University of Derby

College of Engineering and Technology

A project completed as part of the requirements for

BSc(Hons) Computer Games Programming

The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games

---

Author:
Becky Lavender

Supervisor:
Dr. Tommy Thompson

April 23, 2015

Abstract

Procedural content generation (PCG) has been applied successfully to games both in research and in industry. However, there are some genres which remain relatively untouched by PCG. This paper discusses the difficulties inherent when generating dungeons for action-adventure games. A previous technique for solving these issues which utilises different types of grammars for mission and map generation is implemented. The results are then applied to a typical tile-based 2D *Zelda*-style game in order to evaluate the grammar generators in an action-adventure context. The created mission and map generators, as well game application process, combine to form the Zelda Dungeon Generator (ZDG). This work finds that a wide range of functional dungeons can be produced using this method, and generation results can be successfully manipulated by changing the grammar rules input. Moreover, several obstacles rooted in the technique are found, particularly within the space generation stage. Solutions are described in detail but alternative methods are suggested as well.

Contents

1 Introduction

1.1 Rationale

Procedural content generation, the algorithmic generation of game content (Togelius, Champandard, et al., 2013), is gaining in popularity at a significant rate. Having been first introduced as a solution to hardware limitations in the 80s (Yannakakis and Togelius, 2011), PCG has been a staple feature of a number of popular games and is now listed as a selling point for many titles (Steam, 2014). However, there are a number of challenges PCG now faces as a field.

Context can be added to levels and maps in the form of quest or mission structures, which inform the space generation process as it is happening. The generation of quests and maps together has been suggested as a step which can be taken in order to eventually realise the broader goals of PCG research and expand the potential of the field (Togelius, Champandard, et al., 2013). The specific course of action this work takes and the more precise motivation for it are explained in the remaining sections 1.4 to 1.6.

1.2 Zelda Games and Action-Adventure Dungeons

The action-adventure (AA) genre is characterised by a mix of elements from action games (such as real-time combat) and elements from adventure games (such as puzzle solving) (Orland, Thomas, and Steinberg, 2007). *The Legend of Zelda* series is arguably one of the most successful franchises in the action-adventure genre, and consists of a large number of games released over the past 29 years (*Zelda Universe: Games* n.d.). The games traditionally house storylines which revolve around the hero having to collect a number of trophies over the course of the game, typically awarded to the player following the successful completion of dungeons.

Dungeons consist of multiple rooms and floors, containing tests such as defeating monsters and solving environmental puzzles. These dungeons often involve key and lock puzzles, in which the player must obtain a type of key before traversing an obstacle. These can take the literal form of keys and lock doors, or a more abstract form, such as an item which has an effect on the environment in a way which makes new areas accessible (Dormans, 2010).

Puzzles such as these can span entire dungeons, meaning dungeon layout in AA becomes much more complex and integral than in some other genres. Consequently, generation of AA dungeons is more difficult to achieve.

1.3 Grammars

Generative Grammars are a tool originally used in linguistics (Chomsky, 1956). A grammar consists of a set of rules which in theory could generate any plausible sentence in a given language. The concept can be applied to other disciplines, and involves recursively replacing subsections of a set, as dictated by rules. Generative grammars have been used in PCG to produce a wide range of content, and are discussed in further detail in section 2.5.3.

1.4 Dormans' Work

When looking for design patterns inherent in AA games, Joris Dormans analysed a dungeon from *The Legend of Zelda: Twilight Princess* (Nintendo, 2006) and found that levels were comprised of two separate structures: missions (the series of actions a player had to perform) and maps (the space in which to perform these actions) (Dormans, 2010). Dormans concluded that these dungeons should be generated in two distinct steps: graph grammar generation for missions and shape grammar generation for maps. Further detail on his analysis and arguments are presented throughout this work.

1.5 Aim and Objectives

Dormans has presented a promising approach to generating AA content by making use of generative grammars. However, this work has yet to be applied to a typical 2D *Zelda* game. Bearing in mind that the level in which Dormans originally found the mission and map distinction was from the *Zelda* franchise, it will be beneficial to see how his generation process holds up when applied to traditional *Zelda* mechanics.

It is also noted that some aspects of this two-tier grammar generator implementation have not been documented to the level of detail necessary for reproduction, which may hinder further work being done in the area.

The aim of this work is therefore:

To evaluate Joris Dormans' two-tier generative grammar technique when applied to a typical 2D Zelda-Style action-adventure game in two respects; effect on the development process and the expressive range of the resulting dungeons.

In order to achieve this aim, the following objectives are carried out:

1. Conduct a broad review of pre-existing procedural generation techniques for missions and maps in games, with a focus on that applying to the action-adventure genre and generative grammars.

2. Create a graph grammar generator to produce dungeon missions, documenting the development process simultaneously.

3. Create a shape grammar generator to produce dungeon maps which correspond to the generated missions, documenting the development process simultaneously.

4. Create a simple action-adventure demo game which uses dungeon maps created by the graph and shape grammars, and demonstrates key features of a typical *Zelda*-style dungeon.

5. Evaluate the development process, to assess Dormans' grammars as a practical choice for dungeon generation projects.

6. Evaluate the success of the product and the given hypothesis using results based on the expressive range and the functionality of dungeons produced.

1.6 Hypothesis

Based on the undertaking of the given objectives, the project hypothesis states that:

Applying Dormans' grammar based mission and map generation process to a 2D action-adventure Zelda-Style game will yield a diverse array of functional dungeons, which reflect the structure of their generated missions successfully.

This hypothesis focuses on the results of development rather than the process itself. In order to determine whether the dungeons are diverse, functional, and reflect their mission structure, relevant information will be documented during the development process. These qualities will then be further evaluated by performing a large number of tests and analysing the results found.

2 Literature Review

2.1 Introduction

2.1.1 Review Aim

The aim of this literature review is to assess whether or not the application of Dormans' graph and shape grammars to a 2D *Zelda*-Style game is valuable work, and whether or not the process would make a useful contribution to PCG in general.

2.1.2 Review Structure

- **2.2: Dormans' Analysis of Action-Adventure Dungeons** – The review starts with an examination of the nature of AA, discussing the observations which motivated Dormans' implementation.
- **2.3: An Overview of Generation Techniques** – This section provides background knowledge on pre-existing generation techniques for the types of content the project is concerned with - quests and levels. An understanding of the general case and currently accepted methods is necessary in order to grasp the specific issues AA generation faces, reviewed in the next section.
- **2.4: The Challenges of Generating Action-Adventure Dungeons** – Once the basic concepts of the relevant generation disciplines have been covered, the reasons that methods of generation discussed in section 2.3 pose problems when applied to the types of dungeons discussed in section 2.2 are brought to light.
- **2.5: Methods of Solving These Challenges** – This section presents several methods of solving these problems, which are used in industry and research. Section 2.5 also introduces concepts inherent in Dormans' work, including the combination of map and mission generation and generative grammars.
- **2.6: Dormans' Implementation and Further Work** – Once all necessary background knowledge has been presented and alternative techniques have been considered, Dormans' suggested implementation for solving the unique challenges of AA generation is described in detail. Dormans' further work, and work which has been inspired by his Adventures in Level Design paper, are investigated.
- **2.7: Discussion and Conclusions** – Limitations of Dormans' work are discussed and compared to other AA generation methods presented in the review. Relevant conclusions which can be drawn from the Literature Review are then presented.

2.2 Dormans' Analysis of Action-Adventure Dungeons

Dormans undertook a detailed analysis of level structure in modern action-adventure games in his Adventures in Level Design paper, subsequently referred to as AiLD (Dormans, 2010). It was found that these levels were divided into two distinct structures – the map (the geometrical layout), and the mission (the sequence of tasks the player had to perform to traverse the dungeon). The mission sequence, which can be represented as a graph, is independent from the geometric layout, but the two can be isomorphic in

places (see Figure 1). The same mission can be mapped onto many different spaces, and the same space can house many different missions. A non-linear mission can be mapped onto a linear space, and a non-linear space can house a linear mission.

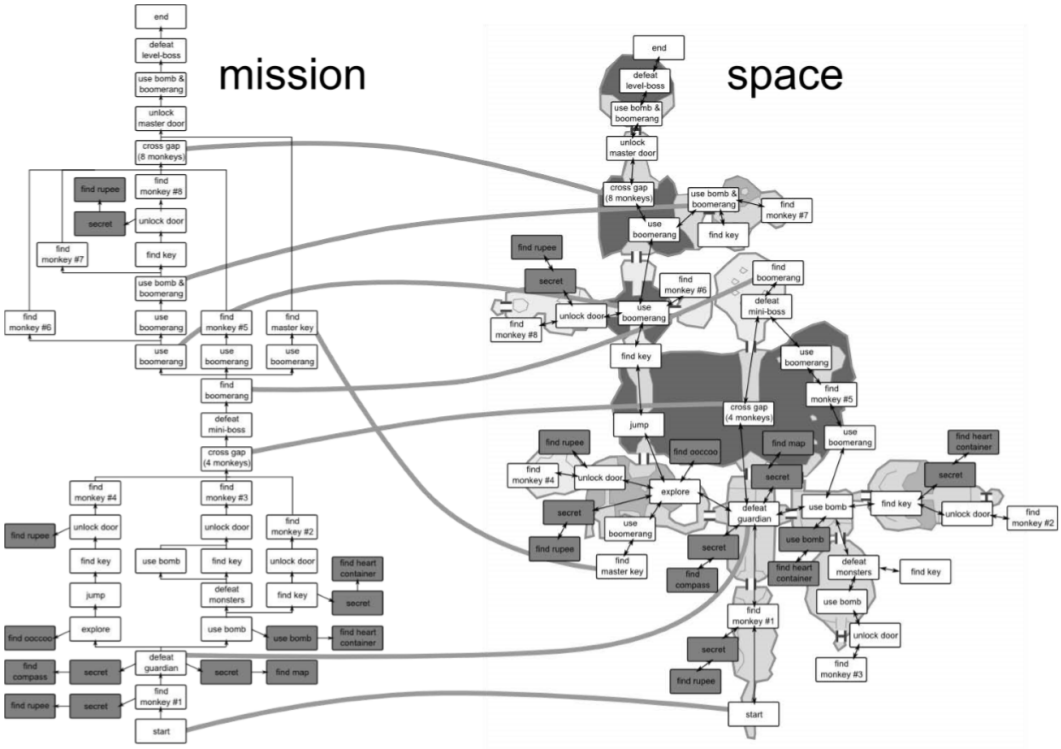


Figure 1: Dormans' Mission and Space Diagrams of the Forest Temple in *The Legend of Zelda:Twilight Princess* (Nintendo, 2006) (Dormans, 2010).

Dormans also found that the level analysed exhibited many common techniques in level design. Common in Nintendo games, the level followed a martial arts training strategy with several stages (Dormans, 2010). The player learns a simple technique in isolation, repeats the technique, learns how the technique can be combined with others, and is then tested in a final trial (often a battle). On top of this, the mission examined also follows the monomyth structure. Here, the player is confronted with a threshold guardian before entering the dungeon. Mid-dungeon, the player fights a mid-level boss and if victorious is rewarded with a new skill or item. This new item then allows them to traverse the rest of the dungeon and defeat the final boss. It acts as a key to many locks in the dungeon (disguised in different forms) (Dormans, 2010). The role of keys and locks in adventure games is also discussed in detail in Dormans' paper (see Section 2.4).

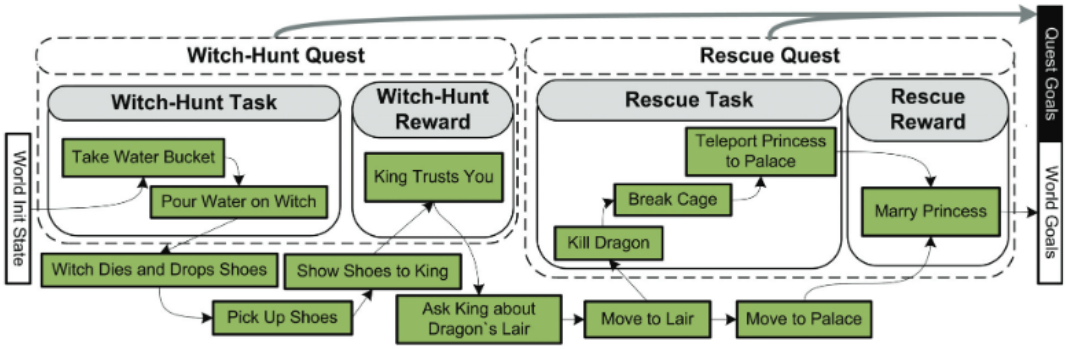
Dormans argues that the two distinct structures of mission and space have very different compositions, and so need to be generated using distinct techniques which have been specifically tailored to each element. To understand what makes generating missions and generating maps so dissimilar, and how the two processes can be linked together, it is necessary to first look at level and quest generation techniques in general.

2.3 An Overview of Generation Techniques

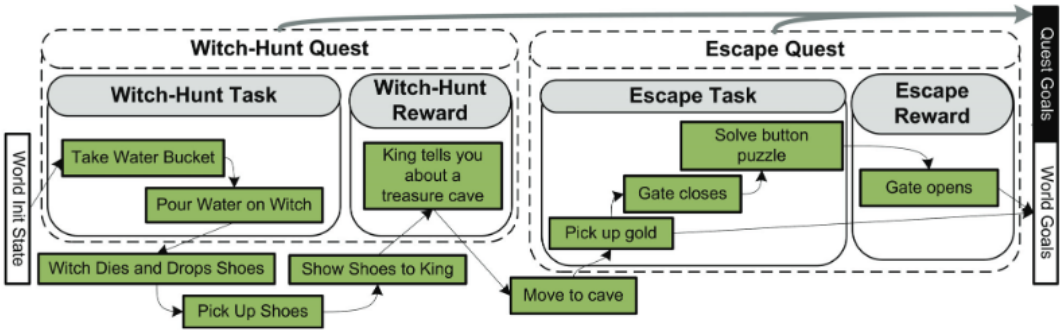
2.3.1 Mission and Quest Generation

Here, missions and quests refer to the series of actions players must perform in order to achieve certain goals. Generating these quests can have interesting applications, for instance to adapt the next events in the story as the user plays through (see Figure 2).

Human-authored plotlines can be adapted to suit individual players using an offline planning approach, while maintaining plot line coherence and preserving as much of the original plotline as possible, to reflect the human author's intentions. Li and Riedl's adaptation algorithm works in two steps, first rewriting the initial world state and outcome situation to match the plot requirements, and then progressively making adjustments to the plot until '(a) all plot requirements are met, (b) the plotline is sound, and (c) the plotline is coherent' (Li and Riedl, 2010).



(a) *Plotline before adaptation.*



(b) *Plotline after adaptation.*

Figure 2: A plotline, before and after it has been adapted based on a player's disinterest in rescuing and marrying a princess, and their interest in acquiring gold (Li and Riedl, 2010).

2.3.1.1 Narrative Generation

Much more investigation into quest and mission generation has been done under the mantle of narrative generation, or automatic narrative. These terms sometimes refer to narrative more strictly in the form of dialogue or back story (Mateas and Stern, 2005; McCoy et al., 2013), but can also be applied to the kind of 'action mapping' approach this project is concerned with.

For example, planning trajectories are used to control narrative generation for interactive storytelling systems (Porteous and Cavazza, 2009). Porteous and Cavazza recognise

the difference between an optimal series of actions to achieve a goal and one which has more reader interest, with errors and more complex sequences. To create the latter effect using narrative structuring, features of planning language PDDL3.0 are used to impose a number of constraints, and a novel method for planning with them.

Chang and Soo develop an incredibly complex planner which NPCs use to perform social reasoning, using logical inference about others' minds and a plan search to perform actions which change others' minds and achieve its goal (Chang and Soo, 2009) (see Figure 3). This technique is shown to produce interesting narrative segments.

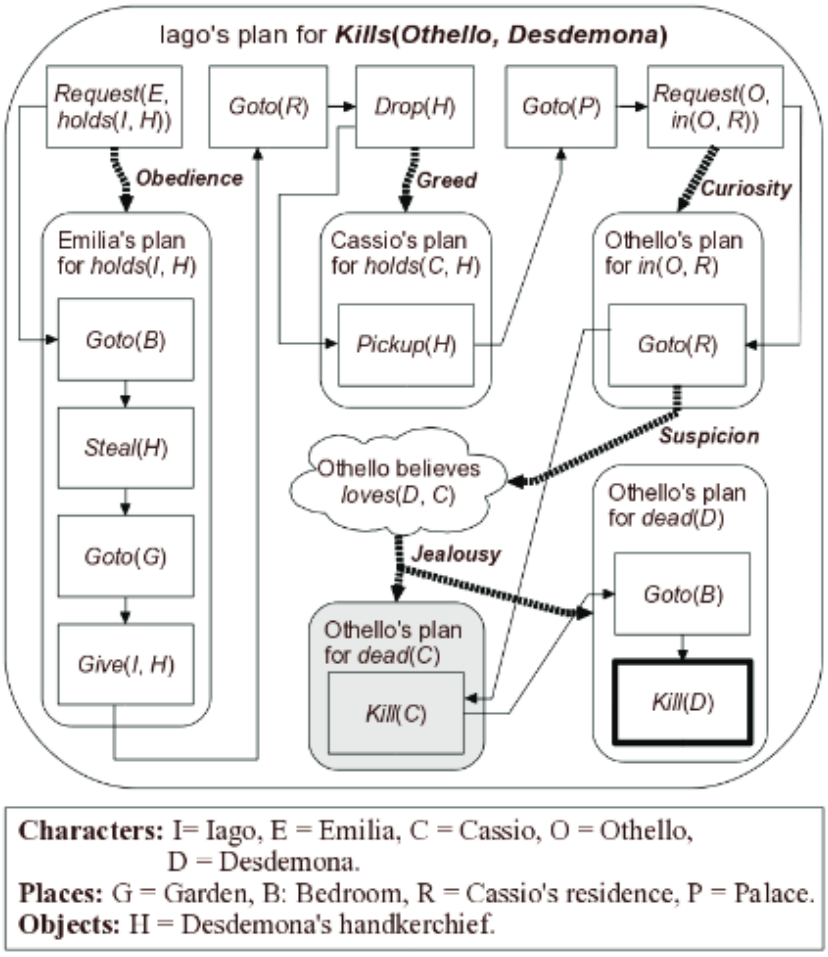


Figure 3: An example of a character's plan (Chang and Soo, 2009).

The Liquid Narrative group have produced the Mimesis system, 'an architecture for intelligent interactive narrative incorporating concepts from artificial intelligence, narrative theory, cognitive psychology and computational intelligence' (Young, 2007). At run-time, Mimesis generates plans and maintains their coherence when presented with unanticipated user behaviour and an unpredictable game-world state (see Figure 4). (Young et al., 2004).

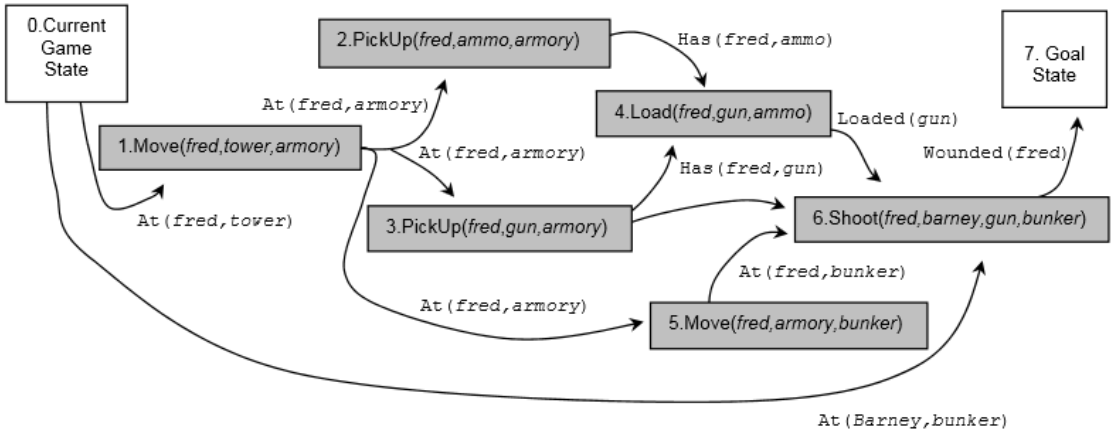


Figure 4: A Mimesis storyworld plan (Young et al., 2004).

Young used Mimesis to experiment with a bipartite model of narrative generation (Young, 2007). Story elements such as plot and character act as plans to affect a dynamic virtual environment. Discourse elements, on the other hand, act as plans to communicate and convey the story world's plan structure (causing events which advance the main plot).

2.3.2 Level and Map Generation

Levels and maps vary drastically across genres, as do the techniques used to generate them. One of the more common applications of PCG is terrain generation. For example, *Minecraft* (Mojang, 2009) uses 3D Perlin noise and linear interpolation to generate its terrain (Persson, 2011). In infinite/endless runners such as *Canabalt* (Saltsman, 2009), the generator must be running during play itself, continually generating more of the track (Smith, 2014).

In 2008, *Infinite Mario Bros* (Persson, 2008) was created with the purpose of offering an endless number of randomly generated *Mario*-style levels each time the game was loaded. The Mario AI competition was built around this game, and included a level generation track, in which participants would implement various PCG techniques to tailor levels to an individual's playing style (Shaker, Togelius, et al., 2011). More novel applications of level generating PCG in platformers have also been explored, for example the arrangement of geometry around a rhythm the player should feel in their hands whilst playing (Smith, Treanor, et al., 2009).

Togelius et al. use a search-based approach to generate maps for strategy games based on a number of objectives for predicted player experience (Togelius, Preuss, and Yannakakis, 2010). The space of possible maps is then searched for candidates which satisfy pairs of these objectives (though these may be partly conflicting). This is achieved using a multiobjective evolutionary algorithm. Generated Pareto fronts are used to display how the conflicting criteria can be compromised, in order to choose appropriate maps which balance them both.

2.3.2.1 Procedural Generation of Dungeons

One of the most significant uses of PCG in industry is the *Diablo* series' (Blizzard Entertainment, 1996-present) dungeon generation. The entire roguelike genre is characterised by character progression and permanent death, and often incorporates random

dungeon generation (Almgren et al., 2014). Dungeons in *Diablo* are completely randomised. Rooms and halls are positioned differently each time, some offering unique quests if randomly selected. Blizzard admit that randomised dungeons can look generic if not done well, but cite complex placement of tiles and good artists as the reason theirs work (*Random Map Generation in Diablo III* 2012).

2.3.2.1.1 Technique - Spatial Partitioning

Binary Spatial Partitioning (BSP) can be used to generate 2D dungeon layouts. An initial space is recursively split in two, randomly selecting whether to split horizontally or vertically, and to what ratio (Shaker, Liapis, et al., 2015). This results in disjointed distinctive subsets of the space, which do not overlap (see Figure 5). Williams explains that any point in the space will lie in exactly one of these subsets, or 'cells' (Williams, 2014). Then one room is generated in each cell, which could be tiled, rectangular, or an entirely different shape (see Figure 6). These rooms are then connected based on their relationships in the BSP tree, either using a brute force method which may intersect other rooms, or a more complex pathfinding algorithm.

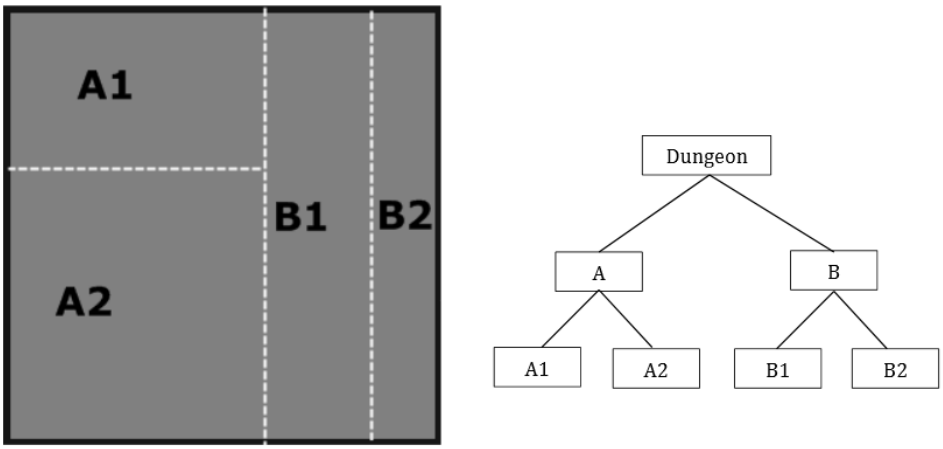


Figure 5: A space which has been partitioned twice and the BSP tree (a.k.a k-d tree) which represents it (Williams, 2014).

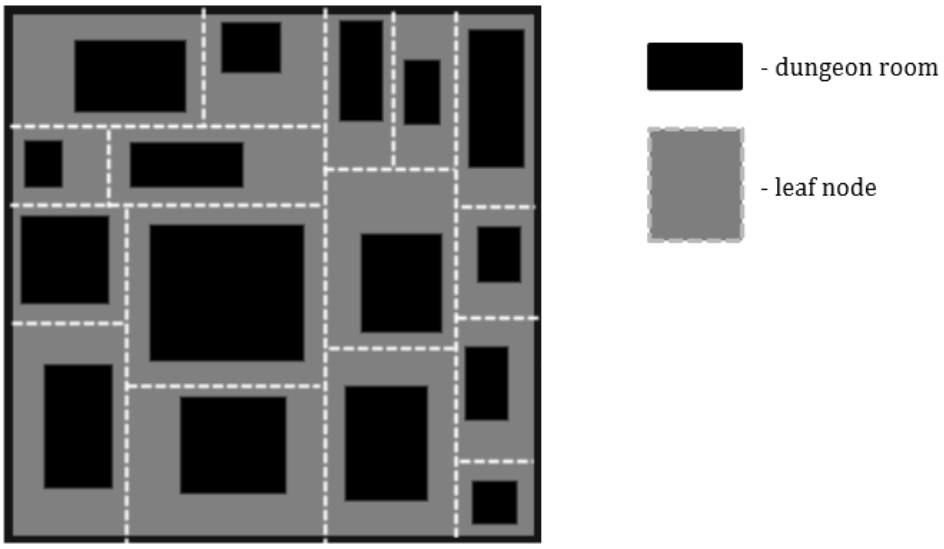


Figure 6: Rooms placed in cells at a later stage of generation, yet to be connected (Williams, 2014).

Shaker et al. describe this as a macro approach, as the algorithm acts as an 'all seeing architect' (with the full map available to query and use for decision making at all times) rather than a more blind agent (Shaker, Liapis, et al., 2015).

2.3.2.1.2 Technique - Cellular Automata

A cellular automaton consists of an n-dimensional grid, a set of states and a set of transition rules (Shaker, Liapis, et al., 2015). Cells in a map are in one of these finite states, for instance 'on' or 'off'.

Cellular Automata (CA) can be used to 'evolve dungeons in discrete steps, with cell states transitioning based on their current state and the state of those around them' (Shaker, Liapis, et al., 2015). These surrounding cells are referred to as 'the neighbourhood' and can either be of Moor (square) or von Neumann (cross) type.

Williams gives one example of a CA process is as follows. First, a grid in which all cells are turned off is 'scattered' with random non-empty/'on' cells. This noise is distributed on to the grid. Then several generations of CA are applied, during which a cell will examine the state of its neighbourhood and transition based on its prescribed rule(s) (Williams, 2014). The results of this process can vary greatly, based on the different rule sets and neighbourhoods used.

CA is often used for exterior maps and other natural looking content, as it creates very organic structures (Dormans, 2010). It has been demonstrated that CA can be used to create playable infinite cave maps in real-time, which mimic the natural aesthetic found in real caves (see Figure 7) (Johnson, Yannakakis, and Togelius, 2010).

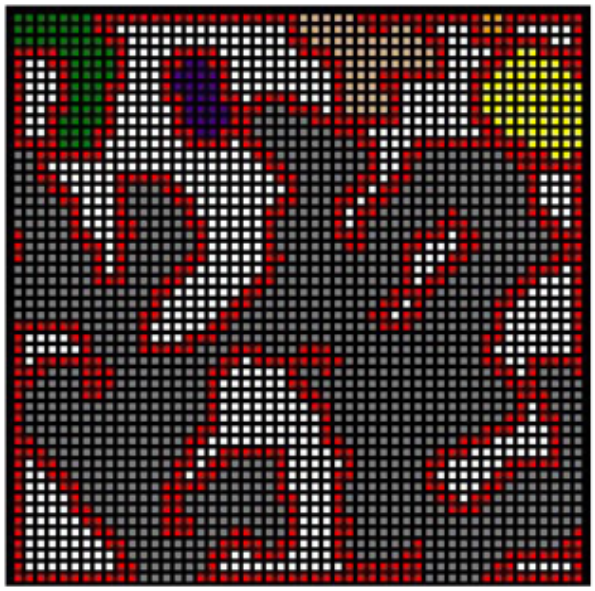


Figure 7: A map generated with Cellular Automata (Johnson, Yannakakis, and Togelius, 2010)

2.3.2.1.3 Technique - Agent based 'drilling'

In this approach, a single agent begins at a certain location in the 2D space and then 'digs' tunnels and rooms outwards (Shaker, Liapis, et al., 2015). The AI moves through the space creating corridors, with varying chances of turning left or right, or adding a room, each step.

The AI can be tweaked to make different decisions, resulting in heavily varied dungeon styles (Shaker, Liapis, et al., 2015).

Shaker et al. describe this as a micro approach, as the agent does not have full knowledge of the map so far, outside of its immediate vicinity, when making decisions. The result is more organic, chaotic dungeons than produced with methods such as spatial partitioning. However, the amount it can 'see ahead' can be tweaked, to generate less unruly, more practical dungeons (Shaker, Liapis, et al., 2015).

Each of these methods has a wide variety of benefits and drawbacks, an in-depth analysis of which is beyond the scope of this review. The implications some of these techniques have on AA dungeon generation in particular are discussed briefly in the next section.

2.4 The Challenges of Generating Action-Adventure Dungeons

The action-adventure genre, and the nature of the puzzles it contains, pose unique problems for random generation. In dungeons in particular, these puzzles are often of the 'key and lock' type (Dormans, 2010). A key does not explicitly have to be a literal key, it could be an item which must be used to trigger a switch, or an ability that allows the user to traverse previously unnegotiable terrain. With dungeon generation, keys cannot be generated behind the locks they correspond to. This sounds like a simple problem at first, but can be quite complex to solve with different generation techniques.

Some techniques lend themselves more to the idea of key and lock puzzles than others. For instance, the spatial partitioning technique described in section 2.3.2.1.1 has several

benefits due to the way the dungeon is stored in a BSP tree. By placing a key in a dungeon space on a lower level down the tree than its corresponding lock, you can ensure it will be possible to find the key before unlocking the door (Willems, Sascha, 2010).

Dormans and Bakkes discuss why dungeons in action-adventure games pose a very different problem. AA games are described as 'story-driven games where exploration, puzzle-solving, conceptual and physical challenges make up the majority of the gameplay' (Dormans and Bakkes, 2011). Dormans and Bakkes then explain that AA does not rely on any elaborate character development system, and instead use level-design as the prime source of gameplay. Levels for this genre must incorporate devices such as a structured learning curve, clever pacing of action, challenges and puzzles. These devices are difficult to implement with a lot pre-existing dungeon generation algorithms, as the structures span several rooms and form the level as a whole.

When discussing the challenges of PCG, Togelius et al. acknowledge that most generated content looks generic, and most procedurally generated maps, such as those in roguelike games, are often only superficially different from one another. *The Legend of Zelda* dungeons are given as examples of contrasting structures which are masterfully designed, are aesthetically pleasing, clearly convey pace and progression, and 'often [offer] some original and unique take on the design problems of action-adventure games' (Togelius, Champandard, et al., 2013). This not only explains why it is so desirable to be able to generate such levels, but why the task is so difficult when compared to other genres.

2.5 Methods of Solving These Challenges

2.5.1 Combining Generation of Quests and Levels

Combining the generation of Quests and Levels is a very beneficial area of work in PCG research. In the paper Procedural Content Generation: Goals, Challenges and Actionable Steps, Togelius et al. discuss the main goals and directions of research which would best fulfil PCG's potential (Togelius, Champandard, et al., 2013). Two of the challenges described as standing in the way of these goals are: interaction and opportunistic control flow (collaboration between different types of generators) and general content generators (which create multiple types of content). Togelius et al. give linked quest and map generation as a step which would move towards solving both these challenges. Vast research has been done into the generation of quests and the generation of maps, but 'surprisingly little' has been done to generate these two facets together (Togelius, Champandard, et al., 2013). One of the main goals outlined, multilevel PCG, could be showcased in this way, with a map helping to tell a story and a story helping to explore a map. Togelius et al. also previously cited the bland, random feel to PCG content and gave *Zelda*'s 'unique take on the design problems of action-adventure games' as an example of a concept which has not yet been successfully generated (Togelius, Champandard, et al., 2013). Linking the generation of missions and maps gives much needed context to resultant worlds and dungeons.

