Proceedings of the Fourteenth Artificial Intelligence and
Interactive Digital Entertainment Conference (AIIDE 2018)

A Design Pattern Approach for Multi-Game Level Generation

Spencer Beaupre, Thomas Wiles, Sean Briggs, Gillian Smith
Worcester Polytechnic Institute
Worcester, MA, USA
{smbeaupre, tgwiles, srbriggs, gmsmith}@wpi.edu

Abstract

Existing approaches to multi-game level generation rely upon level structure to emerge organically via level fitness. In this paper, we present a method for generating levels for games in the GVGAI framework using a design pattern-based approach, where design patterns are derived from an analysis of the existing corpus of GVGAI game levels. We created two new generators: one constructive, and one search-based, and compared them to a prior existing search-based generator. Results show that our generator is comparable, even preferred, over the prior generator, especially among players with existing game experience. Our search-based generator also outperforms our constructive generator in terms of player preference.

Introduction

While scholars and practitioners have made great progress in developing techniques and methods for procedural content generation in specific game design contexts, research in how to approach the algorithmic creation of content for multiple games is only in the preliminary stages. Human designers are capable of creating levels for a wide variety of games because of their ability to interpret new contexts from written rules and to apply design processes and pre-existing knowledge across contexts. In this paper, we present a design pattern-based approach to generating levels in multiple game contexts. We work in the GVG-AI framework (Khalifa et al. 2016), a testbed for multi-game analysis, though we are hesitant to apply the term "general" to it (see Discussion).

Current approaches to multi-game level design typically rely upon level structure emerging organically; levels often look disorganized, cluttered, and laid out without underlying intent for specific experiences. Design patterns are an approach for formalizing design knowledge (Alexander, Ishikawa, and Silverstein 1977) and reasoning about game and level structure. But it is unclear how to identify useful design patterns that can span multiple games in a concrete-enough way to support level generation.

We take the approach of identifying design patterns in a bottom-up manner, through automated analysis of the existing corpus of human-designed levels in the GVG-AI framework. These patterns are then used as building blocks for newly generated levels. To use level information from different games, each human-made level was converted into a set of abstracted design patterns, each of which are 3x3 tile segments. Levels are generated by assembling combinations of these patterns, selected based on their frequency in the overall pattern corpus. A selection of these constructively-generated levels form the initial population for a genetic algorithm that optimizes the level based on simulated player performance. We then evaluated our design pattern-based generator with human players, and compared the two generators we created–one constructive, one search-based–with a prior search-based generator built in the same framework.

Our contributions in this paper are a design pattern-based analysis of an existing game corpus, an approach for multi-game content generation that performs equally well, if not better, than an existing search-based approach, and our derived insight into the challenges and opportunities for using design patterns in multi-game generation.

Related Work

The GVG-AI competition focuses on studying PCG and game playing in a multi-game context (Khalifa et al. 2016). In addition to the evolutionary approach followed by Khalifa et al., there is also a GVG-AI generator that combines answer set programming with evolutionary algorithms (Neufeld, Mostaghim, and Perez-Liebana 2015).

The GVG-AI framework currently supports over 90 different ports or variations of classic two-dimensional arcade games. Games are expressed in the VGDL language (Schaul 2013), which provides a consistent structure and language for expressing both game rules and game levels. The framework also comes with a set of sample, hand-authored levels for use in the game-playing track, which we use as a corpus for design pattern extraction.

