Patterns as Objectives for Level Generation

Steve Dahlskog
Malmö University
Östra Varvsgatan 11a
205 06 Malmö, Sweden
steve.dahlskog@mah.se

Julian Togelius
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen, Denmark
julian@togelius.com

ABSTRACT

This paper discusses how to use design patterns in procedural level generation, with particular reference to the classic console game Super Mario Bros. In a previous paper, we analyzed the levels in this game to find a set of recurring level design patterns, and discussed an implementation where levels were produced from concatenation of these patterns. In this paper, we instead propose using patterns as design objectives. An implementation of this based on evolutionary computation is presented. In this implementation, levels are represented as a set of vertical slices from the original game, and the fitness function count the number of patterns found. Qualitative analysis of generated levels is performed in order to identify strengths and challenges of this method.

Keywords

Game Design Patterns, Procedural Content Generation, Super Mario Bros., Genetic Algorithms

1. INTRODUCTION

This paper tries to combine design patterns and procedural content generation (PCG) in the domain of level design and is aimed toward the 2-dimensional platformer game genre and especially the classic game of *Super Mario Bros.* (SMB) [27]. Out approach is based on a previous article where we analyzed *SMB*-levels to find 23 different reoccurring patterns of 5 "families"; Enemies, Gaps, Valleys, Multiple paths and Stairs. In that paper, we also used the suggested level patterns to implement a level generator that concatenated these patterns with some variation in height, length or difficulty. In this paper we consider a different take on the relationship between patterns and PCG, seeing patterns not as building blocks but as objectives. This inverts the relation between pattern generation and detection, and conceptually separates building blocks from design objectives. In order to increase the variation of the content but still incorporate the suggested level design patterns of *SMB* we implemented a prototype based on evolutionary computation. In this implementation the representation of level content is made by taking recurring pieces from *SMB*. These pieces are just one tile wide and thus we will refer to them as *vertical slices*. By chopping up Mario levels into finer elements in this way, we can considerably increase the variety of generated levels while still only using existing material. The fitness function for the evolutionary algorithm counts the number of patterns found in the candidate levels. This approach allows us to create levels that resemble the original Super Mario Bros levels both on a micro level, by using existing vertical slices, and on a macro level, by incorporating the same design patterns.

2. BACKGROUND

In this section we intend to clarify where our approach is grounded by shortly review patterns, design patterns in games and automatic generation of game content as well as what game content in games may be.

2.1 Design patterns

During the seventies, Alexander et al. developed a language of patterns for architectural use with the goal to allow others to express design abilities. "Each pattern describes a problem which occurs [...] in our environment, and then describes the core solution to that problem..." [4]. The advantage of the pattern idea is the allow a designer to use a general pattern to solve reoccurring problems. This idea has gradually spread to other areas. In object oriented software development Gamma et al. have defined a set of templates for solving general design and programming problems [16].

2.1.1 Design patterns and games

Design Patterns in digital games (*DPG*) can be used for different activities ranging from scholastic approaches to practical development of (digital) games. Suggested use range from creative aid in design activities where DPG can support knowledge transfer between designers while they generate, communicate and modifiy design ideas and concepts. DPGs can also be used as an analytical tool as well as a learning tool for scholars in both game analysis and game design activities. The design patterns may be used as a tool to understand player behavior during play-testing.

Björk and Holopainen have extensively documented design patterns for game design [7, 6]. Design patterns have also been explored in a variety of aspects, ranging from pattern-related design in relation to game mechanics [2] to specific game contexts, like old-school action games [9], RPGs [30], FPSs [18] and social network game applications

in the style of "'Ville" games [22].

2.2 Game content and game development

Digital games can be seen as the union of two types of digital artifacts: game content and a game engines. The game engine's purpose is to handle user input, to control AI-agents and to present the game content to the user. Typically the game content consists of polygon meshes, texture maps, level geometry, non-player characters, player-characters, missions, quests, items, etc.

2.2.1 Cost drivers

Digital games have become more expensive to develop [33, 20], at least for the average commercial project. The driver of costs is primarily that the advancement of the technical platform that runs the digital game (hardware and game engine) requires more labour to be spent in content development. Customers expect more media and with higher quality [15] which in turn demands more of the engine that uses this media which therefore inherently becomes more complex and may even drive the cost to develop and maintain further.