2.5.1.1 Charbitat

To conquer the tendency towards 'random' feeling content, Ashmore and Nitsche state that 'procedurally determined context is necessary to structure and make sense of [procedurally generated] content' (Ashmore and Nitsche, 2007). This concept has been applied

in practice to the project Charbitat, an experimental game which uses a full modification if *Unreal Tournament* (Games, 1999-present) and a Java backend (Nitsche et al., 2006).

2.5.1.1.1 Designing Procedural Game Spaces

Charbitat was created to demonstrate a world generated on-the-fly, based on the player's previous in-game actions (Nitsche et al., 2006). The world consists of square tiles, 500m across, which remain empty until the player steps on to them (see Figure 8). Then, an individual seed is generated for the tile based on the player's actions, the current elemental state of the world and the elemental state of neighbouring tiles (for instance, a fire tile cannot be placed adjacent to a water one). The seed is used to generate the tile with a specific element's theme. Terrain, enemies and objects are all generated to match the theme of the tile.

Figure 8: A pre-modeled goal tile in *Charbitat* (Nitsche et al., 2006).

When generated levels were tested, Nitsche et al. found a problem in that navigating the tiled, randomly generated environment was very difficult. This was improved with the creation of distinctive terrain features and landmarks which spanned several tiles at once.

2.5.1.1.2 The Quest in a Generated World

The team were led to a desire for context in the *Charbitat* world, because though players could explore and expand the world, there was no immediate direction or contextual link between tiles (Ashmore and Nitsche, 2007). Even though there was a final goal which required the player to reach all goal tiles, and in doing so heal different sections of the world, the space was not limited in any way. Thus a quest structure comprising

lock and key type limitations was devised to organise the generated space, and give it purpose.

In order to integrate this quest into the procedurally generated world, Ashmore and Nitsche implemented a process involving a graph representing the relationship between keys and locks in the Java backend. Nodes of the graph represent game spaces, and are connected to other game spaces within the same tile and within adjacent tiles. Edges between nodes can have locks, for instance a river acting as a swim skill lock (see Figure 9). Each time a new tile is generated, this graph is thoroughly analysed to ensure that all locks remain lockable and keys are stored in the correct sections of the map. All tiles which could possibly be generated are analysed by evaluators, which score each tile based on programmatically defined rules. For instance, one evaluator scores highly only when at least 3 instances of a lock will likely be found by the player before they find the corresponding key. The highest scoring possibility then determines what kind of keys and locks could be placed. Ashmore and Nitsche describe the key and lock puzzle as the bridge between the generated space and the quest (Ashmore and Nitsche, 2007).

Figure 9: A tile has been split by a river, which acts as a navigational 'lock' (Ashmore and Nitsche, 2007).

Ashmore and Nitsche state that all quests in *Charbitat* are linear, but this is counteracted by keys having multiple functions, giving the player more diverse freedom. This concept draws on the inspiration of games like *Zelda* and *Metroid* (Nintendo, 1986-present), which have keys act as weapons or make it easier to backtrack, as well as primarily being used for progression. The search for keys is also heavily dramatized with multiple obstacles and challenges.

2.5.1.2 From Plot Points To Maps

Story-based map generation can be achieved using a collection of plot points (Valls-Vargas, Ontanón, and Zhu, 2013). Valls-Vargas et al. recognised the tight relationship

between stories and virtual worlds (for instance, not wanting the player in a murder mystery game to leave a certain room until a certain event has occurred) and identified the long term goal of producing maps that supported multiple stories. The system created takes a collection of plot points (e.g. 'The player discovers the existence of a hidden door') as input, and generates spatial configurations. An intermediate graph structure is used to describe the spatial relationships between locations on the map. It then generates all possible stories (given the input plot points) that can unfold in each of those configurations, using an approach found in planning based story generation systems. These possible stories are then assessed on thought flow, activity flow, manipulation, intensity, action consistency, and length. The system takes the results of this assessment and uses them to pick one of the spatial configurations to graphically realise. In doing so, it can build maps that house one or more stories from the given story space. These stories can include cycles and allow non-linear storylines.

2.5.1.3 Game Forge

The project *Game Forge* also uses a plot-point based system to generate a game world (Hartsook et al., 2011). Plot points are defined similar to previous, as 'high level specification of a period of time with a semantic and recognisable meaning', for instance fighting/killing enemies, buying/selling objects, and conversing with non-player characters. Hartsook et al.'s system takes a pre-written story as input, which could either be human-authored or generated by a separate PCG system (see Figure 10). The system also incorporates the user's preference for play style in the world generation (see Figure 11). This information is received via a questionnaire which is taken before the game is played. This player model affects 'bridge' length, number of sidepaths, chance of random encounters and likelihood of finding treasure.

1. **Take** (paladin, water-bucket, palace)
2. **Kill** (paladin, baba-yaga, water-bucket, graveyard1)
3. **Drop** (baba-yaga, ruby-slippers, graveyard1)
4. **Take** (paladin, shoes, graveyard1)
5. **Gain-Trust** (paladin, king-alfred, shoes, palace)
6. **Tell-About** (king-alfred, treasure, treasure-cave, paladin)
7. **Take** (paladin, treasure, treasure-cave)
8. **Trap-Closes** (paladin, treasure-cave)
9. **Solve-Puzzle** (paladin, treasure-cave)
10. **Trap-Opens** (paladin, treasure-cave)

Hero (paladin), NPC (baba-yaga), NPC (king-alfred), Place (palace),
Place (graveyard1), Place (treasure-cave), Thing (water-bucket),
Thing (treasure), Thing (ruby-slippers), Type (baba-yaga, witch),
Type (king-alfred, king), Type (palace, castle),
Type (graveyard1, graveyard), Type (treasure-cave, cave),
Type (water-bucket, bucket), Type (ruby-slippers, shoes),
Type (treasure, gold), Evil (baba-yaga) ...

Figure 10: A story made up of plot points and its initial state. The format is described as 'consistent with AI-planning based story generation systems' (Hartsook et al., 2011).

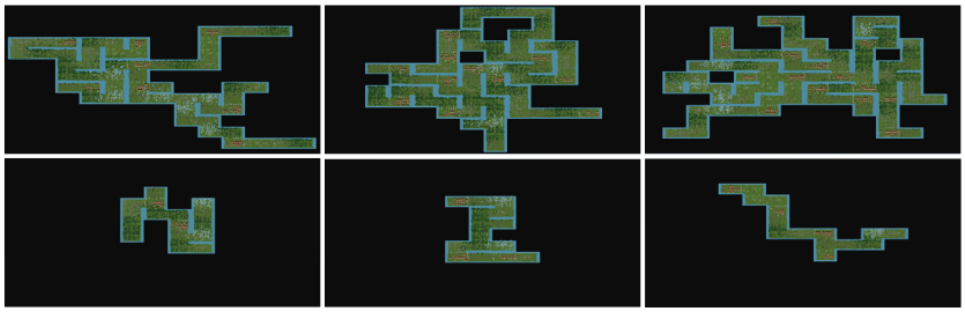
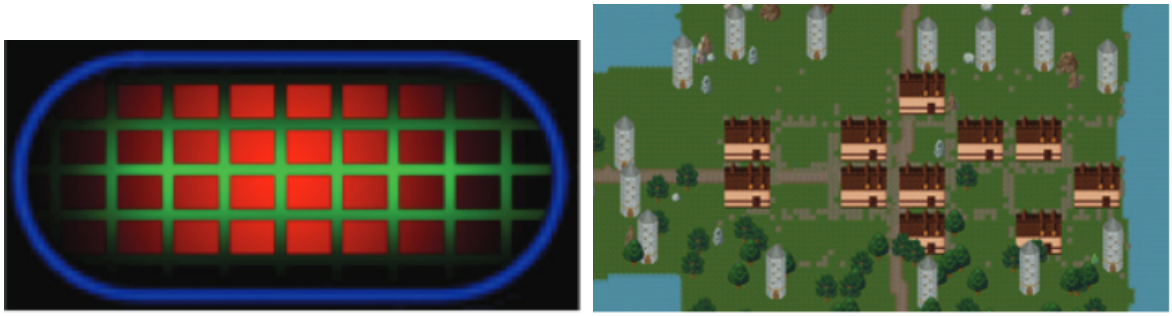


Figure 11: Each column contains maps generated for the same plot. The top row reflects parameters for a large, branching world whereas the bottom row reflects parameters for smaller, more linear ones (Hartsook et al., 2011).

The human-made plot and set of play-style preferences are input into the system, which then searches for a sequence of changes which will transform the plot into one which meets those preferences, by adding and removing plot points (Hartsook et al., 2011). The map is then generated using a genetic algorithm guided by a fitness function which incorporates the game world model, the player model and the plot points. Map generation uses a metaphor of islands and bridges, with islands being main settings like a castle or graveyard (see Figure 12), and bridges being the spaces between them, which facilitate random encounters and non-essential treasure discovery. This system can only work with linear stories consisting of a main storyline and no sidequests.



(a) *Tile distribution map.* (b) *Generated castle layout.*

Figure 12: Tile placement is also generated. Figure 12b is the result of castle tiles being distributed as dictated by the distribution map in Figure 12a (Hartsook et al., 2011).

2.5.2 Lenna's Inception

In his talk on Flow in Procedural Generation at PROCJAM 2014, Tom Coxon describes his procedurally generated game *Lenna's Inception* (see Figure 13), in which the player progresses though gaining items to be used in lock and key based systems, as in the *Zelda* series (Coxon, 2014). Coxon states that the game is often described as 'addictive', and explains this is because it takes account of flow when generating levels, following a pattern which builds up difficulty and frustration before peaking, and then constructs more easy going periods of gameplay afterwards. This creates a flow in which progression becomes a rewarding experience in itself.

Figure 13: A screenshot of *Lenna's Inception* which displays the similarity in aesthetic and gameplay style to *The Legend of Zelda* games (*Lenna's Inception announced* n.d.).

Coxon talks about the added complexity when difficulty scaling for multi-path dungeons, as opposed to linear ones in which difficulty can scale with distance from entrance (Coxon, 2014). Dynamic approaches such as basing enemy level on player level, or the AI director from *Left 4 Dead* (Turtle Rock Studios, 2008), were not appropriate for *Lenna's Inception*. Coxon instead uses a static approach involving lock and key puzzles, which are useful not only for controlling spatial difficulty, but also for incorporating hand written story into the procedurally generated world (see section 2.5.2.2).

2.5.2.1 Lock and Key System

To implement his lock and key system in dungeons (in this case rooms connected by corridors), Coxon uses a sequence of keys and determines how many keys a player needs to be able to access a certain room (*Overworld Overview - Part 1* n.d.) (if the player has the third key, they must also have 1 and 2). These keys are then positioned in rooms which require only the previous keys to enter, accordingly (see Figure 14).

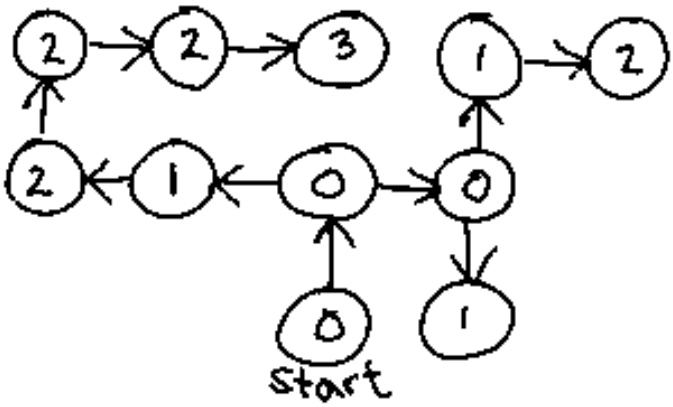


Figure 14: Each room has a key number, which increases from the start of the dungeon. Locks are placed between these differently numbered rooms and the keys placed in rooms between the entrance and the locks they are required to open (*Overworld Overview - Part 1* n.d.).

To avoid generated dungeons with tree structures which force the player to spend a large amount of their time retracing their steps, extra corridors (see Figure 15) are then added between rooms with the appropriately numbered lock (this will always take the highest key index from the two rooms being connected). Difficulty of these rooms is then scaled based on the key value and desired flow curve. As action-adventure games do not rely on easily scaled stats the same way RPGs do, difficulty must instead be determined by number/type of enemies encountered.

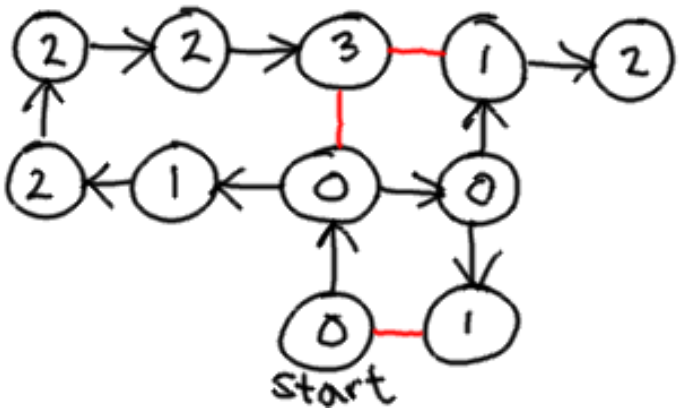


Figure 15: Red connections represent extra corridors between rooms, which prevent tedious backtracking (*Overworld Overview - Part 1* n.d.).

Using this system, it is possible to hand design the overall dungeon space, and then keep generating rooms until that space is filled.

2.5.2.2 Overworld Generation

Not only are dungeons generated in *Lenna's Inception*, but also the overworld. Coxon discusses other generation techniques in games and identifies that some games use templates of particular rooms or areas, which they then piece together with PCG (*Overworld Overview - Part 1* n.d.). Coxon's method moves away from this, and is instead based

on another pre-existing method: generating a solution path to traverse and filling the remaining space by generating other content. This path is extended with lock and key puzzles. The advantage of this method is that rooms or areas can be any size and not just one per screen. To briefly summarise, Coxon's technique is comprised of 3 stages:

1. Terrain generation
2. Lock and key generation
3. Spatially aware room content generation

For terrain generation, Coxon uses Perlin noise to generate mountains, seas etc. The locks and keys in *Lenna's Inception* are more like those in *DOOM* (id Software, 1993-present), where a certain key only fits doors that correspond to it, rather than *Zelda* small keys, where several different keys can each fit several different locks. Coxon states that *Zelda*-style small keys would be more difficult to implement with PCG. For the spatially aware content, Coxon generates the placement of individual tiles in the overworld, using techniques such as 'growing bubbles' of terrain to create more natural looking locks (*Overworld Overview - Part 2* n.d.). Figure 16 shows the result of this growth process after the top screen path, left screen path and stairs to the right were blocked by water, rocks, and grass respectively. The generator ensures that spaces are left for eg, standing in front of the grass to cut it or standing next to a rock to lift it.

Figure 16: A result of the 'bubble' tile placement process (*Overworld Overview - Part 2* n.d.).

Coxon explains the way this structure rewards the player with more freedom as they progress. 'Each time you get a new key item, there are more areas open to you, more routes back to previous important areas, and more ways to travel, until at the very end when you actually are in an open world. So it often doesn't feel very linear to a

player' (*Overworld Overview - Part 1* n.d.). It also allows subplots to take part in the overworld at specific points in the story. For instance a subplot could start in an area accessible at keycount *n*, and continue only in an area with keycount *n + 1*. Smaller sub-areas can be structured in the same way, creating key systems within key systems.

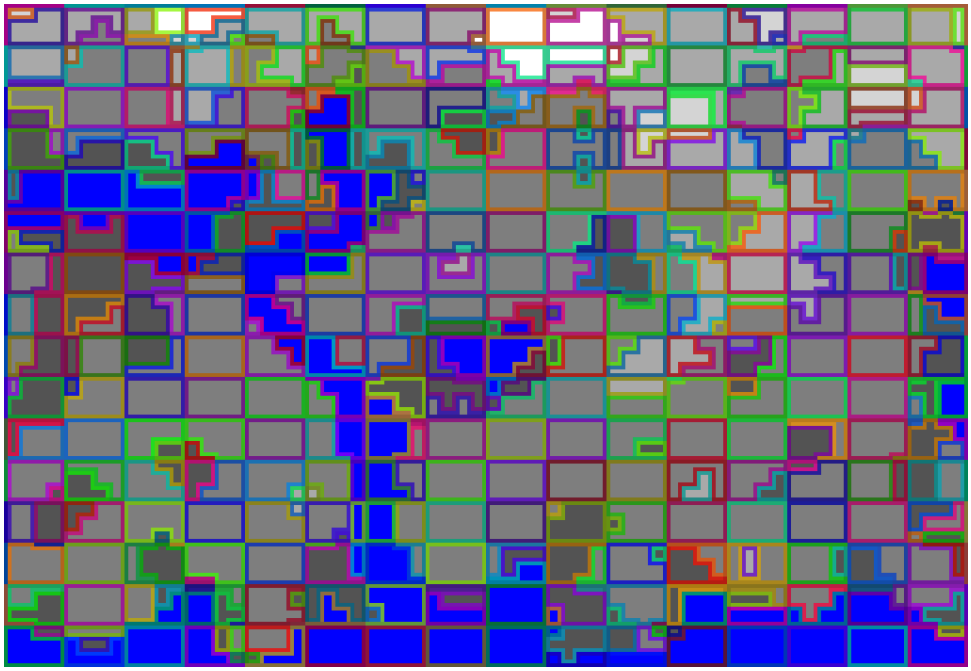


Figure 17: An Overworld divided into 'rooms' (*Overworld Overview - Part 1* n.d.).

Figures 17 and 18 show how the key and lock system can be used to divide an overworld. The world is split into rooms, both from natural blockages in the terrain generated such as cliffs, and edges between screens are treated as potential room borders. Stairs and bridges can be used to link these 'rooms' where necessary.

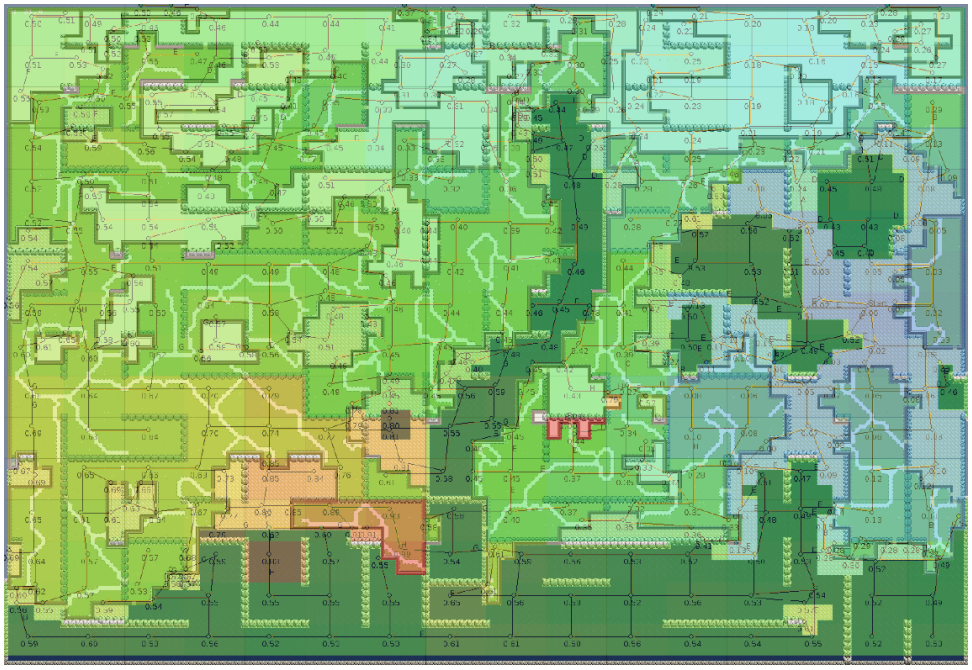


Figure 18: Map from Figure 17 with lock and key diagram such as that in Figure 14 placed over the top. Colour corresponds to difficulty of monsters and obstacles, which increases as the player gets further thanks to the lock and key system (*Overworld Overview - Part 1* n.d.).

2.5.3 Generative Grammars

Dormans and Bakkes argue that aspects of AA design such as flow, pacing and structured learning curves can be excellently handled by large abstract structures in generative grammars (Dormans and Bakkes, 2011). Using these structures, generation can focus on both the big picture and fine detail. In doing so it can ensure that all parts fit together in a structured way, instead of being placed one after the other in a purely random manner.

2.5.3.1 Background

Chomsky coined the term generative grammars and first used them as a linguistic tool, for working with the structure of linguistic phrases (Chomsky, 1956). They are used commonly to study syntax. Grammars involve a rewriting technique, in which complex objects are generated 'by successively replacing parts of a simple initial object using a set of rewriting rules or productions' (Prusinkiewicz et al., 1990). A grammar consists of an alphabet containing words or symbols, and a set of rules which determine which symbols can be replaced with which others. In linguistics, it is in theory possible to create a grammar which could be used to get to any possible grammatically correct sentence in a language from a starting rule.

However, the nature of grammars makes them applicable to study and implementation in many different disciplines. For instance, Lindenmayer Systems (or L-systems) were first proposed as a mathematical theory of plant development, and are now used in many areas of computer science (Prusinkiewicz et al., 1990). In Chomsky grammars, rules are applied one by one, sequentially, whereas in L-systems several rules can be applied at once – the same symbol when appearing in multiple places, can be replaced simultaneously in parallel (see Figure 19). It is this quirk which makes L-systems applicable to biologically focused

systems, replicating multicellular organisms in which many cell divisions may occur at the same time. Several geometric interpretations of L-systems have been proposed as versatile tools for modelling plants. (Prusinkiewicz et al., 1990). Other applications of L-systems range from creating music (DuBois, 2003) to generating designs for houses and architectural plans (Goel and Rozehnal, 1991).

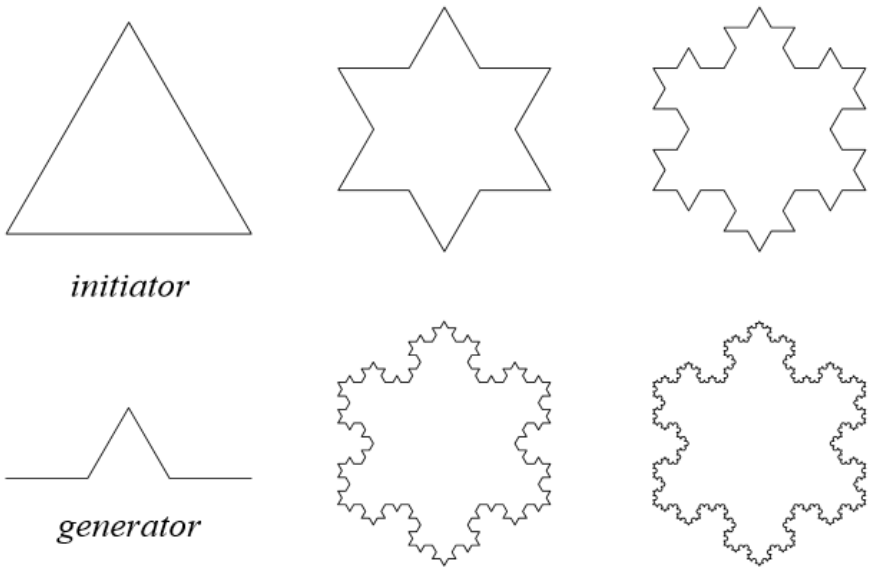
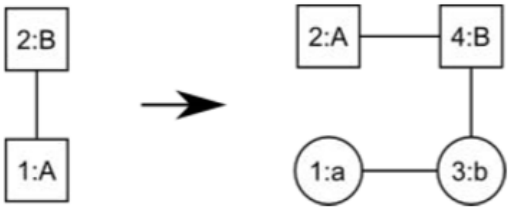


Figure 19: An initial shape and a rule which adds a spike shape to a straight line are used to construct a 'snowflake curve' (Prusinkiewicz et al., 1990).

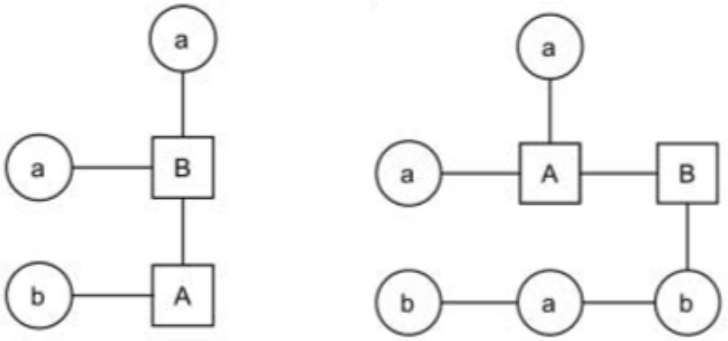
2.5.3.1.1 Types of Grammar

Though grammars originated in the context of strings, the concept can be easily applied to other domains, as the snowflake curve demonstrates.

Figure 20a displays an example of a rule from a graph grammar, with the alphabet a, b, A, B. Instead of describing which characters in a string are replaced with which, it describes which nodes and edges in a graph should be replaced with others. Applying the rule in 20a to the graph in 20b gives the output shown in 20c.



(a) *The rule to apply.*



(b) *The initial graph.*

(c) *The resulting graph.*

Figure 20: An example of a simple graph grammar rule and its application (Dormans, 2010).

Figure 21 demonstrates a shape grammar for producing very simple dungeon layouts. The alphabet is displayed in 21a, and consists of an unresolved connection, a door (or space between rooms) and a wall, respectively. 21b displays the 3 rules, in this case all describing what an unresolved connection can turn into. When they are applied randomly to a starting node of one unresolved connection, it is possible to get the resulting map displayed in 21c.

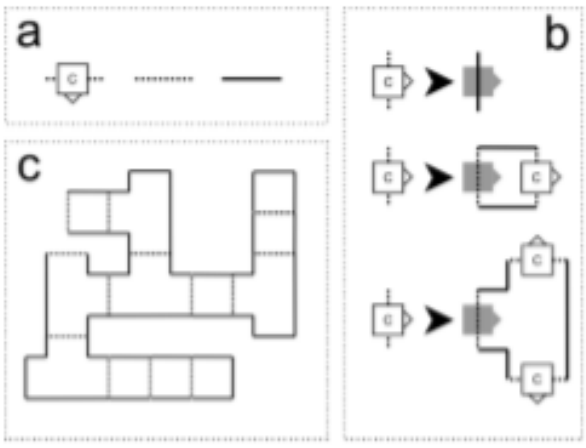


Figure 21: A simple shape grammar for map layout generation (Dormans, 2010).

2.5.3.2 Application to Games

Grammars have been applied to video games in a wide variety of contexts. The aforementioned L-systems have been used in many games (and movies) via the software SpeedTree,

a toolkit for creating 3D animated plants and trees (IDV, 2014).

The rhythm based platformer mentioned in section 2.3.2 is an example of how different types of grammars can be combined in the generation process: first using one grammar to generate rhythm, and then using a different grammar to generate geometry based on those rhythms (Smith, Treanor, et al., 2009).

Grammar-based methods were also used in the level generation tracks of the Mario AI competition. Shaker et al. evolved playable platform levels using grammatical evolution (Shaker, Nicolau, et al., 2012). The grammar 'allows simple encoding of important level design constraints, and allows remarkably compact descriptions of large spaces of levels.'

Epic games used a collaborative system between grammars and artists/designers to generate a wide range of different buildings, which demonstrated automatically generated LODs (lower-detail versions of 3D objects which can be displayed when the object is far from the camera, and are used for optimisation) (Golding, 2010). These buildings could be easily tweaked by changing the rules of the grammar, and kept the artist and level designer's workflows from colliding.

Merrick et al. demonstrated a system which used a shape grammar formalism and a model of creativity to produce maps which exhibited the usefulness and value of human-made designs whilst introducing novel variations (Merrick et al., 2012). The project aimed to explore whether a system could judge content for novelty, surprise, usefulness and value, taking a similar approach to a typical human designer.

Several university research papers have explored the use of grammars to generate dungeons for games. Ince's work is one example, using context-free grammars to represent levels as regular expressions and create dungeons which are either: purely random, weighted based on difficulty or weighted based on size (Ince, 1999). Adams created a custom, general-purpose graph grammar system and used it to create a dungeon level generator (Adams, 2002).

2.5.3.3 Application to Action-Adventure Dungeons

Dormans explains how dungeons can be represented using grammars when the grammar's alphabet contains appropriate symbols such as keys, locks and monsters (Dormans, 2010). An example is given of the rules in Figure 22a generating the results in Figure 22b.

1. Dungeon → Obstacle + treasure
2. Obstacle → key + Obstacle + lock + Obstacle
3. Obstacle → monster + Obstacle
4. Obstacle → room

(a) *Rules.*

1. key + monster + room + lock + monster + room + treasure
2. key + monster + key + room + lock + monster + room + lock + room + treasure
3. room + treasure
4. monster + monster + monster + monster + room + treasure

(b) *Possible Results.*

Figure 22: Basic example of how dungeons can be conceptually laid out using grammars. Symbols beginning with upper case are non-terminal (still to be replaced), lower case symbols are terminal (final) (Dormans, 2010).

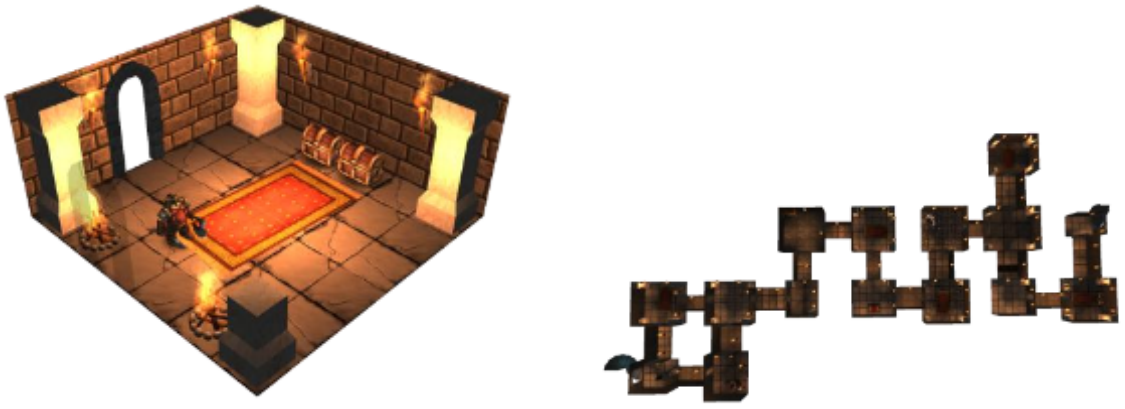
This is obviously a very high level example, but successfully introduces the concept of representing dungeons as symbols.

2.5.3.3.1 Case Study - van der Linden

The need for PCG to be more accessibly controlled by specifying and limiting what PCG systems can produce has been pointed out (Linden, 2013). Van der Linden acknowledges that the majority of PCG research has focused on areas other than gameplay-based control on the generation process. A system is then introduced in which designers input ordinary, high level, gameplay oriented vocabulary. A gameplay grammar, made up entirely of tasks the player has to perform throughout the dungeon, is produced. This gameplay based control allows player-based rhythm, game narratives and game missions to steer the level generation process. Van der Linden aimed to create a system which was more generalizable than other existing grammar-based approaches to dungeon generation, and could be applied to a wide range of games and genres.

In van der Linden's system, parameters to control the generative grammar, as well as different types of spatial relationships, can be specified and controlled. Co-located actions must occur in the same spot, for instance killing a dragon and looting that same dragon. Semantically connected actions, such as finding a key and putting that key in a lock to open a door, have a more complex relationship and do not necessarily have to occur in the same place. The system incorporates branching, generating alternative paths to take in order to achieve certain goals.

Van der Linden then applies this system to the dungeon crawler game *Dwarf Quest*, which contains maps orthogonally placed on a 2D grid with a maximum of four connections per room (see Figure 23b). The system takes the gameplay-grammar generated action graph and yields a room graph. It does this by first turning nodes into rooms and edges into hallways. It then uses layout pre-processing to make the resulting graph planar, with no overlapping edges and no more than 4 edges from each node. Long edges are compressed, with extra rooms being added into those which cannot be compressed further. To apply the result to a *Dwarf Quest* map, it selects possible predefined room configurations based on the content of the graph's action nodes (see Figure 23a). Finally, it marks semantically connected pairs like doors and switches to be dealt with appropriately by the engine.



(a) *Room generated to fulfil 'Loot Treasure' action.*

(b) *Example map layout.*

Figure 23: A *Dwarf Quest* room and an example of a generated dungeon layout (Linden, 2013).

Van der Linden's system has been evaluated for responsiveness, effectiveness, danger

and complexity (Linden, 2013). However, it is planned that the project be evaluated through designer use in the future for metrics such as intuitiveness, as the system's high level vocabulary input was created with designers in mind.

2.5.4 Smart Terrain Causality Chains

Dart and Nelson approach the adventure game puzzle problem in a very different way (Dart and Nelson, 2012). Smart Terrain Causality Chains (STCCs) are introduced to solve and generate puzzles, taking advantage of a duality between the two processes. Once a solution to a puzzle is generated, the necessary objects to complete the solution are placed in the puzzle environment.

Dart and Nelson highlight the lack of replayability common in AA games and consider different methods of rectifying this. It is acknowledged that the technique of having branching narrative requires the considerable addition of assets, many of which may not even be seen by the player. Replayability without high level narrative variation is then proposed, by procedurally generating puzzles that reuse existing locations and fit into the existing narrative progression (Dart and Nelson, 2012).

