithm used to produce our dungeons searches a small corpus of maze levels produced using the DCNN described in the previous section. The process of building a dungeon begins with defining a control graph for the algorithm to follow. We

purposely chose to restrict our generation to a maximum of three puzzle rooms per graph as this is the maximum number of rewards the player can receive in our game, these being a shoot powerup, a magnet powerup, and a final treasure for completing the dungeon. Like many of the limitations presented in this work, the decision to restrict our solution to three puzzle rooms per graph is a reflection of our specific game's design and is by no means a constraint of our proposed solution. Figure 4.6 provides example levels built using each of our three control graphs presented in Figure 4.5. In these figures, dungeon *A* appears to only have two maze rooms even though its control graph specifies three. This is an extremely rare case where two identical mazes on the west wing of the dungeon were selected, one of which is on Floor 1 and the other on Floor 2.

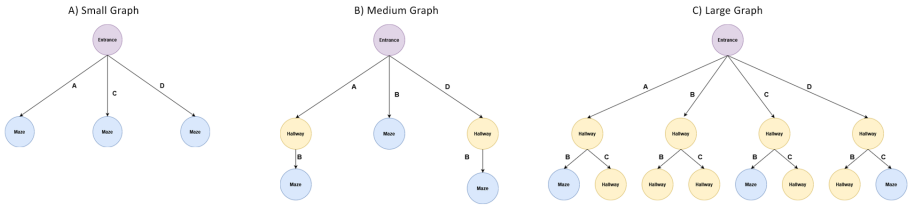

Figure 4.5: The three graphs used during the evaluation of this system. Graph *a* is referred to as the *small* control graph, *b* as *medium*, and *c* as *large*.

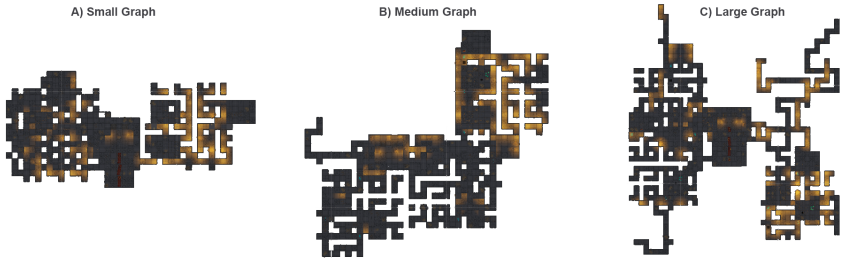

Figure 4.6: Example levels built using each of our three control graphs using a seed of 0. Dungeon builts using the small graph take approximately 15 to 20 minutes to complete; medium dungeons, approximately 20 to 30 minutes; and large dungeons, approximately 30 to 40 minutes.

While building a dungeon, the system is guided by a Depth-First Traversal (DFT) of these control graphs. We chose a depth-first scheme because if the system fails to instantiate a room after trying all possible permutations of our source mazes, the algorithm jumps back and regenerates the last room, which for a DFT will either be the parent or a sibling node in the control graph; however, if we would have used a Breadth-First Traversal (BFT), it is possible that we would jump back to an unrelated node in a completely different sub-tree than the one experiencing issues. This process of searching for room permutations differs depending on the room type; for mazes, this involves choosing a random source maze from our dataset and rotating it until its starting tile lines up with its parent's connection tile, and for hallways, this process is similar except we allow for either the start/orb/exit tiles to connect with the previous room. If the room placed is either an entrance or a hallway, we instantiate a number

of connection tiles based on the number of children nodes in the graph. For hallways, these points are found along any of the outermost floor tiles and face in the direction of most available space. While for entrances, these connection points are based on the label of an outbound edge, for example, edge *A* will place a connection point on Floor 1 of the west side of the room, while *B* is also on Floor 1 but on the east side of the room. Connection points *C* and *D* are similar, except they are placed on the second floor of the dungeon.

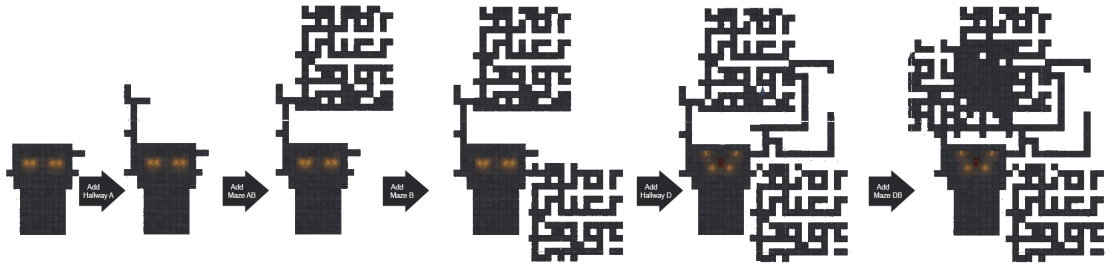

Figure 4.7: The process for building a dungeon using medium control graph *B*. The graph is traversed using a depth-first scheme, with rooms labelled using a concatenation of the edges followed to reach its respective node. For example, following edge *A* leads to hallway room *A*, then edge *B* to maze room *AB*.

Entrance rooms themselves are always the first rooms placed as they are at the root of our graphs, these rooms are always rectangular in shape, and have random dimensions with the smallest being 7x5 and the largest being 9x13. We also force the dimensions of these rooms to be odd numbers such that there is always a centerline for the staircase to follow. These rooms have the player spawn at the south end with a staircase placed at the north end. This staircase leads to a one tile wide balcony section which runs along the entire north face of the room and a random ways down its west and east faces, serving as a means of access to the upper floors of the dungeon. Figure 4.7, illustrates the construction of a dungeon using the medium control graph starting with the entrance, then proceeds down the branches of the graph using a Depth-First Traversal (DFT) starting on left most branch labeled *A*.

When instantiating hallways, our algorithm follows the solution path included in the map file. If any value greater than 0 is set for the hallway padding parameter *p*, the algorithm will also instantiate any neighboring floor tiles a distance of *p* away from each tile on the solution path. We prematurely stop the spread of hallway padding if it is about to intersect with another room. The intention behind this was to preserve the original solution path of puzzle rooms that may become compromised by a neighboring hallway. This is an artifact from an early stage of design where instead of all the dungeon's rooms being accessible from the beginning, certain wings of the dungeon would have been locked behind doors that the player would have to unlock by obtaining keys from other wings. If this were still the case, any additional routes into a puzzle room introduced by the unrestricted spread of hallway padding would have undermined the purpose of having a locked door, but with this feature never realized, more interesting dungeon structures may emerge by removing this check.