2.2.2 Game development and PCG

Procedural content generation (*PCG*) is the process of automatically or semi-automatically generating game content. Game developers used PCG successfully to generate game content for different purposes. Examples range from saving developer time with the game *Darwinia* [19], or saving money as for *Just Cause* [5], exploring possible game content as with *The Sentinel* [14], to have variation and creating unique game content as with *Minecraft* [25] and to saving main memory as in *Elite* [1, 34].

Lately *PCG* have spurred interest from researchers and several aspects have been explored as a result. Examples vary from search-based methods to find maps for RTS games [37], levels for platform games [23], and answer set programming for generating mazes [29].

PCG could be used in game design and game development in several different ways, depending on whether the algorithm is seen as a tool, an expert, a designer in its own right etc [21].

There are a couple of approaches that could combine PCG and design in fruitful ways. Firstly, we could use PCG for specific, well defined (and therefore well tested) design tasks, as for instance in *Civilization* [24] where it is used to generate new world maps during run-time (i.e. *online*) or during development as to generate a large set of varying game items, like in *Borderlands* [17], where PCG were used in several aspects but put to extensively use for generating a large set of weapons. Secondly, we could let the PCG process be the central content provider to the whole title as in *Galatic Arms Race*, *FTL*, *Terraria*, *Minecraft*, *Dwarf Fortress* [13, 35, 28, 25, 3]. A third approach would be to start with handcrafted content and letting the PCG process emulate or copy the designers choice with some kind of variation. This approach can be utilized both for *online* or *offline* PCG. The *offline* version could be used both during the principal development of the game title or after release in order to provide more content to generate more sales of the main title or it could be used to create add-on packages for players who has finished the main game.

2.3 Fitting into the pattern

Let us recall what level designers do; "Level designers use a toolkit or 'level editor' to develop new missions, scenarios, or quests for the players. They lay out the components that appear on the level or map and work closely with the game designer to make these fit into the overall theme of the game." [15]. In order to be able to do this together with PCG we have previously applied a content analysis based on a combination of heuristic analysis [12] and rhythm groups [10, 31] and we suggested a set of level design patterns for the game *Super Mario Bros.* (SMB) [27] and presented a prototype of a level generator[1] [11]. In the previous prototype, we used parameterized but fairly straight forward patterns taken from *SMB* that were randomly picked, modified for difficulty and placed in sequence to form a level. The level generator prototype was able to generate *SMB* levels with high similarity to the original game but some limits in the aspect of variation of the content. The only variation the prototype demonstrated was due to the parameterization (different hight, different length, amount of rewards, amount of risk) of the existing patterns.

For the work-in-progress level generator that we present in this paper we try to approach the problem of generating content by using patterns as the objectives for the evolutionary method rather than generating pattern building blocks as we did with a previous prototype. Our motivation to seeking this approach rather than the reverse is to allow for a greater "creative freedom" or in computer science terms; a larger[2] design space for the PCG-engine than the previous prototype could.

Our goal is to be able to produce a level generator that recreates the particular design and look-and-feel of the levels from the original *SMB*, while still being novel and offering new challenges.

2.3.1 Examples of patterns

Due to the limited space in this paper we will only partially include the suggested patterns that we utilize to generate content (see table 1, figures 1, 2, 4, 5 and [11]).

2.4 Related work

Previous work in the same are as ours is the Tanagra mixed-initiative level generator [32] founded in analysis of platform game levels by Smith et al. [31]. Tanagra is an interactive tool for level designers that utilize a constraint solver for the creation of level geometry according to a number of patterns. The patterns are of two types, single-beat (like "gap pattern" and "spring pattern") and composite patterns (like "valley" and "mesa") which are implemented with some flexibility that can be extended with the aid of the constraint solver to fit the the level. Another approach is the "occupancy-regulated extension" (ORE) [23], that works by adding bits and pieces of jigsaw-puzzle-like parts from levels in a compositional way to solve patterns in level generation.

The use of patterns as objectives, e.g. as fitness functions in search-based PCG similar to ours is the "choke point" evaluation function while evolving maps for *StarCraft* [8, 37]. The function assigned a higher fitness to maps that

[1]Presented at the *First Workshop on Design Patterns in Games, 2012.*

[2]But still more distinct and limited than full randomization of levels.