Prior work in using design patterns for GVG-AI level generation is limited. Sharif et al. (Sharif, Zafar, and Muhammad 2017) have done preliminary work in manually identifying design patterns in the GVG-AI level corpus following an analysis of player movement through the space. Though their goal is to use these patterns in a generative context, their work is still ongoing. The 23 patterns they identify are largely based on how different sprites move and are grouped in level space. Our analysis is automated and results in the detection of more patterns and their frequency in the levels,

```
1.............................    300000000000000000000000000000
000...........................    111000000000000000000000000000
000...........................    111000000000000000000000000000
..............................    000000000000000000000000000000
..............................    000000000000000000000000000000
..............................    000000000000000000000000000000
..............................    000000000000000000000000000000
....000......000000.....000...    000011100000011111100000111000
...00000....00000000...00000..    000111110000111111110001111100
...0...0....00....00...00000..    000100010000110000110001111100
................A.............    000000000000000020000000000000

Aliens Level Mappings:            General Type Mappings:
    . > background                    0 > Other
    0 > background base               1 > Other Other
    1 > background portalSlow         3 > Other Harmful
    2 > background portalFast
    A > background avatar
```

Figure 1: Left: the original hand-authored level for Aliens. Right: the same level, converted to an abstracted type system.

but does not consider how sprites move across space.

There is a rich literature on design patterns, which originated in architecture (Alexander, Ishikawa, and Silverstein 1977), as a tool for game analysis and production (Bjork and Holopainen 2004; Nystrom 2014), and especially for PCG for platformers (Dahlskog and Togelius 2012). Typically design patterns are identified by human authors and include contextual information for when and why they should be used. Our design patterns are identified automatically, and context is inferred from frequently. There has been work from the machine learning game AI community on producing generators based on learned patterns from datasets (e.g. (Snodgrass and Ontañón 2014; Summerville and Mateas 2016)), and though our approach does not currently incorporate a learned relationship between patterns in generation, this is an area of future work.

Design Patterns

All our design patterns in this paper are 3x3 “micro”-patterns of different types of sprites that are frequently grouped together in levels across games. This section describes our approach for deriving these patterns and an analysis of the patterns themselves.

Abstracting Levels

In the VGDL language, levels are comprised of game-specific information that cannot be directly translated to another game expressed in the same language. To use all levels in the GVG-AI framework as a corpus, we must first convert the levels to a common format. Game-specific information is reclassified into a abstract type system that can be used to translate sprite combinations from one game into the most similar sprite combinations for any other game.

VGDL has five sprite classifications. `Avatars` are any sprites directly controlled by the player. `Solids` are sprites that cannot be moved through and have no other interactions. `Collectables` are non-harmful and are destroyed by the player upon interaction. `Harmfuls` either destroy the player or spawn other sprites that do. `Others` are any sprite that does not meet all qualifications for another category. We lean on this representation in building abstract patterns (using the first letter of each classification as a symbol in the pattern), since the classifications provide a rough sense of the role that their specific game sprites play in design.

Table 1: Our abstract format for levels includes both single sprite and combination sprite types.

Token	Type	Token	Types
A	Avatar	1	(Other, Other)
C	Collectable	2	(Other, Avatar)
H	Harmful	3	(Other, Harmful)
O	Other	4	(Other, Collectable)
S	Solid	5	(Other, Solid)
		6	(Other, Avatar, Harmful)
		7	(Other, Harmful, Harmful)

VGDL game levels are stored as matrices of mappings, meaning each tile can contain one or more sprites. Since we want our design patterns to include such combinations of sprites, our abstract format includes symbols for combinations of sprites, denoted by numbers.

There are 12 tokens in the abstract level description language: 5 single sprites and 7 sprite combinations, as described in Table 1. An example of a specific level (for *Alien*) converted to an abstract level is shown in Figure 1.

Pattern Extraction

97 games are represented in the pattern library, out of a total of 103 games in the GVG-AI corpus. Six games in the corpus were removed (*eggomania, eightpassenger, jaws, painter, realsokoban*, and *thecitadel* because of errors reading the game description file and/or levels. Every game in the framework has 5 hand-authored levels, resulting in 485 levels in total from which we could draw patterns.

A 3x3 sliding window was passed over each abstracted level, and every 3x3 token matrix in the level was then stored as a potential pattern. This resulted in 12,941 unique patterns. The patterns, along with the frequency of their occurrence in the corpus, are stored in a pattern library for use in the generation process.

