Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Hierarchical WaveFunction Collapse

Michael Beukman*, Branden Ingram*, Ireton Liu, Benjamin Rosman
University of the Witwatersrand, Johannesburg
michael.beukman1@students.wits.ac.za, branden.ingram@wits.ac.za,
ireton.liu2@students.wits.ac.za, benjamin.rosman1@wits.ac.za

Abstract

Video game developers are increasingly utilising procedural content generation (PCG) techniques in order to generate more content far quicker than if it were designed. Although promising, much of the successful work to date has been achieved in simple 2D environments or has required significant hand-designed effort. This is due to the difficult nature of defining plausible metrics, fitness functions or reward functions which can quantify the quality of generated levels. Our work aims to avoid this difficulty by utilising minimal human design to build up constraints, and generating diverse levels that maintain these constraints. We achieve this by hierarchically applying the recent WaveFunction collapse (WFC) algorithm. Our approach allows designers to specify larger-scale components, and additional constraints that are difficult to enforce using standard WFC. We empirically demonstrate that our approach does indeed incorporate these higher-level structures, and is more controllable than our baselines. Despite these benefits, our levels do not suffer from a lack of diversity. Finally, we illustrate the scalability and flexibility of our approach by applying it to both 2D and 3D domains.

Introduction

Procedural content generation (PCG) has become increasingly important in the field of game development due to its ability to efficiently create complex and varied content at a fraction of the cost of manual development (Hendrikx et al. 2013; Smith 2017; Korn et al. 2017). WaveFunction collapse (WFC) is a new, powerful technique used for generating levels in games (Gumin 2016; Karth and Smith 2022). This approach has been shown to generate impressive results similar to that of a designer due to its emphasis on enforcing designed constraints within the generated levels (Karth and Smith 2022; Stålberg 2022). Furthermore, these constraints can be specified naturally by providing an example level, as opposed to needing to manually define each constraint (Gumin 2016). However, this technique has some limitations (Cheng, Han, and Fei 2020). In particular, the generated levels often have a lack of structure, due to constraints being mostly local. Secondly, while a designer can specify constraints in terms of an example level, controlling the occurrence of certain elements, it is challenging to enforce more complex constraints, such as having exactly one of a certain structure in the generated level.

To address these limitations, we propose an extension to WFC that leverages hierarchical elements, which can provide more structure to the generated levels. In particular, our approach performs several steps, where the first steps add high-level structure to the level in the form of large elements, and the later steps fill in details around this structure. Each step performs the normal WFC algorithm, with a different set of rules and constraints to, in the end, generate a structured, but interesting level. By introducing hierarchies, our method allows for a more modular approach to level design, making it easier to specify desired features and generate high-level structures. We demonstrate the benefits of our approach through a series of experiments in different games, showing that our method can generate structured and diverse content that meets specific design goals while maintaining the advantages of WFC. In particular, our approach allows for more control over the generation, while not compromising on the diversity of content. Furthermore, by leveraging hierarchy and larger-scale components, our method can generate more complex levels than a non-hierarchical baseline.[1]

WaveFunction Collapse

The WFC algorithm is effectively a constraint-solving method that can generate tilemap levels, with several distinct steps (Karth and Smith 2022). A tile is the lowest-level element we consider, where a tile is a single pixel in an image, or a single block in a level. We next consider *patterns*, which are small, rectangular grids of tiles (e.g., a 2×2 or 3×3 window). To execute the WFC algorithm, we require constraints that specify which patterns can be placed next to which other patterns. There are several ways to obtain the patterns and constraints, and they can also be designed manually. A common technique, however, leverages an example level and extracts patterns and adjacency constraints from this example (Gumin 2016). Given an example level of size $N \times N$, we consider overlapping sliding windows of certain sizes (such as 2×2 or 3×3), obtaining a set of patterns. This process is shown in Fig. 1. We now also have our constraints: pattern a can be placed adjacent to pattern b if there

*These authors contributed equally.

[1] See https://github.com/Michael-Beukman/HWFC for code.

is at least one row or column of overlap between a and b.

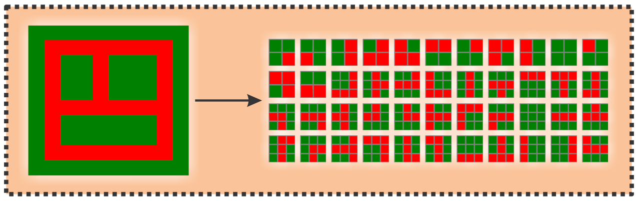


Figure 1: Illustrating the set of patterns (right) that were extracted from an example level (left).

The rest of the WFC process is shown in Fig. 2. We consider the *map* to be a 2- or 3-dimensional matrix of *superpositions*. A superposition is simply a set of tiles, indicating which possible tiles can be placed at that particular location in the map. The map is initialised as a matrix where each element is a superposition of all possible tiles, i.e., each cell in the map is $c = \{T_1, T_2, \ldots, T_k\}$, if we have k tiles. This represents that, initially, each cell can be filled by any tile.

We now perform the following three steps until the level is completely generated. First, we **Select** a tile to collapse (far left in Fig. 2). We do this using the concept of *entropy*, where the entropy of an $n \times n$ window is proportional to the number of different tile options available in this window. In essence, it is the sum of the number of tiles in each cell's superposition $\sum_i |c_i|$. We choose the location with the lowest entropy to reduce the chance of obtaining a state where no more tiles can be placed, but the level is not yet finished—a contradiction. Once we have selected a tile, say (i, j), we find the patterns that are *compatible* with the current state of the map around that location. We do this by considering the window with its top-left corner at location (i, j). A pattern is then compatible with the current state of the map if, for each corresponding tile, the superposition in the map contains the tile in the pattern at that location. If there are multiple compatible patterns, we select randomly, weighted by how often each pattern occurred in the example level.

Once a pattern is selected, we perform the **Collapse** step. This involves taking the current region, and replacing each cell's superposition with a single tile, that of the currently chosen pattern. This then reduces the number of active superpositions and collapses the region into a concrete pattern. The final step is **Propagation**. This considers the new state of the map, and updates the superpositions of surrounding cells to take the new information into account. In particular, we consider all compatible patterns that partially overlap with the recently-collapsed region. The superpositions of the surrounding tiles are then updated to include only tiles from these patterns. This process is repeated in a breadth-first manner until all superpositions have been updated. WFC progresses by repeating these last three steps, selection, collapse and propagation until the entire map is collapsed, at which point the level is completely generated.