Table 1: Examples of patterns for *Super Mario Bros.*

Enemies	
Enemy	A single enemy
2-Horde	Two enemies together
3-Horde	Three enemies together
4-Horde	Four enemies together
Roof	Enemies underneath a hanging platform making Mario bounce in the ceiling
Multiple paths	
2-Path	A hanging platform allowing Mario to choose different paths
3-Path	2 hanging platforms allowing Mario to choose different paths
Risk and Reward	A multiple path where one path have a reward and a gap or enemy making it risky to go for the reward

contains choke points and the result of the level generator is thus likely to have that pattern.

3. MARIO

The original SMB platform game, initially published by Nintendo in 1985, has been the inspiration for Markus "Notch" Persson's public domain Java-based clone which in turn have been modified for the Mario AI and PCG competitions [36]. In *SMB* Mario has to traverse 8 worlds with 4 levels each. The 4th level of each world is a "Boss-fight"-level with different layout than the other levels.

3.1 The original representation

The original game from *Nintendo* that was released in 1985 (on NES-cartridge) is a bit problematic to analyze and use as a base for content generation since the implementation is optimized for space rather than readability. This basically means that in order to build something within the original implementation you have to first know the "hex value"[3] of the geometry you want to build, in what page (each level is implemented as a string of pages) you want to place it and where you want to place it in this page. The first level in *SMB* (World 1–Level 1, which we for future reference purposes will call W1L1) contains 13 pages and ends a few tiles after the flag pole[4]. The second piece of geometry you encounter as a player is (see figure 4) implemented as a horizontal brick with the length of five with two "Question mark"-block placed on top in the second and fourth tile (one containing a mushroom and the other a coin). This form of representation is effective when representing the horizontal and vertical block lines (including rows of coins), pipes (different heights), rock-stairs and the castles flagpoles. In fact, some geometry and enemies have double functions, as in the underwater setting in W2L2 and W7L2 (which can be changed to a level on land by manipulating a few hex values). In this case the vertical underwater vegetation becomes vertical bricks and some *Cheep-cheep* enemies becomes *Bullet Bills*. The horizontal green blocks becomes land-based blocks. The representation and optimization for *SMB* level geometry may have affected the design of the levels in a similar way that the limited memory capacity of *Atari 2600* affected the design of the games on that platform [26], since it is more costly in the *SMB*-representation (in terms of memory) to draw more objects and the cost of longer sections are relatively cheap (low or no extra cost compared to draw a single tile).

[3]A value represented in a positional numeral system with a base of 16 (where the symbols usually are 0-9 and A-F). A byte value is conveniently represented as 00-FF instead of 0-255.

[4]Including the first "empty" screen.

4. REPRESENTATION AND GENOTYPE-TO-PHENOTYPE MAPPING

In this section we present our approach to prototype a search-based level-generator for *SMB*. Our intentions were to apply an evolutionary algorithm that evolves levels containing the identified patterns. Three different representations were explored: $\rho 0$, $\rho 1$ and $\rho 2$. Since the method we apply is in the realm of stochastic optimization and metaheuristics the problem of how to represent the genotype (the data structure that the evolutionary algorithm acts on) and its relation to the phenotypes (the data structure that is evaluated by the fitness function) was given critical concern [38].

We initially approached the representation problem in the most direct way, thinking of representing levels as two-dimensional matrices with integer values for each block mapping directly to the phenotype ($\rho 0$). This would lead to a very large search space with only a small region consisting of playable levels. This idea was therefore quickly discarded. Next, we considered viewing the geometry of *SMB* as a range of integers spanning from 0 (representing a hole in the ground) to 10 (the maximum height of an obstacle in *SMB*). This representation will be referred to as $\rho 1$.

However this idea was abandoned when we inspected and compared the levels of the original *SMB* with our generated levels and noticed the scarce presence of ground based obstacles. In the original *SMB* most non-moving obstacles are combinations of rocks, pipes and land-based or mushroom-based platforms[5]. Apart from that, the enemies in *SMB* were not present in $\rho 1$ but gave some food for thought for the next idea of how to represent the genotypes when we tried to introduce them in this representation. The fact that a *Goomba* is placed in a certain piece of elevated geometry suggest an explosion of possibilities concerning a specific type of the genotype. Simply put; the need to differentiate between a specific enemy type (11 different ones) and the height (11 different ones) of a geometry type (12 different ones) grows towards a search space that is computationally expensive when we factor in the length of a level and the size of the population. This computational expensive solution may not pose a practical problem until one decide on applying this PCG solution in an online[6] situation with actual users and tries to generate content on the fly.