The pattern library also holds apart certain types of patterns for use in specific parts of the generation process. All patterns containing an avatar are tagged, since each level can only contain one avatar. Additionally, all patterns that are all `Solid` on one or more sides are classified as “border” patterns that are candidates for edge or corner pieces in a level.

Resulting Patterns

Though all 12,941 patterns are used in the final generative process (randomly selected, weighted by their frequency), in this section we report upon the six most common patterns found in the GVG-AI corpus. Table 2 reports the pattern and its percentage frequency in the corpus, as well as the games that are most and least influential on that pattern’s appearance in the corpus. In parentheses after each game name is the percentage by which the pattern is over- or under-represented in the game compared to its representation in the overall pattern library. For example, the pattern that con-

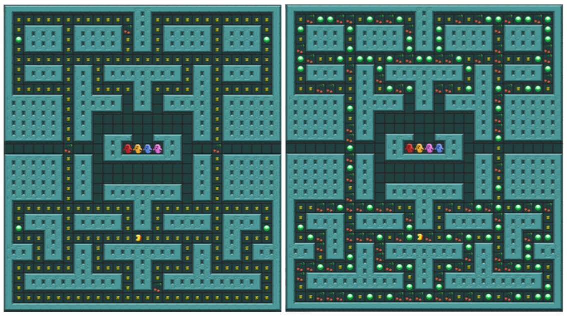

Figure 2: Left: rendering of a hand-authored Pacman level. Right: the same level, reconstructed from its abstract description. Notice greater variety in collectible items, due to loss of specifics about which collectible item appears at that point in the level.

sists of all O sprite types appears 66.38% more often in the game *x-racer* than in the overall pattern library, indicating that this game greatly contributes to its presence in the pattern library.

It is notable from this table that one pattern dominates the library: the empty space pattern accounts for almost a quarter of all game levels. This pattern is also the most polarizing, with the pattern either vastly over- or under-represented in many games (including those in the table). There are also two patterns that appear with greater frequency due to only a single game: the all solids pattern is overrepresented in the game *modality* and the all enemies pattern is overrepresented in the game *defem*. How influential a game is over the pattern's representation is not taken into currently taken into account during pattern selection in generation, though could be in future work.

Generative Approach

We use an evolutionary approach for level generation, building from an initial population of levels created using the constructive generator. The constructive generator selects and instantiates abstract patterns in the context of the game description provided to the generator.

Instantiating Abstract Patterns

The abstraction of levels into a common format results in information loss. For example, if a game has two sprites that are classified as the same type (such as the pellets and fruit in *pacman*), then the encoded level will lose the distinction between the two when recording them both as the same type (see Figure 2, depicting a hand-authored level that has been converted to an abstract representation and then back again).

When instantiating abstract patterns for a specific game, the generator selects sprite combinations that match each type from the list of level elements in the VGDL description for that game. In the case of multiple specified sprite combinations matching a single type, a sprite is chosen at random. In the case of no match to the type, the closest match in terms of number of matching elements is selected. For example,

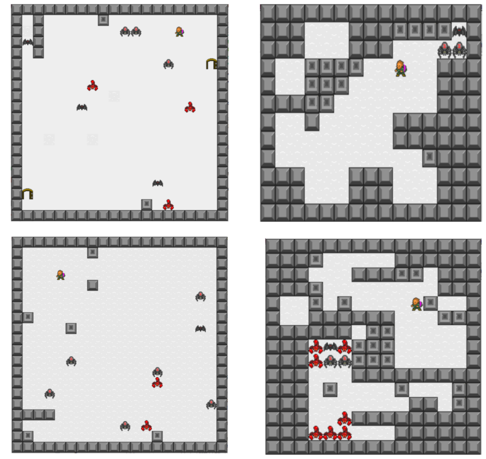

Figure 3: Top left: Khalifa et al. constructive generator, Top right: our constructive generator, Bottom left: Khalifa et al.'s search-based generator, Bottom right: our search-based generator.