Smart Terrain takes the form of objects within the game that can be queried for actions they can perform. To solve a puzzle, the player needs to use an object directly or use it to affect another object, which will eventually lead to the puzzle solution. This creates a chain of causal dependencies between these objects as a solution sequence (see Figure 24).

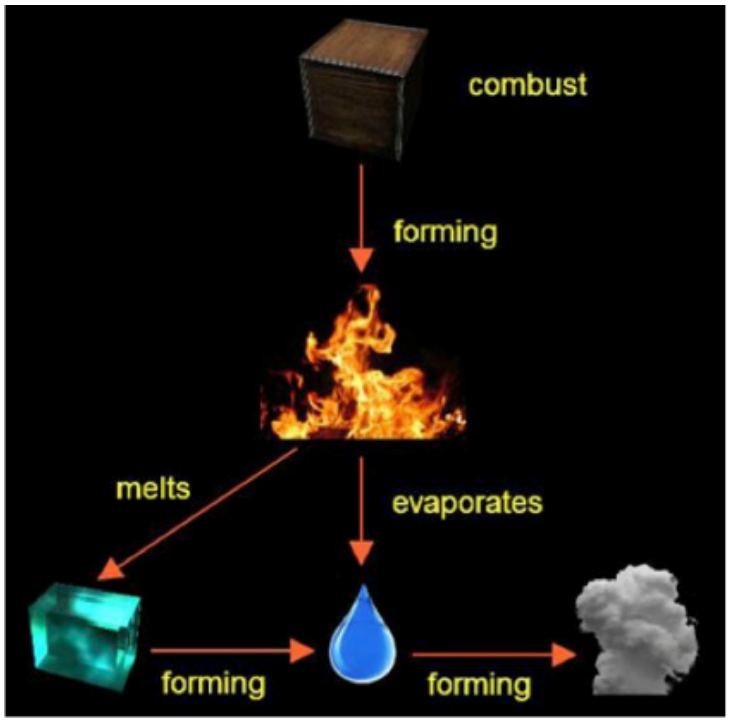


Figure 24: An example of an STCC (Dart and Nelson, 2012).

Dart and Nelson's system has very interesting implications for a number of tropes common to adventure games. It is common for a player to think of a solution to a problem which should be perfectly viable, but is not, simply because the game does not allow it. Dart and Nelson give the example of needing to find a specific newspaper

to light to make fire, when in the player's inventory there already exists a small piece of paper they used earlier to read a code from. STCCs are less concerned with hard-coded interactions such as 'if combine lighter and newspaper, make fire', and more with environmental forces such as 'if paper heat greater than x, combust'. In this way, items can affect other items which they were not strictly made aware of, or that may not even exist yet. Puzzles can therefore be solved in ways the original designer did not think to implement.

This concept was implemented in *Space Dust*, an experimental game which uses both physics simulation and causality chains to demonstrate this technique (Dart and Nelson, 2012). Figure 25 demonstrates several different generated methods for solving the same 'escape the room' type puzzle. The difficulty level of *Space Dust* can be altered by changing the number of parallel solutions to a puzzle generated – a survey of players resulted in 70% saying puzzles with more parallel solutions were easier.

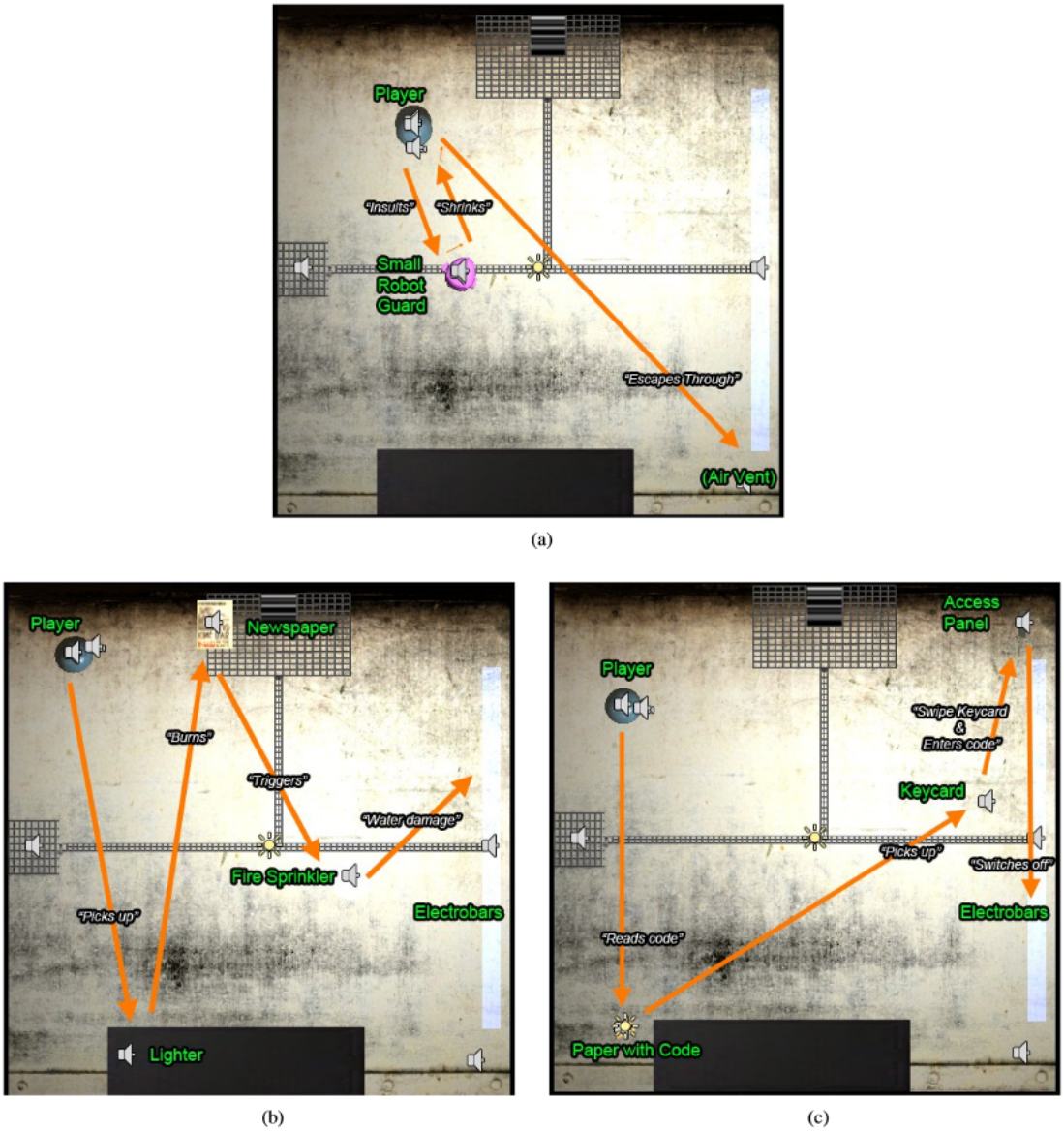


Figure 25: 3 Different *Space Dust* puzzles generated with the same objective: to escape the room (Dart and Nelson, 2012).

In most level generators, the higher level structure is explicitly defined by the generator or explicitly defined as input, and causal links between tasks and objects are then generated implicitly to fit the overall design of the level. However, Dart and Nelson explain that in the STCC approach, causal relationships are explicit and enforced by construction. Furthermore, higher level structure is implicit, enforced only by the combination of these individual causal constraints and explicitly defined constraints (such as objects required for the puzzle). The sum of these constraints form the overarching puzzle structure indirectly. This has interesting implications when comparing STCC to different generation techniques (see section 2.7.1.1.2).

However, this approach is certainly not applicable to a variety of adventure games yet, and Dart and Nelson admit that their process produced a rather 'extreme' example. For instance, in the play tests of *Space Dust*, the average player had to restart a level 10 times before beating it. Each replay of the game has a newly generated puzzle with the same objective, and the player retries until they manage to solve one of these successfully (Dart and Nelson, 2012). Most adventure games make it impossible for the player to get themselves into a situation from which they cannot recover, contrary to *Space Dust*'s approach.

All techniques discussed in this section are evaluated further in section 2.7.1.1.2.

2.6 Dormans' Implementation and Further Work

As all necessary background knowledge has now been covered, this section discusses Dormans' two-step approach to generating action-adventure dungeons in detail.

2.6.1 Implementation Details

Dormans' implementation in AiLD links two of the proposed methods: generative grammars and the combination of mission and map generation (Dormans, 2010). After the analysis of AA dungeons described in section 2.2, Dormans decided that missions and maps were such distinct concepts that they needed to be generated with entirely different methods in entirely distinctive steps.

Dormans' implementation uses graph grammars to generate missions and shape grammars to generate space. AiLD argues that graph grammars are well suited to the mission, which is best described as a complex non-linear structure for games which are centred on exploration, and that they are capable of accommodating the complex level structures described in section 2.2, such as the threshold guardian etc. AiLD then argues that shape grammars should be used to generate the space, as not only can they lay rooms out according to the associated mission graph, but they can be used to recursively create fractal, more organic shapes. Separating the generation process into two 'allows us to capitalise on the strengths of each type of grammar' (Dormans, 2010).

2.6.1.1 Mission Generation

Dormans' mission generation consists of a series of rewrite iterations on a mission graph. These iterations are carried out in a series of steps (Dormans, 2010). Using the example of a rule in Figure 26, the steps to apply the rule are displayed in Figure 27.

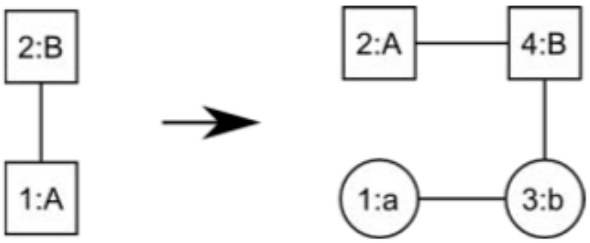


Figure 26: Dormans' example of a graph grammar rule (Dormans, 2010).

The rule in Figure 26 dictates that a pair of consecutive nodes with symbols B and A should be replaced with the structure of 4 nodes on the right hand side. The numerical identifiers assigned to each node dictate which nodes' symbols should turn into which. In this case, B will become A (as dictated by numerical identifier 2), and A will become a (as dictated by numerical identifier 1). As there are no nodes with identifiers 3 or 4 on the left hand side of the rule, B and b will simple be added to the graph.

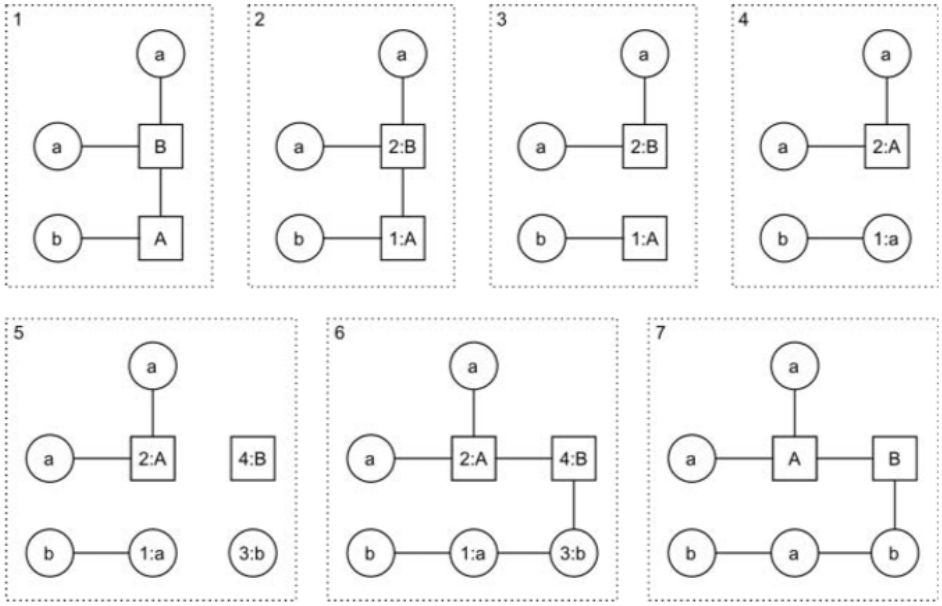


Figure 27: Dormans' replacement operations according to the rule in Figure 26 (Dormans, 2010).

Figure 27 dictates that first, selected nodes to be replaced are numbered according to the right hand side of the rule (step 2). All edges between these nodes are removed (step 3) and then the numbered nodes are replaced with the corresponding nodes on the right hand side of the rule (step 4). Any extra nodes on the right hand side are then added (step 5), edges connecting the new nodes are added (step 6), and the numbers are removed (step 7) (Dormans, 2010).

The alphabet Dormans uses in his mission grammar consists of non-terminal symbols (high-level nodes yet to be replaced, such as a 'Linear Chain' of actions to perform, 'Parallel Chain' and 'Final Chain'), and terminal symbols (final nodes such as entrance, 'key', 'lock', 'test' and 'boss'). (See Figure 28). The grammar also includes a special type of edge, referred to as a tight coupling. Represented as a double arrow, it signifies that the subordinate node must be placed behind the super ordinate in the generated space.

The coupling does not represent relationships such as key and lock, which are handled by the normal edges in the directed graph, but relationships such as a chest which should be placed directly after killing a monster or completing a test, as a reward. This coupling is used later in the space generation process, to determine the dungeon's layout (Dormans, 2010).

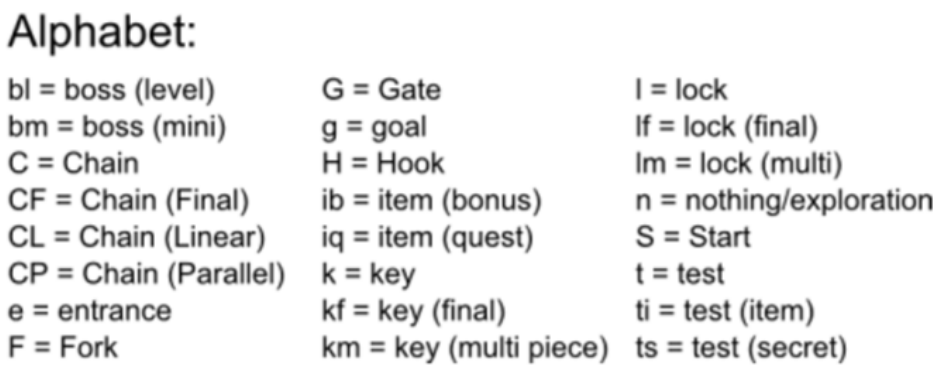


Figure 28: Dormans' mission alphabet. Non-terminals begin with uppercase letters, terminals begin with lowercase (Dormans, 2010).

The rules in Figure 29 can create the mission graph in Figure 30.

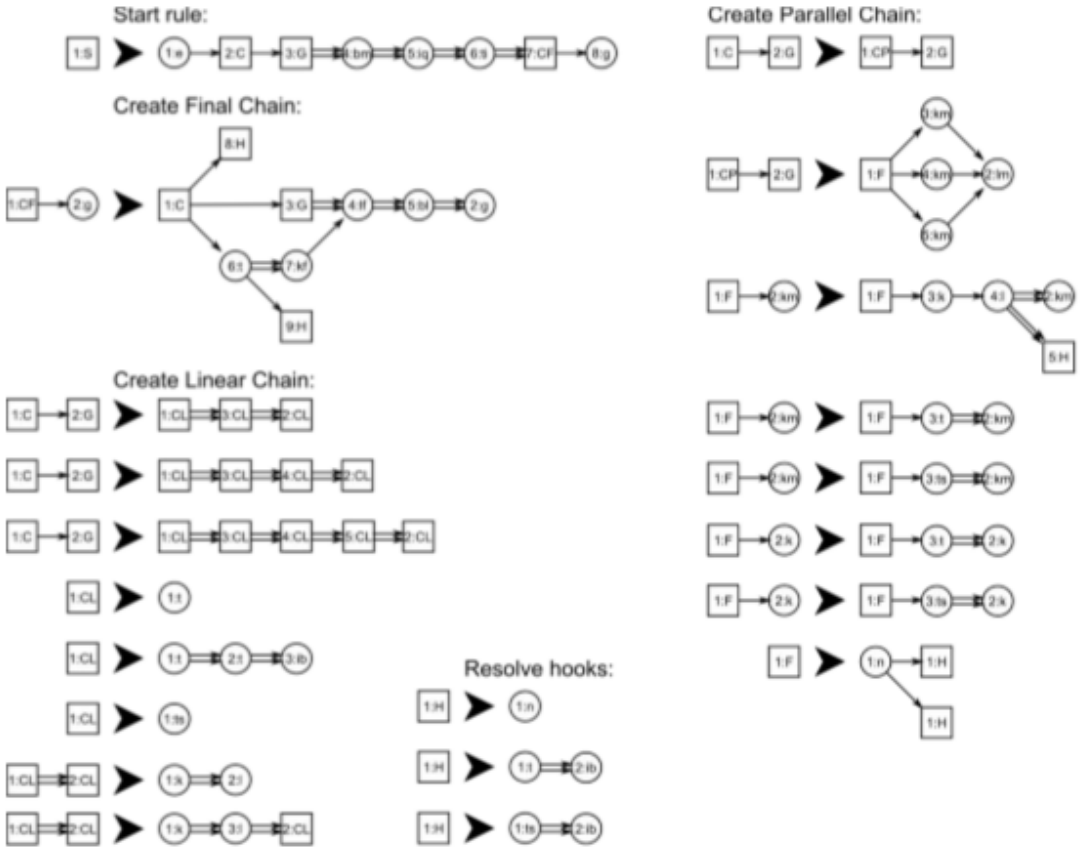


Figure 29: Dormans' set of rules used to create his mission graph. Symbols in the alphabet are described in Figure 28 (Dormans, 2010).

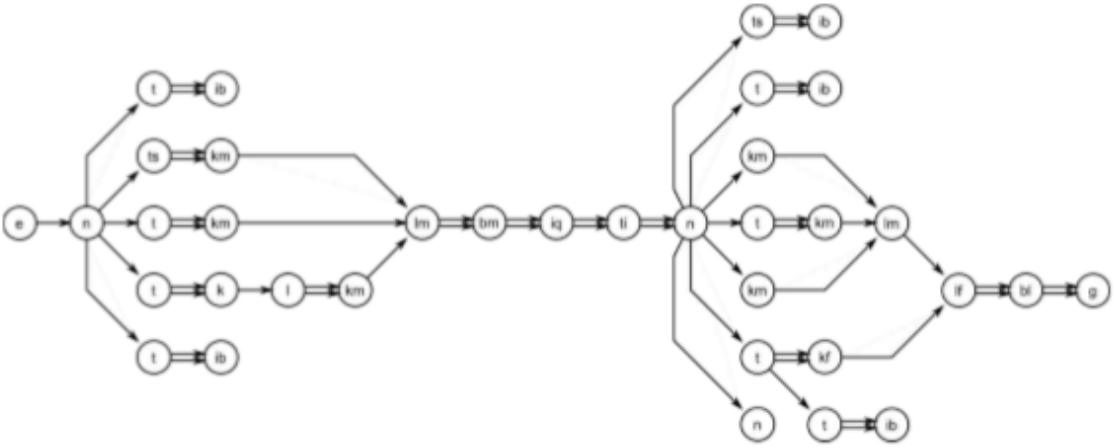


Figure 30: A mission generated from the rules in Figure 29 (Dormans, 2010).

Dormans' mission grammar is context sensitive, meaning there can be multiple nodes on the left hand side of a rule. Therefore the position of a node relative to others in the host graph can affect how and when that node is replaced (Prusinkiewicz et al., 1990).

2.6.1.2 Map Generation

Dormans uses a space grammar to piece together rooms which house the various tasks/events in the mission graph (Dormans, 2010). Each rule in the shape grammar is associated with a terminal symbol from the mission graph, so that when the mission graph is iterated through, each node will cause a corresponding room to be placed down (see Figure 31).

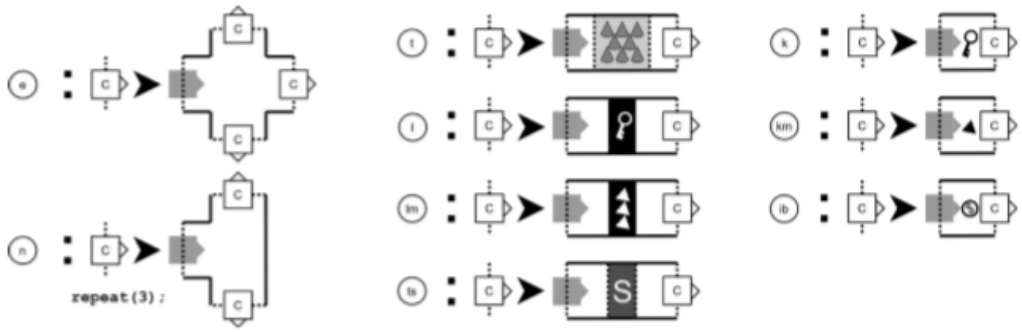


Figure 31: A selection of Dormans' space rules. Each node from the mission graph becomes a room which contains the symbol's described content (for instance t becomes a test) (Dormans, 2010).

The space generation algorithm retrieves the next symbol in the mission, selects a rule which implements that symbol at random (based on relative weight), and randomly picks the position for that rule to be applied on the map (based on the spot's relative fitness). Rooms keep track of which mission nodes they were created for, in order to implement the tight coupling described in section 2.6.1.1, and ensure keys and items are placed in meaningful 'reward' locations. This process is displayed in Figure 32.

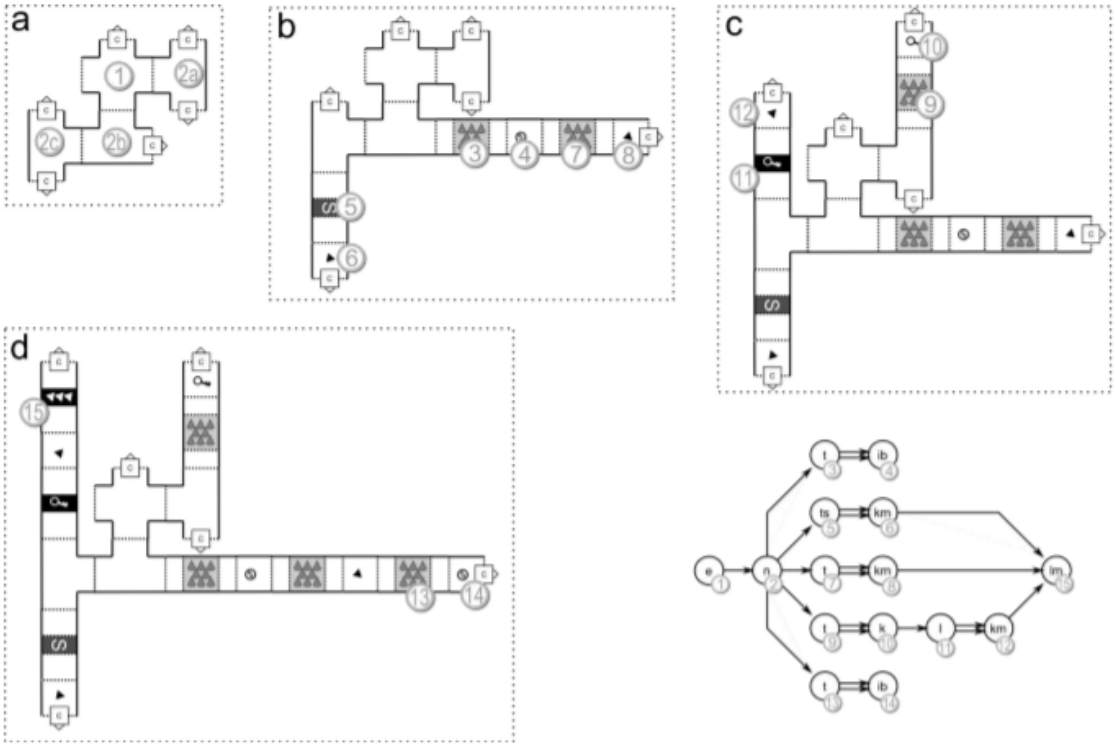


Figure 32: Dormans' example of a generated space, based on the shown section of the mission from Figure 30, and iterating with the type of space rules described in Figure 31 (Dormans, 2010).

Dormans' space implementation also encompasses several other features, such as registers which can be used to create progressive difficulty by modifying the probability of hazardous obstacles being generated, and a method for reconnecting the space to itself based on city modelling algorithms, creating cycles and reducing the need to backtrack. Once the mission has been fully realised in the space, the shape grammar iterates normally until all non-terminal symbols which remain are replaced and the space is finalised (Dormans, 2010).

2.6.2 Further Work

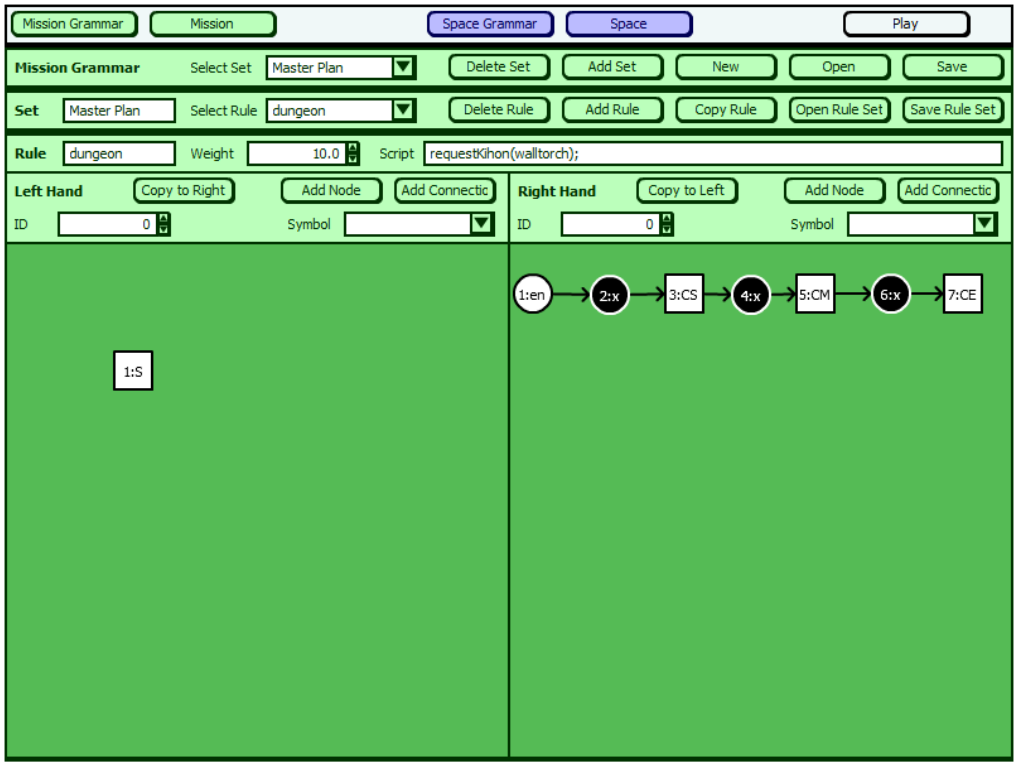
In order to securely establish the present state of research on Dormans' grammars, further and related work which specifically applied to his Adventures in Level Design paper were investigated. In one of the follow up papers to AiLD, Dormans and Bakkes drastically alter their space generation technique, retiring space grammars as a means to generate the layout of the dungeon and instead using shape grammars only to give more natural shapes to pre-existing rooms (Dormans and Bakkes, 2011). The layout itself is first generated by taking the mission graph layout and translating nodes into rooms and edges into doors (with some alterations). Dormans and Bakkes cite the reason for this change of technique as being that the shape grammar method had difficulty generating spaces for missions which allow multiple paths to converge at the same target. This is a valid criticism of the shape grammars, which tend to branch out randomly and would require complex alterations to allow multiple paths to grow with purpose and direction to meet at a certain target. It is also easily solved by the graph layout approach. However, the graph layout approach has problems such as crossing connections, but possible solutions such as

teleporters are discussed. It could also be argued that this step undermines the message of the original paper, which stated that these two grammar generation steps should be the principle methods of generating mission and space, and should be entirely separated from each other. Directly converting the mission graph layout into space feels contrary to the two step grammar principle, with shape grammars now primarily performing the task of aesthetic polish.

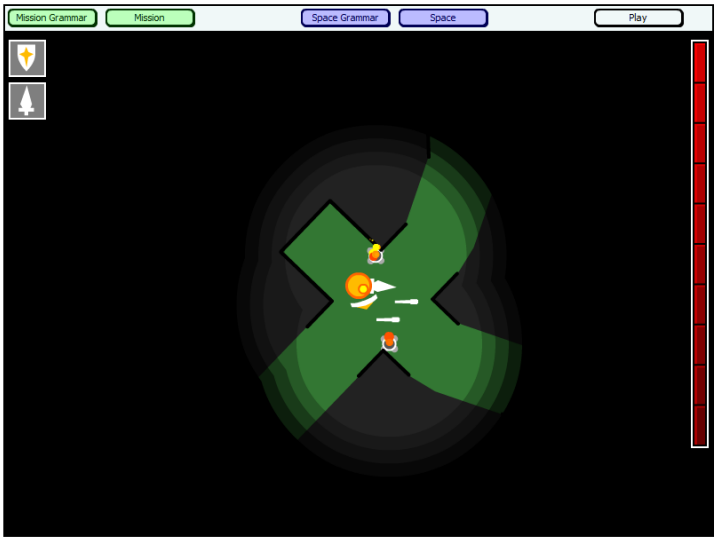
In the same paper (which concentrates on how the mission and space grammars can be used to incorporate player models at runtime) Dormans and Bakkes use a much simpler mission grammar, almost entirely consisting of tests, keys and locks (Dormans and Bakkes, 2011). The paper focuses on the distribution of these keys and locks throughout the graph, for instance with rules which move locks forwards and keys backwards, to try and enforce the design practice of a locked door being encountered before the key that opens it. In a more recent paper on combinatorial and exploratory creativity in PCG, Dormans refers back to the 'much more sophisticated' grammars used in AiLD and says that they did not generate the same variety of possible lock and key structures (Dormans and Leijnen, 2013).

In his later work, Dormans uses graph grammars for the more general task of utilising model transformations to partly automate level design (Dormans, 2011). The work focuses more on rewrite systems in general, and on the relationship between level designers and the system.

Two relevant demos can be found online – one using shape grammars and the other using the graph layout technique (Dormans, Joris, n.d.[b]; Dormans, Joris, n.d.[a]). Both demos exhibit a complex UI (see Figure 33), giving a very large amount of control to the user who can use a mix of automatic and manual iterations to generate the mission and space. This approach, where human and system work together to generate content, is referred to as Mixed Initiative (Yannakakis, Liapis, and Alexopoulos, 2014).



(a) *The grammar editing interface.*



(b) *The game produced with the grammars created.*

Figure 33: Dormans' demo (Dormans, Joris, n.d.[b]).

It appears that the pictured demo was indeed an implementation of the techniques written up in the AiLD paper, but nowhere does it make clear how a number of implementation issues with the space generation (which are not described in sufficient detail for the implementation to be copied in the paper itself) were solved. Extra steps in the space generation process such as commands executed before or after certain rules are placed are alluded to but not described in detail (Dormans, 2010). One aspect of the implementation which is not even mentioned is the order of traversal of the mission graph, in order to generate a coherent space room by room (see section 3.3.2).

2.7 Discussion and Conclusions

In order to evaluate whether or not the proposed project will be valuable, Dormans' implementation must be assessed on how well-suited it is as a method for achieving his aims, and on how it compares to other techniques presented in the review.

2.7.1 Discussion

Dormans and Bakkes discuss the general disadvantages of grammars as being initially very difficult and time intensive to set up (Dormans and Bakkes, 2011). This sentiment has been echoed from multiple sources, with Dormans' work being described as containing 'formidable challenges' (Almgren et al., 2014), and van der Linden stating that 'it takes a lot of effort to understand and work with these grammars' (Linden, Lopes, and Bidarra, 2014). This leads to one of the most common criticisms of Dormans' work – that though created to be used by designers, the grammar interface is not intuitive, and contrastingly would require a lot of specialist knowledge from a designer, who would have to be specifically trained (Linden, Lopes, and Bidarra, 2014). Shaker et al. also criticise the lack of control in Dormans' project, citing the absence of intuitive input parameters, which can aid designers when interacting with level generations (Shaker, Liapis, et al., 2015). Another problem with the shape grammar technique used is that it has difficulty 'generating spaces for missions which allow multiple paths to converge at the same target' (Dormans and Bakkes, 2011), discussed in section 2.6.2.

As Dormans and Bakkes point out, the correct implementation of these grammars means that there is no need to generate a large sample of possible results and then pick between them: it will not be necessary to verify the validity or quality of the missions generated, as the rules should ensure that the mission is valid and of good quality by default.

2.7.1.1 Comparison to Alternative Methods

2.7.1.1.1 Van der Linden's Gameplay Grammars

Van der Linden (see section 2.5.3.3.1) acknowledged the multitude of similarities between his work and Dormans'. One of his main aims was to tackle the lack of intuitive control offered by Dormans' system. Shaker et al. describe van der Lindens' work as successfully empowering designers by allowing them to use natural design oriented vocabulary. The combination of this and the designer-specified parameters on generation produce 'even more fine-grained control' (Shaker, Liapis, et al., 2015). However, it is difficult to know for sure until the system is formally evaluated by employing designer participants.