The instantiation of puzzle rooms is very similar to that of hallways as they follow the same process, with the exception that they do not simply follow the optimal solution path outlined in the source map file, but rather the entire maze as it was originally produced by the DCNN

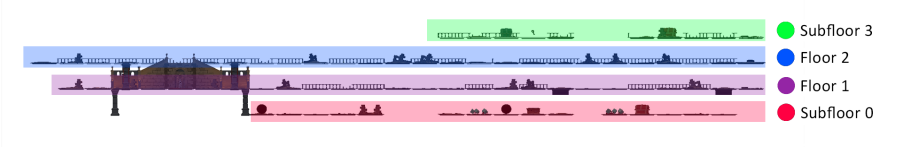

Figure 4.8: The profile of a dungeon rendered without walls to display the contents of the two floors and subfloors.

in Stage 1. For these dungeons, we chose to ignore pit tiles from the original maze generation scenario as we did not feel that their inclusion would add any meaningful contributions to our dungeons' design. Finally, as discussed in Chapter 3, puzzle rooms also include lifts to the puzzle chambers. We place access to these chamber at the start, orb, and exit tiles with the intention of having the player trace the solution path taken by our BFS agent during the creation of these original source mazes. Also mentioned in Chapter 3, in order to have these chambers not interfere with the rest of the map, we place them on Subfloors 0 and 3. Puzzles rooms on Floor 1 have their puzzle chambers placed below them on Subfloor 0, and puzzle rooms on Floor 2 have their chambers placed above them on Subfloor 3. These chambers are simple 4x5 rectangular rooms that are placed such that their bottom left-hand corner is one tile above their respective start/orb/exit tile. Figure 4.8 presents the profile of a dungeon with two puzzle rooms generated to the right of the dungeon's entrance. In this figure, the two puzzle chambers on Subfloor 3 (green) are associated with the puzzle room on Floor 2 (blue), and the three chambers on Subfloor 0 (red) are associated with the puzzle room on Floor 1 (purple).

4.3 Stage 3 – Puzzle Production

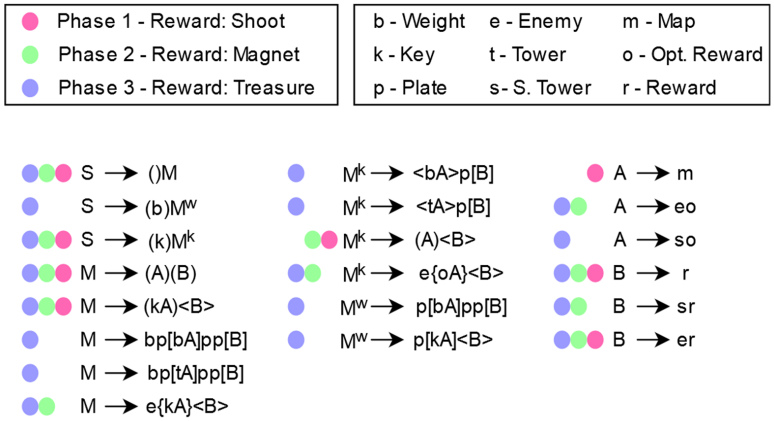

Figure 4.9: The production rules used to generate the dungeon's puzzles. The phase that each rule belongs to is marked with a coloured indicator, and a legend for the terminal symbols' corresponding in-game objects is provided. Generation begins with nonterminal symbol "S" and progresses with nonterminals "M", "M^w", and "M^k", where superscripts "w" and "k" stand for weights and keys respectively. Finally puzzles end with nonterminals "A" and "B", where the items needed to unlock the lifts of puzzle chambers "B" are provided in "A".

The placement of the dungeon's puzzle elements is closely related to the algorithm in the second stage of production as it is responsible for instantiating all the dungeon's interactable objects. In fact, whenever the algorithm successfully places a maze room in the dungeon, a puzzle string is generated using a set of production rules specified by the control graph. We mentioned in Chapter 3 that we separate our production rules into three phases that should reflect the abilities obtained by the player. The first set of production rules generates puzzles which the player can solve without the ability to shoot or move weights and rewards them with the shoot powerup; the second set introduces enemies that the player can now fight with their newly obtained shoot powerup, rewarding them with the magnet powerup; and the final set introduces weights that the player can manipulate to receive their final reward for completing the dungeon. These rules, along with their respective phases, are outlined in Figure 4.9.

As shown in the right-most legend of Figure 4.9, our puzzles include 9 interactable objects. Denoted by "b", weights are placed onto pressure plates "p" to unlock elevators denoted by "[...]". Keys "k" are used to unlock plate lifts denoted by "<...>". Enemies "e" can be defeated to unlock enemy lifts "{...}". "t" and "s" are used to represent tower enemies, and when destroyed, produce a weight object "b". Map "m" is only awarded to the player in the second chamber of phase 1, providing them with a tool for studying the dungeon's structure and the areas that they already explored. Optional reward "o" is a replacement for the map phases 2 and 3, this treasure would typically be a non-crucial reward such as currency, weapons, or potions. Finally, the left-most legend of Figure 4.9 reiterates which reward "r" is obtained at the end of each stage in the third and final chamber of each puzzle.

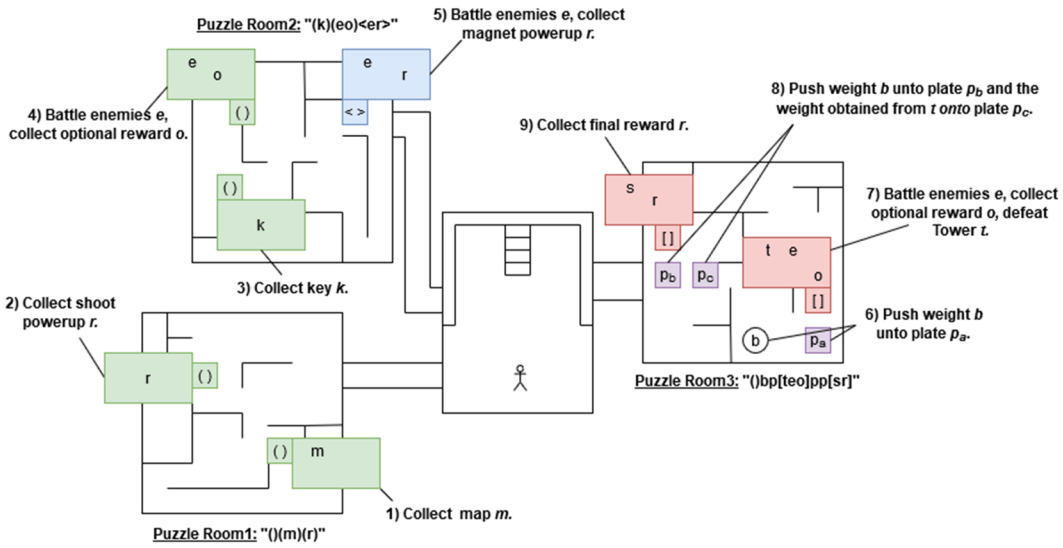

Figure 4.10: An example dungeon demonstrating puzzles from all three phases of production. Puzzle Room 1 uses the puzzle string "()(m)(r)", Puzzle Room 2 uses "(k)(eo)(er)", and Puzzle Room 3 uses "()bp[teo]pp[sr]". A numbered solution through the entire dungeon is provided.

Figure 4.10 provides an example dungeon containing a puzzle string generated for each of our three phases. The first puzzle room simply requires the player to collect the dungeon's map, and shoot powerup. The second puzzle chamber has the player collect a key, and an optional treasure from the first two chambers before unlocking the third and final chamber containing

the magnet powerup. Finally, the player progresses to the third puzzle room, here, they unlock the first of two chambers by pushing a block onto a pressure plate to unlock the lift. Inside this chamber, players retrieve an optional treasure, defeat some enemies, as well as a Tower enemy. With the destruction of the Tower, the player obtains a second weight. To gain access to the dungeon's final chamber, the player places both weights on the chamber's two accompanying pressure plates were they encounter the dungeon's final reward.