Thus, further studies of the original content of *SMB* (see section 3) led us to decide on a different approach than in our previous prototype (see section 2.3) approaching the content as individual pieces and thus keep with the pattern approach.

4.1 Vertical slices

[5]See *SMB* level W1L3 for an example of land-based platforms and level W4L3 for an example of mushroom-based platforms.

[6]I.e. during *runtime*.

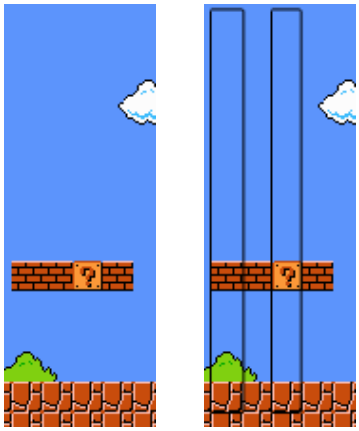

Figure 1: A simple 2-Path-pattern instance in *SMB* to the left. This can be reproduced with only 2 vertical slices indicated with black frames shown to the right.

Our current solution (which we refer to as $\rho 2$) is based on the idea to approach content from the perspective of *Mario* and not the view of the player. In the perspective of the player we travel from left to right but as *Mario* we travel *forward* one step at the time jumping onto objects with varying vertical placement. From this perspective the content of *SMB* can be viewed as vertical slices that together with other slices make up our previous suggested patterns.

Level genotypes are represented as strings of length 200 with an alphabet of 24 symbols. Each symbol corresponds to a vertical slice of Mario level with a length of 1 block and a height of 13 blocks. Levels (phenotypes) are constructed by simply appending vertical slices, giving all levels a length 200 blocks. The 24 slices used for the alphabet are representative samples from patterns extracted from the original *SMB*, mostly from W1L1 and L2.

Initially, we were concerned that since the vertical slices are not always compatible with each other, we might need an extra constraint checking function that would be time consuming to design, implement and complex to maintain and debug. However, with the *Mario*-viewpoint and the more detailed analysis of the content in the original *SMB* we concluded that the variation of vertical slices is surprisingly limited. If we observe figure 1 we have a section of *SMB* W1L1, that can be classified as a simple instance of the "2-path"-pattern. In our representation this section is simply a series of vertical slices of two types; the first one is used three times (in position 1, 2 and 4) and the second one is used once (in position 3). The two types contains a ground block at the lowest hight of the level and a brick-block or a question mark-block at height 4. In order to separate the instance of the pattern from other instance we also need a simple piece of ground at height 1.

See figure 3 for an explanation of how slices are appended to create levels.

4.2 Putting pieces together

In order to explain the vertical slices and how we combine them into patterns we will use an example with an instance of the *Enemy: 3-Horde*-pattern [11] (as in figure 2). The instance of the pattern could then be described as a sequence

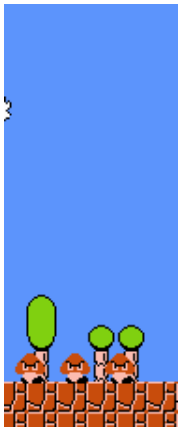

Figure 2: A 3-horde-pattern in the wild (*SMB* World 8 Level 1).

Figure 3: Adding vertical slices to form an instance of the pattern in figure 2.

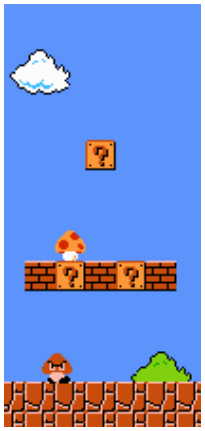

Figure 4: A 3-Path-pattern.

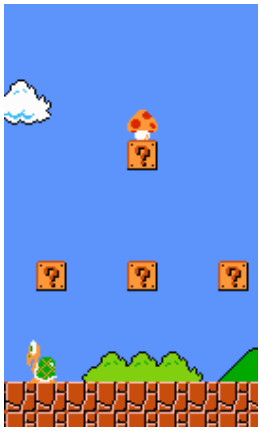

Figure 5: Another 3-Path-pattern.

of three identical vertical slices. Each of the slices are simple geometry (in the example; a ground-tile at "ground" level) with an enemy in a manner portrayed in figure 3.