Finally, WFC runs the risk of ending up in an unsolvable state, where some tiles have empty superpositions. This indicates a contradiction, which means that the generation process cannot progress. There are multiple ways to address this. One way is to simply restart the generation process, hoping that it does not lead to another contradiction (Gumin 2016; Karth and Smith 2022). A more principled approach involves backtracking, where the recent steps are reversed, and a new pattern is chosen (Karth and Smith 2022).

Related Work

In recent years, PCG has gained significant attention from both the academic (Dormans 2010; Dormans and Leijnen 2013; Summerville et al. 2018) and commercial (Adams and Adams 2006; Yu 2008; Dormans 2017; Ludomotion 2017; Adams 2019) communities due to its potential to reduce development costs and increase the replayability and diversity of games. There are many approaches to PCG, such as machine learning (Summerville et al. 2018; Liu et al. 2021), evolutionary algorithms (Togelius et al. 2011; Earle et al. 2022; Beukman, Cleghorn, and James 2022) and grammars (Dormans 2010; Dormans and Leijnen 2013).

General PCG

Search-based PCG methods use search algorithms to generate content that meets specific functional criteria, for example, solvability (Togelius et al. 2011). However, the search process can be computationally expensive, especially for complex content or large search spaces.

A less computationally expensive approach involves the use of constraint-based methods. This approach entails representing design rules and preferences as constraints and applying constraint solvers to search for solutions (i.e., content) that satisfy the specified constraints (Van Der Linden, Lopes, and Bidarra 2013). However, designing these constraints can be unintuitive in complex domains. This shortfall can also be observed in grammar-based methods, where a grammar specifying the structure and properties of the generated content has to be defined (Shaker et al. 2011, 2012).

Recently, there has been a shift towards PCG via Reinforcement Learning, where an agent is trained to sequentially place tiles to maximise a particular reward (Khalifa et al. 2020; Earle et al. 2021; Jiang et al. 2022). Nevertheless, devising a comprehensive objective or reward function that encourages the creation of intricate and functional structures while remaining amenable to optimisation may be an unintuitive task (Togelius et al. 2013). An alternative method that eliminates the need for an objective function involves leveraging a large dataset of pre-existing content (which may not be available for many games) to train machine learning models, enabling the generation of novel content (Summerville et al. 2018; Liu et al. 2021; Shu, Liu, and Yannakakis 2021).

WaveFunction Collapse

More recently, a technique called WaveFunction collapse (WFC) has gained popularity in the realm of PCG, for the visually aesthetic quality of the generated content (Karth and Smith 2022), and has seen use in two published games, Bad North (Concept 2022) and Townscaper (Stålberg 2022). Initially proposed by Gumin (2016), WFC looks to decompose input examples into smaller blocks while identifying the constraints needed to compose them. This approach then

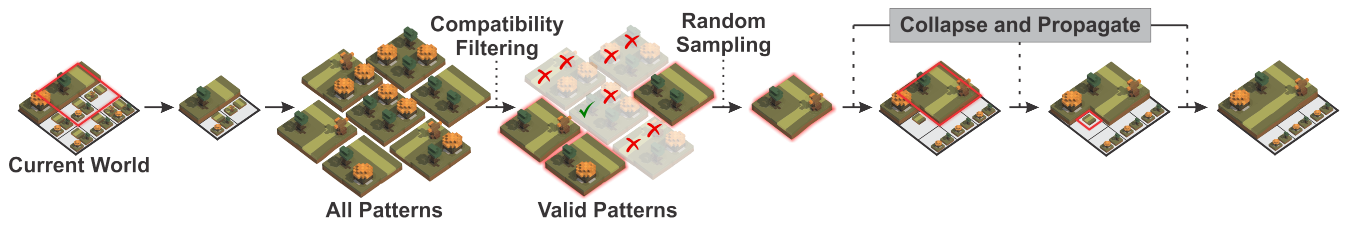


Figure 2: The overall WFC process. First, we select a region to collapse, and find all of the compatible patterns that could be placed at that location. We then randomly sample a pattern, and collapse the previously selected region to that pattern. Then, during propagation, we update the superpositions of the surrounding regions to take this new information into account.

allows for the construction of large outputs from these small blocks based on the identified constraints. Additionally, Kim et al. (2019) expanded WFC to a graph-based domain which allowed for greater variety in content at the cost of intuitive controllability. To address the lack of controllability, Langendam and Bidarra (2022) developed a mixed-initiative approach to WFC, where the user can influence the generation process, thereby changing the final output. To a similar end, Sandhu, Chen, and McCoy (2019) propose dynamically adjusting tile weights—which govern how often tiles are selected—to potentially obtain more satisfactory results. Both these extensions focus on better playability, designer control and increased similarity to that of a human designer.

Compositional PCG

Much work has also been done in compositional PCG, where multiple methods are composed together to generate large and complex structures (Togelius, Justinussen, and Hartzen 2012). Snodgrass and Ontanon (2015) use hierarchy to generate and combine structures at different scales. Given a set of example levels, their method extracts high- and low-level patterns from these. They then fit Markov chains to this data, and generate levels by first placing the high-level structure, and filling in the low-level details.

Beukman et al. (2023) also decompose levels into several layers, where each layer generates an abstract level, to be completed by the lower-level generators. While this approach allowed the designer to specify the hierarchical structure, it required fitness functions to be specified for each generator, which may be cumbersome to design.

Another application of hierarchy is seen in generating different aspects of a level separately. In particular, Dormans (2010) first generates the high-level mission, followed by the physical layout of the level based on the mission. A similar approach has been employed in the commercial game *Unexplored* (Ludomotion 2017), where an abstract graph is transformed into a concrete grid-based level layout.

Combining the idea of a hierarchy with WFC, Alaka and Bidarra (2023) segment the map into abstract, meta-tiles that dictate which low-level tiles can be placed in each region. For instance, they group together several concrete tiles all under the abstract tile "road"—allowing a designer to specify that a particular area must be covered by roads, without needing to focus on the details of which exact tiles should be used. This method focuses more on the mixed-initiative setting, where a designer can sketch out a level layout in abstract, high-level terms. WFC can then use this sketch to replace the abstract tiles with concrete patterns. We instead add structure to WFC itself through additional designed elements, which are organised hierarchically.

Methodology

Our method uses both hierarchical and non-hierarchical expert examples to generate structured video game levels. This process, outlined in Fig. 3, consists of two core components: constraint discovery and our hierarchical variation of WFC.

Constraint Discovery

In order to generate structured levels, we extract a set of patterns from human-designed examples. These patterns represent the lowest-level features which can be present in our generated levels. This set is obtained by passing a rolling window across the entire example while tracking unique occurrences (patterns) along with their frequency. By using these patterns and how often they occur in the level, we are able to ensure that our generated level is constructed from the same patterns as the example, with a similar frequency. This approach is similar to that performed in standard WFC (Gumin 2016; Karth and Smith 2022).