4.4 Stage 4 – Decorative Pass

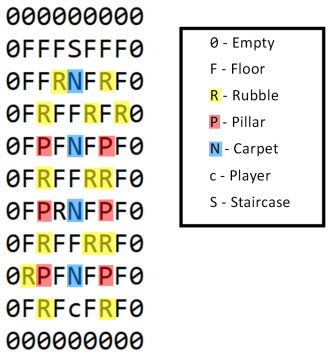

Figure 4.11: Example of entrance data used for the training of Stage 4's n-gram model. Indicated in the colors of yellow, red, and blue are the hand-decorated elements representing rubble, pillar, and carpet/chandelier tiles respectively.

As discussed in Chapter 3, the process of decorating our dungeon entrances involves the use of an n-gram model to place rows of decorative elements such as pillars, rubble, carpets, and chandeliers in order to style the pre-existing empty floor tiles. Figure 4.11 provides an example of the data used to train our model accompanied by a legend of possible tile types. From this data, we see the characters R, P, N , and F to denote rubble, pillar, carpet, and empty floor tiles respectively (recall that chandeliers share the same character N as carpets). We train our model to learn unigrams, bigrams, and trigrams, using 40 of these hand-decorated examples. We train our model to learn horizontal slices of the entrance that are 6 characters in length. We chose this value based solely on observation, as it seems to give the most consistent results across most entrance rooms regardless of their size—experimental results for other word lengths are provided in Chapter 5.

In order to select which slice to place, our model uses the probability of each slice given the previous two, a trigram. The likelihood of each slice appearing next in the sequence is based on the conditional probability:

$$P(w_3|w_1, w_2) \approx \frac{count(w_1, w_2, w_3)}{count(w_1, w_2)}$$

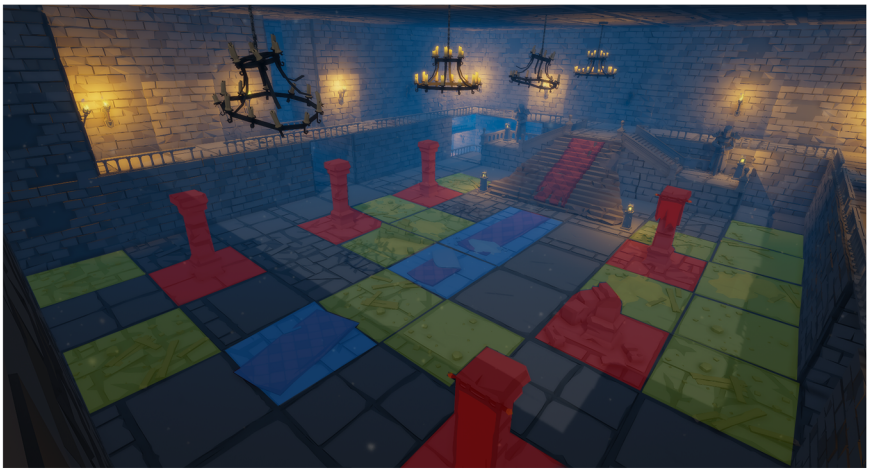

Figure 4.12: An example level generated using a trigram model, with a slice length L of 6. Decorative elements placed by this model are indicated using the same color scheme as Figure 4.11: yellow for rubble, red for pillars, and blue for carpets.

Unlike traditional n-gram models, our slices are being placed into pre-existing, immutable structures. This means that we must discard any slice which does not fit the current context in which we are attempting to place it. For example, if the current room segment we are trying to dress is "0FFcFFF", then we only will accept possible slices of the form "0xxcxxx" such as "0FRcFFR" or "0RRcFRF", where "x" represents a mutable tile position. This constraint ensures that our map's topology is not altered in any way by this process; additionally, we place a second constraint on the system such that any of the slices which appear to fit the current context but also contain 0's in any mutable position "x" are also discarded so that floor tiles are not accidentally replaced with empty spaces. For example, if we accept slices of the form "0xxcxxx", "0RRcF0F" would be discarded because it alters the map's topology by introducing an empty space in position 5. A consequence of discarding possibilities in this manner is that we will no longer have a proper distribution that sums to 1. This problem can be easily addressed by constructing a new distribution using the trigram counts of only the slices found to be feasible for the current context. This distribution is captured in the following modification to the previous equation:

Where I is the set of all infeasible slices,

$$P(w_3|w_1, w_2) \approx \frac{count(w_1, w_2, w_3)}{count(w_1, w_2) - \sum_{w_i \in I} count(w_1, w_2, w_i)}$$

or alternatively, where F is the set of all feasible slices,

$$P(w_3|w_1, w_2) \approx \frac{count(w_1, w_2, w_3)}{\sum_{w_f \in F} count(w_1, w_2, w_f)}$$

The problem with the original distribution is that the bigram counts in the denominator still account for the number of times they occurred before a contextually infeasible word. What

we are doing in this new calculation is temporarily excluding these infeasible bigram counts, and only considering those which were followed by feasible slices w_3. Figure 4.12 provides an example level decorated using a trigram model with a slice length L of 6. In this example, we can see how the model has learned to follow a structure similar to that of the example file in Figure 4.11 where carpets follow the room's centerline, pillars are evenly placed on eitherside of this line, and rubble is scattered around these elements.

Chapter 5

Results and Evaluation

This chapter presents the methods used to evaluate the various phases of our generator. While the majority of this work attempted to convey sufficient evidence that the design of our PCG system adequately considers qualities such as speed, reliability, controllability, and creativity, much of the discussion concerning the expressivity of our solution has been deferred to this chapter, as much of our results focus on this criteria.

5.1 Exploration of Latent Space

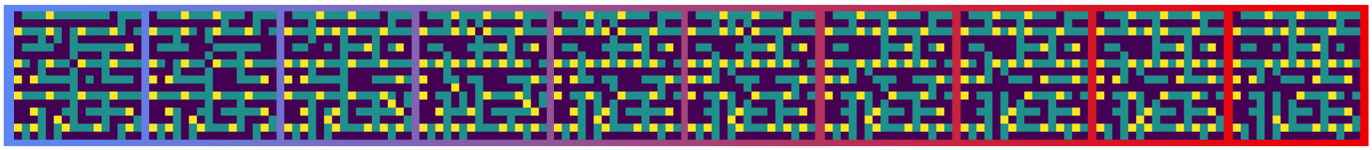

Figure 5.1: Example mazes obtained by interpolating between two randomly selected points in latent space. The mazes on either end of the diagram are those produced by our Stage 1 DCNN at each of these points. Purple tiles represent the floor; yellow tiles, pits; and green tiles, walls.

The GAN architecture is well-known for its ability to map input vectors to resulting images it produces. These input vectors are usually sampled from a Gaussian distribution and have no particular significance until assigned one by training the GAN. Through this training process, the GAN learns to map its output to corresponding inputs. By the end of training, we will be able to explore this distribution in what is called latent space. As we saw in [39], the latent space of the author's *SMB* level generator was searched for levels that maximized a series of performance metrics. What's interesting about latent space is that the interpolation between two points yields a smooth transition between the images produced by the model [25].

In this work, we trained a DCNN whose architecture resembles that of a GAN's generator network using an ES and examine the resulting latent space by choosing two random vectors and perform a spherical linear interpolation (Slerp) to obtain eight intermediate vectors. Results are shown in Figure 5.1, where the two mazes other either end of the diagram do slowly

transition between each other. In Chapter 3, we discussed how this could be used to find levels that have a combination of interesting structural features; unfortunately, as our mazes are only 16x16, we fear that they are too simple to contain any identifiable characteristics which warrant this approach for the time being.