Van der Linden claims that his own work makes a significant improvement in generalisability, and can be applied to a wider array of games and genres than previous work. Van der Linden's implementation also incorporates alternative paths to achieve certain goals, which is not supported by Dormans' grammars.

One drawback of van der Linden's approach is that it uses the same mission to space transformation as Dormans' later work (see section 2.6.2), and in doing so does not explore the generation of mission and space as entirely separate processes.

2.7.1.1.2 Smart Terrain Causality Chains

Section 2.5.4 discusses how STCCs make different things explicit to other techniques such as Dormans' grammars. The STCCs enforce lower level causal relationships, and the higher level structure is the sum of these smaller links. However grammar based methods enforce the higher level structure explicitly, and only influence the lower level causal relationships to prevent causally impossible things from being produced, if the rules are specified correctly. Being able to specify the higher level structure of a level, such as in Dormans' implementation, is integral if control over patterns such as monomyth is required.

The STCC approach applies to relationships between many different types of objects, and so is more applicable to games further towards the adventure side of the action-adventure spectrum, for instance point and clicks with a constantly changing inventory. Whereas this work focuses on trying to recreate pre-existing design structures in a procedurally generated context, STCCs will create entirely new emergent types of gameplay which vastly differ from that seen before.

In Dart and Nelson's approach it is also possible for the player to get themselves stuck and not be able to recover from their situation without restarting the game. This is not possible in the majority of *Zelda* games and so not desired for this work. In Dormans' approach, correct grammars ensure that the player can always proceed.

2.7.1.1.3 Lenna's Inception

Lenna's Inception is the project most similar to *Zelda* in both aesthetic and gameplay style looked at in the literature review (see section 2.5.2). Coxon's work is a good example of how design patterns can be incorporated into procedurally generated environments. Instead of concentrating on Dormans' martial arts and monomyth structures (see section 2.2), the generator concentrates on difficulty flow. Coxon deals with the issue of difficulty scaling in multi path generated dungeons in a way Dormans doesn't, at least in the AiLD paper. However, his method of solving this issue resolves around his lock and key system, which isn't compatible with *Zelda*-style small keys. Coxon states that small keys would be more difficult to incorporate into his PCG system. The generalisability of Coxon's lock and key system is demonstrated by its application to the Overworld in *Lenna's Inception*, and by having key systems within key systems. Attempting to apply Dormans' style of grammars to an overworld scenario would be very interesting future work.

2.7.1.1.4 Charbitat

The *Charbitat* system bears more resemblance to Dormans' than it appears to at first, largely because of the differences in the games they were applied to 2.5.1.1. However, one of the largest differences comes from the fact that the *Charbitat* world is generated 'online', as the player is moving through it, and is therefore able to incorporate previous in-game actions and the state of the game world into its generation preferences. Bakkes and Dormans presented plans for using this approach to incorporate player modelling into their dungeon generation system, and it makes for a promising area of further work (Bakkes and Dormans, 2010).

2.7.1.1.5 Plot-Point Based Approaches

The systems identified in sections 2.5.1.2 and 2.5.1.3 successfully incorporate very detailed and complex stories which are closely linked to the space generation process. However, it could be argued that they are more suited to the diverse array of actions present in overworld situations than those in dungeons. The application of such concepts to dungeons could produce very interesting, deep emergent dungeon design, but is beyond the scope of this work. The success of *Game Forge*'s incorporation of player preferences into map design also provides hope for the proposed incorporation of player modelling into Dormans' work in the future (Bakkes and Dormans, 2010).

2.7.2 Conclusions

The aim of this literature review was to assess whether application of Dormans' grammars to a 2D *Zelda*-style game would have a positive contribution to PCG research (see section 2.1.1). Relevant conclusions which can be drawn with respect to that aim are as follows:

2.7.2.1 Exploring the addition of context into PCG systems is valuable research.

A tendency for existing PCG to appear random and superficial has been acknowledged. This needn't always be the case, as 'Generated spaces have the potential to intrigue and inspire the player and not merely be an open expanse or infinite dungeon' (Ashmore and Nitsche, 2007).

In order to bridge the gap between generated content and human-authored content, PCG should look to imitate pre-existing design patterns in its generation. As Ashmore and Nitsche state, 'procedurally determined context is necessary to structure and make sense of [procedurally generated] content' (Ashmore and Nitsche, 2007).

2.7.2.2 Combining the generation of missions and maps is a constructive way to apply this context, and in doing so solve generation problems of Action-Adventure games.

Section 2.4 explained the difficulties encountered when generating AA content, which requires extra context because of the more complex nature of the tasks it entails. Section 2.5.1 explains the motivation for linking the generation of missions and maps, and how doing so will take a step towards overcoming challenges such as general content generation. Togelius et al. claim that such projects showcase multilevel PCG, and give needed context to game spaces (Togelius, Champandard, et al., 2013).

2.7.2.3 Adventures in Level Design makes a strong case for using a two-step grammar approach to combine missions and maps, which warrants further investigation.

Dormans makes a sound case for his two step graph and space grammar implementation, which has benefits and drawbacks when compared to alternative methods of generation in section 2.5. However, of all techniques considered, his approach has the most potential for incorporating design structures such as monomyth.

When used correctly, Dormans' grammars produce valid and good quality results by default. It will be important to preserve this strength when implementing the grammars in a new domain, taking care not to add any factor which may result in dud graphs that need to be regenerated.

Section 2.6.2 discusses Dormans' further work and how it moves away from the original aims and assertions of the AiLD paper. Therefore, this project will focus on the grammars presented originally in AiLD.

2.7.2.4 Applying Dormans' grammars to a more typical Zelda-Style game will assist in bringing to light any limitations of the method and expand the scope of games these generation techniques have been applied to.

As Dormans' conclusion came from an analysis of a *Zelda* game in the first place, it will be note-worthy to see if his grammars can be successfully applied to one (or a very similar tiled game such as that of *Lenna's Inception* (Coxon, 2014) or *Game Forge* (Hartsook et al., 2011)). Such a project could evaluate whether the design structures specified in the grammars replicate typical *Zelda* gameplay elements effectively. It is also necessary to apply generation methods to a number of diverse AA games (either through a general system or specific, individually tailored generators) before the problems of AA game generation can be considered solved (at least in part).

2.7.2.5 Some hurdles of the two-step grammar implementation are yet to be described in detail, so it will be worthwhile to spend time documenting the implementation process comprehensively.

Detailed solutions to several implementation problems, such as the graph traversal issue described in section 2.6.2, were found to be lacking in AiLD. Consequently, this paper will explain in clear detail each step in the implementation process, and how various problems were overcome (see section 3.3.9 for a solution to the graph traversal).

3 Methodology

3.1 Overview

The project implementation process was split into three steps; mission generation using an implementation of the graph grammar designed by Dormans; space generation using an implementation of the shape grammar designed by Dormans; and application of the result of these grammars to a *Zelda*-style 2D action-adventure game. Conclusions taken from the Literature Review, such as the need to clarify certain solutions which have not yet been documented clearly, are taken into account throughout the process. The system created, which combines all three steps, will subsequently be referred to as the Zelda Dungeon Generator, or ZDG.

Section 3.2 addresses project objective 2: Mission generation. Mission generation was built to reflect Dormans' described grammar as much as possible. This was due to the fact that Dormans' rules and alphabet had been carefully designed to ensure a functional, completable result whilst reflecting a traditional *Zelda* level structure (Dormans, 2010).

Section 3.3 addresses project objective 3: Map generation. Space generation was implemented in a similar manner to that described in AiLD, with several tweaks towards what would be most appropriate for a tiled game with individual dungeon rooms. Complications discussed in section 2.6.2 are successfully resolved and documented here, in detail.

Section 3.4 addresses project objective 4: Application of generated maps to a 2D *Zelda* game. The application process of finished space to game was done to reflect as many typical *Zelda* features as possible, while retaining the general layout and structure dictated by the grammars.

In order to achieve the project aim of evaluating Dormans' grammar technique, both the development process and the product itself were evaluated based on several factors including functionality and expressivity. The game maps created were also evaluated on whether or not they successfully reflected the series of player actions that the mission graphs supplied. The method of evaluation is covered in section 4.2.1.

3.2 Mission Generation with Graph Grammar

3.2.1 Iteration Method

The concept of graph grammars are covered in section 2.5.3.1.1, while the step by step process Dormans described for applying rewrite rules to the mission graph are described in detail in section 2.6.1.1. This implementation followed the steps described in section 2.6.1.1 exactly.

3.2.2 Graph Framework

All grammar work was to be done in C# with an appropriate graph framework in order to clearly display the graphs themselves. Graph# (*Graph#* 2010) was chosen for its layout algorithms and use of WPF. It also has the potential to display two nodes with the same text, as long as they have unique IDs behind the scenes (something that other graph libraries lack), and the ability to specify different formatting for certain arrows, which was necessary to convey the tight coupling (TC) described in section 2.6.1.1. ZDG Mission Graphs' visual style was taken from (Barber, 2010).

3.2.3 Graph Structure

Graphs are implemented as a list of nodes which themselves contain a list of edges. Nodes contain information such as their mission symbol, unique ID, and their temporary rule node identifier needed for steps 2 to 6 of the rewrite process (see section 2.6.1.1). Edges contain the unique ID of the target node and whether or not the connection is a tight coupling. A rule consists of two graphs (left and right hand side) of the same described structure. An alphabet is a dictionary of string symbols (such as 'bl') and bools specifying whether or not the corresponding symbol is a terminal one.

3.2.4 Generation Method

The mission graph begins as a single 'S' (Start) node (see Figure 34), and is then iterated on until all symbols it contains are terminal. Steps referred to in the following iteration process description are those brought to light in section 2.6.1.1.

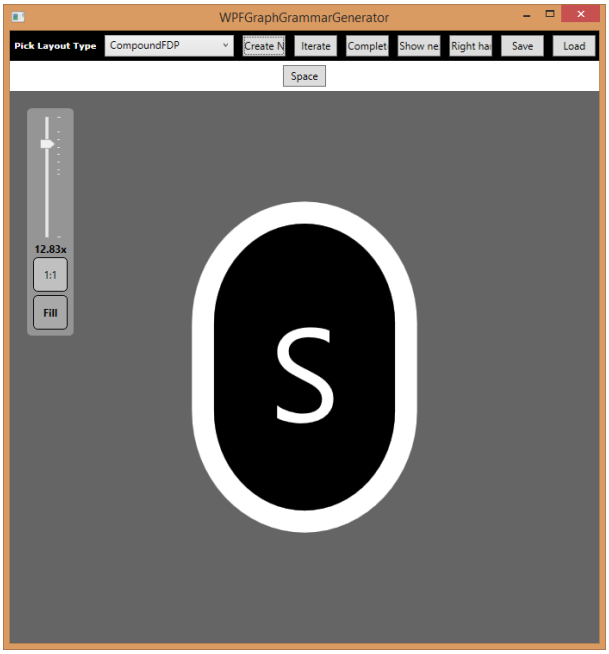


Figure 34: The initial state of the mission graph.

Each iteration, a list of applicable rules (whose left hand sides are sub-graphs of the current main graph), is put together. One of these rules is then randomly selected before being applied in the previously described step by step process. Each node in the left hand side of the rule is recursively compared to a node in the main mission graph, to check that the subgraph is fully present. This comparison is also sensitive to tight couplings: two nodes connected by a tight coupling and two nodes which are identical to the previous, but connected by a normal connection, will not be matched. As each node is being compared, nodes are assigned their corresponding number identifiers for the rule (step 2), and edges are marked for deletion. If the comparison succeeds, the marked edges are deleted (step 3). The right hand of the rule is then iterated through. If the node in question has a corresponding identifier in the main graph, that node is replaced by the corresponding rule node (step 4). If it does not, the rule node is added into the graph (step 5). Then, edges are added based on those on the right hand side of the rule (step 6), and the rule node identifiers are removed (step 7).

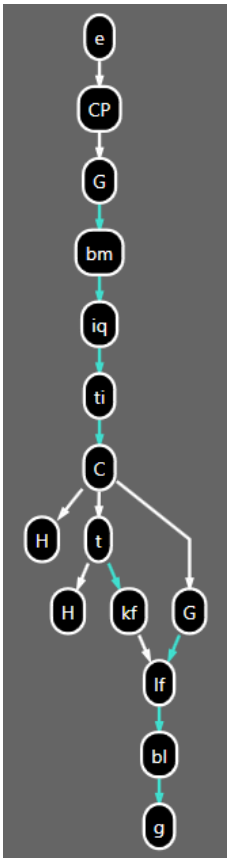


Figure 35: An iteration of the mission graph mid-generation.

As Figure 35 demonstrates, tight connections are represented by blue arrows, normal connections by white. Figure 35 shows a mission mid-generation, meaning some non-terminal symbols (represented by capital letters) remain in the graph.

Weights which alter the probability of certain rules being chosen are also implemented. Generally speaking, a rule with weight 3 has 3 times as much chance of being selected to incorporate into the graph than a rule with weight 1. This feature was also later applied to space generation rules in the same way. Figure 36 shows a completed mission graph.

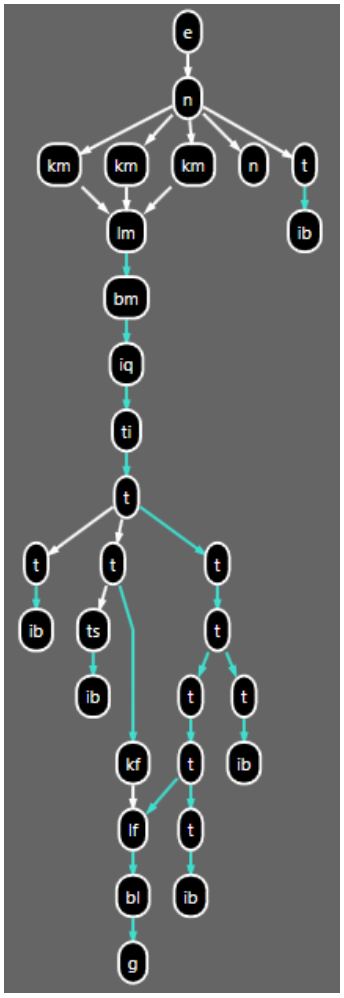


Figure 36: A completed mission graph.

3.2.5 Debugging, Save and Load

Special functionality was implemented to enable the viewing of each side of individual rules in order to debug them. Save and load of graphs was also implemented using XML serialisation. The same method was later used to save and load generated spaces as well.

3.3 Space Generation with Shape Grammar

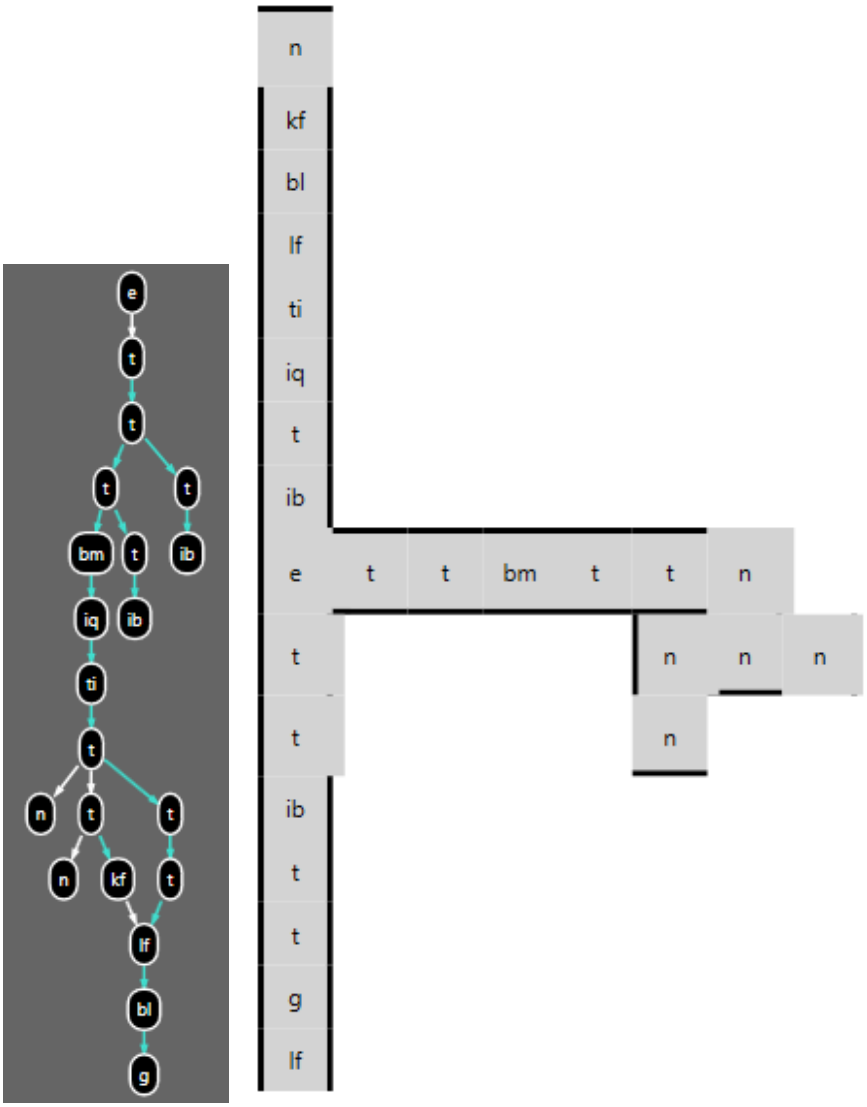
3.3.1 Base Generation Method

The concept of shape grammars is covered in section 2.5.3.1.1, while the process Dormans used for applying mission to space is described in detail in section 2.6.1.2. This work's map generator implementation follows the steps described in section 2.6.1.2. This process involved a large number of steps in order to generate valid maps which reflected mission structure correctly. Each step is explained and an example given of a map at that stage.

3.3.2 Step 1: Original Method

Figure 37 shows the result of following the space rules specified in Dormans' paper (refer back to Figure 31) in what could probably be described as a more literal, exact way than they were intended. The initial map realisation process went as follows. The mission

graph is traversed using depth first search (DFS). For each mission symbol, a room with one entrance and one exit on opposite sides is placed down using a randomly selected open connection from a previously placed room. Only the 'n' (nothing) rooms are given more than one exit direction. The result is a very linear, very boring dungeon layout.



(a) *Input: Mission*

(b) *Output: Map*

Figure 37: Original space result. Open connections are represented by the lack of a black wall outline.

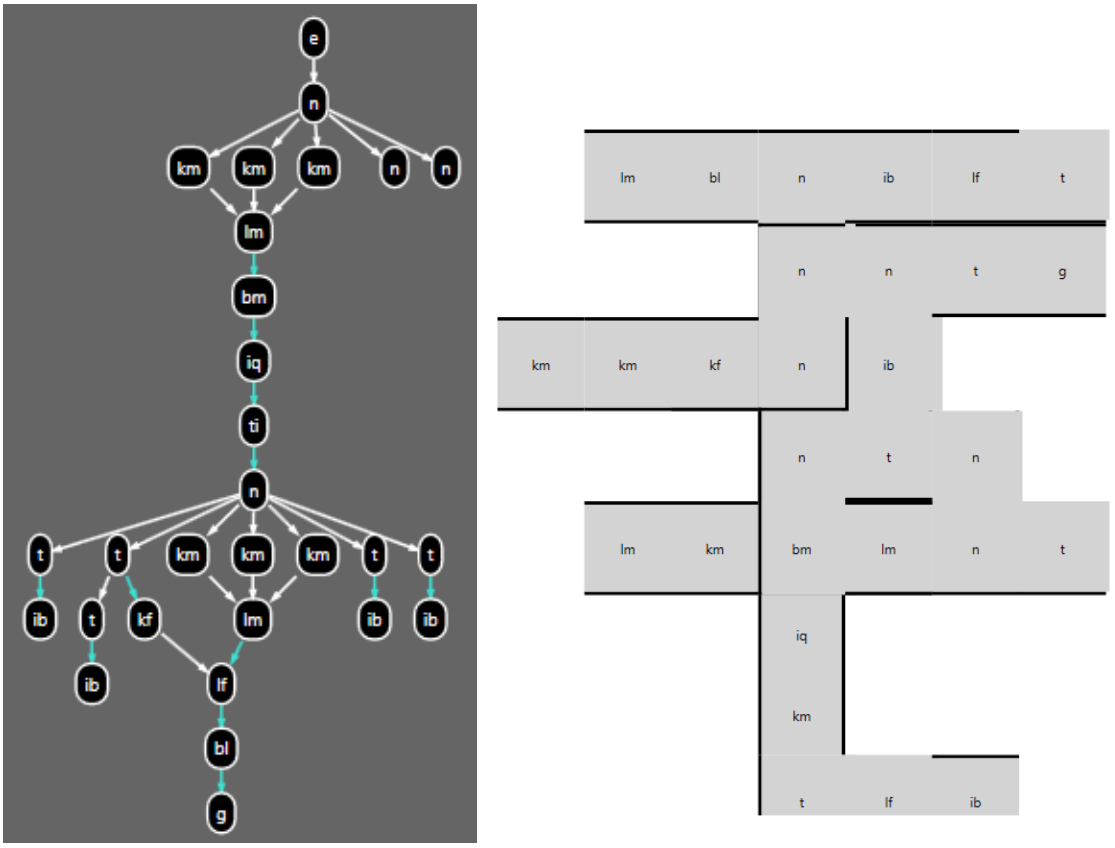
A problem caused by DFS traversal is also made apparent: both n nodes, which correspond to three n rooms, are on the left-most side of the graph, and therefore accessed last and added to the layout last. The result is a cluster of rooms at the edge of the layout, which were supposed to be spread throughout the dungeon to mix up the structure.

The originally planned solution was a breadth first search traversal (BFS) of the mission graph, which would effectively place the rooms layer by layer and in doing so preserve more of the intended layout order. However, this would mean that the tight couplings specified would have very little impact on the final space layout. Referring

back to Dormans' paper, the example in which a section of a mission graph is applied to a space graph does not appear to follow DFS, BFS, or a combination of the two based on tight couplings (refer back to Figure 32). The traversal algorithm used is not mentioned anywhere in the AiLD paper, and remains a mystery.

3.3.3 Step 2: Adding Variation

In a first attempt to solve the over-linearity shown in Figure 37, the probability of a parallel chain being placed in the mission graph was increased. A variety of different rules were also added, with different numbers and combinations of exit doors instead of the standard solitary exit opposite the entrance.



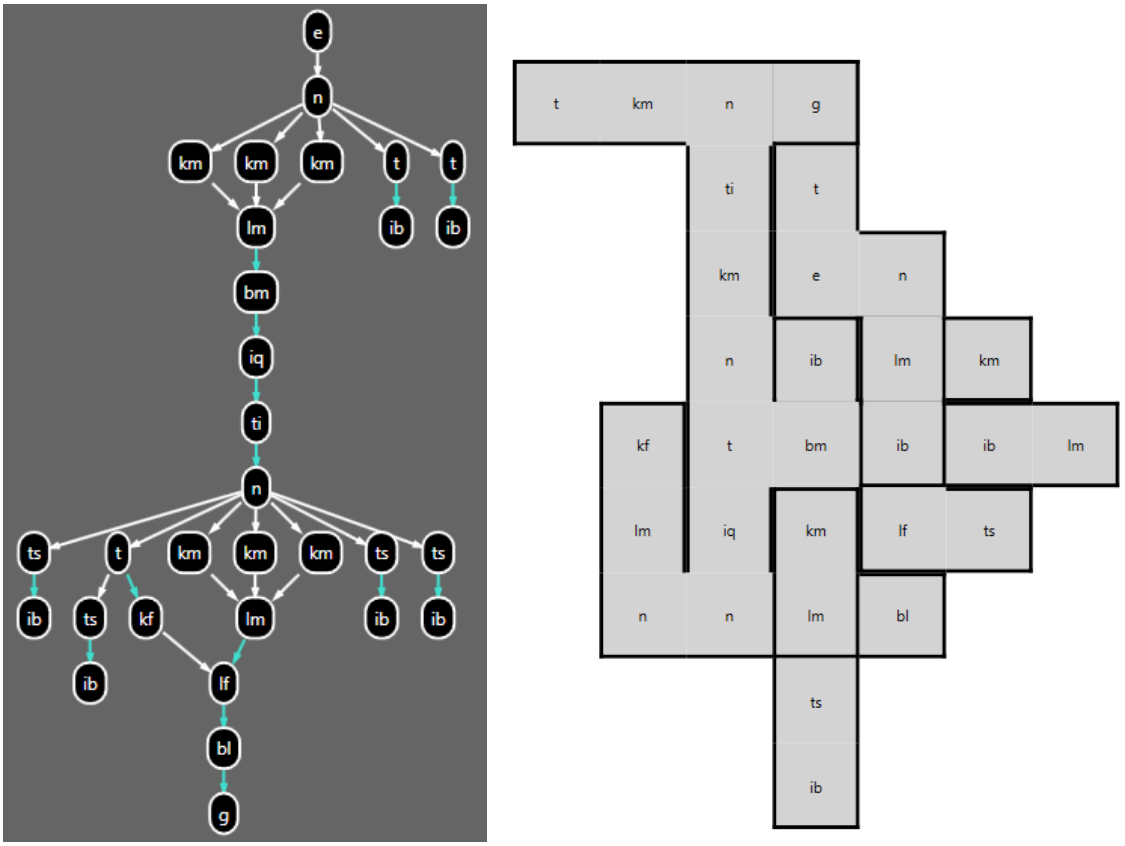
(a) *Input: Mission* (b) *Output: Map*

Figure 38: Result after variation added.

The result (see Figure 38) was reasonably successful, but clearly demonstrated two large problems with the shape grammar so far: many rooms overlap with each other, and some connections remain open when the graph is finished generating.

3.3.4 Step 3: Finishing the Map by Closing Connections

The remaining connections were then closed whenever the mission was fully implemented in the space. The resulting maps looked more coherent, but still confused because of the overlapping rooms (see Figure 39).

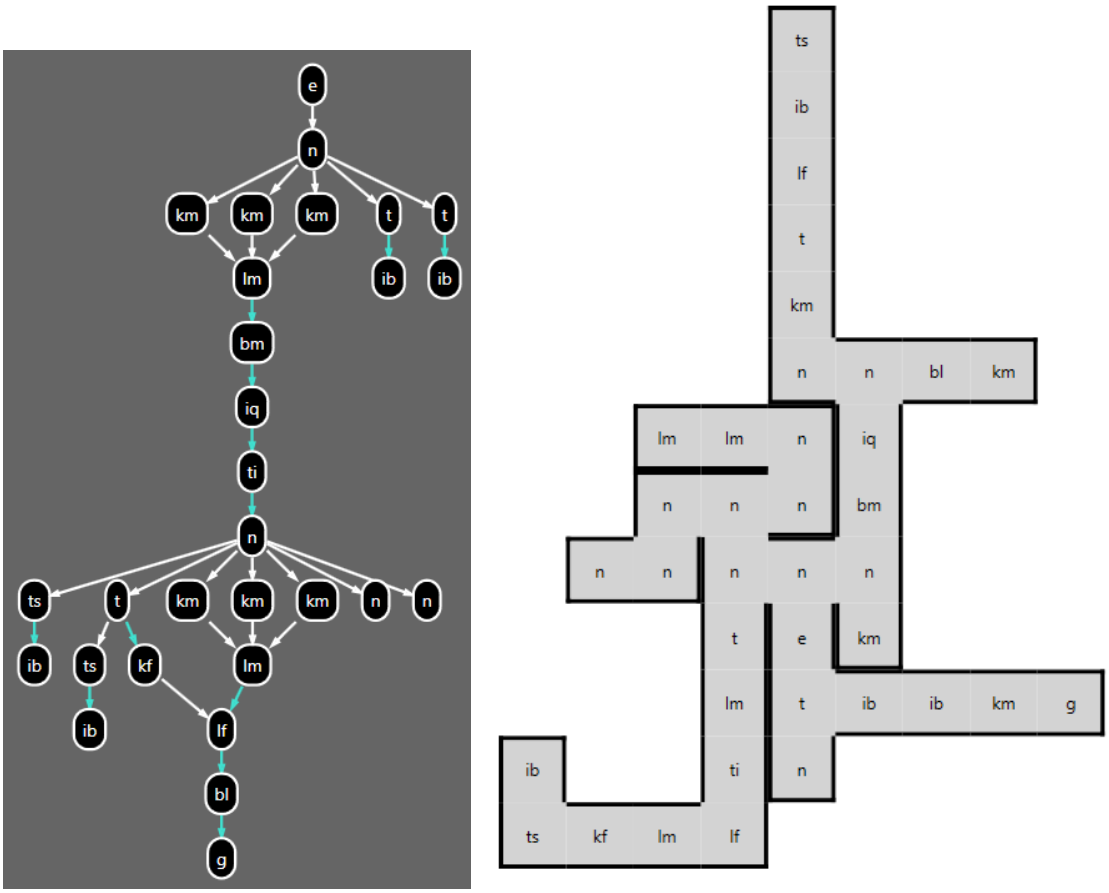


(a) *Input: Mission* (b) *Output: Map*

Figure 39: Result after remaining connections closed.

3.3.5 Step 4: Fixing Overlap of Rooms

To deal with overlapping rooms, a method was implemented to check for any open connections directly next to pre-existing walls after every iteration, and close them (see Figure 40).



(a) *Input: Mission* (b) *Output: Map*

Figure 40: Result after overlapping rooms fixed.

3.3.5.1 Potential Issue - Closed Map with Incomplete Mission

This solution had the potential to lead to a generation 'dead end'. Theoretically, with the chance of connections being closed if placed in unsuitable positions, all possible connections could be lost before the mission was entirely accounted for in the space.

Some possible solutions to this issue, such as adding an open connection somewhere in the map whenever one was closed, were considered, but found to be unnecessary. A 'quick complete' button was added to the editor and a very large number of generations tested by using it. However the problem never emerged, so the fix was put into the backlog until necessary.

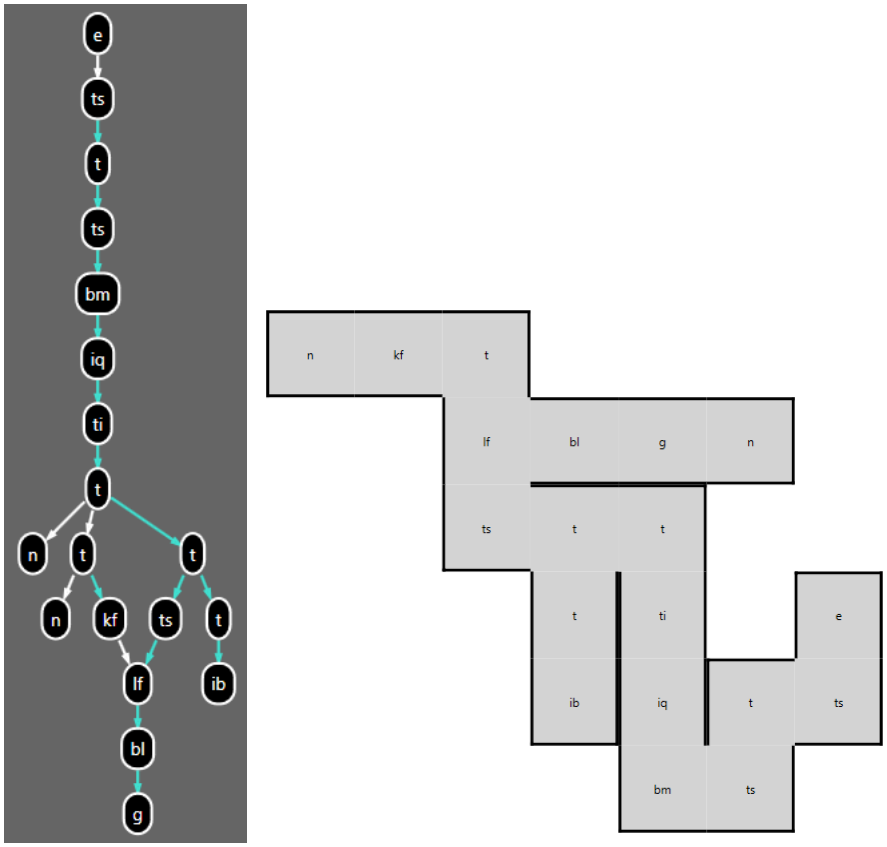
3.3.6 Step 5: Implementing Tight Coupling

Tight Coupling, represented as a blue arrow in the mission graph, demonstrates a tight consecutive geometrical link between two rooms. Instead of representing causal links such as the fact that a key and a lock are related, it represents that for instance, a certain chest should be positioned directly behind a secret test.

A technique was devised in order to take account of these in the space implementation. Upon placing a room with a tight coupling, the graph traversal would effectively turn into DFS, moving down the chain of TCs as necessary and placing rooms consecutively. This would avoid the chain being interrupted by other rooms.