Besides the example levels which we use to obtain low-level patterns, we additionally require the designer to specify hierarchical elements. These are generally larger patterns that represent a single component that can be placed in a level. These components can be high-level, in which case the patterns contain *superpositions*, i.e., some tiles are not filled in by the designer, and must be populated by the algorithm. Medium-level patterns do not contain superposition tiles, and can be placed within these high-level structures.

Hierarchical WaveFunction Collapse

Given the hierarchical patterns, as well as the example level, our method (detailed in Algorithm 1) generates levels as follows: We start with a map completely populated with superposition tiles. Then we perform several passes of WFC (line 6), with each pass using a different set of patterns. At each stage, we also have separate termination conditions (line 7); for instance, specifying that k elements at a particular level of a hierarchy must be placed, or that $x\%$ of the level must be filled up from elements at that hierarchy level. Being able to specify constraints in this way gives the designer immense control over the final generated level, which is a beneficial property in PCG (Sorenson, Pasquier, and DiPaola 2011).

Algorithm 1: HWFC

```
1:  procedure HWFC(Patterns, Map, $i$, Term)
2:      if $i$ is lowest level then
3:          return Map
4:      end if
5:      repeat
6:          Perform WFC using Patterns[i] on Map
7:      until Term[i](Map)
8:      Replace Unfilled tiles with a full superposition
9:      Propagate(Map, Patterns[i])
10:     return HWFC(Patterns, Map, $i + 1$, Term)
11: end procedure
```

1. We first perform a pass using only the high-level patterns.
2. We next perform a pass using the medium-level patterns; however, these patterns are only allowed where the superpositions were placed by the first step (i.e., *within* the top-level hierarchies).[2]
3. Finally, we generate the remainder of the level using the low-level patterns.

In general, we can have multiple levels of hierarchy that will each be executed in sequence; for instance, two high-level steps, followed by three medium-level ones, etc.

Additionally, while we use only the corresponding patterns at each level when collapsing, we use an expanded set of patterns when performing the propagation step. In particular, at level i of the hierarchy, we consider all patterns from levels i to the lowest-level patterns during propagation. This ensures that the algorithm does not fail due to a contradiction if a lower-level pattern can satisfy all constraints. Finally, we also allow the designer to specify the maximum number of components of each type that can be placed. Once a pattern's number of occurrences reaches this limit, we remove it from the set of compatible patterns.

To implement our method in practice, we introduce a new tile denoted as `Unfilled` (see the far left pink tile in Fig. 5). When designing hierarchical elements, the designer can place these tiles to indicate areas that must be populated by the lower-level hierarchies. During generation, when these tiles are placed, the map considers it as collapsed at that location. In between each hierarchical level, however, we reset these tiles as superpositions (line 8), which can then be filled by the next pass of WFC. We also perform a full propagation step here (line 9), to ensure these superpositions are compatible with the already-placed structures within the level. In essence, the termination conditions govern *when* we move to the next hierarchical level, whereas the `Unfilled` tiles indicate *where* the next level's elements must be placed. To ensure that the medium-level patterns are only placed within the high-level components, at each high-level hierarchical step, all tiles that remain uncollapsed are marked as invalid positions for the next hierarchical level.

[2]While we have opted to restrict medium-level patterns to be contained within high-level patterns, designers are entirely at liberty to modify or remove this constraint.

In sum, HWFC differs from standard WFC in two notable ways:

1. While WFC would place patterns of the same scale throughout the execution, our method expands upon this by incorporating patterns of multiple different scales.
2. Instead of limiting ourselves to a single completed iteration, we recursively perform multiple iterations of the WFC algorithm at different scales until completion.

Experiments

This section details the experiments we perform to empirically demonstrate the effectiveness of HWFC. We first discuss the games we consider, and then describe the baselines we compare against. Finally, we detail the metrics we use.

Environments

We evaluate our method across three different tile-based domains, each with its own unique characteristics. We first consider a 2D binary Maze game. We next examine a much more complex 2D game, with 38 different tiles. We finally investigate Minecraft, a more complex 3D game.

Maze A Maze level is modelled as a 2D grid of cells, each being either occupied or unoccupied. The simplicity of this domain makes it ideal for efficiently benchmarking our system. Fig. 1 depicts the example level and low-level patterns, while Fig. 4 shows the hierarchical components.

Roguelike Each level is also a 2D grid of cells, but each cell can exist in one of 38 distinct states. This domain is a more complex version of the Maze, and is similar to games used in prior works, such as *The Legend of Zelda* (Khalifa et al. 2020). The tiles, hierarchical patterns and example levels we use are illustrated in Figs. 5 to 7, respectively.

Minecraft Minecraft provides a useful domain for PCG due to its open-world sandbox environment. Additionally, Minecraft's tile-based nature makes it ideal for expanding our system to a more complex and challenging 3D setting.

Baselines

We consider two methods as our baselines. The first is the standard WFC algorithm, which extracts 2×2 and 3×3 patterns from the examples. It also extracts patterns from the hierarchical designs, but does not use them otherwise. The second baseline is Multiscale WFC (MWFC), which is exactly the same as WFC, except that it also uses the hierarchical examples directly as patterns, which it can place as normal. In this way, we can measure how much hierarchy improves performance compared to default WFC. We can also compare the effect of having multiple stages—where we first place the large hierarchies, and then the next levels within this predefined structure—against naïvely using the large patterns within the unmodified WFC framework.

Metrics

We use four metrics to evaluate the performance of HWFC.