Type 7 - Other, Harmful, Harmful does not have a match in pacman, where harmful sprites cannot occupy the same space. In its place, an (Other, Harmful) mapping is made, such as placing a Floor tile and a Red Ghost.

Constructive Generator

The constructive generator combines patterns from the pattern library to create the layout of a level, and then converts the abstract types from the patterns into game-specific level mappings. Due to the 3x3 shape of the patterns, levels must have lengths and widths divisible by 3. When building a level, patterns are randomly chosen from the library, weighted by the frequency of that pattern's appearance across all games.

If the game includes a `Solid` sprite, then a border is created around the edges of the level by randomly selecting edge and corner patterns. Otherwise, random patterns are selected for the entire level.

There are also two greedy pattern selection heuristics: 1) Every time a pattern is selected as a candidate, the system checks to see if it contains an avatar: if it does and if no avatar has been placed in the level yet, then the avatar pattern will be included, otherwise it is ignored and a new pattern is chosen. This is because there can be only one avatar in currently-expressed GVG-AI games. 2) When a pattern is placed into the level, we check to see that all non-solid spaces in the pattern are reachable. If they are not, then the pattern is discarded and another one included.

Evolutionary Approach

We adapted the existing search-based generator included in the GVG-AI framework to use our constructive generator to create an initial population. We also modified the mutation function to operate on patterns instead of individual sprites. There are two different mutation operators. In the

Table 2: Table showing the six most frequent patterns in the pattern library, how frequently they appear across all levels, and the games that most and least contribute to their presence in the library.

Pattern	Frequency	Most Influence	Least Influence
OOO OOO OOO	24.05%	x-racer (+66.38%)	pacman (-24.05%)
OOO OOO SSS	2.56%	pokemon (+13.91%)	lemmings (-2.16%)
SSS OOO OOO	2.16%	pokemon (+14.51%)	lemmings (-2.16%)
SSS SSS SSS	1.95%	modality (+59.59%)	bomberman (-1.95%)
OOO SSS OOO	1.27%	donkeykong (+1.60%)	whackamole (-1.27%)
111 111 111	1.21%	defem (+68.74%)	boulderdash (-1.21%)

first, when levels are mutated, one pattern is randomly selected and swapped with another pattern; if the selected pattern is on the border, then it is replaced with another border pattern. In the second, two patterns are selected at random and swapped with each other.

Our evolutionary generator used the same parameters as Khalifa et al.'s search-based generator: an initial population of 50, with a crossover probability of 70% and a mutation probability of 10%. Our generator uses the same simulated player fitness function as the original Khalifa generator as well.

Evaluation

In this section, we present example levels generated using our design pattern-based generator, and a comparative evaluation between our generator and Khalifa et al.'s search-based generator (Khalifa et al. 2016).

Sample Levels

Figure 3 shows a side-by-side comparison of typical levels for the game Bomberman against prior state of the art generated levels.

In contrast to the minimal aesthetic in existing GVG-AI levels, for which the generator added just enough sprites to make the level playable, our generated levels have more structure and include more game objects.

Player Evaluation

We conducted a study comparing player preferences between our search-based and constructive generators, and Khalifa et al.'s search-based generator. We compared only against their search-based generator because they reported that as the best of their generators. We did not compare to a random baseline.

