Multi-Level Hybrid Procedural Dungeon Generation

Luke Wanless
u7120506@anu.edu.au
Australian National University
Canberra, Australia

Penny Sweetser
penny.kyburz@anu.edu.au
Australian National University
Canberra, Australia

Abstract

Procedural content generation (PCG) is the algorithmic creation of content. PCG can allow for rapid and diverse level design in dungeon style games. Previous research has been undertaken to partition the level generation process into two components: content generation and space generation. However, these methods have not yet been applied to generate playable maps across a multi-level dungeon game. This paper extends upon previous work by creating a framework that allows for multi-level dungeon-style game design. We achieve this by using context free grammars to generate multi-level content before translating this into a physical representation using a cellular automata inspired space generator. The concepts explored in this paper provide a method for start-to-finish map creation, a process which has the potential to rapidly reduce the time spent manually sequencing individually generated levels together.

CCS Concepts: • **Applied computing → Computer games**; • **Theory of computation → Grammars and context-free languages**.

Keywords: procedural content generation, dungeon game, PCG, context free grammar, game design, multi-level, roguelike game

1 Introduction

Procedural content generation (PCG) is the process of automating content creation either in part or in full using predefined algorithms. The goal of such work is to expedite the process of tedious level creation for game designers. The focus of this paper is on applying PCG to roguelike dungeon games. Roguelike is a term originating from the game Rogue (1980) and is used to classify a general class of games in which levels are usually randomly generated and character death is permanent. Other common features of roguelike games include grid-based level layouts that sequence together individual rooms.

PCG is a popular design tool for such games since levels are unique and randomly generated during gameplay. The level generation style explored in this paper features dungeon style levels similar to games such *The Legend of Zelda* [9] that use lock-key pairs to design challenging levels. More specifically the problem of multilevel generation is considered. We define multi-level generation as the process of generating multiple levels as a single atomic output. Thus, content can extend over the scope of multiple levels; for example, a key found on one level may be used to unlock a door on the next. The aim of such generation is to increase game cohesion and allow game developers to more easily control game flow between levels as they are no longer contextually independent. Secondly this framework can be extended to allow for individual level generation that considers elevation.

This paper builds upon pre-existing single level hybrid approaches that partition the generative process into content and space. More specifically Context Free Grammars (CFG) are used to output a mission description that can then be interpreted by a Cellular Automata (CA) inspired space generation algorithm. This paper extends upon previous work [3, 4, 7] in 2D single level procedural dungeon generation and considers an approach to generate a multilevel roguelike game.

The goal of this paper is to present a simple and generic multi-level dungeon generator. Additionally, it explores how this model could be adapted to generate single level games that consider elevation.

2 Related Work

2.1 Constructive approaches

Constructive approaches are the most common class of Procedural Dungeon Generation algorithms in game development [5]. Such approaches considers algorithms that generate content once. To safeguard against invalid content, such as unreachable rooms, rules must be followed that ensure no invalid content creation occurs. Popular games such as *The Binding of Isaac* [7] consider a constructive approach. The game layout is built upon a 2D grid of cells that represent the placement of rooms. Level generation originates from a genesis room that constructs a dungeon layout in a breadth first manner by sequentially iterating over a cells neighbours in all cardinal directions (Von Neumann neighbourhood).

2.2 Techniques

Various techniques exist for creating Dungeon generation algorithms [4]. Each technique has associated benefits and limitations, these are detailed below.

2.2.1 Evolutionary algorithms. Evolutionary algorithms are a class of algorithms that optimise through the process of selection from a candidate population using a fitness function. Valtchanov and Brown [12] use genetic algorithms to select partial levels that are stored in tree structures. During the generation process the algorithm attempts to place rooms in the order specified by the trees layout whereby children are directly connected to parents. Impossible placements lead to the pruning of the corresponding tree branch. The major drawback of the provided fitness function is that it doesn't encode any game-play oriented placement of rooms. A common drawback of evolutionary approaches in subjective domains such as level generation is the reliance on finding an appropriate fitness function [13].