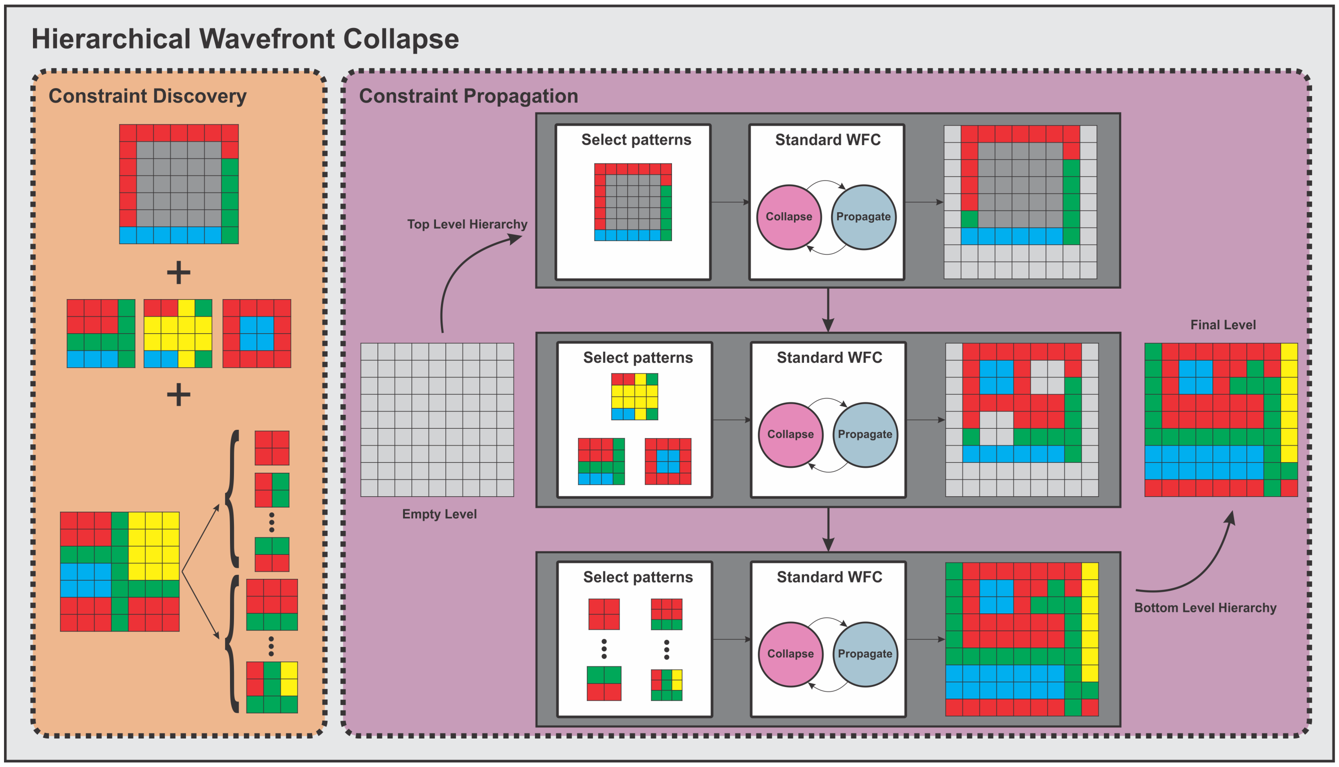


Figure 3: Overall model for HWFC. Here, we demonstrate the process of generating a full level with 3 levels of hierarchy. During the constraint discovery phase, we hand-design the high- and medium-level patterns, and extract all the unique low-level patterns from a hand-designed example level. High-level patterns contain superposition tiles (indicated by the dark grey squares). During the constraint propagation phase, we modify a single starting world by sequentially performing WFC for each hierarchical level. The high- and medium-level hierarchies terminate without completing the world; this current world state is then used as the starting point for the next hierarchical level. The generation process is completed once WFC has terminated in the lowest level of the hierarchy.

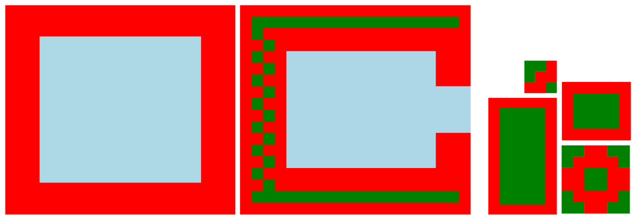


Figure 4: The designed hierarchy patterns for Maze. Blue tiles represent superpositions within the high-level patterns; red indicates wall and green indicates open space.

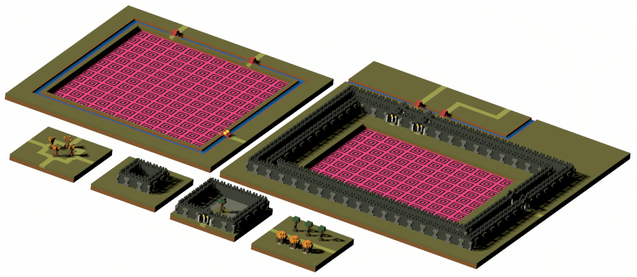


Figure 6: The designed hierarchy patterns for Rogue.

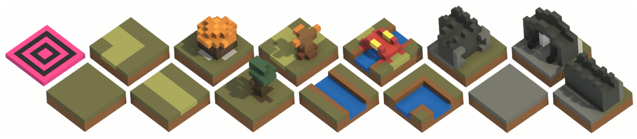


Figure 5: The tiles in Rogue (excluding rotations, which we also use). The pink tile on the left represents a superposition.

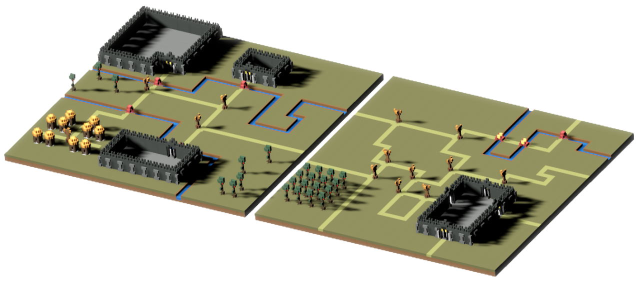


Figure 7: The Rogue example levels.

Diversity First, we consider the diversity of the generated levels, as measured by the Tile-Pattern KL divergence (Lucas and Volz 2019). This metric computes the KL-Divergence between the probability distributions of $n \times n$ tile patterns in a level. Concretely, given a level, we construct a probability distribution over all possible $n \times n$ patterns (where n is fixed, e.g. $n = 3$). The distance between two levels, then, is $d(A, B) = \frac{KL(A||B)+KL(B||A)}{2}$. Given K levels, we compute the distance between each pair of levels; and average these distances to obtain an overall diversity score for a particular method. We also consider another metric, the Hamming distance. This is simply the number of corresponding tiles that are different between two levels. Over K levels, we again average the pairwise diversity. Overall, a meaningful level of diversity is required to ensure that a generator is useful, i.e., generating content that is unique and interesting enough to maintain engagement.

Action Variance To more precisely narrow down the diversity of our levels, we consider another metric, the *action variance*, which measures the variance in patterns that were selected and placed in the level. The intuition of this is that a high variance is undesirable, as it indicates that the method places some patterns very often, and others very seldomly. A lower variance is preferred, as it indicates that the method does not focus on only a small set of patterns.

We calculate this as follows: We find the number of times each pattern was selected over K levels. We then multiply the frequency by the number of tiles in the pattern, which accounts for larger patterns taking up more space in the level. We finally compute the standard deviation of these values.