5.2 Expressive Range

Expressive range is a popular method for measuring the expressivity of a generator's content by plotting two separate evaluation metrics on a heatmap. The goal is to visualize the range of potential content that can be expressed by a procedural content generator [31] as well as identify any biases it may possess through regions of especially high intensity. Using this method, we will evaluate the expressivity of both our mazes produced in Stage 1, as well as the dungeons produced in Stage 2.

5.2.1 Expressive Range of Stage 1 Mazes

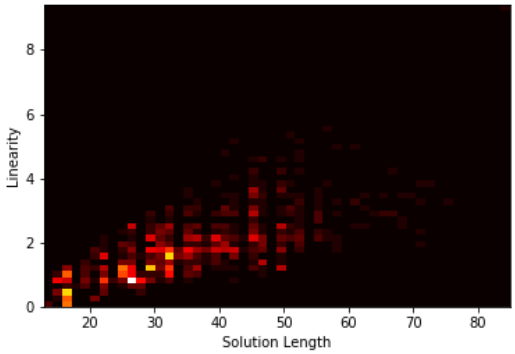

Figure 5.2: The heatmap generated for the linearity vs. solution length gathered from 1000 mazes produced by our Stage 1 generator.

To test the expressive range of this generator, we plot linearity vs. solution length, with results shown in Figure 5.2 and a table providing the max, min, mean, and standard deviation (SD) for these two metrics provided in A.1 of Appendix A.

Based on the spread of data found in this plot, we would say that this generator does indeed produce a reasonably diverse range of mazes with very little bias towards generating any of a specific kind as there are no regions of significantly high intensity. As an example of what this bias could look like and what it would signify in terms of our generator, imagine if a pocket of yellow/orange points were present in the bottom left-hand corner of the plot, we could conclude that the generator is biased towards producing mazes with short linear solution paths. While in actuality, there are a some of these high-intensity points on this plot, we do not view these regions as large enough to consider them a source of biased behavior. This said, it is disappointing, but not entirely unexpected, that these two features are so highly correlated.

It appears reasonable to say that less linear solutions—those with a high linearity measure—would also have longer solution lengths as the measure of a maze's linearity is the sum of all turn angles in the agent's solution path: longer solution paths lead to more turns taken, and as a result, lead to higher linearity values. While it is not a major source for concern, we would ideally like to see all kinds of mazes represented, such as those with a high linearity and a low solution length. We suspect that our decision to generate mazes without notifying the generator as to where start/orb/exit tiles will be placed is impacting its expressivity as it is reasonable for the generator to compensate for this by designing more generic mazes that support multiple solution paths. In the future works of Chapter 6, we will discuss the potential benefits of including a mechanism for notifying the generator on the position of these three critical tiles as this improvement would likely provide even better results both in terms of content quality and expressivity.

5.2.2 Expressive Range of Stage 2 Dungeons

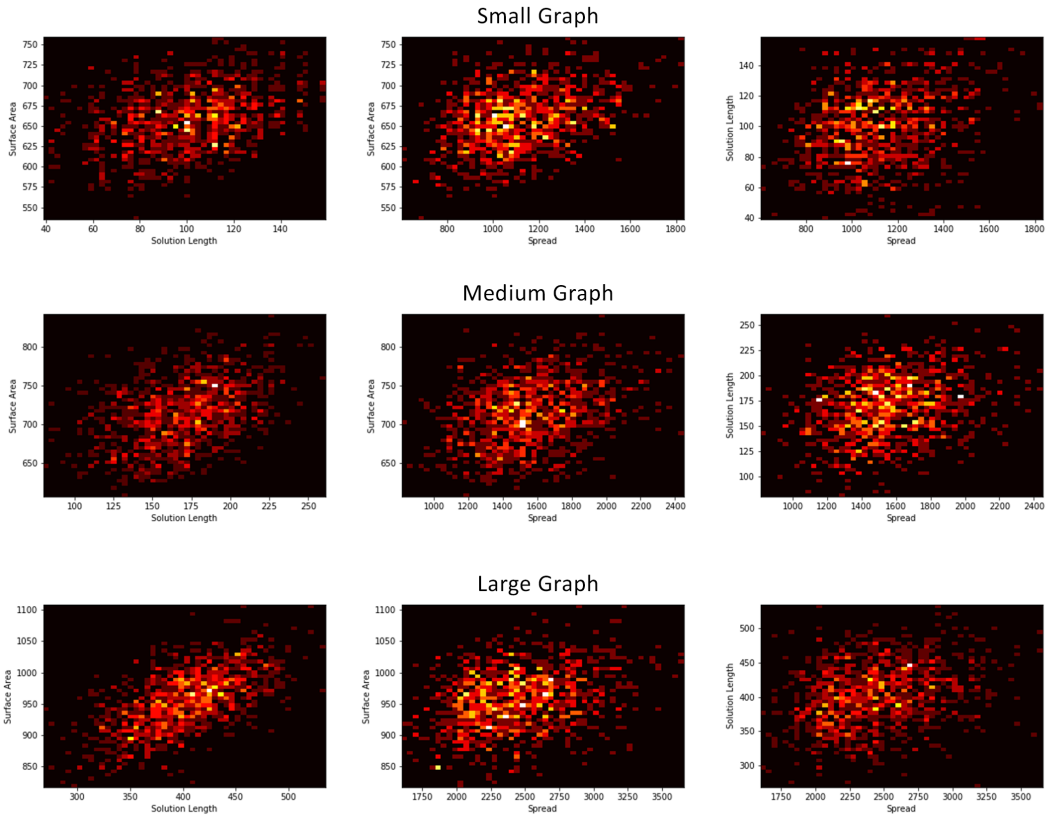

Figure 5.3: Heatmaps generated for the surface area, solution length, and spread from dungeons produced using our small, medium, and large control graphs; each producing 1000 levels. Surface area counts the number of floor tiles in the dungeon, solution length sums the length of the optimal solution path of each hallway and maze room, and spread reports the surface area of the smallest rectangle which can encapsulate the entirety of the dungeon.

For the expressive range of the dungeon generator in Stage 2, we choose to measure surface area, solution length, and spread. The surface area metric counts the number of floor tiles, solution length sums the length of the optimal solution path of each hallway and maze room, and spread reports the surface area of the smallest rectangle which can encapsulate the entirety of the dungeon. Like in Stage 1, we produce 1000 sample dungeons, but this time, for each of the three control graphs: small, medium and large. Figure 5.3 presents heat maps comparing surface area vs. solution length, surface area vs. spread, and solution length vs. spread for each of the three control graphs, with Tables A.2, A.3, and A.4 of Appendix A providing the max, min, mean, and SD for each of these metrics.

Based on the results presented in Figure 5.3, we would say that this generator undoubtedly produces a wide variety of dungeon environments, as all 9 of the heatmaps show a wide variety of levels produced with very little bias shown towards any particular feature. We do see some signs of bias towards levels with a medium-amount of surface area and spread, particularly for the small and large control graphs, but like with our mazes, these high-intensity regions are not dense enough for us to consider this generator to be overly biased towards any specific type of dungeon. What is encouraging about these results is that these dungeons are built using a very small set of 16 mazes as room templates, meaning that any shortcomings in that generator's expressivity can easily be recouped by this one's. Initially, this small corpus of 16 mazes was chosen to help with the exhaustive search being performed in Stage 2's constructive algorithm, as we feared that a large number of mazes would have taken too long to search through.