2.2.2 Cellular Automata.

Cellular automata are discrete computational simulations that populate grids of cells using predefined rules. Using specifically chosen rules, CA can be used to model dungeon layouts. Previous work has leveraged the computational efficiency and inherent chaos of CA to create natural and infinite cave like maps [6]. Such approaches generate realistic content however suffer from a lack of parametrization and control that is needed for real game environments. In order to encode useful information about the underlying game, additional structure must be utilised. Pech et al [10] used an evolutionary algorithm to select CA seed layouts based on how interesting humans found the generated mazes. The evolutionary algorithm was able to increase the scores of the generated mazes to within 90% of a a predefined optimal solution, showing the efficacy of such hybrid solutions.

2.2.3 Grammars.

Grammars are logical structures that describe a language or set of possible orderings of terminal symbols. Grammars have been used to generate both content and space. Adams [8] uses Space grammars whereby terminal symbols are shapes used to construct spatial dungeon layouts. A search based algorithm is used to select the most optimal solution of each production rule of the grammar. The use of grammars as spatial construction tools is limited by the natural rigidity of the production rules and the fact these themselves must be hard coded. Dormans et al [3] used a graph grammar to create a directed graph that emulates the direction of the players actions through time. This abstract representation of the games objective is then parsed by a space grammar into a final dungeon layout. As noted by Linden et al [13] the consideration and subsequent reification of a mission creates a more meaningful spatial generation.

2.2.4 Machine Learning.

Machine learning (ML) refers to a broad category of statistical based algorithms that implicitly learn desirable and salient features using large datasets. Torado et al [11] applied Generative Adversarial Networks (GANs) to the task of generating *The Legend of Zelda* levels. The fact that 42% of the generated levels were unplayable highlights that statistically based generation methods fail to replicate the underlying logic of games. Furthermore the reliance of ML methods on large datasets reduces its practicality for real world development and new types of games.

2.2.5 Hybrid techniques.

Hybrid solutions pluralize the aforementioned techniques in the generation process. The guiding principle of such approaches is to utilise the relative strengths of different approaches and localise these to the relevant parts of the generative process. This has the additional benefit of mitigating the limitations of various techniques.

In Dormans [3] and Gellel [4] both adopt a hybrid approach in order to effectively partition the dungeon generation into a mission generator and space generator. Dormans [3] suggests that well designed levels generally have two structures: a mission and a space.

In this paper we consider a hybrid approach that uses Cellular Automata inspired behaviour for space generation and Context Free Grammars for mission description generation.

3 Design

The approach taken in this paper utilises the hybrid approach discussed in Gellel [4] and Dormans [3] whereby a mission description is generated, before being translated into a physical space. Procedural dungeon generation naturally lends itself to this type of generation partitioning. Here the specific logic of a game can be handled solely by the mission generator which can be made as rigid as desired. Further, this removes any problems related to the stochastic nature of the space generator in terms of placing invalid content.

The dungeon generation style consists of a 2D grid layout spanning across multiple levels.

3.1 Context Free Grammars

Mission descriptions are generated using context free grammars (CFG). A context free grammar is defined as a set of terminal and non-terminal symbols and set of production rules. Non-terminal symbols represented by upper-case words are those that have an associating production rule; in other words they are not *terminal* or final. Conversely, terminal symbols are lower cased and have no associating production rule; these typically represent physical non-absract entities such as an enemy or key. The language of a CFG consists of all possible strings that can be generated.

The advantage of using CFG's to generate mission strings is we can describe a language that consists only of playable mission descriptions. We define playable mission strings as those that can be traversed from left to right with all keys appearing before the corresponding locks; in this way we can ensure no content becomes inaccessible. Below is an altered version of the CFG presented in Gellel [4] and is used in this papers implementation:

1. Dungeon → start + room + Content + room + end
2. Content → Content key Content lock Content
3. Content → room Content
4. Content → enemy room
5. Content → level
6. Content → Content Content

Here the terminal symbols can be interchanged with specific content as required by the game, however for this implementation we focus in on lock-key pairs as they naturally require correct ordering on behalf of the content generator. To extend the generation over multiple levels, the *level* terminal symbol was introduced. Using production rule 1 it becomes evident that this allows for dependent content such as lock-key pairings to span across levels.

3.2 Cellular Automata

A Cellular Automata (CA) inspired approach is used in the space generating process. As discused in Gellel et al [4] and Dormans [3], cellular automata like behaviour maps nicely onto grid layout generation. More specifically (CA) inspired behaviour is used to describe the process of using some kind of neighbourhood function to determine the existence of a cell based on its immediate surroundings.