Implementing this concept brought to light another problem with the grammar not addressed by the AiLD paper. Dormans' grammar can create nodes with two tight couplings stemming from them, but only gives examples of those rooms with one exit in his paper (Dormans, 2010). To solve this issue, a rule was added with a version of each potentially double TC room holding two exits. When two TCs stem from the same node, the two exit version of that room is chosen by default. However, with this technique there remained a chance that a two exit room may be chosen to carry out the tight coupling, but when placed, may be put directly between some pre-existing rooms and consequently have one or more of its exits blocked off. This would make fulfilment of the TC impossible (see section 3.3.8).

At this point, the three iterations of 'n' (nothing), as described in AiLD (refer back to Figure 31), were removed. For one room to be tightly coupled to each 'n' room it pointed to in the mission graph, it would need three open connections to be filled with 'n' rooms, for one sole iteration. The 'n' rule was instead reverted to a normal single room, which didn't cause a big increase in linearity thanks to the new multi-exit versions of rooms added previously. The result of this step is shown in Figure 41.



(a) *Input: Mission* (b) *Output: Map*

Figure 41: Result after tight coupling implemented.

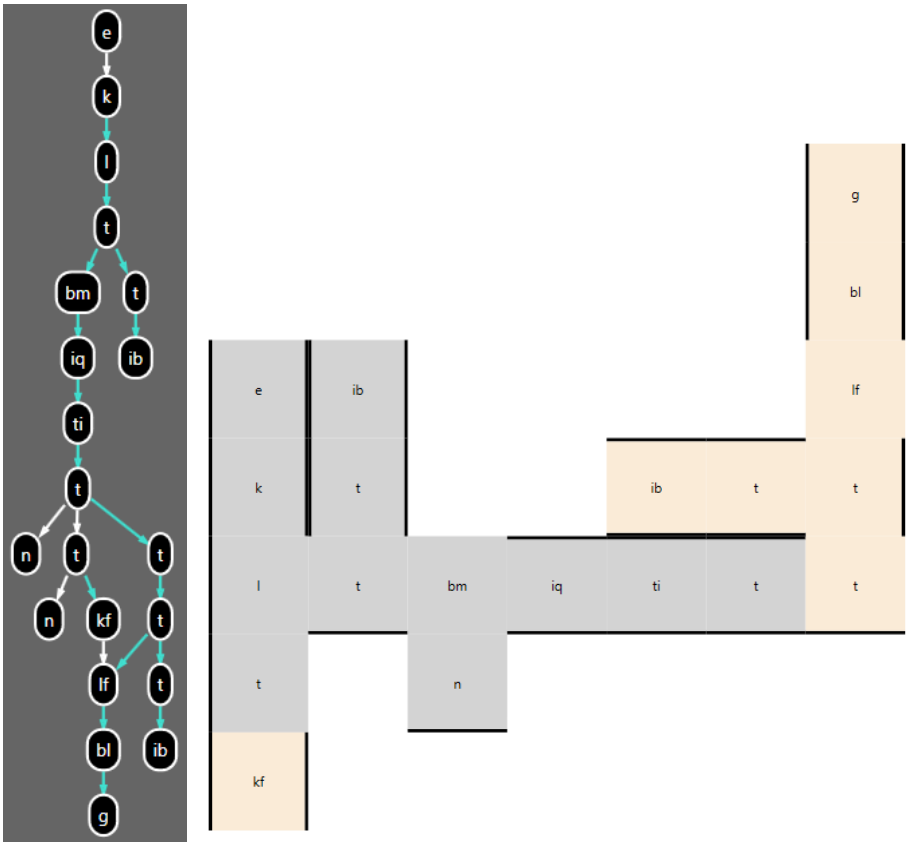
3.3.7 Step 6: Dealing with Graph Traversal: BFS and DFS

A significant problem with the graph traversal as it was is revealed in Figure 41. 'kf', the final key, is behind 'lf', the final lock. This would make the resulting dungeon impossible to complete.

The root of this issue stems from the fact that in order to guarantee that one node not be placed behind another, with the latter blocking the route between the entrance and the first node, **the first node must be placed down and connected to the graph before the second**.

At first, a solution was devised which involved both BFS and DFS: BFS for all regular nodes to preserve the desired flow of the mission, and DFS for tight coupling chains as described previously. However, even with this technique being implemented, the described problem would still remain. As can be seen from the mission graph of this map, traversal would almost immediately lead to a DFS TC placement, putting down all nodes in the blue arrow chain. One of the only nodes not connected to this chain is 'kf' (final key). The key will be placed, in BFS traversal, after the lock counterpart it is supposed to point to, and consequently could be placed behind it. Another bug with similar results was found in the system, in which a TC would push back one too many nodes and flag a regular non TC for urgent placement. In 41b, the 't' (test) and 'kf' chain would also push back the 'lf' (final lock) node, meaning that when the 'ts' (secret test) node, which should have been placed directly before 'lf', came to be processed, it would be unable to point to it. This also resulted in incorrectly ordered room placement and the placing of rooms before their direct predecessors.

Consequently, a system was devised in which grid spaces would be 'reserved' for rooms in a TC chain when necessary, but not 'fully' placed (see Figure 42). The rooms would not be made 'real' until their correct turn in the BFS traversal. Only after the rooms had been made 'real' could other, normal rooms be connected from them. The main consequence of this being that the TC could not be interrupted, and no other rooms could be prematurely connected from the TC, corrupting the structure. This would stop a normal key being attached to or after its corresponding door (which may have been placed prematurely in a TC). The idea being that the key is always 'actually' placed before the door, and can be reached from the entrance without the door blocking its path. This solution (though necessary!) was complex to implement, and involved a stack of tight coupling rooms to reserve and a queue of normal rooms to place in the BFS.

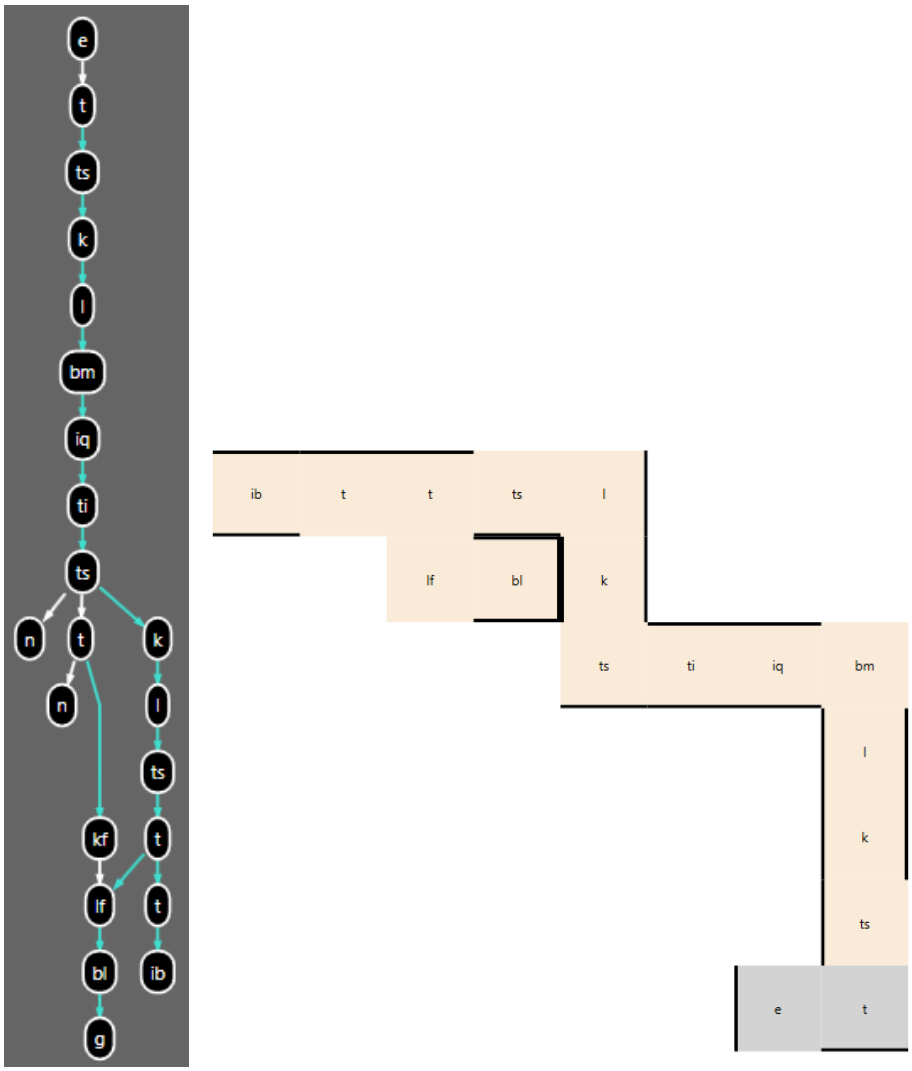


(a) *Input: Mission*

(b) *Output: Map*

Figure 42: This image shows a dungeon mid-creation using this 'reservation' technique. Cream rooms are reserved but not yet placed fully. As such, no grey 'real' rooms are branched off of them. Therefore, 'kf' and the 't' it is connected to are always placed from a grey room which was traversed previously in the BFS. They are not placed off of a tight coupling room which was reserved a space in the DFS, but may actually come 'below' it in the BFS of the mission graph. The situation of placing the key behind its lock is then avoided.

3.3.8 Step 7: Avoiding Dead Ends



(a) *Input: Mission*

(b) *Output: Map*

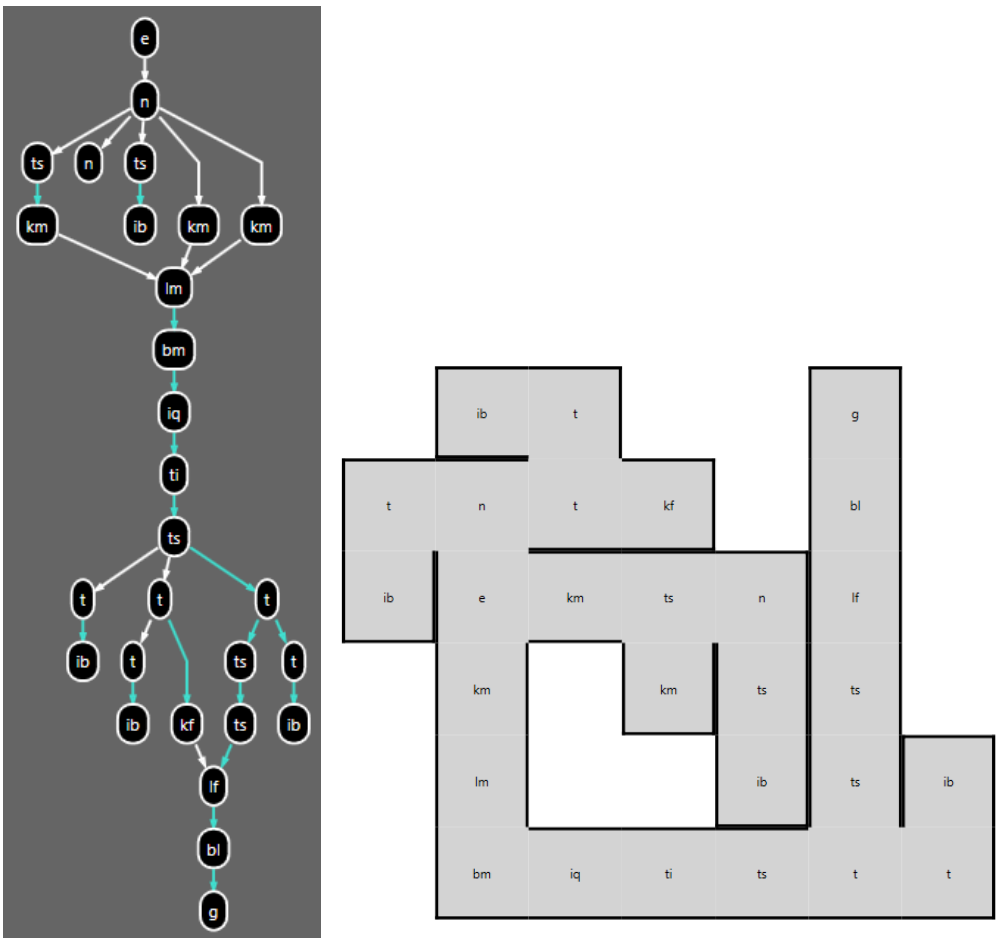
Figure 43: A map which has hit a 'dead end' while reserving room slots.

Figure 43 displays a problem in which TC spots are reserved, but the last node in the chain, 'g' (goal), cannot be placed. Although a version of the 'bl' (level boss) room must have been chosen with at least one exit to account for the TC coming from it, the 'bl' room's connection, which would have been on its east side, has been closed off, as it is directly in front of another wall. An apparent solution would be to add another connection in the southern direction instead, but then the problem is merely transferred onto the scenario of 'bl' being surrounded on all sides. This problem applies not only to single rooms individually, but to entire chains – for instance if a TC with 4 rooms remaining started to be reserved in a spiral-like path towards the centre of the dungeon, which was surrounded on all sides but only had 3 spaces left in which to place rooms. This is a common problem, occurring in general at least once every single time a space is generated.

On the scale of a single iteration, this problem seems simple to solve: just check all

exits will be clear before placing the room and if not, chose a different location. However, when placing a chain of consecutive rooms down in a situation such as the spiral described above, what if it becomes apparent that many steps back, the placement of one of the initial rooms entered a 'dead end', and no possible permutation could now fit the rest of the chain in the space available?

A solution was implemented in which any chain of TCs is reserved all at once in a single iteration. If at any point a room has no possible successful position in which to be reserved, the entire chain restarts and attempts to reserve itself from the beginning again (see Figure 44).



(a) *Input: Mission* (b) *Output: Map*

Figure 44: An example of a graph which would have led to a dead end if it followed its natural course, but instead was corrected and chose a different path instead.

3.3.8.1 Potential Issue - Infinite Trial and Error

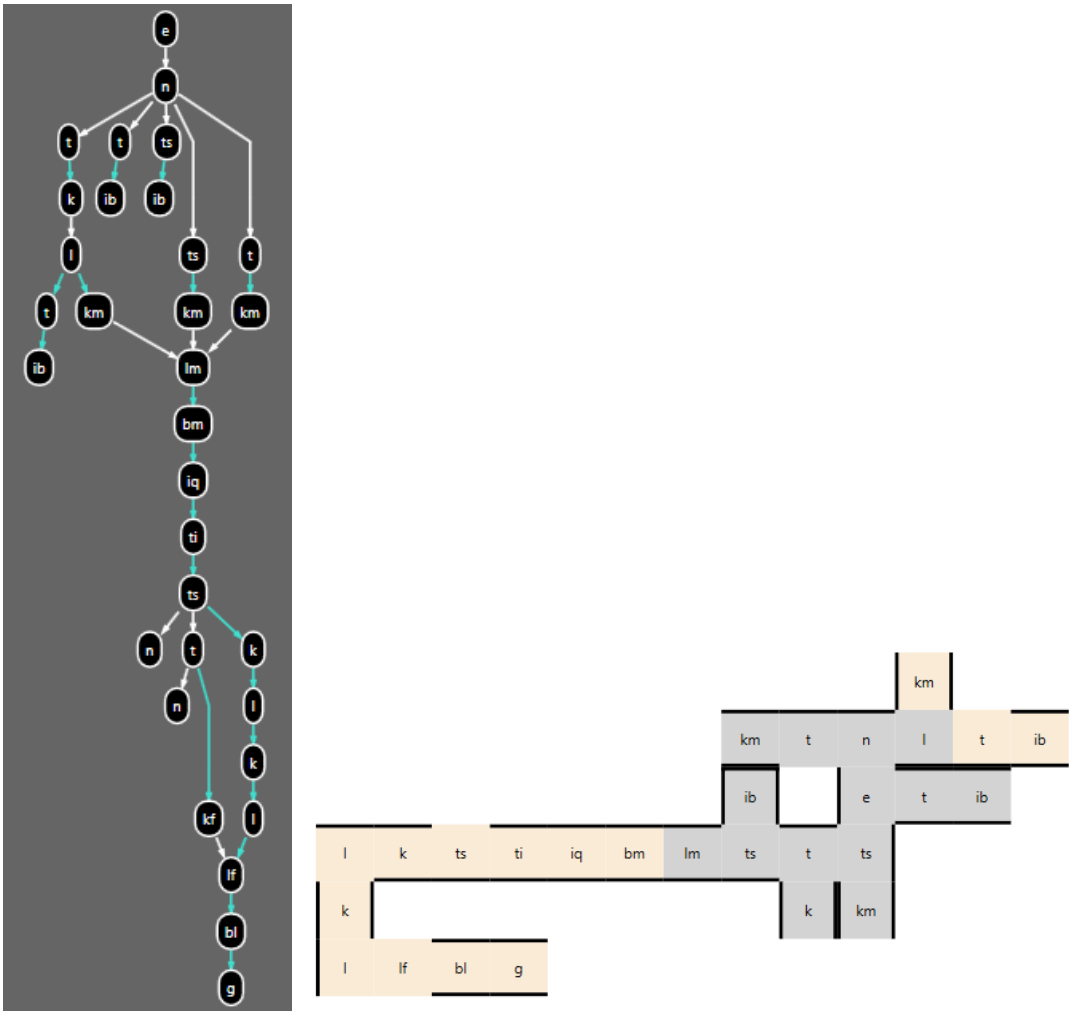
A potential issue of any trial and error method of random iteration is that it could lead to an infinite loop of unsuccessful trials. In this case, some quirk of a graph could mean that no successful layout is possible and an infinite loop would occur. Moreover, in general, any unsuccessful iterations could substantially increase loading times.

The ideal solution to this issue would be an algorithm which keeps track of a stack of rooms placed, and tries all combinations of rooms in order when it finds a dead end, removing the most recently placed room and trying all subsequent combinations one

by one until a successful path is found. This would ensure every single possibility is attempted, instead of relying on chance to achieve something correct. If every single combination is exhausted, the graph is definitely a dud and an error can be displayed.

However, a very large number of spaces have been generated with an insignificant increase in generation time and no dud graphs have been found. Therefore, the trial and error method was deemed appropriate and not changed.

3.3.9 Step 8: Adapting Traversal: Topological Sort and DFS



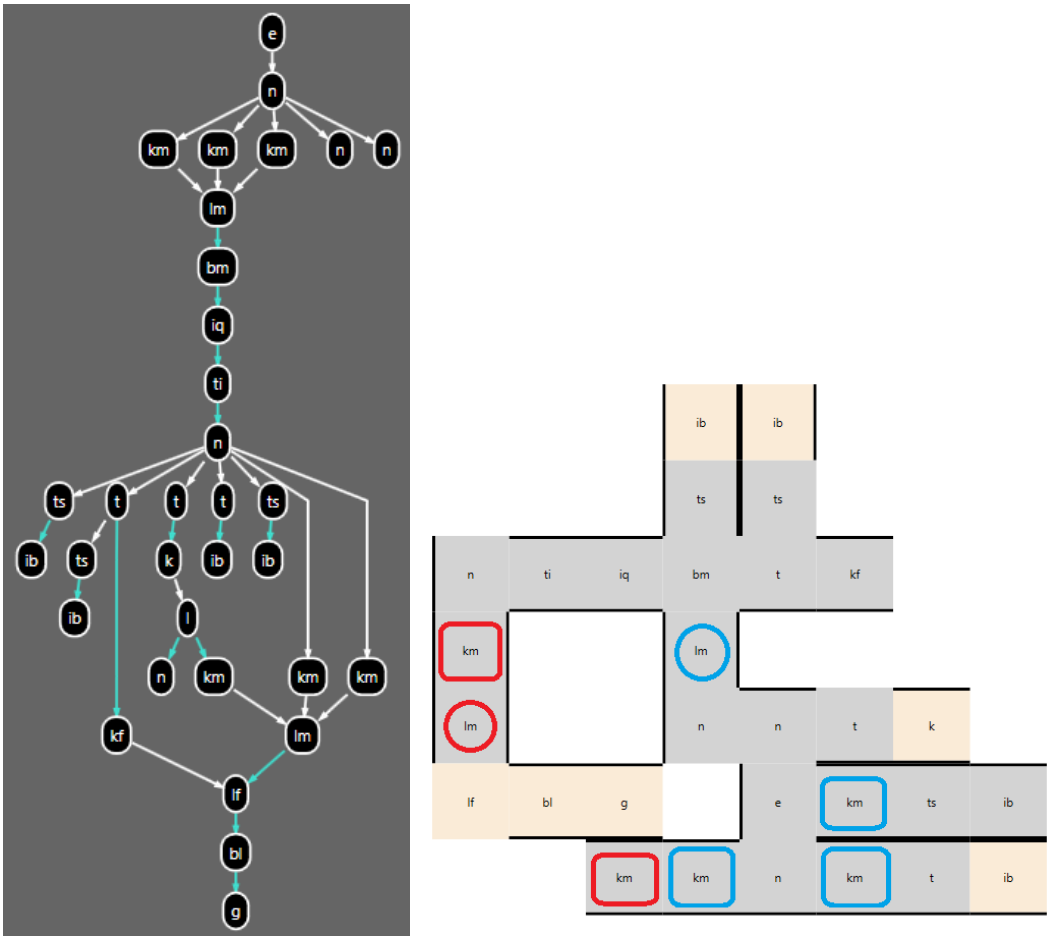
(a) *Input: Mission* (b) *Output: Map*

Figure 45: A map which displays symptoms of incorrect traversal order.

Figure 45 shows a symptom of remaining problems with the mission graph traversal, which could lead to impossible maps. As nodes are traversed in the BFS to be placed properly, so that other nodes can be placed from their exits, all keys should be placed in grey before their corresponding locks. A 'km' (multi-part key) in this map has not been properly placed down before its corresponding 'lm' (multi-part lock). In the case shown it will not cause an impossible dungeon, as luckily the 'km' has already been reserved elsewhere. Nevertheless, this is a symptom implying several impossible scenarios are possible with traversal as is.

The cause of this can be seen by looking at the top section of the graph and noting that even though nodes are lined up in coherent rows in the graph display, BFS will count 2 'km's in the fourth row, and one in the sixth. When following the right hand side of the graph down, the 'lm' would be in the 5th row, and so would be placed in the BFS before the key which was supposed to come before it.

Figure 46 shows a clearer symptom of the problem in which the position of a key has not even been reserved before the lock it is supposed to precede has been placed.



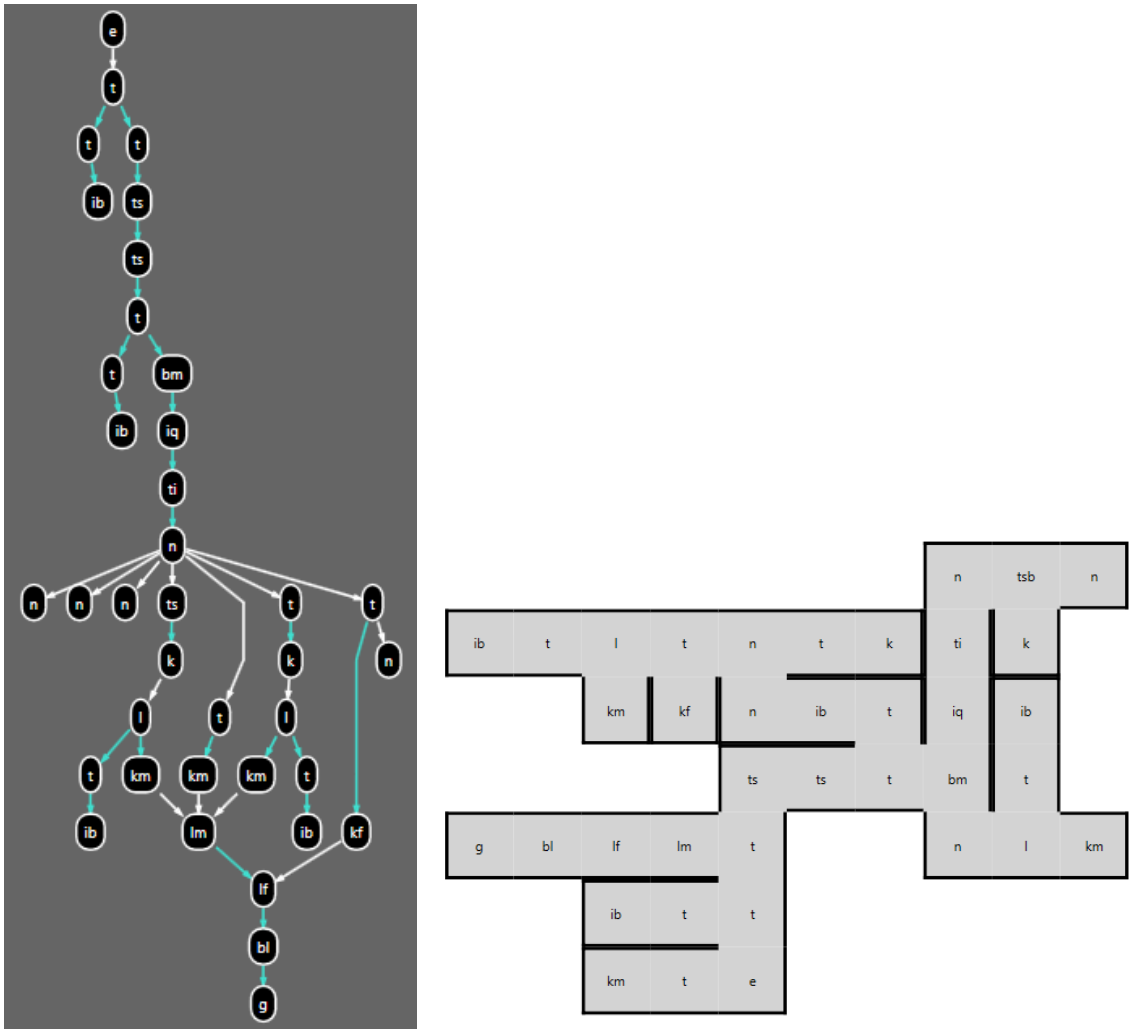
(a) *Input: Mission* (b) *Output: Map*

Figure 46: This mission graph has a similar disparity between rows, this time in the bottom section. The result is that 5 'km's and 2 'lm's have been placed, while both sets of 3 'km's should have been put down before the 'lm's they correspond to. In theory it would be possible for the remaining 'km' to be placed behind its 'lm' in a situation such as this, rendering the dungeon impossible.

This issue came from a misunderstanding of the nature of BFS. BFS is defined as 'a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node' (Myers, Andrew, 2012). In this case, it is more than likely that some locks will be placed before their keys: as demonstrated in Figure 46, sometimes a lock has a smaller number of nodes between itself and the entrance than its key does.

This meant that a different traversal algorithm was needed, which ensured all parent nodes were traversed before their children. Topological sort was found to do the trick, and

an implementation of the algorithm was used to replace BFS. Therefore, the final traversal technique used is a combination of DFS for reserving tight couplings, and topological sort to place all rooms properly. Figure 47 shows a fully completed mission and map.



(a) *Input: Mission* (b) *Output: Map*

Figure 47: A fully completed mission and map.

3.4 Application to Zelda-style Game

3.4.1 Choice of Platform

A pre-existing game or engine was desired for the project, as the focus was on dungeon layout rather than generating any other game content or functionality itself. The method of applying procedural dungeon generation to a pre-existing game was also seen to be successful in several projects found in the Literature Review, including van der Linden's application of gameplay grammars to the game *Dwarf Quest* (Linden, 2013). Luckily, reproducing *Zelda* functionality in quest and map makers is relatively popular, and there were many engines to choose from.

These map makers were compared on a range of factors which would affect incorporation of the grammars and procedurally generated content. The results of the best three

(Open Zelda (*Open Zelda* n.d.), Solarus (*Solarus: An ARPG game engine [sic]* n.d.), and Zelda Classic (Armageddon Games, n.d.)) are shown in Table 1.

	Free?	**Open Source?**	**Engine Language**	**Functionality, Quality**	**Quest Designer**
Open Zelda	Y	Y	C++	Dungeons good, but glitch in placed. Animations very slow.	Very intuitive map editor, unsure if can edit programmatically.
Solarus	Y	Y	C++, scripted Lua quests	*Link to the Past* level of quality and functionality, much more so than Open Zelda.	Intuitive map editor, can also edit dynamically through Lua.
Zelda Classic	Y	?	?	NES Style	Fine, not as accessible.

Table 1: Comparison of Zelda-style map makers and engines.

The Solarus engine was chosen for several reasons. Firstly, several full games had been created with the engine, exhibiting the majority of functionality from the *Link to the Past* (Nintendo, 1991) (LTTP) game (see Figure 48). This proved that the desired game functionality was achievable. Secondly, the combination of a C++ backend and scripted Lua quests gave a lot more scope to change maps dynamically, whilst retaining the potential to access and change the engine source if absolutely necessary. The engine itself produced impeccable quality games, and demonstrated an intuitive, accessible (and fun!) map and tile editor. A possible problem came from the fact that a lot of the tutorials and documentation for the engine were in French, but there was a thorough and comprehensive Lua API available which contained the majority of information needed.

Figure 48: Screenshot of *Zelda: Mystery of Solarus*, a game built with the Solarus engine (*Zelda: Mystery of Solarus, en Archlinux* n.d.).

3.4.2 Dungeon Layout

3.4.2.1 Connecting Rooms

Each Solarus map has two files associated with it – a .dat containing map information such as where tiles and entities are placed, and a .lua, which describes any dynamic

behaviour the map may have. A map's Lua script can be used to dynamically place tiles and entities. Entities used for transitioning between maps in Solarus are called teletransporters and destinations. When collided with, a teletransporter teleports the player to the specified destination point (see Figure 49). A plan was devised in which a map's Lua script, specifically the contents of its on_started event, would be generated from the C# grammar backend. This Lua script would describe the appropriate door tiles, teletransporters and destinations the map should have.

Figure 49: A screenshot of a Solarus map in-editor. Yellow arrows are teletransporters which the hero will be transported to a different map from, and green arrows are destinations that the hero will be transported from a different map to.

The regular practice in Solarus is to lay an entire floor of a dungeon out in one map, creating walls and doors between rooms with tiles. However, the ZDG consists of one room per map, so that the positioning relationships between rooms can rely solely on teletransporters and destinations, with the hero being transported from door to door, room to room. This meant that the editor could still be used to design and decorate rooms, but that certain perks of Solarus, such as the automatic dungeon map generation, could not be used.

3.4.2.2 Rotating Rooms

The idea of rotating rooms based on the orientation from which they were placed in respect to the previous room was considered, to maintain consistent entrance and exit layouts which would supply a wider array of possibilities for asymmetric puzzles. However, the complexity this would add to the generation process was not worth it. Consequently, a slightly different experience is had in each room depending on which direction the player entered from. Also, the contents of some rooms had to be designed symmetrically, in order to take account of the hero being able to come and go from any direction (see Figure 50).

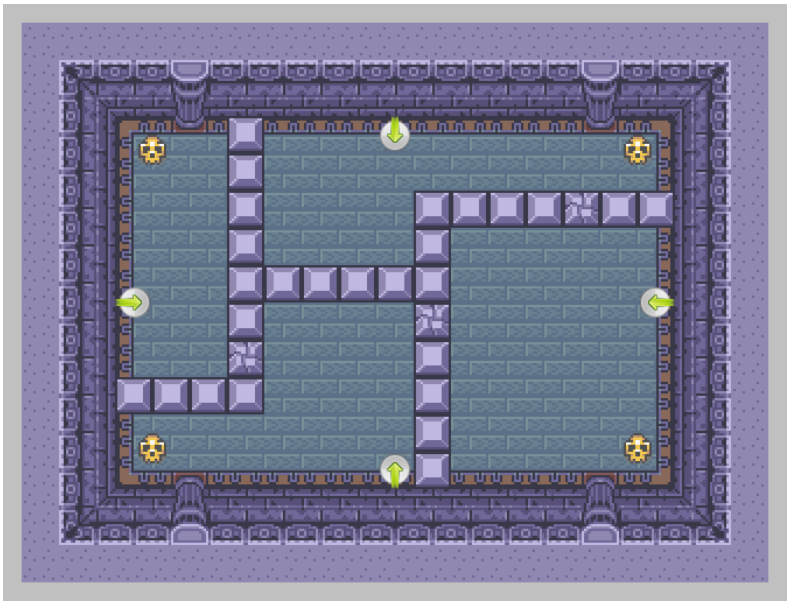


Figure 50: The 'ti' (item test) room in-editor, in which the hero must use bombs to get to the other side. As we don't know which direction the hero will come from and which they will need to get to, all directions must be blocked from each other with bombable blocks.

3.4.2.3 Duplicating Rooms

Ideally, enough rooms would be created in-editor to accommodate all possible layouts of the dungeon with each room being unique. However, for the scope of this project, reuse was necessary. Solarus does not offer any built-in way of reusing maps, as map names are used as unique identifiers in a number of scenarios, and enemies killed/doors opened in one room would persist across the duplicated room. The same issues would also apply to a room's destinations and transporters, as the engine would be unable to differentiate between them. A method of achieving this effect, by 'tricking' the engine into creating two instances of the same map using the project database file, was investigated, but not successful.

Therefore, the C# backend was used to access the dungeon file system, copy map files, rename them, and add references to them to the database file. These were flagged as generated in their names so that the previous iteration's files could be deleted each time maps were applied.