5.3 Evaluation of the Stage 3 Grammar

Compared to every other stage, the evaluation of our puzzle generating grammar in Stage 3 is the most basic as we will only be reporting the size of the languages described by our three sets of production rules. We view this as a simple evaluation of our grammar's expressivity. As a reminder, we divide our grammar's production rules into three phases which guarantees to only produce puzzles that reflect the abilities currently possessed by the player. Figure 4.9 of Chapter 4 presents each of these production rules and their respective phases. Phase 1 assumes the player does not possess either the shoot or magnet power-up and contains 8 production rules that produce a total of 6 unique puzzles. Phase 2 assumes the player has possession of the shoot power-up and contains 12 production rules that can produce 15 unique puzzles. Finally, phase 3 assumes the player possesses both the shoot and magnet power-ups and contains 18 production rules which can produce 60 unique puzzles. Assuming each dungeon contains one puzzle from each of these three phases, there will be exactly 5,400 possible combinations.

5.4 Evaluation of Stage 4 N-grams

The evaluation of our Stage 4 n-gram model is purely qualitative as we will only be judging the stylistic qualities of the entrance rooms it decorates. We base our evaluation on adjustments made to both the length L of its horizontal slices as well as n in the n-gram. Recall that in Stage 4 of our solution, we train an n-gram model using rows of characters representing the different decorative elements of our dungeon's entrances (pillars, rubble, and carpets/chandeliers) which

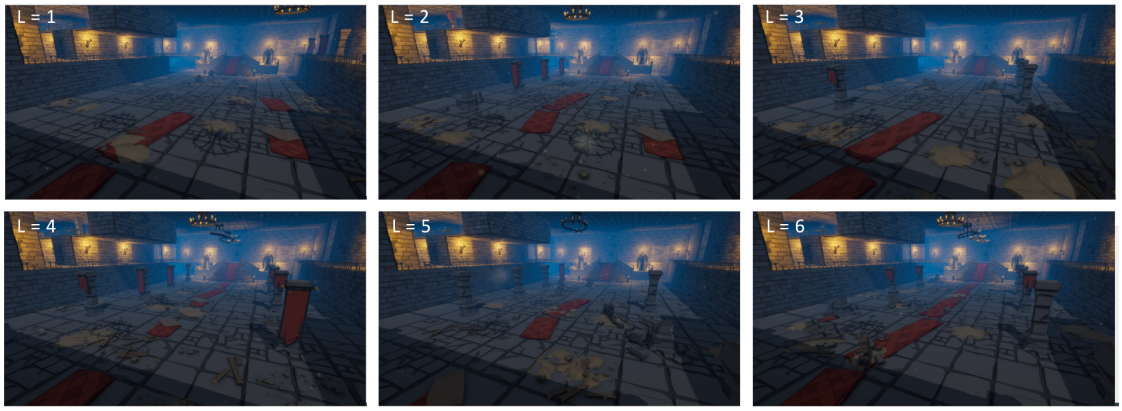

Figure 5.4: A dungeon decorated using a trigram trained with slices of length L ranging from 1 to 6.

we refer to as horizontal slices. The process of decorating our entrances involves the n-gram model stacking these slices starting at the south end of the room, using previously placed slices as the only context for which should be placed next. We let L represent the number of characters in each of these slices, while n refers to how many previous slices are used as context by the n-gram model.

Figure 5.4 presents an entrance decorated using a trigram model with L ranging from 1 to 6. Because this is a trigram model, we are determining which slice to place using only the previous 2. From this figure, we can observe that an L of 4 or higher produces satisfactory results as there is a series of carpet tiles which follows the room's centerline with evenly spaced pillars running down both sides of the room. When L is less than 4, we can see blatant violations to the symmetry we are trying to achieve, with carpet tiles straying beyond the bounds of the centerline, and pillars scattered on either side of the room. Through observing dozens of entrances, we find that a value for L of around 6 works best in most cases.

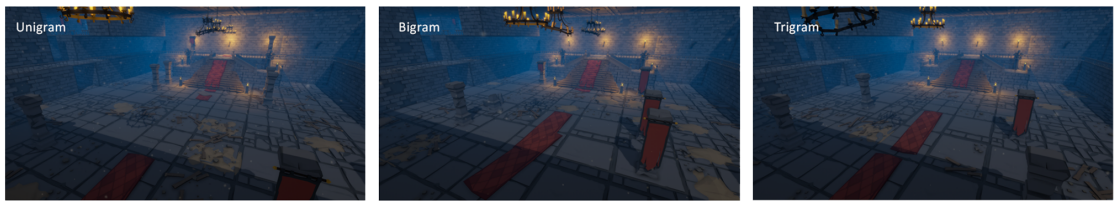

Figure 5.5: A dungeon decorated with an L of 6, using a unigram, bigram, and trigram model.

When observing entrances decorated with a unigram, bigram, and trigram models, we find that the trigram model is best at maintaining the spatial relationships of elements such as carpets and pillars. As seen in Figure 5.5, the unigram model places decorative slices with too much irregularity, as the spacing between pillars is too large in some sections and too small in others. The bigram model makes many of these same mistakes, but overall does perform considerably better than the unigram model. Finally, the trigram model appears to not make any mistakes in this instance, spacing the carpets and pillars appropriately apart. Of course, it is not a

coincidence that the trigram model would perform the best out of these three, as the space between pillars and carpets in our training data is exactly two tiles apart, aligning perfectly with the contextual window of a trigram.

5.5 Analysis of our System's Desirable Properties

While much of this chapter focused on metrics such as expressivity, we will now conclude this chapter with a discussion which frames our results in terms of all of the desirable properties including: speed, reliability, controllability, expressivity, and creativity/believability in a manner similar to that of Section 3.4. With the exception of subsection 5.5.1, we will not be introducing any new forms of evaluation, but instead, we highlight and review how many of these desirable properties have already been discussed in the earlier findings of this chapter.

5.5.1 Speed

	Small	Medium	Large
Mean	0.078	0.086	0.296
Min	0.061	0.071	0.128
Max	0.254	0.277	23.268
SD	0.010	0.009	0.883

Table 5.1: The mean, min, max, and SD for the combined generation times of all four stages of our solution using a ***small***, ***medium***, and ***large*** control graph. All values are listed in seconds.

For much of this work, we have discussed and analyzed each of our solution's four stages individually, but for a property like speed, we will judge the totality of our system by reporting the combined generation times of all four stages using the same 3000 dungeons gathered for Figure 5.3. In this case, we generated 1000 dungeons from our small, medium, and large control graphs and recorded their average generation times in Table 5.1 along with their min. times, max. times, and SD. It is important that our solution runs using consumer-grade hardware as this is the target destination for many, if not all, video games; we provide a list of hardware specifications below:

- RAM: 16.0 GB
- Processor: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
- GPU: NVIDIA GeForce GTX 1070 with Max-Q Design
- OS: Windows 10 64-Bit

In Subsection 3.3.3 of Chapter 3 we stated that "we will only consider online generation times as those who are no longer than several seconds", as we view any amount of generation time beyond this as an inconvenience to the player. From the results presented in Table 5.1,

we would say that we succeeded in producing a rather fast generation method as our average generation times were less than 1 second for all three control graphs. The only point of concern would be the 23 second outlier that occurred while generation dungeons using the large control graph. And while we expect the generation times of dungeons to increase with the number of nodes present in the control graph, this extraordinary long time was undoubtedly caused by backtracking many times until a valid solution was found. Based on this, we conclude that a future improvement to this system would be to simply restart generation in such cases where too many conflicts occur.