The space generation algorithm is inspired by the constructive approach used in *The Binding of Isaac* [7]. The dungeon generation in this paper is built upon a (30,30) 2D grid. Rooms occupy individual cells and adjacent rooms can be accessed in all 4 directions unless a room is classified as locked. Because all rooms can be accessed from all directions unless locked, we prevent loops and shortcuts by enforcing that a cell can have at most one neighbour before being created. To offset the fact that such dungeons would resemble a snaking corridor without branching, breadth first search is used to create more natural and explore-able dungeons.

This quality of a dungeons sub-branching shall be referred to as *linearity*. High linearity refers to corridor like mazes with minimal branching away from the critical path. The critical path is the minimal distance path in order to complete a level. To control the linearity of generated dungeons we have implemented a *convergence factor*. The set of convergence factors CF is formally defined as

$$CF = \{cf \in \mathbb{R} \mid 0 \leq cf \leq 1\}$$

and controls the likelihood of a particular branch surviving.

The breadth first searching algorithm generates outwards from a genesis room placed in the middle of the grid of the first level. From here each neighbouring cell in the four cardinal directions (also know as the Von Neumann neighbourhood) is used to try and place the next room. The CA inspired rules when attempting to place a room are as follows:

1. If the room is already occupied return false
2. If the cell has > 1 neighbour (Von Neumann neighbourhood) return false
3. If the number of rooms in this level has already been reached return false
4. If a randomly generated number in the range $[0, 1]$ is less than the convergence factor return false
5. Otherwise place the room

To fully appreciate how a the convergence factor is used to control the linearity of a dungeon it is necessary to first understand the *retry mechanism*. If the search algorithm has been unsuccessful at placing a new room, we retry each cell once more before giving up and restarting. This follows from the observation that the reason for abandoning a cell may have been due to the convergence factor and not because such a placement was impossible.

Hence by using a retry mechanism in conjunction with a high convergence factor we can reduce the likelihood that multiple divergent paths will be generated. This is because a high convergence factor reduces the likelihood that multiple paths will survive, but a retry mechanism doubles the chance that at least one path will survive. To ensure the chances of a successful room placement is not doubled in each direction, we use lazy evaluation such that only the first successful room is generated upon invoking the retry mechanism.

3.3 Implementation

3.4 Mission Generator

The mission generator used in this paper is a modified version of that created by Gellel et al [4]. It is implemented in python and based on code from [1].

3.5 Space Generator

The space generator was implemented in the open source JavaScript library p5.js. The algorithm was modelled from the breadth first search used in *The Binding of Isaac* [7] and [2]. A link to a usable online version of this papers implementation can be found in the appendix.

4 Evaluation

4.1 Mission Generator

Upon adding production rule 2 (see 3.1), the mission generator was tested using the same method as in [4]. As stated in the aforementioned work, testing the quality of mission strings is difficult. The validity of the grammar has been shown to produce correctly ordered lock-key pairs such that all strings describe playable levels. The selection heuristic was modified in order to favour dungeons with more even room distributions across levels. The effect on the generators runtime is shown below.

Table 1. Average mission generator runtimes (seconds)

100 Missions	1000 Missions	10000 Missions
6.63 x 10^{-3}	4.14 x 10^{-2}	3.58 10^{-1}

4.2 Space Generator

The quality of a generate and test algorithm can be measured by the number of candidate solutions it generates before finding a correct solution. Further, we can also analyse the quality of these solutions. For a quantitative analysis of the space generator, the number of generations that are rejected is measured for a variety of levels and level sizes.

Table 2. Average number of restarts without retry mechanism (100 trials)

Levels	Rooms/level	Average no. restarts
1	5	0.50
	10	1.12
	15	4.17
2	5	1.37
	10	6.83
	15	30.70
3	5	2.82
	10	25.22
	15	171.40
4	5	5.02
	10	27.32
	15	136.90

Shown in Table 2, we can see that the complexity of the naive approach increases drastically as the number of rooms per level and the number of levels increase. This is because the probability of a restart occurring is the product of each levels probability as the events are not disjoint; each room relies on the successful completion of the one before it. By using a retry mechanism we significantly lower the probability of each level failing.