Controllability This metric calculates how many of the medium-level hierarchies were generated inside the top-level hierarchical components. This measures the degree to which a designer can control where the medium-level structures are placed.

Structural Complexity Here we measure the number of large hierarchical elements that were generated in the levels. We further measure the number of subpatterns of these hierarchies in each level, to determine if some methods skew more towards generating small parts of the hierarchies, compared to generating the entire structure outright.

Experimental Setup

Our experimental setup is as follows. To ensure fair comparisons, each method has access to the same examples. In particular, all methods use the same base example to extract patterns from. HWFC and MWFC use the hierarchical components as patterns, whereas base WFC simply extracts patterns from these components—ignoring patterns that contain superposition tiles—and uses them alongside the base patterns. We consider 2×2 and 3×3 patterns. For MWFC, since the interior of the high-level components may be filled with superpositions (in the form of `Unfilled` tiles), once the algorithm has terminated, we replace all `Unfilled` tiles with superpositions and then run the algorithm again, until the entire level is filled. Finally, since the edge of the levels often causes problems for the constraint-solving WFC method, we terminate the generation once all of the non-boundary tiles are collapsed, and then simply crop out these boundary rows and columns. We generate 100 levels from each method and use these to calculate the abovementioned metrics.

Results and Discussion

This section presents our results. We first examine the quantitative results using the metrics defined above and then provide a qualitative demonstration of HWFC's effectiveness.

Diversity

Fig. 8a measures how diverse a set of levels from the same generator is. Most methods perform similarly as the KL-Divergence tile size increases, indicating that the levels are similarly diverse. Fig. 8b measures diversity using the Hamming distance, and here HWFC has slightly less diversity than the other levels, but the large standard deviation indicates that each method's diversity is similar.

Next, we consider the *action variance*, shown in Fig. 9. This plot indicates that MWFC has a much higher action variance, indicating that it frequently places the same pattern, which leads to repetitive-looking levels. This occurs most often in the medium-level hierarchy, but also happens generally over the entire generation process.

Controllability

Fig. 10 illustrates the fraction of the medium-level hierarchical elements that were placed inside vs. outside the top-level hierarchies. HWFC is able to place a much larger fraction of these elements inside the top-level hierarchies, compared to MWFC. This demonstrates that our method can effectively use the hierarchical components to generate more controllable levels that better match the desired structure. The reason why HWFC does not generate all of its medium-level components inside the top-level hierarchies is the same reason why normal WFC has a non-zero (albeit small) number of medium-level hierarchies: Some of these hierarchical elements are quite simple, and were generated by normal WFC directly, instead of being placed as a single pattern.

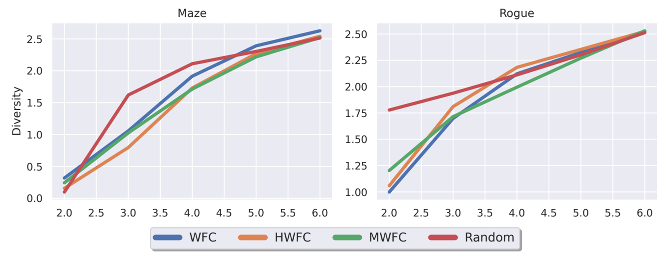
Structural Complexity

Fig. 11 contains the structural complexity results. In the relatively simple Maze domain, each method generates a large number of each pattern. As the size we consider increases, the quantity of patterns does drop off, which makes sense, as each subpattern then takes up more space in the level. In the more complex Rogue domain, WFC's generation of large patterns decreases. By contrast, both MWFC and HWFC can generate many of these larger structures.

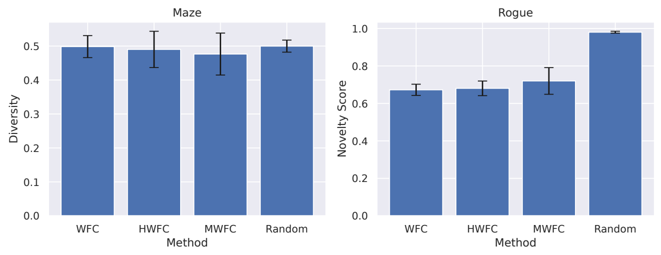
In simpler domains, there is less opportunity for complexity within the environment, which is why we see less of a drop-off for WFC. This could also be caused by the fact that our hierarchical elements in the Maze are similar to the example, causing WFC to generate similar-looking structures.

Qualitative Results

To gain qualitative insights into our findings, we present Fig. 12, which illustrates several levels generated by each



(a) Illustrating the KL-Divergence diversity metric, over 100 levels, as the tile size increases (represented by the x-axis).



(b) Diversity of each method, as measured by the Hamming distance. The mean and standard deviation are shown here.

Figure 8: Diversity of the generated levels using (a) The KL-Divergence and (b) Hamming metrics.

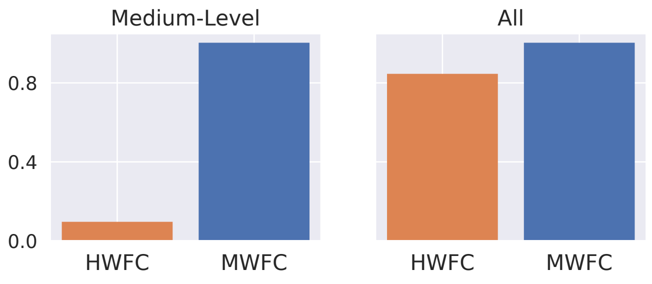


Figure 9: Comparing the action variance of MWFC to HWFC, for Rogue (normalised such that MFWC = 1). The left panel shows the variance for only the medium-level patterns, whereas the right panel shows it over all of the patterns.

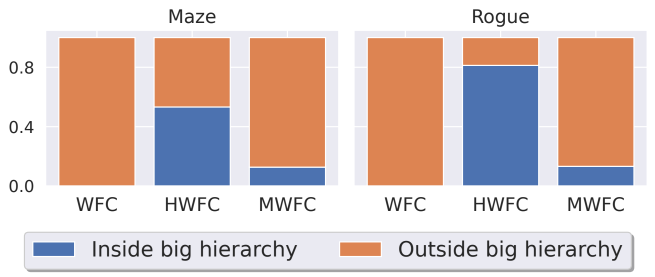


Figure 10: Illustrating the controllability of each method for the (left) Maze and (right) Rogue. Each bar has the fraction of medium-level hierarchical elements that are placed inside compared to outside the top-level hierarchies.

of the three methods for (a) Maze and (b) Rogue. Our observations reveal that while WFC adheres to the constraints, it tends to produce simplistic levels, lacking the emergence of complex structures such as large castles or town layouts. Conversely, both MWFC and HWFC demonstrate the ability to generate such intricate structures. However, MWFC exhibits a notable drawback (evidenced by the Rogue levels), as the majority of levels consist of a limited set of larger patterns repetitively placed in neighbouring locations. In contrast, HWFC strikes a balance between MWFC and WFC, allowing for the emergence of large complex structures such as castles in proportionate quantities. Additionally, Fig. 13 shows a collection of Rogue levels generated using HWFC, employing the 2.5D tile assets shown in Fig. 6. This presentation highlights the diverse range of levels, their complexity, and the controllability achieved through our approach.