3.4.2.4 Placing Tiles

After this process, the system loops through rooms in the space graph, generating tiles and entities for all doors as it goes. This is a surprisingly complicated procedure, as a range of intricate tiles with different sizes and offsets are required for each door (see Figure 51). Different versions of each tile must also be accounted for, depending on the direction the object is supposed to face.

Figure 51: One of the first attempts at tile generation. Demonstrates the importance of getting it right – Link couldn't leave the room entrance!

A problem was found in which destinations did not exist by the time they were needed if they were generated from C# and Lua. Whenever a door with a specified destination is walked through, that destination needs to already exist in the specified map in order to be transported there. This was simple to fix, however, by placing four pre-existing destinations in each map file ('from_north', 'from_east', 'from_west' and 'from_south'). The extra destinations do no harm, and can be accessed whenever from wherever.

3.4.3 Accommodating Gameplay

Though some unrelated aspects of implementing gameplay were arduous in their own ways, only those that were impacted by or had implications on the PCG system are covered in detail here.

Trying to implement aspects such as enemy behaviour scripting from scratch was taking too much time, and as it was not integral to the project, a pre-existing Solarus game was adapted instead: *Zelda: Mystery of Solarus DX* (Solarus, 2008) is a free, open-source game made using the Solarus engine by the engine creator (*Zelda Mystery of Solarus DX* n.d.). By taking advantage of the pre-existing implementations, features such as enemy behaviour, rupees and a HUD were obtained for free. However, several aspects still required manual completion.

3.4.3.1 Dungeon Item and Multi-Part Keys

As the scope of the project was not large enough to account for different dungeon themes and main items, it was necessary to choose instances of certain game concepts to implement. For example, which dungeon item would the player receive after defeating the miniboss, and use to traverse the remainder of the dungeon? (See section 2.2). Also, as Dormans describes, keys and locks in a dungeon can refer to much more abstract concepts (Dormans, 2010). It was necessary to decide what exactly a multi-part key and multi-part lock in the ZDG would entail. Dormans uses the example of the monkeys from

Twilight Princess (Nintendo, 2006), necessary to throw Link over large gaps (Dormans, 2010).

Bombs were chosen as the dungeon item, as not only could they be used as a key for traversal, but also as a way of hiding secrets as specified in the mission grammar. There also existed an enemy in the *Mystery of Solarus DX* game who could only be defeated using bombs, fulfilling the kumite stage Dormans points to in *Zelda* level structure (Dormans, 2010). For relative simplicity, the multi-part key and lock utilised certain monsters as keys, who when killed would cause a lantern to be lit in the lock room (see Figure 52). When all four lanterns were lit, the lock would be opened and the hero could traverse through the next door. This concept can be seen in several *Zelda* games, including *Ocarina of Time* (Nintendo, 1998) and *Twilight Princess* (Nintendo, 2006).

Figure 52: The lantern room with the shut door, acting as the multi-part lock.

As only one multi-part key/lock concept was implemented, it became apparent that having any more than one set of multi-keys and locks in a dungeon would cause too much confusion. The grammars as they are do not account for placing one set of multi-keys after a previous multi-lock, and assumes they can be mixed up without issue. Unfortunately, a problem arises with this multi-key implementation when sets of keys are not distinguishable. Killing a monster in the starting area could lead the player to believe a lamp will be turned on, as with the other monsters of that type he or she has killed. However, this monster could actually correspond to a different set of lanterns later in the dungeon, and so appear to have no effect that the player could see (see Figure 53).

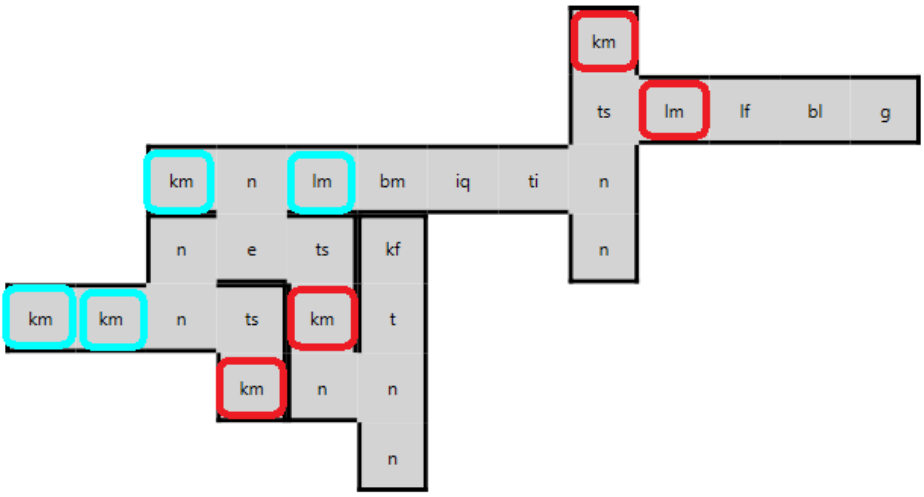


Figure 53: An example of the problem. 'Km's (multi-part keys) are certain types of monsters which make a 'secret' sound when killed and turn on a lantern somewhere in the dungeon. In this example, the player could defeat 5 of them (3 blue, corresponding to first lock, and 2 red, corresponding to second) before walking through the first 'lm' (multi-part lock), and 2 of them would have no effect on the lanterns in that first room.

This would be assumed a glitch, or at most very counter intuitive. Therefore, the mission grammar was tweaked slightly, to only allow one set of multi-keys and lock per dungeon. This is a temporary fix in theory, as having two differently implemented sets of multi-keys (such as the monkey example described earlier, in addition to the lamp lighting example currently implemented) would solve this problem with ease, making any change to the grammars unnecessary.

3.4.3.2 Secrets

Utilising bombs as secret tests for 'ts' rooms yielded a problem when combined with the shape grammar as it was originally. 'ts's could appear both before and after the player had obtained the bombs, and could physically block the player from progressing any further along the main path if they hadn't picked up the bombs yet. Therefore, the shape grammar was slightly tweaked to differentiate between secrets that were placed down before the bomb room, and secrets that were placed down after ('tsb's). This meant that the hero could still encounter a weak wall before they had obtained the bombs, but it would never block the path to the room with the bomb bag in it (see Figure 54, Figure 55 and Figure 56).

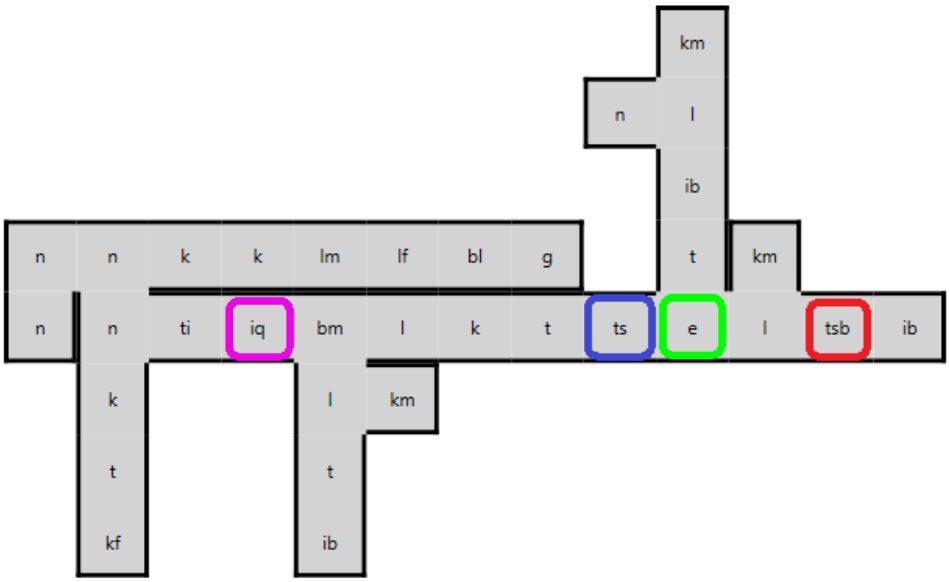


Figure 54: Demonstrates solution – where a ‘ts’ (blue) blocks the path to the ‘iq’ (pink) room, where the player receives the bombs, it remains a ‘ts’ and contains a secret which does not require bombs to solve. However, another ‘ts’, though near the entrance, blocks nothing more than an item bonus, so has become a ‘tsb’ (red) which the player can backtrack to after receiving the bombs to collect the bonus.

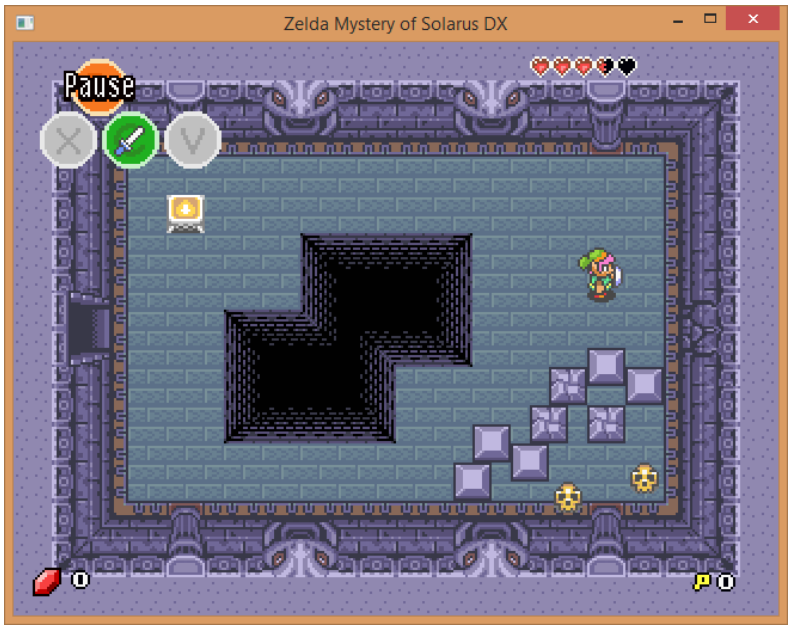


Figure 55: A ‘tsb’ room which requires bombs to traverse.

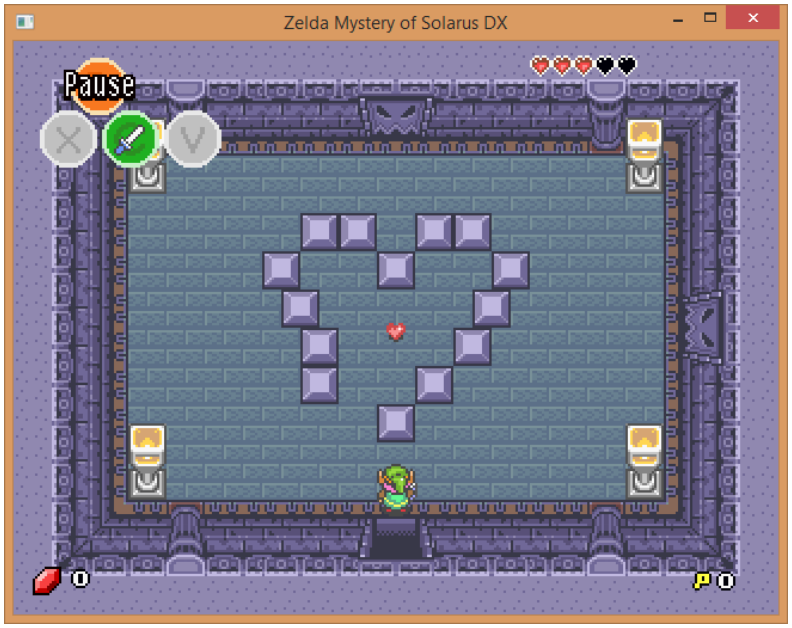


Figure 56: A 'ts' room which does not require bombs to traverse.

3.4.4 Generating Save Game Variables

In order for the state of an entity to persist between rooms in a Solarus game, it must be assigned a unique save game variable string. These are usually manually assigned from the map editor (see Figure 57), but in the case of the ZDG had to be generated to account for either being duplicated, or being created from the C# backend entirely. The state of doors and chests being opened, treasure being obtained and enemies being killed all need to persist between rooms, and therefore require their own unique variables.

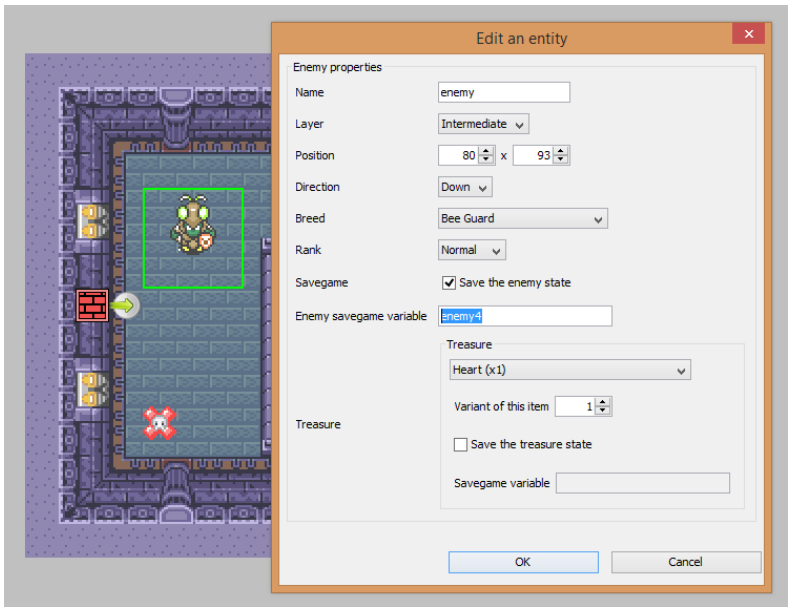


Figure 57: Manual assignment of savegame variables in-editor.

Initially, this was thought to be a simple case of having the Lua script loop through all entities in a map and assign them a string based on a combination of the map name, entity type, and running entity count.

However, a problem was found. Various Lua functions exist in Solarus for the purpose of getting and setting certain properties of pre-existing entities, but they do not exist for all properties in both directions. For instance, one can get_breed of an enemy but not set_breed. In the case of savegame_variable, Lua cannot access the property at all, so the initial plan was unachievable.

Two options were considered at the time:

• Generate all entities in Lua, passing a unique save game variable in as an argument to the entity constructor. This would mean being unable to place or edit any entities in the editor, and having to specify co-ordinates and other irrelevant properties in Lua.

• Delve into the engine source and add a function which exposes the variable to Lua. Adding the function would be the easy part, as getting the engine to compile seemed to be a very complex process.

The latter option was chosen, but the warning Solarus' developer had put on his website turned out to be true. 'In both cases, the hardest part is to install the libraries because under Windows, you have to download all of them one by one (in their development version) and install the headers and the libraries. It's a nightmare.' (*Compilation Instructions* n.d.) The libraries proved horrendous to track down and install. Compiling the engine on Windows seems renowned as an almost impossible task within the Solarus community.

Moreover, the reason that these Lua getters and setters did not already exist could have been that the information they provide for pre-existing entities is necessary on map load from the .dat, and cannot be changed dynamically after the fact. For instance, if an enemy was loaded from the .dat, only to then be assigned a savegame variable afterwards which dictates that he should not have been loaded, it is unclear what the behaviour of the engine would be.

A solution was found in the form of a newly devised third option, as manually arranging entities using co-ordinates would have led to too much hassle. Luckily, maps' .dat files are human readable, and can be edited (see Figure 58).

```
enemy{
  layer = 1,
  x = 160,
  y = 141,
  direction = 3,
  breed = "khorneth",
}
```

Figure 58: An example of an entity's .dat entry.

The C# backend was used to search through the file for any entities of the correct type, and insert a line of data specifying the save game variable (e.g. "savegame_variable" = "enemy4"). This could have been risky, as .dats are generated by the editor and up to this point in the project had been left alone, but the solution worked and was definitely the best option of the three.

4 Results, Analysis and Discussion

4.1 Overview

The aim of the project was to evaluate Dormans' grammars by considering a) the expressive range of dungeons produced by the ZDG and b) the impact the grammars had on the development process. Therefore, the results of the project are split into two sections. Section 4.2 conducts an in-depth analysis of the mission and map expressivity (objective 6), while section 4.3.1 covers the success of the project implementation process and any implications of the grammars (objective 5).

Section 4.2 only focuses on the map and mission generation results; application to the game is not covered. The reason for this is the nature of the implementation: the same map will always lead to the same dungeon. No random generation, and hence nothing expressive, is performed in the stage between map generation and the realisation of a Solarus dungeon. Therefore, no expressivity-related data could be gained from the Solarus map files which could not be obtained from the map graph in the generator. However, section 4.3.1 covers the development process with particular emphasis on how successfully resulting maps could be applied to the Solarus game, and evaluates the procedure this way.

4.2 Product Expressivity Evaluation

4.2.1 Method of Evaluation

The technique of evaluating an expressive range (which classifies the style and variety of content produced) is largely based on the concepts introduced by Smith and Whitehead for analysing a level generator (Smith and Whitehead, 2010). Instead of focussing on the number of different permutations the generator can produce or the amount of time taken to create them, Smith and Whitehead examine the variety of generated levels and the impact of changing input parameters. In doing so, unexpected biases and holes in the expressive range can be exposed. This evaluation follows Smith and Whitehead's 4 step approach: determining appropriate metrics by which to measure results; generating a large amount of content and scoring it based on these metrics; visualising the generative space in the form of 2D histograms; and analysing the impact that changing parameters has on these results.

Smith and Whitehead evaluate expressivity in two respects: design constraints inherent to the generation algorithms, and analysis of the range of content produced. Sections 4.2.2 to 4.2.4 focus mainly on the expressive range which can be achieved, but any algorithm limitations are discussed in their relevant areas of testing and in the summary section 4.3.1. General limitations of the implementation used are also discussed at greater length in sections 4.3.1.3 and 5.2.

A reliable and representative set of results was gathered by including results from 1000 generations in each 2D histogram.

4.2.1.1 Parameters

Dormans' approach involves the ability for a designer to exercise full control over the output of the system by creating and maintaining their own grammars (Dormans, 2010). Therefore, there does not exist a discretised set of parameters which can easily be tweaked

for testing. Instead, different sets of grammars which exhibited extreme characteristics (based on the concept of appropriate parameters) were created. The selection of these rule sets, which differ based on individual rules' weighting (and hence probability of being used) was then used to vary input into the system. Mission rule sets are displayed in Table 2 while map rule sets are shown in Table 3.

Mission Rule set	Description
Branching Missions	A rule set designed to favour branching rules with multiple paths of actions.
Linear Missions	A rule set designed to favour linear rules with a single path of actions.
Long Missions	A rule set designed to favour longer rules which contain more actions.
Short Missions	A rule set designed to favour shorter rules which contain less actions.
Control	A rule set designed to balance the parameters described above.

Table 2: Mission Rule set Descriptions.

Map Rule set	Description
Few Exits	A rule set designed to give a more linear map layout. One exit rooms are given the highest weight, followed by two exit rooms and finally three exit rooms.
Many Exits	A rule set designed to give a more branching map layout. Three exit rooms are given the highest weight, followed by two exit rooms and finally one exit rooms.
Control	A rule set designed to balance the parameters described above.

Table 3: Map Rule set Descriptions.

Sets of control rules were created for both mission and map generation, which were balanced on all factors chosen as parameters. For example, there exist four options for replacing a chain and a gate: one branching rule and three different linear rules. Therefore, in the control rule set, the weighting of each of the linear rules was set to 10 whilst the weight of the branching rule was set to 30. This way an equal chance of getting a linear rule vs a branching rule was created.

Not all rule sets embodied their extremes as much as theoretically possible. For example, the goal room is always the last room to be placed and so all its exits are always closed. Therefore, a 3 exit version of this room was not included in the Many Exit rule set.

4.2.1.2 Metrics

The following measurements were used to evaluate the expressive range of the system. As advised by Smith and Whitehead (Smith and Whitehead, 2010), all metrics are global properties of the missions and maps produced, and are independent from the generation algorithm in that they are not explicitly used as parameters by the generator. Linear and branching rule sets are specified in that rules which exhibit the corresponding attribute are given a higher probability of selection, but the generator has no concept of linearity and does not work towards a target formula such as that used to calculate the linearity metric. All metrics discussed are represented by values between 0 and 1, which represent the minimum and maximum values of the metric respectively. These metrics are shown in Table 4.

Metric	Applies to...	Brief Description
Mission Linearity	Map	A scale between an extremely branching and extremely linear mission graph. Measured by looking at the ratio of nodes included in the shortest path between entrance and goal to those not.
Map Linearity	Map	A scale between an extremely linear map in which all rooms have one exit, and an extremely branching map in which all rooms have three exits and the player is constantly presented with decisions on where to go next.
Leniency	Mission and Map	A scale between an extremely safe map in which no rooms present any danger to the player, and an extremely hazardous map in which every room poses a threat to the player.
Path Redundancy	Map	A scale based on how many rooms serve a purpose, either by presenting the player with a reward or eventually leading to a room which does, and how many do not (e.g. A dead end room with an obstacle which leads to nothing.)

Table 4: Metrics used to score results.

4.2.1.2.1 Mission Linearity

$$\frac{\textit{Number of Nodes in Shortest Path}}{\textit{Total Nodes}}$$

Mission linearity is calculated by finding the sum of nodes in the shortest path between the entrance and goal nodes (inclusively), and dividing by the total number of nodes in the graph (see Figure 59). A graph consisting of a straight line of nodes between entrance and goal would therefore score 1 for Mission Linearity.

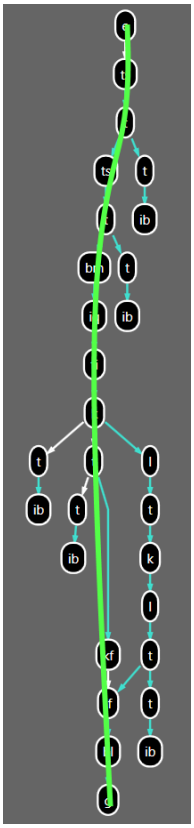


Figure 59: An example mission graph with a green line through it representing the shortest path from entrance to goal. Each node the line passes through is counted as linear. There are 14 nodes in the shortest path and 29 overall in the graph. The Mission Linearity is therefore 14/29, or 0.483.

4.2.1.2.2 Map Linearity

$$\frac{(1 * 1 \textit{Exit Rooms}) * (0.5 * 2 \textit{Exit Rooms}) * (0 * 3 \textit{Exit Rooms})}{\textit{Total Rooms Counted}}$$

Map linearity is calculated by scoring each room based on the number of navigation decisions it presents: 1 for 1 exit, 0.5 for 2 exits and 0 for 3 exits. This score is then divided by the total number of rooms with exits. Dead-end rooms are taken into account neither in the score calculation nor total room count. A map consisting of a straight line of rooms between entrance and goal would score 1 for Map Linearity. A map consisting of a spiral of rooms (a mix between rooms with one exit straight ahead and rooms with one exit to the left) would also score 1 for linearity, as although it turns, it does not present any more decisions for the player to take.

4.2.1.2.3 Leniency

$$\frac{\textit{Number of Safe Rooms}}{\textit{Total Rooms}}$$

Leniency is calculated by dividing the number of safe rooms which pose no threat to the player by the total number of nodes in the graph. The result is the same whether calculated on the mission graph or the map. It is worth noting that this measurement

is very specific to the implementation of the ZDG and would not apply in the same way to other realisations of the maps produced. For instance, another game may make a km (multi-part key) room a safe room in which the player collects the key and leaves, but in the ZDG multi-part keys take the form of monsters which must be killed. The fact that the player could potentially harm themselves with bombs in any room or fall down holes in the majority of them is not taken into account.

4.2.1.2.4 Path Redundancy

$$\frac{\textit{Number of Redundant Rooms}}{\textit{Total Rooms}}$$

It was noted during implementation that some paths in a map do not lead to any reward or useful item, and so present useless and unnecessary work for the player to traverse. Path redundancy is calculated by taking the number of rooms which do not eventually lead to, or themselves contain, any reward, and dividing by the total number of rooms in the map (see Figure 60).

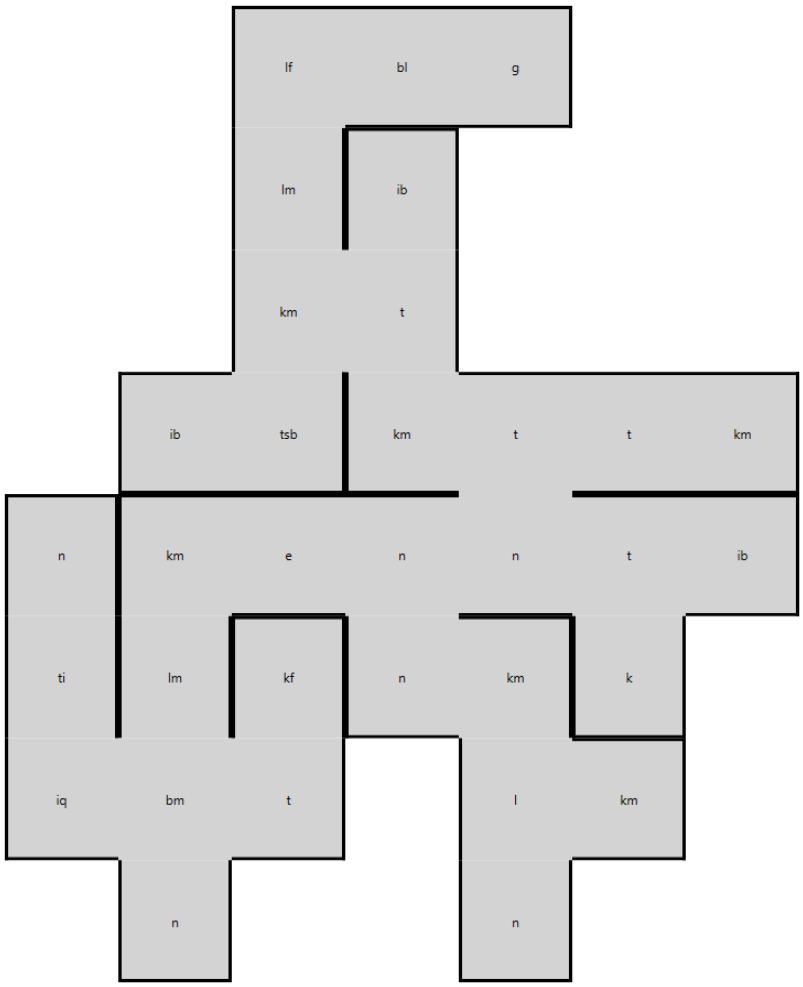


Figure 60: A map with redundant rooms. The two 'n' (nothing) rooms at the bottom of the map lead to no reward, and are therefore redundant. On the left hand side of the graph both 'ti' (item test) and 'n' are redundant, as they do not lead to any reward either. Therefore this map has a path redundancy of 4/33, or 0.121.

4.2.1.3 2D Histograms

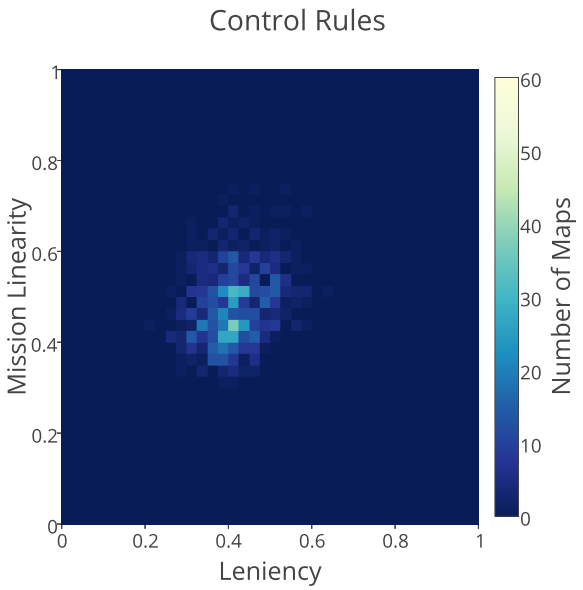


Figure 61: A 2D histogram representing the range of Mission Linearity and Leniency when using the Control Rule set.

The 2D histograms in this paper describe the relationship between two metrics for a set of generated results, one on each axis. The colour of each square bin in the graph corresponds to the number of generations which fell under those metric score brackets at the axes position. Each 2D histogram represents data from 1000 generations. Axes which span from 0 to 1 have been used in all histograms for ease of comparing them. Bin range maxes (e.g. 60 in Figure 61) were made the same for all histograms with the same combination of axes, for the same reason. Bins are set to a width and height of 0.025 for all histograms. All histograms are created using the online graphing tool Plotly (Plotly, n.d.).

4.2.2 Mission Metrics

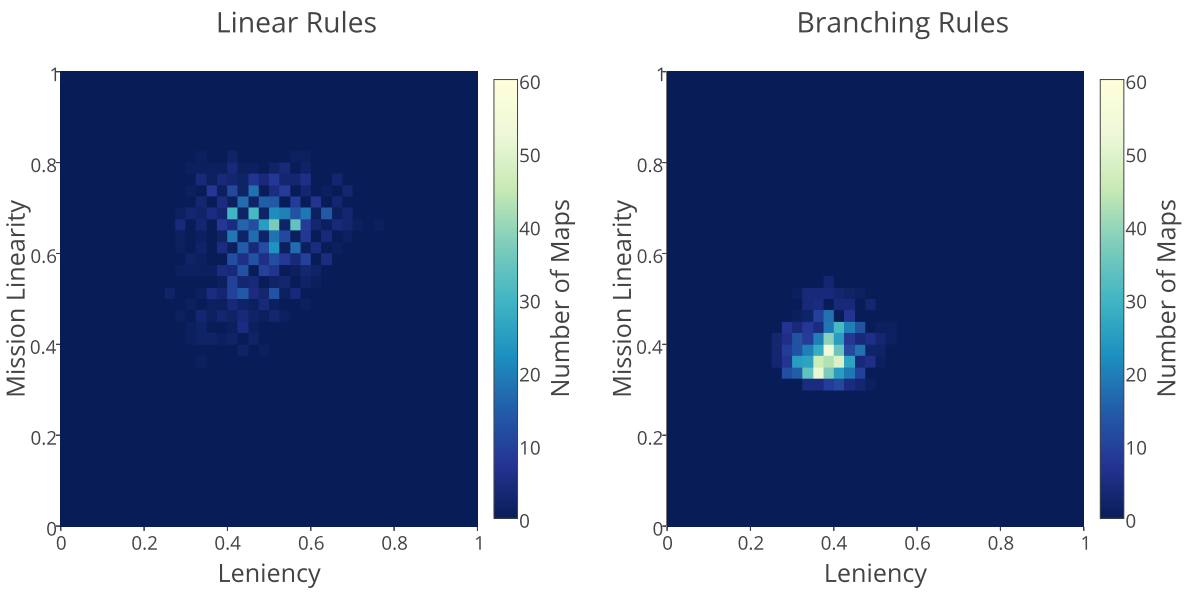
4.2.2.1 Control

Figure 61 shows the results when using the Control mission rule set and testing the Mission Linearity and Leniency parameters.

This shows the normal distribution of mission linearity and leniency when balanced rules are used. The maps seem to span the central values of linearity quite evenly. There is a similar span for leniency but more towards the lower end of the scale. This is easily explained when we consider that there are 9 safe room templates and 16 hazardous ones.

4.2.2.2 Linear vs Branching Rule sets

Figure 62a shows the results when using the Linear mission rule set and testing the Mission Linearity and Leniency parameters. Figure 62b shows the results when using the Branching mission rule set and testing the Mission Linearity and Leniency parameters. Examples of a mission graph from each set can be seen in Figure 63.

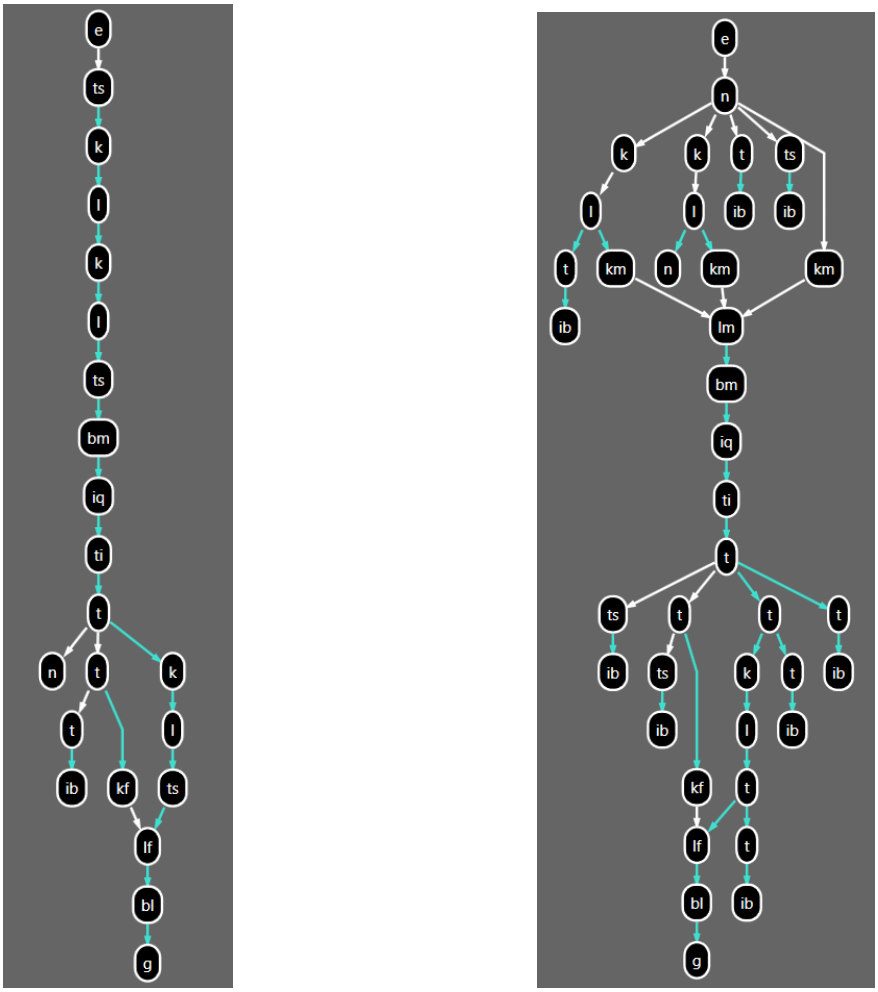


(a) *Linear Mission Rule set* (b) *Branching Mission Rule set*

Figure 62: 2D histograms comparing the range of Mission Linearity and Leniency when using the Linear and Branching Rule sets.