5.5.2 Reliability

In previous sections of this work we have discussed how, in theory, each of our stages should only be capable of producing playable content. It is the reason we chose to use a finite set of mazes in Stage 1, a constructive algorithm for Stage 2, a CFG in Stage 3, and a decorative pass in Stage 4 which makes use of tiles that cannot possibly impede the player's progress; however, in practice, it is nearly impossible to know for sure that all of the dungeon's our system can produce are indeed solvable without generating and testing each one for ourselves. For now, the only quantitative results we have for reliability, are based on the fact that the 3000 dungeons generated for Figure 5.3 did in fact generate without a single failure.

5.5.3 Controllability

Unlike speed and reliability, controllability in most cases only enhances a player's experience with a game, and very rarely detracts from it. Controllability is mostly used for player adaptive systems, which custom-tailor levels to fit a specific player's profile; and unlike the playability or generation times, a complete disregard for controllability over a system could still result in excellent content that the player would enjoy. In this way, we do not need to focus on maximizing controllability in every stage of our approach as we see no such thing as a controllability bottleneck. This said, throughout this work we have discussed ways in which our various stages could be controlled. We have already seen and discussed how our Stage 1 mazes could be controlled using the interpolation of feature vectors, or how our Stage 2 control graphs allow for a designer to directly influence the topological structure of a dungeon; furthermore, a great deal of control over our Stage 3 puzzles are offered through the use of a CFG, and how manipulation of L and n do provide some influence over Stage 4's n-gram model.

In this subsection, we would like to highlight the potential effects of our Stage 2 control graphs, as these mechanisms were deliberately included for the sake of controllability over major structural features of our dungeons. Beyond simply controlling the layout of a dungeon, what we are also able to observe is how the size of these graphs effect the magnitude of certain emergent properties such as those discussed in Section 5.2. By once again comparing the expressive range heatmaps presented in Figure 5.3, we can see how the magnitude of metrics such as spread, surface area, or solution length change based solely on the size of the control graph. This can be observed by comparing the change in any of these three metrics' distribution across any two control graphs. For example, the solution length of dungeons generated by the small control graph ranges from approximately 40 to 160; then in medium, from 80 to 260; and finally in large, from 270 to 540. This shows how much influence these graphs have over

the resulting dungeons being generated. While currently, the three control graphs evaluated in this work only differ in size, the influence of a graph's configuration could have a meaningful impact on these three metrics. An experiment that warrants future investigation would involve the comparison of these three metrics, but this time, in levels produced by control graphs which contain the same number and type of nodes, but differ in their configuration of them.

5.5.4 Expressivity

With much of this chapter's earlier analysis focussing on expressivity, we will not contribute much more now. We will take this time to reiterate the importance of expressivity as it is perhaps the best indication of a generator's success as a PCG system. Without the ability to produce multiple unique and interesting pieces of content, there is very little reason to use of PCG in the first place. In our earlier evaluation of Section 5.2, we saw how expressive range is used to help visual the expressivity of our Stage 1 and 2 mazes/dungeons, where we concluded that our dungeons do indeed exhibit a great deal of variety based on their heatmaps which not only indicate that our system is capable of producing a wide variety of levels, but that it is also relatively unbiased towards any particular type of dungeon. Finally, in Section 5.3 we show how our CFG is capable of generating a vast number of possible puzzles using only a small number of production rules. What we saw is our grammar was capable of producing 6 puzzle strings in phase 1, 15 in phase 2, and 60 in phase 3 for a total of 5400 unique puzzles assuming our dungeons contained one puzzle string from each of the 3 phases.

5.5.5 Creativity/Believability

The purpose of including Stage 4's decorative pass was to increase our dungeons' overall level of believability and, as discussed in Section 5.4, such an evaluation is purely qualitative. Based on the results provided in Figures 5.4 and 5.5 we concluded that a Trigram model with a horizontal slice length L of 6 provided the most consistent and realistic appearance to our dungeon's entrances as the model learned to only place carpet/chandelier tiles down the entrance's center-line with symmetrical rows of evenly spaced pillars on either side.

Chapter 6

Concluding Remarks

In this final chapter, we will summarize the contents of this thesis as well as discuss its novel contributions and future improvements.

6.1 Conclusion

In this thesis, we provide a solution to the problem of designing a procedural content generator for the production of video game dungeon environments. We stipulate that our solution must possess the five major quality metrics of speed, reliability, controllability, expressivity, and creativity in order for it's methods to find use outside of academia. To accomplish this, we present a multi-stage approach which uses methods found in both PCGML, and constructive PCG. First, we train a maze generator using [26]'s ES. The advantage to this method of optimization is that we can involve non-differentiable elements such as gameplaying agents in the training of our models. We show our this approach successfully trains a DCNN to produce a wide range of maze environments. We further expand on these mazes by combining them into larger dungeon structures using a constructve PCG algorithm similar to one found in *Dead-cells*. Results in Chapter 5 show how this algorithm adds a substantial degree of expressivity, and controllibility to the level generation process. With the dungeon's macro-level structures in place, we turn our attention towards micro-level details. We use a context-free grammar to produce a series of puzzles, and an n-gram model to place decorative elements in our dungeon's entrances. Both of these stages add to the dungeon's overall creativity/believability, a property that is commonly overlooked in most PCG works. Ultimately, we would say that this work's goal was to move PCGML methods forward by not only applying a new approach to training, but also by demonstrating a practical application of these methods by using them within an appropriate context.

6.2 Contributions

During our review of PCG methods, we did not encounter any works that made use of the same ES presented in [26], and it is possible that this work is the first to do so. The paper which originally outlined this approach to training was published in September of 2017, and a survey of PCG methods [33] published in May of 2018 makes no reference to any works

which use this approach to training. We realize that these two works were published very close to one another, and at the time, ES were too new for any PCG works to make use of it before the survey's release; but since then, much of the focus of PCG approaches has shifted solely towards GANs, seemingly leaving the potential of ES untapped. To reiterate a point made in Chapter 3, we view this ES as a solution to a major problem faced by GANs in that their evaluation of game levels as images does not coincide with the requirements of the medium: games cannot be fully experienced through passive sight, but rather, only through active play. In the first stage of our solution, we use a DCNN architecture inspired by the generator of a GAN, but instead of training it using a discriminator network and a dataset of pre-existing levels, we train it purely on the minor structural characteristics of the level, and more importantly, on the solution path taken by a game-playing agent.

The second contribution we believe our solution presents is a combination of methods which bridges the divide between those PCG methods typically seen in academic literature, and the constructive methods commonly used in commercial game releases. Our motivation behind designing a generator which possesses speed, reliability, controllability, expressivity, and creativity, is directly influenced by the fact that almost any modern commercially-released games which feature procedurally generated levels attempts to capture all five of these desirable qualities, while many academic contributions do not. Our solution to this problem is to create a multi-stage generator that combined two methods from both PCG and constructive PCG. The two main deficiencies of PCG methods would certainly be reliability, as we can not guarantee that a level produced by these models will always be playable, and controllability as there are often very few opportunities to interact with a fully-trained ML model outside of its input space.