Controllability Our approach imparts several aspects of controllability to the generation process. First, we can control the number of hierarchical elements that are placed in the level. Fig. 14 illustrates this aspect. Here we modify the number of top- and medium-level hierarchical elements that are allowed in the level, and obtain visually distinct results.

Next, we can control the relative positioning of the top- and medium-level components, as seen in Fig. 15. Here, the medium pattern is consistently nested within the larger pattern. In the case of MWFC, the medium pattern can be observed both inside and outside of the large hierarchy. If we consider the medium pattern, e.g., as a blacksmith and the big pattern as a castle, it is reasonable to expect the blacksmith to be generated within the castle. While WFC struggles to generate such complex structures, MWFC produces them without spatial constraints, whereas HWFC maintains the desired hierarchical relationships between patterns.

Minecraft We finally turn our attention to Minecraft. To make our approach feasible in 3D, we make some additional changes. First, we seed the starting level's ground boundary with roads, to ensure we do not have uncompleted structures on the edges. Similarly to Barthet, Liapis, and Yannakakis (2022), we use a different block for air inside our buildings compared to outside. Finally, we have different blocks for the corners of houses, and different ones for each different height level, to ensure the generated buildings have a reasonable height. When generating, however, all of these are replaced with identical "wall" tiles. Fig. 16a shows the examples and hierarchical elements we used, and Fig. 16b shows three generated levels. Overall, we can see that our approach generates coherent levels that contain specific desired structures—a walled area and a moat in this case.

Summary

Overall, the results in this section demonstrate that HWFC can effectively be controlled, and structural complexity can easily be added. Despite this, HWFC does not suffer significantly in terms of diversity; on the contrary, it has a better action variance than MWFC. Furthermore, from our qualitative results, we can see that HWFC can generate structured levels—without being overly repetitive and uncontrollable.

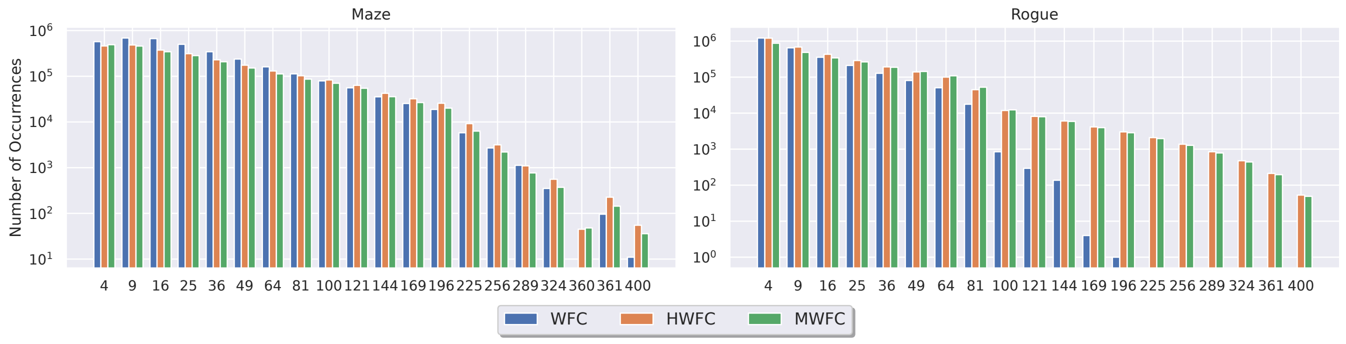


Figure 11: Structural complexity results in (left) Maze and (right) Rogue. Here, the x-axis represents the size of the pattern (or subpattern) we consider, while the y-axis is the number of occurrences of that pattern over 100 levels. For instance, a pattern size of 100 considers, e.g., 10×10 subpatterns of the large hierarchies. It is worth noting that there was a zero occurrence at $x = 360$ for WFC in the Maze domain. In this domain, we used two large hierarchical patterns, one was a simple 20×20 pattern; the other was a more complex pattern of size 18×20. Due to its simplicity, WFC could generate the first pattern. Therefore, where $x = 400\ (20 \times 20), 361\ (19 \times 19), 324\ (18 \times 18)$, etc., WFC had a non-zero number of occurrences. The other, 18×20 pattern, however, was more complex, and WFC never generated it. Therefore, where $x = 360\ (18 \times 20)$, WFC had zero occurrences. This anomaly is an artifact of the particular choice of top-level hierarchies we used.

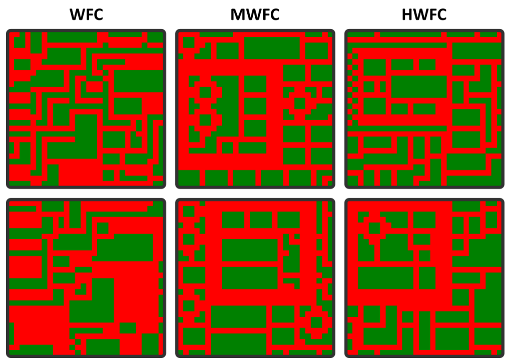
Limitations and Future Work

While we demonstrated that HWFC usefully extends WFC by adding structure, our method has some limitations. Similarly to WFC, there is generally no way to guarantee that a level fulfils specific functional requirements, such as being solvable. Additionally, while designing a single example is simple, it may require more effort to design a useful example that leads to desirable levels. Finally, we found that, for large—and particularly 3D—levels, WFC and HWFC are computationally expensive, requiring large amounts of memory and time to successfully generate levels.

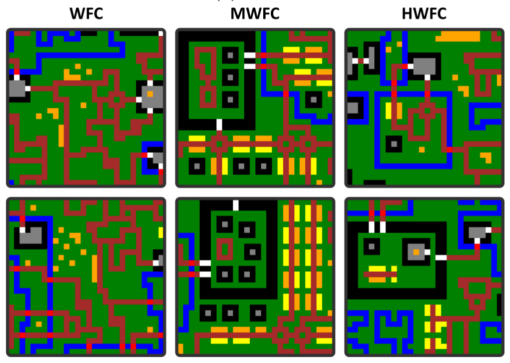
In future work, we would like to combine our approach with another method (such as an evolutionary search-based method) to benefit from HWFC's aesthetic levels and the functionality of objective-based methods. Furthermore, we would like to investigate how to scale the efficiency of the approach to generate larger levels. Additionally, automated extraction techniques could be considered, where the hierarchical elements are partially extracted from some example levels. Finally, it would also be worthwhile to explore 3D domains in more depth, considering how best to design constraints and patterns in this more complex setting.