Linear rule sets cause the map results to spread towards higher linearity and higher leniency. This confirms that a rule set favoured towards individual linear rules will produce more linear missions overall. The slight increase in leniency is probably due to the necessity of placing vital safe rooms such as the dungeon item, item test and goal for the main quest, with no extra hazardous tests added for extra branching side-quests.

Branching rule sets cause the results to converge towards the less linear end of the original results. The smaller, more concentrated set of results is likely a reflection of the fact that there are only 4 rules considered highly branching and 11 rules considered linear in their respective rule sets. The small set of branching rules for the generator to pick from therefore led to similar results each time.

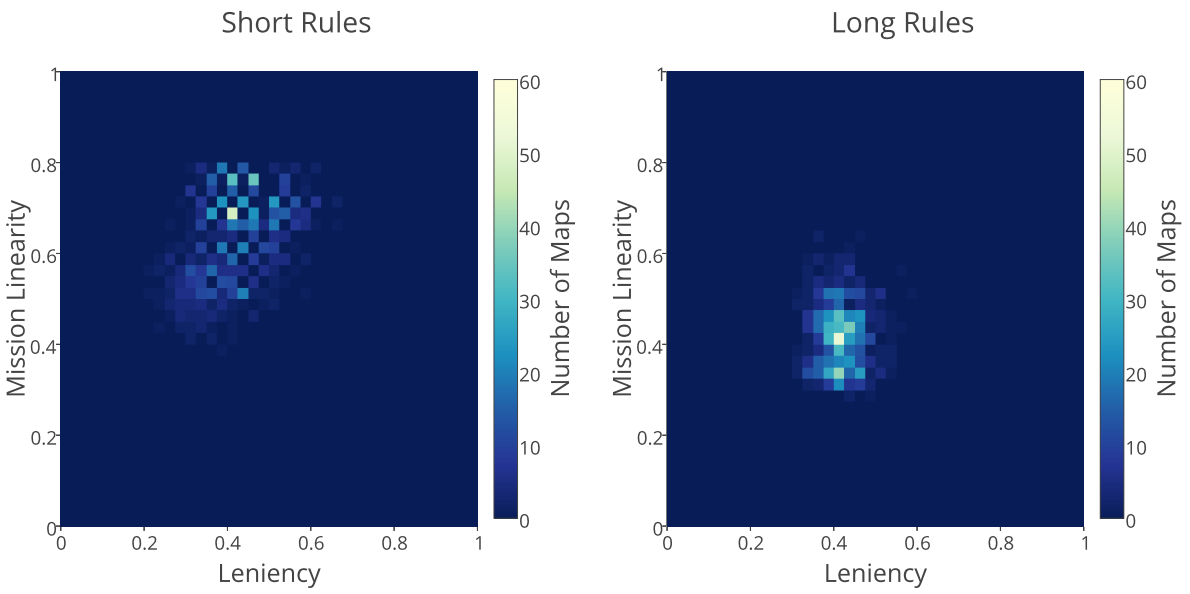


(a) *Linear Mission Rule set* (b) *Branching Mission Rule set*

Figure 63: Comparison of mission graphs produced when using the Linear and Branching Rule sets.

4.2.2.3 Short vs Long Rules

Figure 64a shows the results when using the Short mission rule set and testing the Mission Linearity and Leniency parameters. Figure 64b shows the results when using the Long mission rule set and testing the Mission Linearity and Leniency parameters. Examples of a mission graph from each set can be seen in Figure 65.

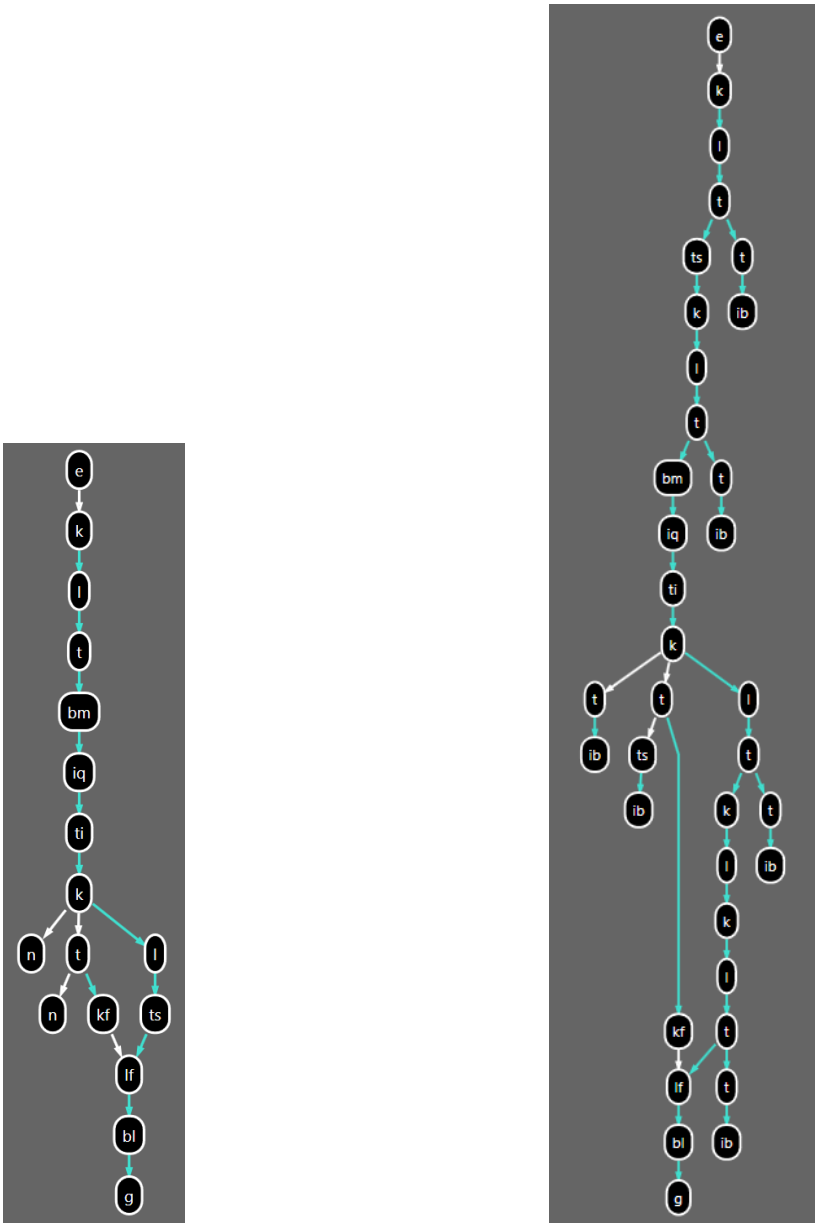


(a) *Short Mission Rule set* (b) *Long Mission Rule set*

Figure 64: 2D histograms comparing the range of Mission Linearity and Leniency when using the Short and Long Rule sets.

The short rules graph spreads out in a similar manner to the linear one, probably because most short rules are also linear. Both histograms contain gaps of 0 density in between high density pixels. This can probably be explained by the fact that the graphs were short. With less nodes to average between, the chances of getting the same results are increased. It was probably impossible to obtain a result within the boundaries of those 0 density, dark bins.

The long rules results converge similarly to the branching rules, but with an extension towards higher linearity. This likely means that the converging effect is opposite to the spread of 0 density issue explained above, and is caused by an abundance of nodes in the graph.



(a) *Short Mission Rule set* (b) *Long Mission Rule set*

Figure 65: Comparison of mission graphs produced when using the Short and Long Rule sets.

4.2.3 Map Metrics

All results shown in this section are produced when generating a map to correspond to the mission in Figure 66.

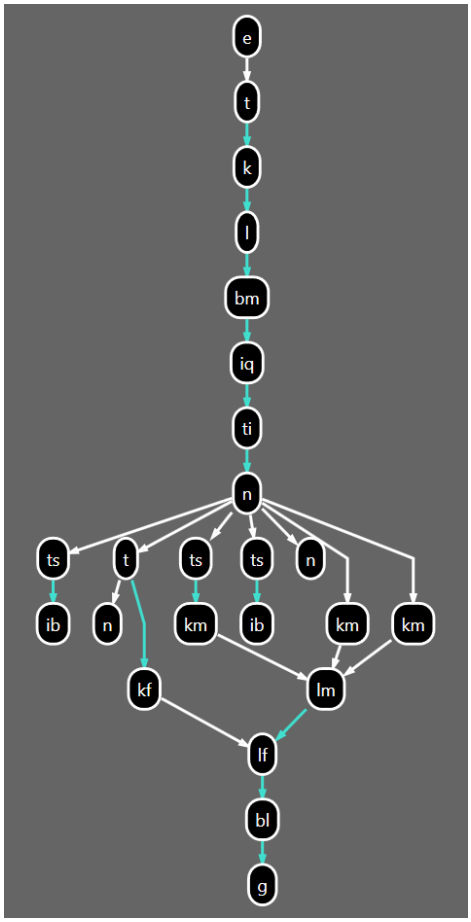


Figure 66: A typical mission graph used as the basis for the map expressivity tests.

4.2.3.1 Control

Figure 67 shows the results when using the Control map rule set and testing the Map Linearity and Path Redundancy parameters.

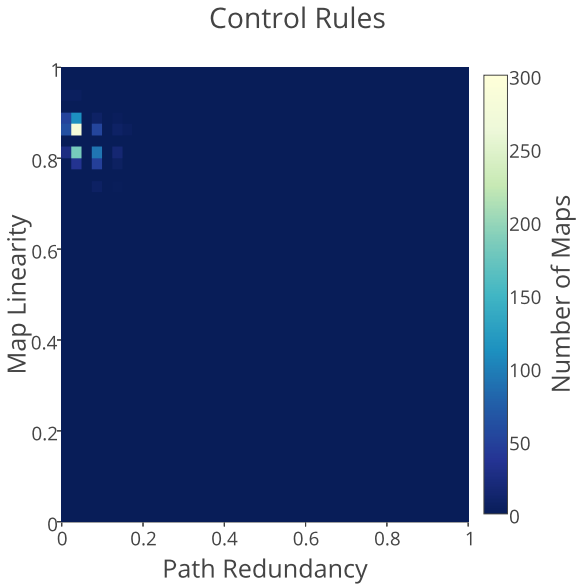


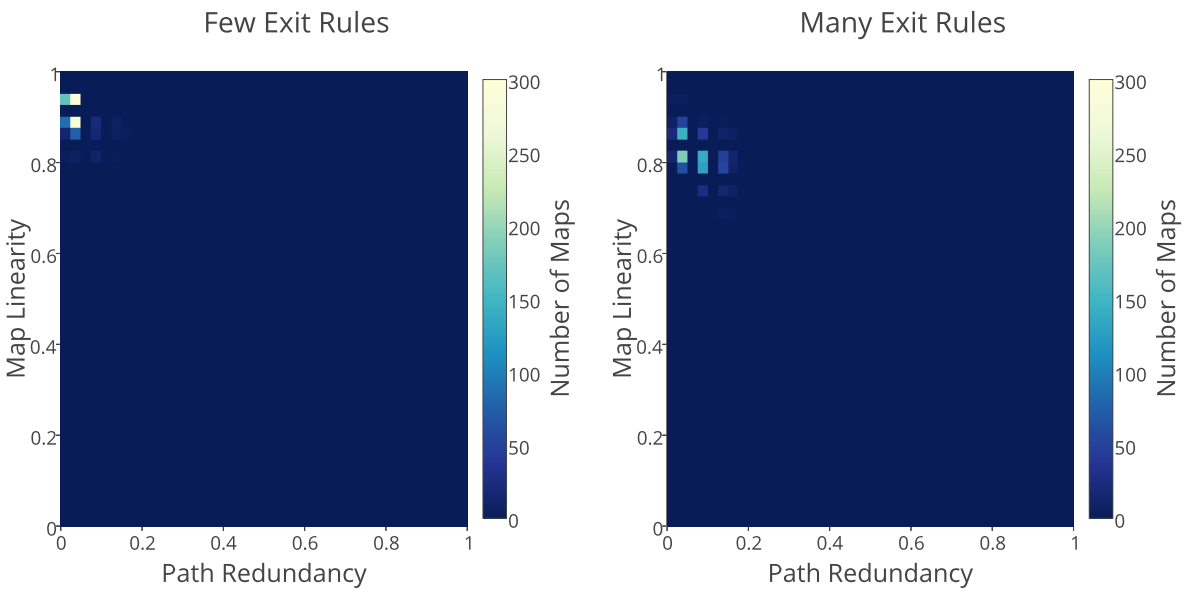
Figure 67: A 2D histogram representing the range of Map Linearity and Path Redundancy when using the Control Rule set.

The empty bin symptom described in section 4.2.2.3 can be seen to an extreme here creating a grid like effect, and highlights the fact that this type of data is less suited to being displayed in a 2D histogram. It appears that map configurations taken from the same mission offer very few different possible results for linearity and path redundancy. Path redundancy makes a lot of sense – there were probably at most 4 rooms which could possibly be in a redundant position in the map, leaving only 4 possible values of the redundancy metric. Map linearity is more surprising – it appears that in this case, only 5 possible values of map linearity could come about from the same mission. This reveals that varying the number of exits different rooms can have does not lead to an extensive set of linearity values.

The small range of map linearity is less surprising, as the equation used means that the value has always been between 0.6 and 1. This is probably due to the way that paths have to be closed off if they are next to a wall, and the fact that all remaining connections are closed at the end of the generation process - even if multi-exit rooms are placed to begin with, it does not mean they will be multi-exit by the time the algorithm finishes.

4.2.3.2 Few Exit vs Many Exit Rooms

Figure 68a shows the results when using the Few Exits mission rule set and testing the Map Linearity and Path Redundancy parameters. Figure 68b shows the results when using the Many Exits mission rule set and testing the Map Linearity and Path Redundancy parameters. Examples of a map from each set can be seen in Figure 69.

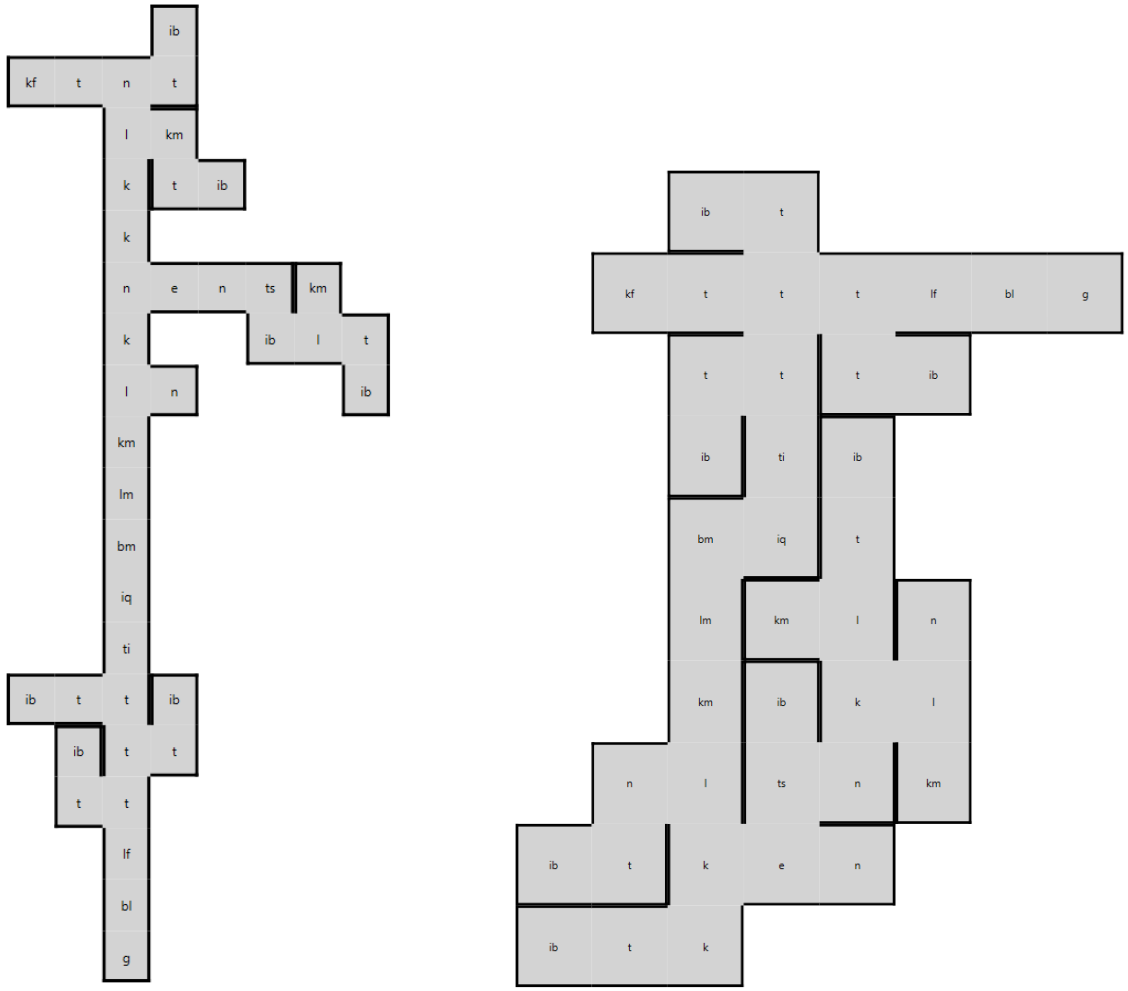


(a) *Few Exit Map Rule set* (b) *Many Exit Map Rule set*

Figure 68: 2D histograms comparing the range of Map Linearity and Path Redundancy when using the Few Exit and Many Exit Rule sets.

The Few Exit rule set histogram indicates that even a mission which produces relatively linear maps with control rules can produce even more linear maps when a set of linear rules is applied. The results change from a span of 0.7-0.9 to 0.8 to 1. We also appear to get less path redundancy with Few Exit rules, as less choice between positions to place rooms will lead to less rooms being placed off of a random path to then lead to nothing.

The Many Exit rule set is much more similar to the control graph. This is likely because there will only ever be a certain amount of rooms that need placing, so not all open connections can be filled. The likelihood is that there remained many more open connections when the more branching maps were placed, but they all had to be closed off when the algorithm ran out of rooms to place.



(a) *Few Exit Map Rule set* (b) *Many Exit Map Rule set*

Figure 69: Comparison of maps produced when using the Few Exit and Many Exit rule sets. Note: Not based on the mission used for the expressivity tests. Map 69a contains one redundant node of type 'n', while map 69a contains two redundant rooms, also of type 'n'.

4.2.4 Combined Metrics

4.2.4.1 Control - Relationships between Mission and Map Metrics

Figure 70 shows the results when using the Control mission rule set and the Control map rule set, and testing the Mission Linearity and Map Linearity parameters.

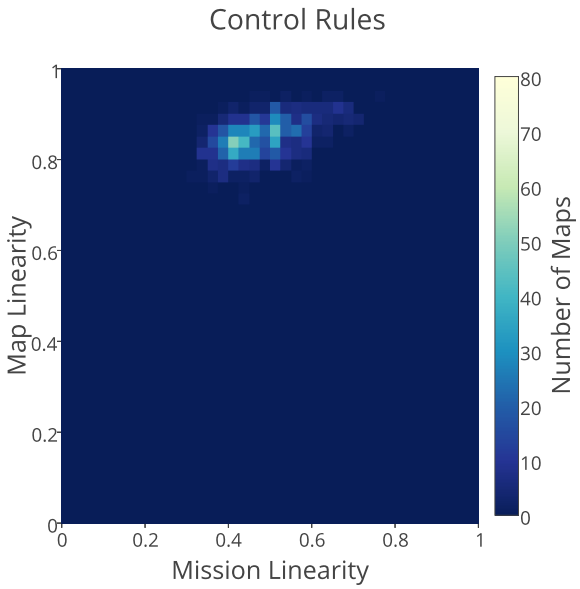


Figure 70: A 2D histogram representing the range of Map Linearity and Mission Linearity when using the Control Rule sets.

As this histogram effectively represents the typical output of the system, it is worth noting that it outputs a relatively small range of results, especially when it comes to map linearity. The reasons why map linearity is so constrained in general are discussed in section 4.2.3.1. The histogram does show a diagonal trend, implying map linearity increases in general with mission linearity, but not strictly. Linear missions causing linear maps makes sense, as there will be fewer side-missions to position from the main rooms.

Figure 71 shows the results when using the Control mission rule set and the Control map rule set, and testing the Mission Linearity and Path Redundancy parameters.

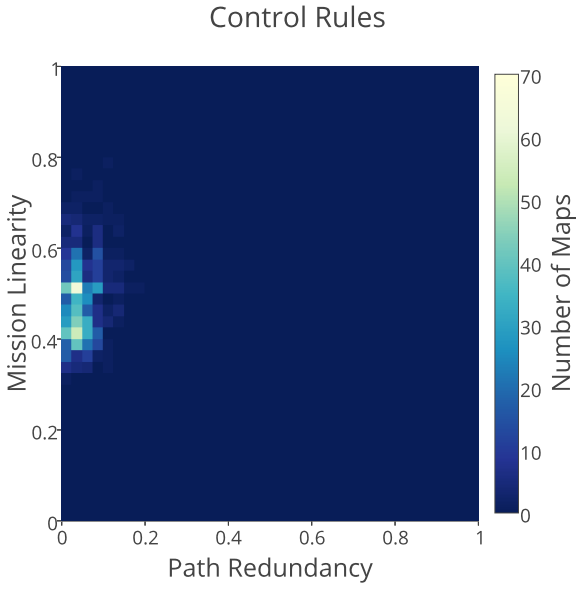
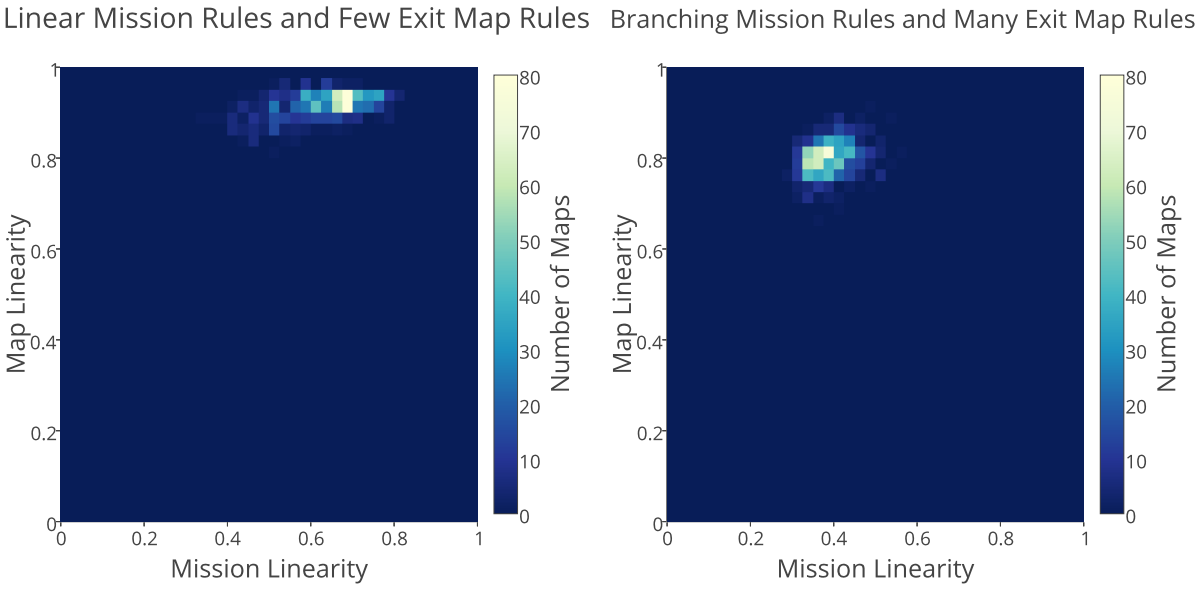


Figure 71: A 2D histogram representing the range of Path Redundancy and Mission Linearity when using the Control Rule sets.

Path redundancy is expectedly low in all cases, as only a few rooms can be redundant in any map configuration. It does not appear that mission linearity has any effect on path redundancy in the corresponding maps.

4.2.4.2 Double Linear vs Double Branching Rule sets

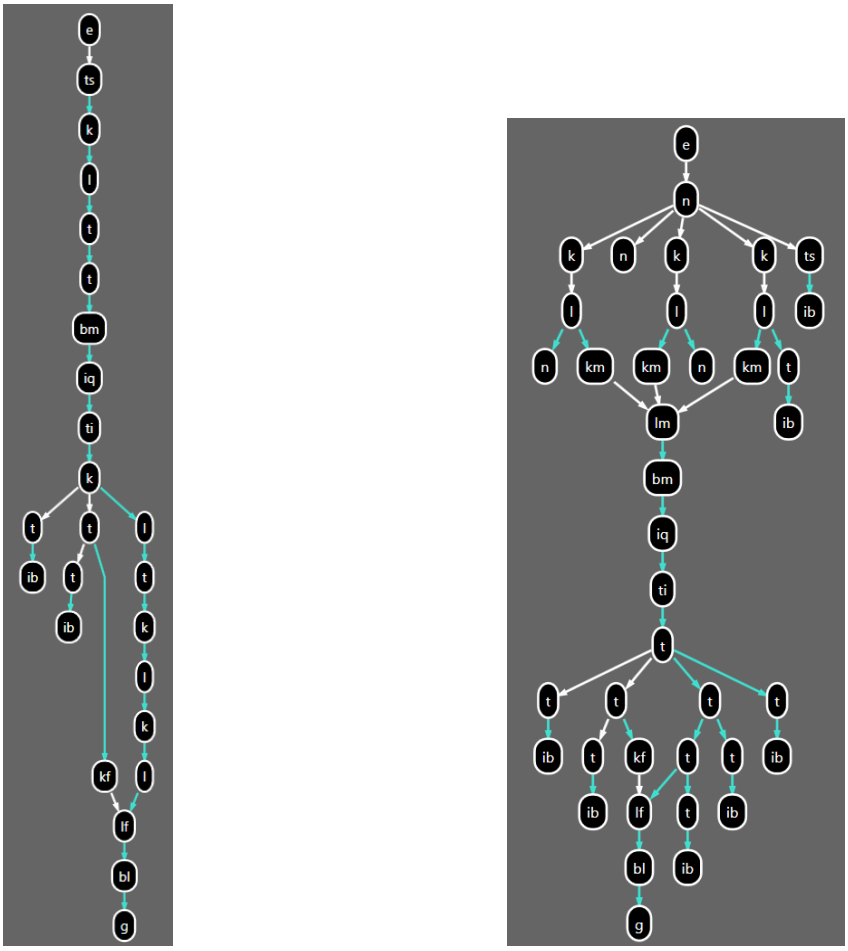
Figure 72a shows the results when using the Linear mission rule set and the Few Exits map rule set, and testing the Mission Linearity and Map Linearity parameters. Figure 72b shows the results when using the Branching mission rule set and the Many Exits map rule set, and testing the Mission Linearity and Map Linearity parameters. Examples of mission graphs and maps from each set can be seen in Figure 73 and Figure 74 respectively.



(a) *Linear Mission and Few Exit Map Rule sets* (b) *Branching Mission and Many Exit Map Rule sets*

Figure 72: 2D histograms comparing the range of Map Linearity and Mission Linearity when using both Linear Rule sets and both Branching Rule sets.

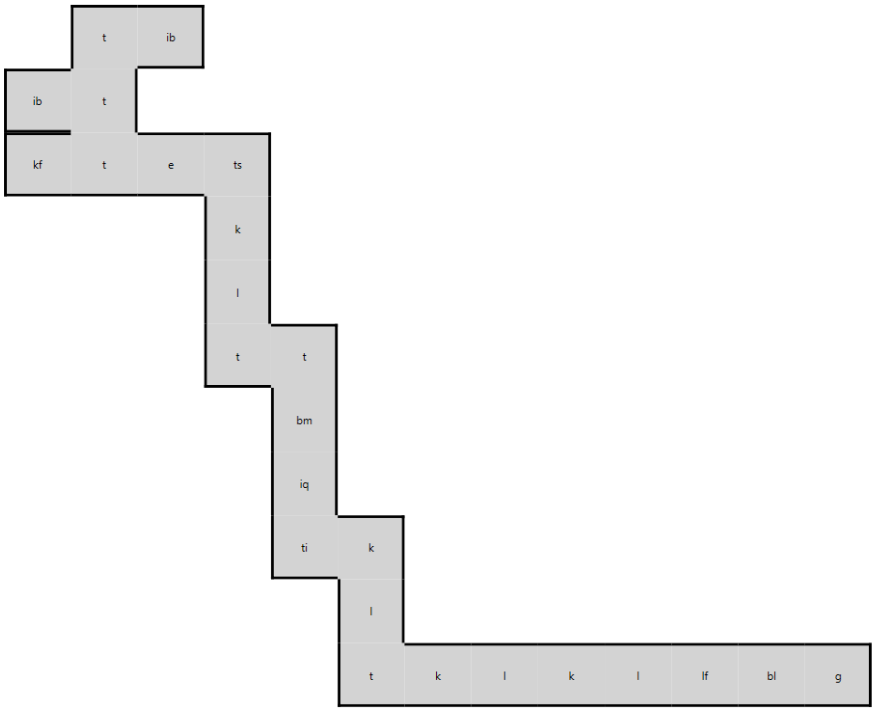
As expected, both histograms show an increase in density towards the linear/branching end of the spectrum of possible results presented in Figure 70. Both also display the diagonal trend, implying that there is a significant link between mission and map linearity.



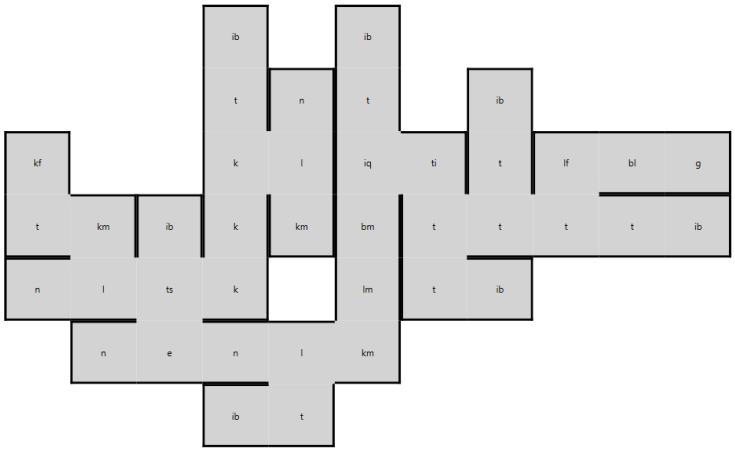
(a) *Linear Mission and Few Exit Map Rule sets*

(b) *Branching Mission and Many Exit Map Rule sets*

Figure 73: Comparison of missions produced when using both Linear Rule sets and both Branching Rule sets.



(a) *Linear Mission and Few Exit Map Rule sets*



(b) *Branching Mission and Many Exit Map Rule sets*

Figure 74: Comparison of maps produced when using both Linear Rule sets and both Branching Rule sets.

4.2.5 Discussion and Proof of Hypothesis

One of the most prominent biases seen, displayed across all graphs the metric is included in, is high map linearity, which more often than not falls between the 0.7 and 0.95 marks, even with branching rule sets applied. This is due to the implementation used – even if a 3 exit version of a room is placed down, there are multiple reasons that the exit may be closed before the map is complete. For instance, if it is placed next to another wall, or when the system runs out of rooms to be placed. Conceptually, if every room being placed down has three exits, probability dictates that only one of those exits will most likely be used, as the number of rooms will be a third of the number of exits. The rest will

be closed when the algorithm completes. There are also visible holes in the generation space at either end of the mission linearity spectrum. Again, this is a limitation of the algorithm/grammar implementation. A fully linear graph is impossible to achieve, as the rules of the grammar as Dormans described it dictate that a branching section of the graph must be placed in order to realise a final chain of events (refer back to Figure 29). A fully branching graph is also impossible, as the rules of the grammar indirectly dictate that at least 13 nodes must be placed in the shortest path between entrance and goal inclusively.