As seen in Table 3, after implementing the retry heuristic the number of restarts dropped significantly.

Table 3. Average number of restarts with restart mechanism (100 trials)

Levels	Rooms/level	Average no. restarts
1	5	0.07
	10	0.20
	15	0.70
2	5	0.19
	10	0.65
	15	1.30
3	5	0.18
	10	1.33
	15	2.85
4	5	0.45
	10	1.56
	15	4.45

For a qualitative representation of the space generator we can consider how singular levels differ in appearance for different convergence factors. Figure 1 and 2 compare two different space generations for the same mission string generated with different convergence factors.

Legend: Green: Start room, White: Normal room, Dark-blue: Lock room, Red: Key room, Light-blue: Enemy room, Yellow: End room.

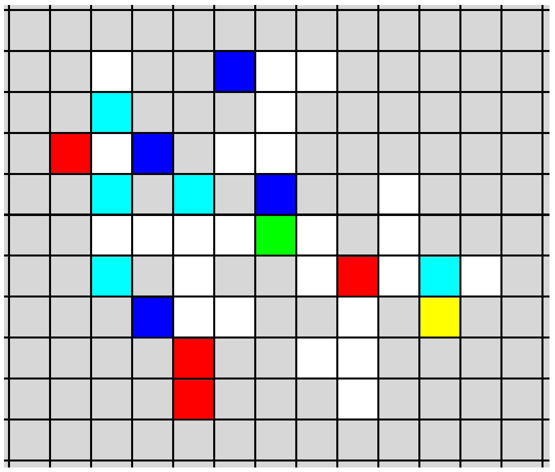

Figure 1. Convergence factor = 0.2. Notice how non-linear the level is, and how easily most of the mission content can be bypassed.

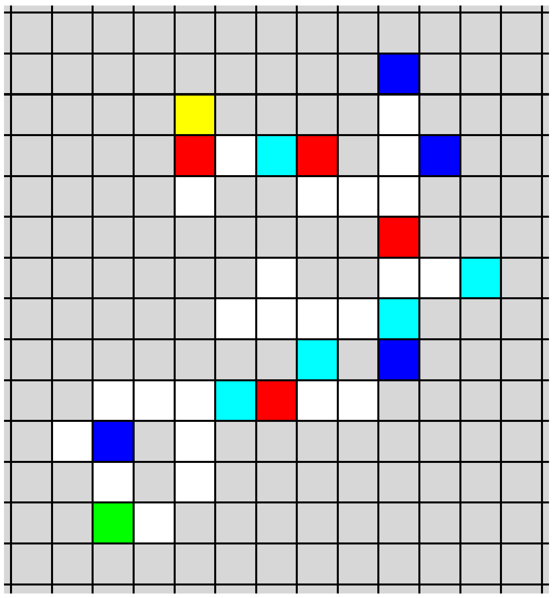

Figure 2. Convergence factor = 0.7. Notice how much more linear the level is and how it cannot be completed without using all of the available mission content.

5 Discussion

It is apparent that a relationship exists between desirable linearity and the frequency of degenerate layouts. Desirable linearity can be seen in figure 2 whereby it allows for exploration yet also makes total use of content provided by the mission string. As shown in figure 1, degenerate levels involve those in which a players path to the next level or end of the game is not obstructed by locks. This is a consequence of using a low convergence factor. Due to a high risk of survival for each branch, low convergence factors lead to levels that generate in multiple directions and can thus result in redundant lock placements.

A retry heuristic proved sufficient at mitigating latency increases and allowing for more linear levels when using higher convergence factors (> 0.55). To consider why the retry heuristic proved so effective at reducing the number of restarts, consider the binomial probability of a single room failing to generate a successor with the restart mechanism. For this we use the assumption that the square being generated has only the previous square as a neighbour and can thus generate in 3 directions. As the restart mechanism is disjoint from the initial trial, we sum the probabilities. This leads to a doubling of the probability that a room will create at least 1 successor. This probability with a convergence factor of 0.6 is given by,

$$P(Success) = P(1-Failure) = 1-\binom{6}{6}\cdot(0.6)^6\cdot(0.4)^0 = 0.953$$

Hence the restart mechanism ensures abandonment of potentially valid room placements decreases from 47.7% down to just 4.7%.