Conclusion

In conclusion, our research presents a novel approach to level generation in procedural content generation that leverages WaveFunction collapse in a hierarchical fashion, alongside constraint discovery to enable greater control over the generation process, resulting in the emergence of high-level structures. Since HWFC is based on WFC, our approach similarly requires minimal human design—one example and some hierarchical elements—and allows for both diversity and the maintenance of desired constraints. Empirical results demonstrate that, through the added utilisation of hierarchies, more complex structures are accurately represented in our generated levels with a high degree of diversity. Moreover, our approach is scalable and flexible, as evidenced by its successful application to both 2D and 3D domains. Our work has significant implications for video game developers seeking to generate more content more efficiently, while also maintaining desired constraints and increasing the level of control over the generation process.



(a) Maze



(b) Rogue

Figure 12: Two generated levels for each method and game.

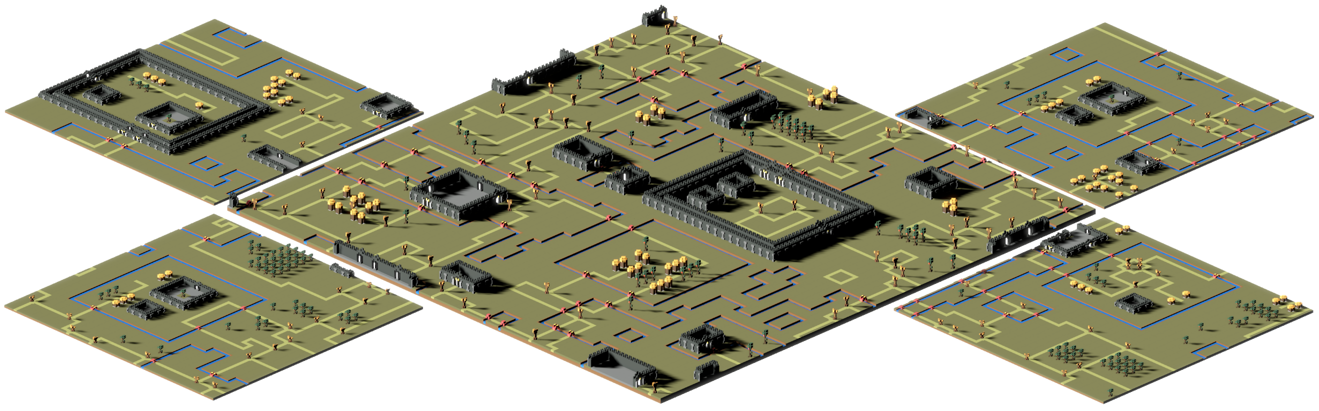


Figure 13: Example levels generated using HWFC on Rogue.

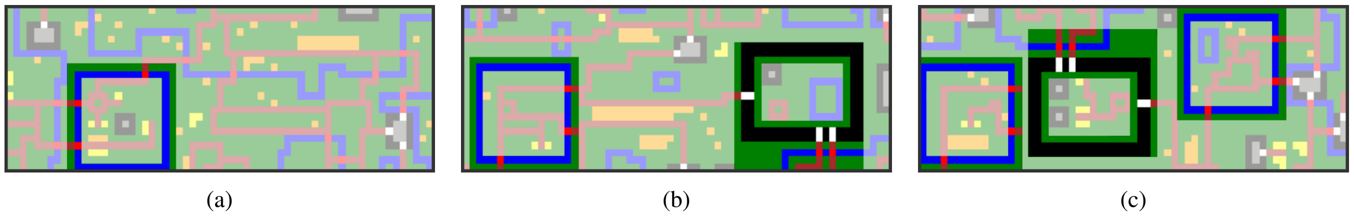


Figure 14: Illustrating the controllability of HWFC by generating levels with (a) one, (b) two or (c) three top-level hierarchies.

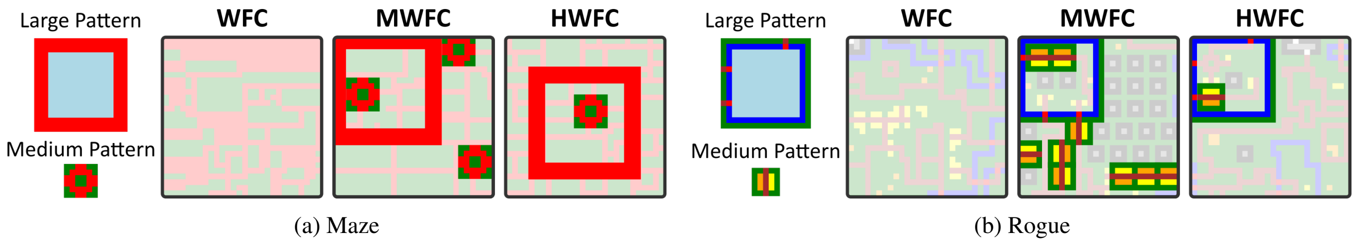
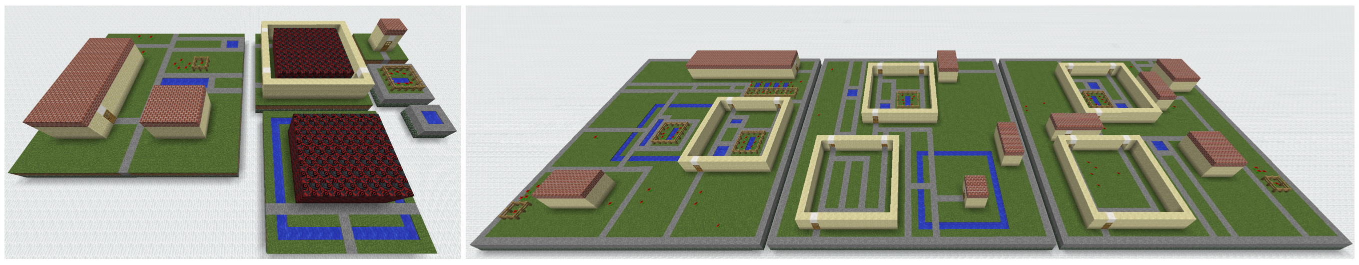


Figure 15: Demonstrating the occurrences of the top- and medium-level hierarchies.