The expressive range tests can be seen as successful in that generally, rule sets have their expected effect: more linear rule sets lead to more linear missions and maps, etc. Limitations on the range of these metrics are demonstrated to be caused by issues specific to the example grammar used. This is backed up by the fact that the rule sets used in testing changed only rule weighting, not structure, in order to preserve functionality. Therefore, it is safe to hypothesise that with differently structured grammars the holes in the expressive range could be filled. Whether or not mission structure would be compromised as a result is a question for a different project!

The project hypothesis claimed that a diverse array of functional dungeons would be produced by the methods presented (see section 1.6). In the ZDG, all dungeons are fully functional by default. The ZDG produces a valid expressive range, with diversity being proven by the histograms given and vastly differing diagrams shown. Though some metrics are limited, we can be confident that they stem from limitations of the grammar used. We cannot say that the hypothesis is definitely true, but we have successfully provided evidence for it which can be expanded on in the future with different grammars.

The fact that the change of input rule sets did have a noticeable impact on these metrics has positive implications for the concept of designer control over the system, but no conclusions can be made on this front until the system is tested with an accessible user interface and a sample of designers.

4.3 Process Evaluation and Discussion

This section evaluates the success of the project process as a whole. Section 4.3.1 focuses on how well the implementation/methodology was carried out, considering successful decisions and limiting ones. Section 4.3.2 looks at the impact Dormans' grammars had on the development process.

4.3.1 Success of Methodology/Implementation

4.3.1.1 Mission Generation

Implementation of the mission grammar generation went very smoothly and without problems. The technique described in AiLD could be followed exactly, and it is highly likely that the implementation in the ZDG works precisely as it should according to the paper. This was largely due to the step by step guide to rewrite iterations reiterated in section 2.6.1.1 of this paper, which described precisely what needed to be done and in what order. GraphSharp turned out to be a good choice of graph library, fully compatible with the back-end representation of graphs as nodes which own edges. It also displayed graphs clearly, as Efficient Sugiyama was the perfect layout style for describing mission chains and dependencies. The debugging view, save and load functionality implemented in the ZDG were effective and fruitful.

4.3.1.2 Map Generation

In contrast to the mission implementation, map implementation was fraught with problems. As acknowledged in section 2.6.2, there were multiple occurrences of solutions not being described fully, or sometimes even mentioned, in the AiLD paper. Bespoke solutions therefore had to be designed and then iterated on until a fully working space generation implementation was found. Dormans' solutions to some of these issues are briefly referred to in AiLD but, as they were not described in detail, probably differ from the implementations in the ZDG quite a lot. Some issues stemmed from quirks of the ZDG implementation (such as the misunderstanding of BFS described in section 3.3.9), while some were inherent in the grammars (such as the lack of two exit rooms to deal with double tight couplings, described in section 3.3.6). As a result, the space generation implementation process took a lot more time than was first allocated to it. Moreover, Dormans mentions some improvements to the algorithm which there was not enough time to implement in the ZDG, such as connecting paths back to each other to enable easier backtracking.

As the shape grammar described in section 6 of AiLD was used as the basis for the ZDG, and effectively involved placing preset rooms, the fractal nature of shape grammars Dormans described could not be taken advantage of. Had a different kind of map been chosen for the ZDG, it would have been easier to investigate this quality of the grammars technique.

Also, some of the potential issues described in the methodology (sections 3.3.5.1, 3.3.8.1) which did not occur during initial testing and were deemed unworthy of addressing, began to show up when extreme rule sets were tested. Linear rule sets in particular led to rooms being placed in dead ends with no way to recover. As it was too late in the project to implement a proper fix for these, the graphs were set up to restart themselves after a number of failed attempts to place rooms. Therefore, the target gained as a conclusion from the Literature Review (see section 2.7.2.3), which stated that the ability of Dormans' grammars to generate valid results every attempt should be preserved, was not met.

However, despite the many problems encountered, the map generation succeeds in creating fully functional dungeons which always reflect their mission structure correctly. All in all, for generating grid-based dungeon maps, the ZDG accomplishes its aim.

4.3.1.3 Application to Game

Applying the resulting maps to a Solarus game was a relatively painless process. The fact that the maps produced were grid-based made them relatively simple to set up in-game. The Solarus engine was an excellent choice, not least because of the teletransporters and destinations which made the map layout so easy to execute, but because of the easily accessible Lua backend. Although when planning the project it was fully expected to have to change the source code of the game engine, the majority of necessary generation could be handled solely by manipulating maps' Lua scripts, while the map editor could be used to easily create all the separate rooms.

One issue caused specifically by the ZDG implementation, not the grammars themselves, was the fact that rooms sometimes needed to be relatively symmetrical in order to serve their purpose (see section 3.4.2.2). This proved to limit rooms which might have be used, but could be avoided if the contents of rooms were set to rotate with the corresponding rotation in the backend map. Hurdles encountered such as the reuse of maps

and the need to generate savegame variables dynamically could be fixed more elegantly by manipulating the engine source, but were adequately dealt with for the purposes of this project. The need to differentiate between secret tests before and after the player had acquired bombs (see section 3.4.3.2) could have been solved by accounting for the distinction in the mission grammar.

Another limitation stemming from the ZDG specifically, and not the grammars themselves, was the use of template rooms. Projects such as *Lenna's Inception* (see section 2.5.2) move away from the reliance on templated areas in PCG, and generate the content of areas as well as their positions relative to each other.

All in all, the resulting game is quite basic and has limitations which stem from its implementation, but it forms a solid proof of concept with a vast potential for future work.

4.3.1.4 Product Evaluation

The product evaluation process went successfully. A thorough analysis of expressivity was conducted in accordance with Smith and Whitehead's guidelines (Smith and Whitehead, 2010), using representative samples of 1000 generations per test. The separation of mission and map expressivity testing also made it simpler to analyse the effects of each distinct section of the generation process.

There would be plenty of scope for expanding this evaluation, however, given time and resources. For instance, it would be very interesting to analyse a range of authentic LTTP maps for these metrics, and compare them to those produced by maps from the ZDG. More accessible parameters could also be exposed for testing, for instance a linearity slider. In order to evaluate the system on intuitiveness and the quality of maps produced by the system, a sample of designers would be needed to create their own grammars. A sample of players would also be needed to evaluate certain parameters such as engagement, or the ability for the maps to pass as human-authored.

4.3.2 Dormans' Grammars and the Development Process

As acknowledged in section 4.3.1.1, reproducing Dormans' well-explained mission generation technique was simple and straightforward. The mission grammar described in AiLD is definitely recommended as a foundation on which to base similar mission generation projects.

Map generation presented many more time consuming hurdles than expected. It is not easy to separate those which came about simply because they had not been described in detail, and those which are problems inherent to Dormans' presented space grammar. Some of the fixes implemented (see section 3.3) were necessarily complex and elaborate. By the end of the implementation process, it seems that the shape layout system was 20% shape grammar and 80% algorithmic fixes. The fact that Dormans moved away from the use of shape grammars in this way in his later papers suggests that some issues were inherent to the technique itself (see section 2.6.2).

The many problems found which were specific to using shape grammars in a grid format imply that several other different issues would occur when trying to apply the shape grammar concept to other types of map. Some of these problems are already solved by the dungeon generation techniques in section 2.3.2.1, and as the grid format cannot take advantage of the shape grammar's fractal aesthetic, it may be worth trying

to incorporate grammatical mission generation with some other form of map generation at another opportunity.

The process of applying generated maps to the Solarus game went well, and demonstrates that applying Dormans' symbols to corresponding rooms works. Allocating one room per symbol does contribute to the generated feel of the map, but Dormans has previously demonstrated that multiple mission nodes can be applied to the same rooms (Dormans and Bakkes, 2011). If the space generation could be improved, Dormans' grammars show promising potential for being applied to other AA games, and more modern *Zelda*-style games. However, the grammar would need to be expanded to cope with more complex puzzles which span several rooms (such as crystals which raise and lower fences, and multi-floor dungeons in which the player must fall through the correct hole in order to get to their destination on the floor below) to truly reflect the game's traditional style.

5 Conclusions and Recommendations

5.1 Success of Aim, Objectives and Hypothesis

The aim, objectives, and hypothesis of this work were formulated at an early point in the project, and are identified in section 1. This section revisits each criterion and assesses whether or not it has been met over the course of the work.

5.1.1 Objectives

The original project objectives can be found in section 1.5.

1. The literature review consisted of a thorough investigation into a variety of map and mission generation methods, existing issues faced when generating these for AA games, and the theory of generative grammars. The review was also used as a means to look into Dormans' further work and to find projects which became inspiration for the ZDG.

2. A graph grammar generator was successfully created to produce dungeon missions in the manner described by Dormans. The process was documented, but not in depth, as there were not many issues to discuss. The algorithm used follows the steps detailed in AiLD exactly.

3. Construction of the shape grammar generator was problematic but eventually solved and well documented. Solutions to various obstacles were presented and described in detail. Whether they are the most efficient possible approaches remains to be seen, but they were sufficient for implementation of the ZDG and should be applicable to similar projects.

4. Application of the resulting missions and maps to a 2D AA game was completed successfully with the Solarus engine. The produced game exhibits many features found in typical *Zelda*-style games, demonstrating that they are compatible with the method of generation used.

5. The development process was reflected on in detail in section 4.3. Weaknesses were considered and it was determined whether each stemmed from either Dormans' grammars or separate quirks of the implementation.

6. Evaluation of the product was conducted using an in-depth expressive range analysis and breakdown of the implementation limitations. Dungeon functionality was ensured by default and therefore not necessary to analyse in the same manner. The results were then used to assess the hypothesis, which is reflected on in sections 4.2.5 and 5.1.2.

All objectives specified for this project were therefore completed successfully.

5.1.2 Hypothesis

First presented in section 1.6, the project hypothesis is as follows:

"Applying Dormans' grammar based mission and map generation process to a 2D action-adventure Zelda-Style game will yield a diverse array of functional dungeons, which

reflect the structure of their generated missions successfully."

Dormans' grammars were successfully applied to a 2D *Zelda*-style game, as evidenced in section 5.1.1. The diversity of the dungeons produced was evaluated in section 4.2 through expressive range analysis. Evidence was provided for a vast range of possible outputs which could be controlled by changing the rules input into the system. However there were holes in the expressive range when tested against the metrics chosen. All evidence points towards the cause of these being the grammar used, or the very specific way space generation was implemented. All dungeons produced are functional and reflect their missions by default – the order in which rooms are allocated in the space means that all dungeons are valid and contain no way for the player to get stuck in an unrecoverable situation. In doing so, the order also dictates that all mission structures will be fully accounted for in the space and in a correct, accessible order.

Therefore, we can conclude that applying Dormans' grammars in this domain produced an array of functional dungeons which reflected the structure of their generated missions successfully. We cannot conclude that these dungeons are diverse, but as stated in section 4.2.5 evidence has successfully been provided for this hypothesis, which can be expanded on in the future with different grammars and varying space implementations.

5.1.3 Aim

First presented in section 1.5, the project aim is as follows:

"To evaluate Joris Dormans' two-tier generative grammar technique when applied to a typical 2D action-adventure game in two respects; effect on the development process and the expressive range of the resulting dungeons."

In order to evaluate the grammar technique, several steps of implementation first had to be undertaken. These steps are detailed in section 5.1.1. A thorough investigation was then carried out into the expressive range of the project and how Dormans' grammars affected the development process.

The implications on development are covered in section 4.3.2. Information was gathered throughout the implementation process and presented in the Methodology. In section 4.3, issues were separated between those inherent in Dormans' approach and those specific to the ZDG. It was found that the grammars had a positive effect on mission generation, being easy to use and apply. The space grammars had negative implications as many hurdles were faced, requiring complex time-consuming workarounds. The generation results, which matched the map format described in AiLD, translated to a game context with relative ease. Some limitations on the game created were found, but the majority were specific to the ZDG.

The expressive range of results is covered in section 4.2. A diverse set of results were gathered, but with holes in the expressive range due to the example grammar used. Access to designers who could create more grammars would have made the project aim simpler to achieve. Mission generation was seen to create a broad expressive range whereas map generation was not.

Both aspects were evaluated thoroughly and had implications drawn from them (briefly described above), meaning the aim of the project was completed successfully.

5.2 Limitations

5.2.1 Lack of Mission Grammar Designers

Game Designers are required to construct the type of grammars Dormans specifies (Dormans, 2010). The creation of such grammars entails intricate insight into causal dependencies of tasks and good game design patterns. As no designers were available, this project was based on the one mission grammar example Dormans included in his AiLD paper. This was sufficient to create a mission generator which produced a wide range of missions, but did mean that holes in the expressive range were found in the evaluation. It also limited the evaluation in that only the probabilities of different rules being chosen could be changed, the rules themselves could not be restructured without risking the structure and validity of the dungeons produced.

5.2.2 Limitations of the Map Generation Implementation

The most obvious limitation of the map generation was the grid-based technique in AiLD chosen for literal implementation. This meant that rooms in the final game were limited to being the same size and shape as each other (in order for the dungeon to be geometrically realistic!) with a maximum of four exits each in the centre of their walls. This also led to many other limitations on the game which could be produced, such as some rooms having to be laid out symmetrically (see section 3.4.2.2). On top of this, it meant that the fractal nature of shape grammars could not be taken advantage of. Though the example of space creation Dormans gives in AiLD is grid-based, his grammars have been applied to non-grid based dungeons in the past (Dormans, Joris, n.d.[b]).

As various implementation hurdles were met when creating the map generator, time was tight and so the generator remains limited in some ways. The intersection-based solution Dormans describes for connecting the space back on itself in order to avoid backtracking was not implemented, and complex fitness functions for possible room positions were not explored (which may have been the cause for the tendency towards linear maps.)

5.2.3 Grammar Limitations on Game Mechanics

It was found that some typical *Zelda* mechanics and puzzles could not be represented using the grammars Dormans provides. For instance, puzzles which span several rooms such as those involving the state of crystals and fences (see Figure 75) could not be represented by the mission or space grammars, as they require complex links between room content and map layout. The same goes for puzzles which span several floors of a dungeon, such as those where the player must fall through correctly placed holes to the floor below (this would require expansive adaptation of the shape grammar, which does not even support multiple floors in its current state.)

Figure 75: An example of a puzzle in which the colour of a crystal affects the state of fences. The state of the crystals and fences persist across multiple rooms and floors. (*Moby Games* n.d.)

5.2.4 Lack of Resources for Evaluation and Limited Conclusions

The lack of designers referred to in section 5.2.1 also had vast implications on the aspects of the ZDG which could be evaluated and how. Without having designers try and create grammars and witness their output, the system cannot be evaluated on ease of use, success of representing many different design patterns etc. To accommodate these designers, the system would need a UI overhaul to increase accessibility of grammars, as all relevant input is undertaken by changing XML files at present. Without a sample of players, the game cannot be evaluated on factors such as engagement or Turing-inspired comparison to human-authored levels.

5.2.5 Offline Generation

One major limitation of the project is the fact that all generation is done offline, before the game is started. Therefore, no adaptation can be done whilst the game is being played, and techniques such as difficulty scaling and player modelling are off-limits. Though these are desired areas of future work for the system, the current implementation relies on the offline approach. For instance, the C# backend is run once, applying its changes to the Solarus maps and Lua files before the game is run. It is unclear what effect altering these files would have if carried out whilst the game was running.

5.2.6 Specific Implementation

It is noted that the implementation created for this project is very specific in the nature of the grammars it is compatible with, and the game it can be applied to. A vast overhaul would be needed if the system was to be generalised for more diverse use.

5.3 Contributions

5.3.1 Analysis and Documentation of Dormans' Grammars

This paper provides evidence for Dormans' mission grammars being easy to use, expressive, and well documented, which may aid others who wish to use them for their own projects. It acknowledges some oversights in relation to space generation in AiLD but provides well-documented solutions to the problems encountered, which should allow others to reproduce the implementation.

5.3.2 Addition of Context to Level Generation

This work successfully adds context (in the form of a mission graph) to maps and consequently the in-game dungeons they produce. As It has been demonstrated that addition of context is a step towards solving various challenges faced by modern PCG and towards more broad goals such as multi-level PCG (Togelius, Champandard, et al., 2013). Togelius et al. give *Zelda*'s unique take on design problems as an example of content it is profitable to generate. The ZDG successfully applies generation to a game which features many aspects and mechanics of typical Zelda games (though limited in scope!)

5.3.3 Demonstration of a Method which Deals With Action-Adventure Generation

The ZDG is an added example of a generator which deals with the problems specific to the generation of AA games. It provides evidence that these problems can be solved and paves the way for similar work which will either expand on the ZDG or attempt to solve other problems with AA generation. This process will be necessary if AA generation is ever to be done to the same scale as other genres.

5.3.4 Demonstration of Research Application to Open Source Game Making Tool

It should be noted that the application of these techniques to the Solarus game/engine was interesting and straightforward, and shows the potential for other generation experiments to be carried out in Solarus and/or other similar game-making tools. In Solarus in particular, the dynamic Lua elements were excellent to work with, and could be improved still further if small tweaks were made to the interaction of properties between Lua and the engine. While procedurally generated content continues to lack in some areas that human-authored content does not, it is beneficial to try and apply PCG to pre-existing games which showcase these aspects. This way, content can be compared and analysed as to what is omitted.

5.4 Future work

5.4.1 Expand Content of the ZDG

The fact that some rooms had to be duplicated in the ZDG was merely due to time constraints. Adding alternative room templates, such as different types of tests and more, could provide a lot of variety and produce a more enjoyable game. More types of multi-key could also be created, so that the gibdo-lantern lock selection of rooms (see section 3.4.3.1) wouldn't be the only choice of content for a branching section of the dungeon. Different dungeon items and dungeon themes could also be added to the game, and this variety of content would move the game further away from a generated feel. An element of random generation could also be applied to the transfer of map to game, with elements such as monsters or item bonuses also being randomised.

When making these changes, it would be important to ensure that no two duplicated multi-key sets appeared in the same dungeon, at the risk of re-awakening the mix up described in section 3.4.3.1. The random selection of content would carefully need to be moderated to ensure the dungeons remained completable by default.

5.4.2 Adapt the ZDG for Online Generation

The potential for a version of the ZDG which generates whilst the game is being played is vast. The many suggestions Dormans and Bakkes made for future work such as the use of player modelling and profiling for difficulty scaling, space adaptation and mission adaptation could be applied (Bakkes and Dormans, 2010). These would be carried out by leaving non-terminal symbols in the space and/or mission after initial generation, and rewriting them with adapted rule sets as the user plays through.

Adapting the ZDG in this way would require a large overhaul, as the backend as it stands at the moment is run before the game, and edits many of the game files. It's unknown at present whether or not the Solarus engine would cope with these files being manipulated while the game is running, and alteration of the Solarus engine may be necessary. Event-based data could then be fed back into the backend from Solarus so that the player's actions could be taken into account. Some of the space generation fixes implemented at present may be incompatible with the remainder of non-terminals in the mission, such as the allocation of room places in a chain of tight connections. Alternative methods of overcoming these issues would need to be found.

5.4.3 Evaluate the ZDG with Designers and Players

It is important that the ZDG could be used by game designers to create their own alphabets and sets of rules, and test the output. Only then could the ZDG be evaluated for a range of factors based on its potential for industry use. Once designer-authored grammars were in play, generated levels' expressive range metrics could be compared to those found in real LTTP levels, and the results used to determine how to tweak the grammars to better reflect design patterns.

Designers could also be used to produce more grammars and provide a more diverse range of input data for the expressive range evaluation. In this way, the relationship between strict mission structure and the expressivity of the resulting dungeons could be explored.

Player participants would also be necessary in order to test the player modelling online adaptation to the system suggested in section 5.4.2, or to test whether or not they could tell generated maps from human-authored ones.

An intuitive grammar editor would be needed in order to carry out this evaluation, with accessible methods of viewing and changing the alphabets and rules used in the grammar.

5.4.4 Apply Dormans' Grammars to Other Action-Adventure Games

It would be very interesting to see Dormans' grammars applied to a 3D game. If the game was grid-based, the challenges faced in generation should not differ from those dealt with in the ZDG, however the 'generated' feel may become more apparent. Dormans' grammars should also be applied to typical AA games which differ from *Zelda*, in order to test this method of AA generation in a multitude of different contexts. Issues individual to those games, such as the puzzles spanning multiple rooms in *Zelda*, may become apparent.

5.4.5 Expand the Capabilities of Dormans' Grammars

Some limitations of Dormans' grammars could be eradicated if the grammars were expanded. For instance, the ability for multiple paths of actions to lead to one goal. This could give the player the choice of how to undertake a particular task. Also, in order to be able to fully realise *Zelda* design, the ability for puzzles spanning multiple rooms or floors could also be added. When implementing these additions, it would be important to ensure each design structure in the grammar remained intact and placed in the correct order, and that the dungeon would still be valid and completable.

5.4.6 Utilise Dormans' Grammars in Conjunction with Other Techniques

When comparing Dormans' approach to others' in the Literature Review, no reason was found that some of the other projects' perks could not easily be combined with Dormans' grammar generation. For instance, an entire dungeon layout could be generated with the two-step grammar approach, and then the rooms within it filled using Coxon's room content tile generation (*Overworld Overview - Part 2* n.d.).

As suggested in section 4.3.2, Dormans' mission grammar could also be combined with many different space generation techniques, for instance those presented in section 2.3.2.1. These alternative techniques could be used to address certain limitations of the map generation process discussed in section 5.2.2. However, it would be necessary to ensure the missions were properly represented in the spaces produced. In order to achieve this, it may be necessary to exploit aspects of certain techniques (see BSP discussion in section 2.3.2.1.1).

5.4.7 Apply Dormans' Grammars to Overworld Generation and Story-Based Quests

Lenna's Inception demonstrates that the same key and lock technique can be used to generate both dungeons and overworlds (*Overworld Overview - Part 2* n.d.). It would be interesting to see whether or not Dormans' approach could be applied to overworld situations as well. Dungeon-relevant nodes such as 'obtain key' could be replaced with

story-relevant actions, such as 'interact with NPC'. Events could be placed in the world in a similar manner to Coxon's key order system, but instead using the position of the node in the story graph as the basis of where to place it in the map to ensure correct progression. Difficulty scaling could be achieved in the same way.

Some of the other additions proposed in this section may be needed before attempting this task. For instance, the grammars would need to be expanded to cope with multiple paths of actions, as overworlds in general are much more branching and present much more navigational choice than dungeons.

5.5 Summary

The ZDG reproduces the two-step mission and map grammar generation introduced in Dormans' AiLD paper and applies the resulting maps to the 2D action-adventure *Zelda*-Style game Mystery of Solarus. All steps work successfully, but the resulting game is quite limited in scope. However, it has been used to undertake a thorough assessment of Dormans' grammars as a method of generating action-adventure dungeons. The majority of results were successful, with the stages of mission generation and application to the game going smoothly. The space generation stage, however, was problematic, and so future work is suggested in which some other form of map generation takes its place. All in all, the grammars have succeeded in applying overarching design patterns and structure to procedurally generated maps, and have exciting implications for the types of content and types of game which could involve procedural generation in the future. Ultimately the addition of context to PCG enables the field to move away from bland, random content which is novel purely because of the fact it is generated, and towards purposeful generated content which is novel in and of itself.

References

Adams, David (2002). *Automatic Generation of Dungeons for Computer Games*.

Almgren, Simon et al. (2014). *Astrogue: A Roguelike*.

Armageddon Games. *Zelda Classic*. `http://www.zeldaclassic.com/`. [Online; accessed 19-April-2015].

Ashmore, Calvin and Michael Nitsche (2007). "The Quest in a Generated World". In: *Proc. 2007 Digital Games Research Assoc.(DiGRA) Conference: Situated Play*, pp. 503–509.

Bakkes, Sander and Joris Dormans (2010). "Involving Player Experience in Dynamically Generated Missions and Game Spaces". In: *Eleventh International Conference on Intelligent Games and Simulation (Game-On'2010)*, pp. 72–79.

Barber, Sacha (2010). *Pretty Cool Graphs In WPF*. `https://sachabarbs.wordpress.com/2010/08/31/pretty-cool-graphs-in-wpf/`. [Online; accessed 15-April-2015].

Blizzard Entertainment (1996-present). *Diablo*.

Chang, Hsueh-Min and Von-Wun Soo (2009). "Planning-Based Narrative Generation in Simulated Game Universes". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 1.3, pp. 200–213.

Chomsky, Noam (1956). "Three Models for the Description of Language". In: *Information Theory, IRE Transactions on* 2.3, pp. 113–124.

Compilation Instructions. `http://www.solarus-games.org/development/compilation-instructions/`. [Online; accessed 15-April-2015].

Coxon, Tom (2014). "Flow in Procedural Generation". In: *PROCJAM*. London.

Dart, Isaac and Mark J Nelson (2012). "Smart Terrain Causality Chains for Adventure-Game Puzzle Generation". In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, pp. 328–334.

Dormans, Joris. *A Mission Space Generator*. `http://www.jorisdormans.nl/missionspacegenerator/` [Online; accessed 19-April-2015].

— *Quest*. `http://www.jorisdormans.nl/dungeoncrawl/`. [Online; accessed 15-April-2015].

Dormans, Joris (2010). "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games". In: *Proceedings of the 2010 workshop on procedural content generation in games*. ACM, p. 1.

— (2011). "Level Design as Model Transformation: a Strategy for Automated Content Generation". In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, p. 2.

Dormans, Joris and Sander Bakkes (2011). "Generating Missions and Spaces for Adaptable Play Experiences". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 3.3, pp. 216–228.

Dormans, Joris and Stefan Leijnen (2013). "Combinatorial and Exploratory Creativity in Procedural Content Generation". In: *Proc. PCGames '13, Society for the Advancement of the Science of Digital Games*, pp. 1–4.

DuBois, Roger Luke (2003). "Applications of Generative String-Substitution Systems in Computer Music". PhD thesis. Columbia University.

Games, Epic (1999-present). *Unreal Tournament*.

Goel, Narendra S and Ivan Rozehnal (1991). "Some Non-Biological Applications of L-Systems". In: *International Journal Of General System* 18.4, pp. 321–405.

Golding, James (2010). "Building Blocks: Artist Driven Procedural Buildings". In: *Presentation at Game Developers' Conference*.

Graph# (2010). `https://graphsharp.codeplex.com/`. [Online; accessed 15-April-2015].

Hartsook, Ken et al. (2011). "Toward Supporting Stories with Procedurally Generated Game Worlds". In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, pp. 297–304.

id Software (1993-present). *Doom*.

IDV (2014). *Speedtree for Video Game Development*. `http://www.speedtree.com/video-game-development.php`. [Online; accessed 15-April-2015].

Ince, S (1999). "Automatic Dynamic Content Generation for Computer Games". In: *University of Sheffield Dissertation*.

Johnson, Lawrence, Georgios N Yannakakis, and Julian Togelius (2010). "Cellular Automata for Real-Time Generation of Infinite Cave Levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, p. 10.

Lenna's Inception announced.

Li, Boyang and Mark O Riedl (2010). "An Offline Planning Approach to Game Plotline Adaptation." In: *AIIDE*.

Linden, Roland van der (2013). "Designing Procedurally Generated Levels". PhD thesis. TU Delft, Delft University of Technology.

Linden, Roland van der, Ricardo Lopes, and Rafael Bidarra (2014). "Procedural Generation of Dungeons". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 6.1, pp. 78–89.

Mateas, Michael and Andrew Stern (2005). "Structuring Content in the Façade Interactive Drama Architecture." In: *AIIDE*, pp. 93–98.

McCoy, Joshua et al. (2013). "Prom Week: Designing Past the Game/Story Dilemma." In: *FDG*, pp. 94–101.

Merrick, Kathryn E et al. (2012). "Computational Creativity and Procedural Content Generation". In: *Unpublished article*.

Moby Games. `http://www.mobygames.com/images/shots/l/396335-the-legend-of-zelda-a-link-to-the-past-snes-screenshot-tower.png`. [Online; accessed 15-April-2015].

Mojang (2009). *Minecraft*.

Myers, Andrew (2012). *Graph Traversals*. `http://www.cs.cornell.edu/courses/cs2112/2012sp/lectures/lec24/lec24-12sp.html`. [Online; accessed 19-April-2015].

Nintendo (1986-present). *Metroid*.

— (1991). *The Legend of Zelda: A Link to the Past*.

— (1998). *The Legend of Zelda: Ocarina of Time*.

— (2006). *The Legend of Zelda: Twilight Princess*.

Nitsche, Michael et al. (2006). "Designing Procedural Game Spaces: A Case Study". In: *Proceedings of FuturePlay* 2006.

Open Zelda. `http://openzelda.net/`. [Online; accessed 19-April-2015].

Orland, Kyle, David Thomas, and Scott Matthew Steinberg (2007). *The Videogame Style Guide and Reference Manual*. Lulu. com.

Overworld Overview - Part 1. `http://bytten-studio.com/devlog/2014/09/08/overworld-overview-part-1/`. [Online; accessed 19-April-2015].

Overworld Overview - Part 2. `http://bytten-studio.com/devlog/2014/09/15/overworld-overview-part-2/`. [Online; accessed 11-April-2015].

Persson, Markus (2008). *Infinite Mario Bros*.
— (2011). *Terrain Generation, Part 1*. http://notch.tumblr.com/post/3746989361/terrain-generation-part-1. [Online; accessed 15-April-2015].
Plotly. *Collaborative Data Science*. https://plot.ly/. [Online; accessed 19-April-2015].
Porteous, Julie and Marc Cavazza (2009). "Controlling Narrative Generation with Planning Trajectories: the Role of Constraints". In: *Interactive Storytelling*. Springer, pp. 234–245.
Prusinkiewicz, Aristid Lindenmayer Przemyslaw et al. (1990). *The Algorithmic Beauty of Plants*.
Random Map Generation in Diablo III (2012). http://us.battle.net/d3/en/forum/topic/6147286293. [Online; accessed 15-April-2015].
Saltsman, Adam (2009). *Canabalt*.
Shaker, Noor, Antonios Liapis, et al. (2015). "Constructive Generation Methods for Dungeons and Levels". In: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Ed. by Noor Shaker, Julian Togelius, and Mark J. Nelson. Springer.
Shaker, Noor, Miguel Nicolau, et al. (2012). "Evolving Levels for Super Mario Bros Using Grammatical Evolution". In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, pp. 304–311.
Shaker, Noor, Julian Togelius, et al. (2011). "The 2010 Mario AI Championship: Level Generation Track". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 3.4, pp. 332–347.
Smith, Gillian (2014). "Understanding Procedural Content Generation: a Design-Centric Analysis of the Role of PCG in Games". In: *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, pp. 917–926.
Smith, Gillian, Mike Treanor, et al. (2009). "Rhythm-Based Level Generation for 2D Platformers". In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM, pp. 175–182.
Smith, Gillian and Jim Whitehead (2010). "Analyzing the Expressive Range of a Level Generator". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, p. 4.
Solarus (2008). *Zelda Mystery of Solarus DX*.
Solarus: An ARPG game engine [sic]. http://www.solarus-games.org/. [Online; accessed 11-April-2015].
Steam (2014). *Daylight On Steam*. http://store.steampowered.com/app/230840/. [Online; accessed 9-December-2014].
Togelius, Julian, Alex J Champandard, et al. (2013). "Procedural Content Generation: Goals, Challenges and Actionable Steps." In: *Artificial and Computational Intelligence in Games* 6, pp. 61–75.
Togelius, Julian, Mike Preuss, and Georgios N Yannakakis (2010). "Towards Multiobjective Procedural Map Generation". In: *Proceedings of the 2010 workshop on procedural content generation in games*. ACM, p. 3.
Turtle Rock Studios, Valve Corporation (2008). *Left 4 Dead*.
Valls-Vargas, Josep, Santiago Ontanón, and Jichen Zhu (2013). "Towards Story-Based Content Generation: From Plot-Points to Maps". In: *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, pp. 1–8.
Willems, Sascha (2010). *Random Dungeon Generation*. http://www.saschawillems.de/?page_id=395. [Online; accessed 15-April-2015].

Williams, Nathan (2014). *An Investigation in Techniques used to Procedurally Generate Dungeon Structures*.

Yannakakis, Georgios N, Antonios Liapis, and Constantine Alexopoulos (2014). "Mixed-Initiative Cocreativity". In: *Proceedings of the 9th Conference on the Foundations of Digital Games*.

Yannakakis, Georgios N and Julian Togelius (2011). "Experience-Driven Procedural Content Generation". In: *Affective Computing, IEEE Transactions on* 2.3, pp. 147–161.

Young, R Michael (2007). "Story and Discourse: A Bipartite Model of Narrative Generation in Virtual Worlds". In: *Interaction Studies* 8.2, pp. 177–208.

Young, R Michael et al. (2004). "An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments". In: *Journal of Game Development* 1.1, pp. 51–70.

Zelda Mystery of Solarus DX. `http://www.solarus-games.org/games/zelda-mystery-of-solarus-dx/`. [Online; accessed 15-April-2015].

Zelda: Mystery of Solarus, en Archlinux.

Zelda Universe: Games. `http://www.zelda.com/universe/games/index.jsp`. [Online; accessed 18-April-2015].