We selected three games on which to conduct this test: *Frogs*, *Bomberman*, and *Zelda*. Of these, *Frogs* and *Zelda* were also used in Khalifa's evaluation. The third game they used, *Pacman*, was replaced with *Bomberman*, as the framework does not support more than one ghost of each color in generated levels, and our generator is somewhat likely to create these as there is no resriction on the number of sprites in any level. Brief game descriptions follow:

- *Bomberman* - Port of original Bomberman. Move around the level and place bombs that explode in a cross shaped pattern (+) to either destroy dark blocks or kill enemies. The goal is to find all doors which are hidden beneath destroyable blocks.
- *Frogs* - Port of Frogger. The objective is to reach the end goal(s) while avoiding vehicles and water which will kill you on contact. The character can only move across water if they are on a log, otherwise they will fall in the water and drown.
- *Zelda* - Port of the original Legend of Zelda game. The player must first collect a key to then be able to unlock the exit door. The character can attack in the direction they are facing to kill an enemy.

Five levels were generated per game, per generator, and new levels were generated using the Khalifa et al.'s search-based generator. A total of 45 levels (15 per generator) were created and stored for this survey. There was no manual curation of which levels would be used for testing.

Our survey had participants play two levels of the same game but from different generators, and repeated this test for each game to account for all generator comparisons, resulting in each person playing a total of six levels. Each person played three comparisons:

- Our search-based vs. Khalifa search-based

Table 3: Results from the user study comparing our constructive (NC) and search-based (NS) generators with each other and against Khalifa et al.'s search-based generator (KS)

.

	A	B	Success	P-Value
NC (A) vs. KS (B)	16	14	53.33%	0.4278
NS (A) vs KS (B)	17	13	56.67%	0.2923
NS (A) vs NC (B)	20	10	66.67%	0.0494

Table 4: Participant preferences separated by prior experience with video games.

Experience	Conf.	Cont.	Success	P-Value
None	2	1	66.67%	0.5
Limited	6	12	33.33%	0.9519
Moderate	24	12	66.67%	0.0326
Substantial	21	12	63.64%	0.08138

- Our constructive vs. Khalifa search-based
- Our constructive vs. Our search-based

The order of conditions and games was randomized.

Players were consented to the study, and had the rules for each game explained to them. The player was asked only to identify which level they preferred, and following playing the games was also administered a demographic survey for gender, age, and experience with video games. A total of 30 players were recruited (20 identify as men, 10 as women), who had varying experience playing video games. Table 3 shows the results of our study, including that our search-based generator was preferred over our constructive generator (significant results, $p < 0.05$), and both of our generators were marginally preferred over the Khalifa search-based generator (not statistically significant).

As in Khalifa's study, the search-based generator significantly outperforms the constructive generator in terms of player preference. This is likely because the simulation-based fitness function is more effective for producing playable levels. However, both our search-based and constructive generator performed similarly when compared against Khalifa's search-based generator. This could indicate that when levels of Khalifa's generator were compared against ours, players would evaluate them with slightly different criteria than when comparing the levels of the two pattern-based generators. Table 4 shows how often participants confirmed or contradicted our hypotheses regarding generator preferences for all three generators, separated by prior game experience.

Players with moderate or substantial experience tended to align with our hypotheses about two thirds of the time, while players with low experience did so just one third of the time. This indicates that prior game-playing experience influenced how the players compared levels.

Table 5 displays the results of our study for just players with moderate or substantial experience with video games. These results are similar, but with higher confidence values,

Table 5: Results for comparing our generators and Khalifa's search-based generator, with players who have low or no experience removed.

	A	B	Success	P-Value
NC (A) vs. KS (B)	13	10	56.52%	0.3382
NS (A) vs KS (B)	15	8	65.22%	0.1050
NS (A) vs NC (B)	17	6	73.91%	0.0173

for the comparison between our constructive generator and the Khalifa's search (small preference for our constructive but not statistically significant), as well between our two generators (statistically significant preference for our search generator over the constructive). Our search-based generator is preferred over Khalifa's among this population, with a confidence of approximately 90%.

Though we did not ask participants for why they preferred one generator over another, several participants volunteered this information. Reasons for selecting one level over another varied widely. Some preferred Khalifa's generator because levels were more open and had less total sprites on the screen, and the level thus seemed more approachable. One participant said they preferred a level simply because it was larger, and that the contents did not influence their choice. Though none of these comments can be considered conclusive evidence, the varied comments we received do point to the need for rigorous qualitative research into why people state design preferences in the context of the multi-game level generation.