(a) The examples and hierarchical components.

(b) Three generated levels in Minecraft.

Figure 16: Illustrating the (a) examples and hierarchical components and (b) three generated levels in Minecraft. In (a), from right to left, we show the example level, the top-level hierarchical components and the medium-level hierarchical components. The dark red tiles represent superpositions.

References

Adams, T. 2019. Emergent narrative in dwarf fortress. In *Procedural storytelling in game design*, 149–158. AK Peters/CRC Press.

Adams, T.; and Adams, Z. 2006. Dwarf Fortress. http://www.bay12games.com/dwarves/. Accessed: 2023-02-01.

Alaka, S.; and Bidarra, R. 2023. Hierarchical Semantic Wave Function Collapse. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 1–10.

Barthet, M.; Liapis, A.; and Yannakakis, G. N. 2022. Open-Ended Evolution for Minecraft Building Generation. *IEEE Transactions on Games*. Accepted.

Beukman, M.; Cleghorn, C. W.; and James, S. 2022. Procedural content generation using neuroevolution and novelty search for diverse video game levels. In Fieldsend, J. E.; and Wagner, M., eds., *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*, 1028–1037. ACM.

Beukman, M.; Fokam, M.; Kruger, M.; Axelrod, G.; Nasir, M.; Ingram, B.; Rosman, B.; and James, S. 2023. Hierarchically Composing Level Generators for the Creation of Complex Structures. *IEEE Transactions on Games*.

Cheng, D.; Han, H.; and Fei, G. 2020. Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm. In *ICEC*, volume 12523 of *Lecture Notes in Computer Science*, 37–50. Springer.

Concept, P. 2022. Bad North - A Minimalistic, Real-Time Tactics Roguelite with Vikings. https://www.badnorth.com/. Online; accessed 3 February 2023.

Dormans, J. 2010. Adventures in level design: generating missions and spaces for action adventure games. In *Workshop on procedural content generation in games*.

Dormans, J. 2017. Cyclic Generation. In *Procedural Generation in Game Design*, 83–96. AK Peters/CRC Press.

Dormans, J.; and Leijnen, S. 2013. Combinatorial and exploratory creativity in procedural content generation. In *Workshop on Procedural Content Generation for Games*.

Earle, S.; Edwards, M.; Khalifa, A.; Bontrager, P.; and Togelius, J. 2021. Learning Controllable Content Generators. In *2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, August 17-20, 2021*, 1–9. IEEE.

Earle, S.; Snider, J.; Fontaine, M. C.; Nikolaidis, S.; and Togelius, J. 2022. Illuminating diverse neural cellular automata for level generation. In Fieldsend, J. E.; and Wagner, M., eds., *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*, 68–76. ACM.

Gumin, M. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. Accessed: 2023-02-01.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural Content Generation for Games: A Survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1).

Jiang, Z.; Earle, S.; Green, M. C.; and Togelius, J. 2022. Learning Controllable 3D Level Generators. *CoRR*, abs/2206.13623.

Karth, I.; and Smith, A. M. 2022. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. *IEEE Transactions on Games*, 14(3): 364–376.

Khalifa, A.; Bontrager, P.; Earle, S.; and Togelius, J. 2020. PCGRL: Procedural Content Generation via Reinforcement Learning. In Lelis, L.; and Thue, D., eds., *Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2020, virtual, October 19-23, 2020*, 95–101. AAAI Press.

Kim, H.; Lee, S.; Lee, H.; Hahn, T.; and Kang, S. 2019. Automatic generation of game content using a graph-based wave function collapse algorithm. In *2019 IEEE Conference on Games (CoG)*, 1–4. IEEE.

Korn, O.; Blatz, M.; Rees, A.; Schaal, J.; Schwind, V.; and Görlich, D. 2017. Procedural Content Generation for Game Props? A Study on the Effects on User Experience. *Comput. Entertain.*, 15(2): 1:1–1:15.

Langendam, T. S.; and Bidarra, R. 2022. miWFC-Designer empowerment through mixed-initiative Wave Function Collapse. In *Proceedings of the 17th International Conference on the Foundations of Digital Games, FDG 2022*. Association for Computing Machinery (ACM).

Liu, J.; Snodgrass, S.; Khalifa, A.; Risi, S.; Yannakakis, G. N.; and Togelius, J. 2021. Deep learning for procedural content generation. *Neural Comput. Appl.*, 33(1): 19–37.

Lucas, S. M.; and Volz, V. 2019. Tile pattern KL-divergence for analysing and evolving game levels. In Auger, A.; and Stützle, T., eds., *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, 170–178. ACM.

Ludomotion. 2017. Unexplored. https://store.steampowered.com/app/506870/Unexplored/. Accessed: 2023-02-01.

Sandhu, A.; Chen, Z.; and McCoy, J. 2019. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 1–9.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'Neill, M. 2012. Evolving levels for Super Mario Bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*, 304–311. IEEE.

Shaker, N.; Togelius, J.; Yannakakis, G. N.; Weber, B. G.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P. A.; Takahashi, G.; Smith, G.; and Baumgarten, R. 2011. The 2010 Mario AI Championship: Level Generation Track. *IEEE Trans. Comput. Intell. AI Games*, 3(4): 332–347.

Shu, T.; Liu, J.; and Yannakakis, G. N. 2021. Experience-Driven PCG via Reinforcement Learning: A Super Mario Bros Study. In *2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, August 17-20, 2021*, 1–9. IEEE.

Smith, G. 2017. Procedural content generation: An overview. *Level Design Processes and Experiences*, 159–183.

Snodgrass, S.; and Ontanon, S. 2015. A hierarchical mdmc approach to 2d video game map generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 205–211.

Sorenson, N.; Pasquier, P.; and DiPaola, S. 2011. A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels. *IEEE Trans. Comput. Intell. AI Games*, 3(3): 229–244.

Stålberg, O. 2022. Townscaper. https://oskarstalberg.com/Townscaper/. Online; accessed 3 February 2023.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Trans. Games*, 10(3): 257–270.

Togelius, J.; Champandard, A. J.; Lanzi, P. L.; Mateas, M.; Paiva, A.; Preuss, M.; and Stanley, K. O. 2013. Procedural Content Generation: Goals, Challenges and Actionable Steps. In Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P.; and Togelius, J., eds., *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, 61–75. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-62-0.

Togelius, J.; Justinussen, T.; and Hartzen, A. 2012. Compositional procedural content generation. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, 1–4.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Trans. Comput. Intell. AI Games*, 3(3): 172–186.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1): 78–89.

Yu, D. 2008. Spelunky. https://spelunkyworld.com/. Accessed: 2023-02-01.