5.1 Multi-elevation levels

The way that the space generator currently outputs levels naturally lends itself to the generation of templates that can represent elevations slices of a single level. Here the distinction of self-contained levels within a multi-level game is a man-made constraint. Tweaking the grammar to produce End rooms on random levels would create this effect. Further modifications would be required to model and prevent degenerate level generations as the path to the end goal would no longer be linear. This remains outside the scope of the presented algorithm.

5.2 Limitations and Future Work

Future work must be undertaken to design an appropriate heuristic that can appropriately model the enjoyment and difficulty of procedurally generated levels. Previous solutions used in Gellel et al [4] use a path difference heuristic to define a levels quality on how much exploration it requires for a player to complete the level. Potential solutions could penalise each time an object from the mission string becomes redundant in completing the level.

Another addition that would make this prototype more useful for real world application would be a difficulty heuristic. This would allow game developers to control the flow and difficulty gradient seen in each level.

Modifications to the space generation algorithm are required to ensure that the placement of content doesn't degenerate into cases whereby the placement of locks has no effect on the players ability to access staircases and end rooms.

Lastly to take full advantage of the variable level sizes produced by the mission generator, modifications to the space generator must me made to allow for variable grid sizes. Currently the grid size is fixed to $(30, 30)$ making individual levels with > 20 rooms extremely time consuming to generate due to the number of times the algorithm must restart after running into an edge of the map. Here a space generator that can infer a variable grid size from the number of rooms to be generated would make for a meaningful extension.

6 Conclusion

In this paper we have built upon previous hybrid procedural content generation techniques by constructing a simple prototype multi-level dungeon generator for roguelike games. This was achieved by partitioning the generative process into a mission generator and space generator. Here we utilise the strictness and reliability of context free grammars to correctly encode the logic of lock-key dungeon games. We then translate this game encoding into a physical map through a cellular automata inspired approach. This processes has been extended to the domain of multilevel games through the use of a retry mechanism to ensure a linear time complexity which is needed for real gaming environments. Finally we explored how such a model could be extended to generate single level content that can force user exploration over multiple levels of elevation. The use of such algorithms in roguelike dungeon games not only saves game developers time but provides infinitely novel content for game players. Specifically the atomic generation of multilevel games can reduce the time spent stitching sequential levels together whilst allowing for content to span multiple levels.

References

[1] Eli Bendersky. 2010. Generating random sentences from a context free grammar. (2010). https://eli.thegreenplace.net/2010/01/28/generating-random-sentences-from-a-context-free-grammar

[2] Boris. 2020. Dungeon Generation in Binding of Isaac. (2020). https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/

[3] Joris Dormans. 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* (Monterey, California) *(PCGames '10)*. Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages. https://doi.org/10.1145/1814256.1814257

[4] Alexander Gellel and Penny Sweetser. 2020. A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels. In *International Conference on the Foundations of Digital Games* (Bugibba, Malta)

(FDG '20). Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/3402942.3402945

[5] Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. 2019. Two-step Constructive Approaches for Dungeon Generation. arXiv:1906.04660 [cs.AI]

[6] Lawrence Johnson, Georgios Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. (09 2010). https://doi.org/10.1145/1814256.1814266

[7] Edmund McMillen. 2011. The Binding of Isaac.

[8] Michael Mendler and David Adams. 2002. Automatic Generation of Dungeons for Computer Games.

[9] Nintendo. 1986. The Legend of Zelda.

[10] Andrew Pech, Philip Hingston, Martin Masek, and Chiou Lam. 2015. Evolving Cellular Automata for Maze Generation, Vol. 8955. 112–124. https://doi.org/10.1007/978-3-319-14803-8_9

[11] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2019. Bootstrapping Conditional GANs for Video Game Level Generation. *CoRR* abs/1910.01603 (2019). arXiv:1910.01603 http://arxiv.org/abs/1910.01603

[12] Valtchan Valtchanov and Joseph Brown. 2012. Evolving dungeon crawler levels with relative placement. (06 2012). https://doi.org/10.1145/2347583.2347587

[13] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. 2014. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 1 (2014), 78–89. https://doi.org/10.1109/TCIAIG.2013.2290371

A Repository Link

https://github.com/lukewanless/PCG_Project