Level Metrics and Expressive Range

For the constructive generator, $4.5 * 10^{98}$ combinations of patterns are possible for non-bordered games like Space Invaders, for bordered games like Zelda, $9 * 10^{76}$ combinations are possible. It is clear from the reactions of our participants that the range of content our generator can produce includes extremely high and low difficulty levels, and levels with both too many and too few objects. It is hard from this to judge what kind of variety there is among levels and what biases are present in the generator. Expressive range analysis has thus been suggested as an evaluation method (Smith and Whitehead 2010).

While a full expressive range analysis of the generator using game-independent metrics is beyond the scope of this paper, we do think it important to comment on the range of levels the generator is capable of producing and to what extent the levels played in our playtest are representative of the generator. A simpler method for assessing the expressive range of the generator is to count how many of each type of sprite gets placed in levels on average. The proportion of objects used in a level influences the experience playing it. Harmful and collectable objects introduce goals and adversaries; solid objects introduce obstacles and reflect how much open space there is for the player to traverse.

Table 6 shows the average, max, and min number of different object types in 1000 generated levels of the *frogs* game, generated using the constructive generator. The

Table 6: Object count metrics for 1000 levels of *frogs* made by our constructive generator.

	Solids	**Collectables**	**Harmfuls**	**Other**
Mean	98.876	5.793	7.037	194.797
STD	13.585	3.992	5.726	31.828
Max	135	28	33	266
Min	64	2	0	144

Table 7: Object count metrics for the 5 *frogs* levels made by the constructive generator and used in our study.

	Solids	**Collectables**	**Harmfuls**	**Other**
Mean	94.8	5	9	187.2
STD	3.493	3.742	5.244	4.324
Max	98	11	16	192
Min	91	2	4	182

amount of harmful and collectable objects has quite a large range that could be a cause for the inconsistency in level difficulty. Of the 1000 levels generated, they ranged from having no harmful sprites to as much as 33. This range of variation makes it important to assess just how representative the levels used in our playtesting study are of the typical levels produced by our generator.

Table 7 shows the same metrics, this time only for the 5 constructive generator levels used in the study. The average amounts of each object type are very close to the averages for the overall generator, except for harmful types being slightly more frequent in the survey levels. The standard deviations of the solid and other types are much lower in the survey levels, indicating that they cover only a small portion of the generator's expressive range. The standard deviations for the collectables and harmfuls are closer, however. As these have a more direct impact on the player, and the mean for all four types are relatively consistent with the expected values, we determined that the constructive levels used for Frogs in our survey are fair representations of our generators expressive range. Similar conclusions were found for the constructive levels for *Bomberman* and *Zelda*, but are not included in this paper for space reasons.

Discussion

In our work, we sought to explore the potential of procedurally generating levels that felt more human-made than ones from more basic generators. Overall, we have shown that a pattern-based approach is a viable and promising method of level generation for 2D arcade-style games of the type expressed in the GVG-AI framework. Qualitatively, the levels produced by our generators had a more organic and flowing structure that created a positive gameplay experience.

The major weakness of our generators was the inconsistency of level difficulty. Some levels would place the player directly next to an enemy at the start or would require the player to path through a region filled with a massive number of enemies. Meanwhile, other levels could have no enemies or place them in such a way the player was unlikely to need to interact with any. A further challenge is in assessing the impact of specific enemy types on game experience: for example, a water tile in *Frogs* is not nearly so dangerous as the alien spawner in *Space Invaders*, yet these are treated as equivalent due to the representation in VGDL and the abstraction enforced by our design patterns.