Our solution addresses both of these issues by using a constructive algorithm inspired by those found in *Spelunky*, *Deadcells*, and *Minecraft*. This algorithm allows us to reliably produce large dungeon structures using only a finite set of mazes produced by our maze generator. We also showed in our results of Chapter 5 how controllable this algorithm is using a graph to inform the algorithm on how to construct the dungeon's macro-level topology in a similar fashion to *Deadcells*. We view our final two stages of generation as supplementary to those in Stages 1 and 2, as their purpose was to simply increase the overall creativity or believability of the system. And while it is important that Stage 3 adds the actual game-specific elements to our dungeons, we view Stage 4 as a more interesting topic of discourse as its application of an n-gram model—a PCG method—was used within a context where its unreliable behavior would be of no overall consequence to the system at large; that, of course, being the production of decorative elements.

6.3 Future Works

Throughout each of our Chapters, we were careful to take note of some areas of future improvement. Specifically, Chapter 3 raised a number of issues, these being the lack of controllability in our Stage 1 approach, the restricted access our dungeon generator has to only a finite set of maze levels, the potential for long generation times due to backtracking, and the limited implementation of our n-gram model.

While discussing the lack of controllability of most PCGML systems throughout this work,

we also acknowledge that a benefit to GANs is that they can be trained to map arbitrary points from a Gaussian distribution to output images. We saw in [25, 39] how this latent space could be explored and presented results for our exploration of our Stage 1 DCNN. Yet, we do not actually utilize this feature of our network in our solution. In Chapter 3, we proposed that one could gather a set of latent vectors mapped to a variety levels which display interesting qualities or characteristics, then interpolate between them such that their individual properties are mixed together. Alternatively, if we wish to obtain a random level with properties similar to another's, we could simply add a small amount of Gaussian noise to the original's latent vector in order to receive a random variation of it. The reason this system was never implemented was tied to the exhaustive search in Stage 2 which can be more easily performed on a finite set as opposed to the entire latent space of our DCNN. Ideally, we could obtain a random level directly from our maze generator, quickly test it for playability, and, if it passes, place it in our level. The problem with this approach is that we can no longer guarantee that generation can be performed in a fixed number of steps while searching the entirety of a network's latent space like we can with a finite set. One simple, yet likely naïve, solution to this problem would be to simply try to generate a dungeon using random mazes sampled from our DCNN, and should generation fail after an arbitrary number of times, fallback to a finite set of levels and proceed with generation.

The expressive range of this Stage 1 DCNN is presented in Subsection 5.2.1 of Chapter 5, where we suggest that this generator would likely produce a wider variety of high-quality content if it was informed on the positions of the start/orb/exit tiles used during training. Currently this network is responsible for generating mazes based solely on a vector of Gaussian noise, with the start/orb/exit tiles being placed without its knowledge during a post-generation step. While we are relatively pleased with the results of this method, we suspect even better results might be attained through notifying the network where these points will be placed during the generation process itself. Ideally, any modifications to the network's architecture would not remove the Gaussian noise at the input layer, as we would still like to maintain the benefits that latent vectors have to offer; instead, we propose that the introduction of these three tile's positional information be inserted elsewhere, such as one of the network's hidden layers. We are not suggesting that this is the only place for possible modifications to be made, but rather as a potential starting point worthy of consideration.

Another issue raised in Chapter 5 was the occurrence of slow generation times due to backtracking. In Subsection 5.5.1 we observed an abnormally long generation time of 23 seconds for a dungeon built using our large control graph which we attributed to the algorithm backtracking until it found a successful configuration of rooms. Considering the average generation time for dungeons using this graph was 296 milliseconds, it certainly would be faster to simply restart generation from the beginning with a different random set of rooms. In fact, many games which use the WFC algorithm first discussed in Chapter 2 frequently encounter similar issues with conflicts with many choosing to forgo backtracking just as we are considering.

We mentioned in Chapter 3 that we would only be using our n-gram model to decorate the entrances of our dungeon. Ideally, this model would be extended to the entire level. For the purposes of this work, we decided that entrances be the most appropriate application for this model due to their large rectangular shape. A potential issue with this model is the selection of slice length L. During a discussion of this model's results in Chapter 5, we mentioned that an L of size of 6 seemed to produce the best results in our entrances. We believe that this is

due to the fact that our entrance has a fixed number of possible widths, in which a slice of length 6 happens to coincide well with. Perhaps, a more interesting and flexible model would abandon one-dimensional slices all together in lieu of a 2D window as we saw in [32] or opt for a completely different method and attempt to use the WFC algorithm. A possible advantage to WFC over these ML models is the size of training data required for them to function properly. In our application we managed to obtain satisfactory results with 40 hand-decorated example entrances. An advantage to WFC over these methods is that it would only require one sample entrance in order to create an infinite number of them in the future.

Finally, while this work focused on presenting how our four-stage approach could be applied to dungeon environments, we would like to address how this method could be adapted to other genres. To do this, it is important to understand the role each stage plays in the system as a whole. In Stage 1, we trained a DCNN to produce mazes, but more generally we can view these as small level fragments that can be combined to form larger structures in the later stages of our system. In practice, changing the ES' reward function as well as its game-playing agent could result in the generation of fragments for any number of game genres, for example, using the agent from [13] and a new reward function that perhaps still focuses on the non-linearity of the agent's solution path, the generator could be trained to build level fragments for 2D Platformers similar to *SMB*. Next, the role of Stage 2 is to combine these fragments into larger environments. It is likely that any changes to this stage would involve the control mechanism responsible for deciding which fragments are both chosen and connected to one another; for this work, we chose to use a control graph similar to that of *Deadcells* to both place and connect our hallways and maze rooms together, however, it was also shown how a grammar was used in the case of *Minecraft* in order to build small villages, or a simple set of rules and constraints in the case of *Spelunky* to build large cavern systems. In general, we view Stages 1 and 2 as those which are responsible for building the level's topology, and reserve the instantiation of any objects the player is meant to interact with for Stage 3. In this stage, we made use of a CFG to place game-critical objects such as enemies, puzzle elements, and rewards, but in other genres of games this may include NPC characters, collectibles, or quest items. Finally in Stage 4, developers are given an opportunity to include any decorative elements they wish to include in their levels. As previously mentioned we had chosen to only decorate our game's entrance rooms, but this system could be trained to decorate the entire level. While we were careful to maintain reliability by using a constructive PCG method in Stage 3, during Stage 4 we purposely train a PCGML method to learn how to copy the structure of decorative objects from a training set and apply them to our newly generated environments with the understanding that these systems do make mistakes on occasion; as a result, it is best that any objects placed in this stage be designed such that, even when placed incorrectly, they not interfere with the player's ability to navigate the level.

Ideally, future applications of this work would not only address each of the concerns we have noted during this discussion but also apply them to a new genre of game in an attempt to not only show the flexibility of this approach but to hopefully expose any other areas of improvement that can potentially benefit it as a whole.

Bibliography

[1] Entertainment Software Association. 2019 essential facts about the computer and video game industry, 2019.

[2] Sebastien Benard. Building the level design of a procedurally generated metroidvania: a hybrid approach, 2017.

[3] Plausible Concept. Bad north. Digital, 2018.

[4] Steve Dahlskog, Julian Togelius, and Mark J. Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference on Media Business, Management, Content & Services*, pages 200–206, 2014.

[5] Isaac Dart and Mark J. Nelson. Smart terrain causality chains for adventure-game puzzle generation. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 328–334, 2012.

[6] Jonathon Doran and Ian Parberry. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, pages 1–8, 2011.