Future Work

Our design pattern-based approach to multi-game level generation is still in-progress, and we are exploring two specific avenues of future research. The first is in modifications to the constructive generator, to take into account frequency of neighboring patterns to the selected pattern. This also involves a deeper analysis of the patterns found in games to determine correlations: are there some patterns that are typically co-located in a level, and can we create hierarchical design patterns that embed this design knowledge? The second avenue is in gaining greater insight into why certain patterns are over or under-represented in certain games. For example, *space invaders* over-influences the presence of the open space pattern in the overall pattern library, which makes sense when considering patterns of play - the game requires an open play field. In future work, we aim to infer such relationships from rules or simulated play, and use them to guide the inclusion or exclusion of particular patterns in the generator.

On "Generality"

Finally, we find it important to note the problematic nature of the term "general" when it comes to the GVG-AI competition specifically, and "general" artificial intelligence more broadly. Generality is always interpreted with respect to the context it can be used in: we arguably produce "general" design patterns, but these design patterns can only apply to games that even have the notion of collectables or harmful entities (and thus that are inherently based on conflict). As Smith has argued previously, the games that this community chooses to use for research reflect and reinforce the values that the community holds (Smith 2017).

The operational logics (Osborn, Wardrip-Fruin, and Mateas 2017) expressed by games in the GVG-AI framework are quite limited, and use of the term "general" to describe it risks contributing to hegemonic thinking in games research (Fron et al. 2007). The other major multi-game level design corpus, VGLC (Summerville et al. 2016), is also quite limited in its scope, including mostly games that appeal to 80s and early 90s nostalgia. One can envision an alternate history of games research in which it is not popular arcade or Nintendo console games that are celebrated and formalized for research purposes, but instead visual novels or adventure games or playground activities. To call the GVG-AI framework and competition an exploration of "general" video games is to implicitly declare that the kinds of games modeled in the framework are reasonable and representative of the space of games itself, which they clearly are not. We strongly urge the research community reconsider and reflect upon the ways in which declarations of "generality" embed implicit assumptions of what is considered a "normal" game, and why.

References

Alexander, C.; Ishikawa, S.; and Silverstein, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Bjork, S., and Holopainen, J. 2004. *Patterns in Game Design (Game Development Series)*. Charles River Media, 1 edition.

Dahlskog, S., and Togelius, J. 2012. Patterns and Procedural Content Generation. In *Proceedings of the Workshop on Design Patterns in Games (DPG 2012), co-located with the Foundations of Digital Games 2012 conference*.

Fron, J.; Fullerton, T.; Morie, J. F.; and Pearce, C. 2007. The hegemony of play. In *Situated Play: Proceedings of Digital Games Research Association 2007 Conference. Tokyo, Japan*, 1–10.

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J. 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, 253–259. New York, NY, USA: ACM.

Neufeld, X.; Mostaghim, S.; and Perez-Liebana, D. 2015. Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, 207–212.

Nystrom, R. 2014. *Game Programming Patterns*. Genever Benning.

Osborn, J. C.; Wardrip-Fruin, N.; and Mateas, M. 2017. Refining operational logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, 27:1–27:10. New York, NY, USA: ACM.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8.

Sharif, M.; Zafar, A.; and Muhammad, U. 2017. Design patterns and general video game level generation. *International Journal of Advanced Computer Science and Applications* 8(9).

Smith, G., and Whitehead, J. 2010. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the Workshop on Procedural Content Generation in Games, co-located with FDG 2010*.

Smith, G. 2017. What do we value in procedural content generation? In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, 69:1–69:2. New York, NY, USA: ACM.

Snodgrass, S., and Ontañón, S. 2014. Experiments in map generation using markov chains. In *Proceedings of the 2014 Foundations of Digital Games Conference*.

Summerville, A., and Mateas, M. 2016. Super mario as a string: Platformer level generation via lstms. *CoRR* abs/1603.00930.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and n'on Villar, S. O. 2016. The vglc: The video game level corpus. *Proceedings of the 7th Workshop on Procedural Content Generation*.