[7] Clara Fernández-Vara and Alec Thomson. Procedural generation of narrative puzzles in adventure games: The puzzle-dice system. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 12, 2012.

[8] Table Flip Games. Sure footing. Digital, 2018.

[9] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. Doom level generation using generative adversarial networks. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 316–323, 2018.

[10] Marco Krüger Ian Wadham. Kgoldrunner. Digital, 2003.

[11] Adrian Carmack Kevin Cloud Tom Hall John Carmack, John Romero. Doom. [CD-ROM], 1993.

[12] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 2006.

[13] S. Karakovskiy and J. Togelius. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[14] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020.

[15] Isaac Karth and Adam M. Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 68, 2017.

[16] Darius Kazemi. Spelunky generator lessons, 2013.

[17] Barbara De Kegel and Mads Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, pages 1–1, 2019.

[18] Marian Kleineberg. Infinite procedurally generated city with the wave function collapse algorithm, 2019.

[19] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 3730–3738, 2015.

[20] Ricardo Lopes, Elmar Eisemann, and Rafael Bidarra. Authoring adaptive game world generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 10(1):42–55, 2018.

[21] Takashi Miyamoto, Shigeru; Tezuka. Super mario bros. [Game Cartridge], 1985.

[22] Mossmouth. Spelunky. [Digital], 2008.

[23] Twin Motion. Deadcells. [Digital], 2017.

[24] Jens; McManus Stephen Persson, Markus; Bergensten. Minecraft. [Digital], 2011.

[25] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR 2016 : International Conference on Learning Representations 2016*, 2016.

[26] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[27] Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, Julian Togelius, and Michael O'Neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, 2012.

[28] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. 2016.

[29] Takashi Tezuka Shigeru Miyamoto. The legend of zelda. [Game Cartridge], 1986.

[30] slicedlime. How villages are generated in minecraft 1.14, 2019.

[31] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 4, 2010.

[32] Sam Snodgrass and Santiago Ontañón. Experiments in map generation using markov chains. In *FDG*, 2014.

[33] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgard, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[34] Adam James Summerville and Michael Mateas. Sampling hyrule: Multi-technique probabilistic level generation for action role playing games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[35] A Nielsen Company SuperData. 2019 year in review, 2020.

[36] T Thompson. Scalable level generation for 2d platforming games. 2016.

[37] Valve. Dota 2. Digital, 2013.

[38] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[39] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference on*, pages 221–228, 2018.

[40] David Williams-King, Jörg Denzinger, John Aycock, and Ben Stephenson. The gold standard: automatically generating puzzle game levels. In *AIIDE'12 Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 191–196, 2012.

[41] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. 2018.

Appendix A

Supplemental Material

[[[1,1],[14,1],[7,1]], [[1,4],[14,4],[7,14]], [[1,7],[14,7],[7,11]], [[1,11],[14,11],[7,4]], [[1,14],[14,14],[7,4]], [[1,1],[14,14],[7,7]], [[1,4],[14,14],[7,11]], [[1,7],[7,14],[7,4]], [[1,11],[7,1],[7,4]], [[1,14],[7,1],[14,14]], [[14,1],[1,1],[7,14]], [[14,4],[1,4],[7,14]], [[14,7],[1,7],[14,11]], [[14,11],[1,11],[14,4]], [[14,14],[1,14],[7,14]], [[14,14],[1,1],[7,1]], [[14,14],[1,4],[7,1]], [[7,14],[1,7],[14,14]], [[7,1],[1,11],[14,4]], [[7,1],[1,14],[7,14]], [[14,1],[7,1],[1,14]], [[14,4],[7,4],[1,14]], [[14,7],[7,7],[7,14]], [[14,11],[7,11],[14,14]], [[14,14],[7,14],[1,1]], [[1,1],[7,14],[7,1]]]

Figure A.1: A List of all start/orb/altar tile coordinates used for training the maze generator.

	Solution Length	Linearity
Mean	34.10	1.75
Min	13.0	0.0
Max	85.0	9.42
SD	11.88	0.95

Table A.1: Solution Length, and Linearity measured for 1000 maze levels.

	Surface Area	Spread	Solution Length
Mean	658.87	1121.66	100.34
Min	535.0	602.0	39.0
Max	760.0	1833.0	159.0
SD	35.84	204.30	23.20

Table A.2: Surface Area, Spread, Solution Length measured for 1000 dungeon levels generated using a ***small*** control graph.

	Surface Area	Spread	Solution Length
Mean	714.57	1560.79	169.57
Min	607.0	814.0	80.0
Max	843.0	2451.0	261.0
SD	40.56	271.50	29.90

Table A.3: Surface Area, Spread, Solution Length measured for 1000 dungeon levels generated using a ***medium*** control graph.

	Surface Area	Spread	Solution Length
Mean	956.78	2443.86	403.13
Min	817.0	1600.0	268.0
Max	1109.0	3660.0	535.0
SD	46.72	359.73	44.25

Table A.4: Surface Area, Spread, Solution Length measured for 1000 dungeon levels generated using a ***large*** control graph.

Appendix B

Dungeon Elements

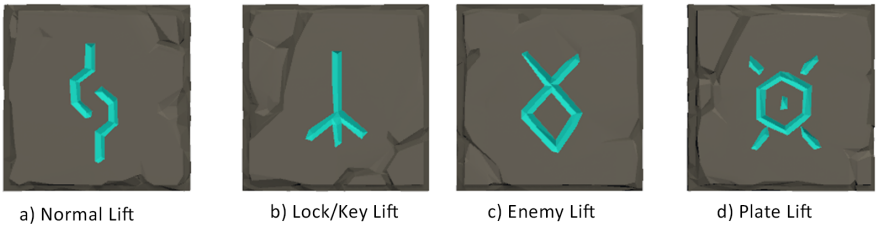

Figure B.1: The 4 lift types featured in our example game: Normal, Lock/Key, Enemy, and Plates.

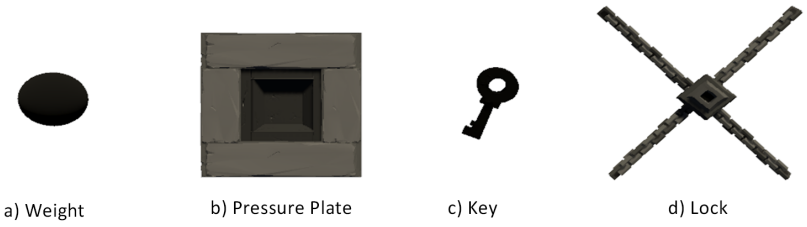

Figure B.2: The 4 interactable objects required to unlock a lift. A weight *a* can be combined with pressure plate *b*, and likewise, key *c* can be combined with lock *d*.

Figure B.3: The 3 enemies types in our game: 4x basic Cube enemies, Tower, and Shield Tower.

Figure B.4: The 3 reward types in our game: optional reward, map reward, powerup/upgrade reward.

Curriculum Vitae

Name:	Mathias Babin
Post-Secondary Education and Degrees:	The University of Western Ontario London, ON 2013 - 2017 B.Sc.
Honours and Awards:	The Western Scholarship of Distinction 2013
	Dean's Honor List x4 2014-2017
	Western Science Entrance Scholarship 2018
	Ontario Graduate Scholarship (OGS) x2 2018-2021
Related Work Experience:	Teaching Assistant The University of Western Ontario 2018 - 2020