4.2.1 *Example 1*

A simple 2-Path-pattern instance in *SMB* which can be reproduced with only 2 vertical slices (one slice with a brick-tile and one slices with a ?-block) see figure 1.

4.2.2 *Example 2*

By adding a vertical slice with two blocks (a brick-tile and ?-block) and reusing two vertical slices from figure 1 we get this instance of a 3-Path-pattern in figure 4.

4.2.3 *Example 3*

By adding a vertical slice with two ?-blocks and reusing a vertical slice from figure 1 we get this instance of a 3-Path-pattern in figure 5.

5. FITNESS FUNCTION

In our implementation (see figure 6) we use a fitness function to decide which level is the best suited to generate offspring and finally be chosen as the level to be played. In essence we perform a linear search through each member of the population and assign a fitness value to each member, the higher the value is (indicated as low, medium, high in

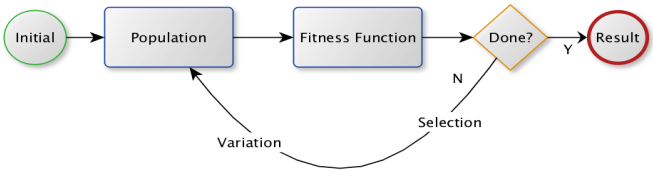

Figure 6: Principal execution of the level generator.

Table 2: Patterns supported in the fitness function.

Enemies	
Enemy	Low
2-Horde	Low
3-Horde	Low
4-Horde	Low
Roof	Medium
Gaps	
Gaps	Low
Multiple gaps	By stacking
Variable gaps	By stacking
Gap enemy	Low–Medium by stacking
Pillar gap	Pillar High
Valleys	
Valley	Low
Pipe valley	Medium
Empty valley	By stacking
Enemy valley	By stacking
Roof valley	By stacking
Multiple paths	
2-Path	Medium–High
3-Path	Medium–High
Risk and Reward	By stacking
Stairs	
Stair up	Low
Stair down	Low
Empty stair valley	Low
Enemy stair valley	By stacking
Gap stair valley	By stacking

table 2), the greater the chance of surviving the next generation is. The more complex a pattern is to replicate – the higher the fitness value. Some patterns are only supported by stacking of beginning and endings of patterns where the parts add up to a higher value (use "medium–high" as a value to compare with other values in table 2) except for *Gap enemy* which only need low–medium since it is a rather simple pattern to replicate with our symbol set.

If a level sequence contains a pattern[7] (see section 4.2) the individual population member gets a higher fitness value. If a level contains a sequence of symbols representing an instance of a pattern, like our example in figure 2, with three consecutive Goombas it is assigned a positive value. Similarly a sequence of rocks with increasing height is assigned a value depending on how long the sequence is. The fitness value assigned is higher if the pattern is uncommon[8] in a random sequence. Unplayable sequences are given a high negative number (but not $-\infty$) allowing breeding with a

[7]Identified previously [11].
[8]Or rather unlikely to appear.

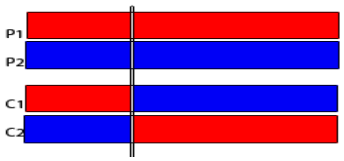

Figure 7: One-point crossover, where parent 1 (in red) and parent 2 (in blue) result in mixed-colored offspring child 1 and 2.

lower chance of survival in order to allow mutation or crossover keeping the good part of the genotype for another generation. Uninteresting sequences are given a low negative number in order to remove uninteresting parts of levels. We allow some uninteresting sequences, like a string of simple ground blocks in order to keep some kind of beat-like[9] [32] expression generated from this search-based approach.

The fitness function also contain beginnings and endings of patterns thus allowing stacking of patterns on top of each other. A beginning or ending is typically rewarded less than a full complex sequence. However, if a beginning, a full pattern and perhaps another beginning or ending is in a sequence this will give a cumulative higher value and thus solving the suggested improvement of stacked patterns in the previous prototype [11].

6. EVOLUTIONARY ALGORITHM

In each evolutionary run, we use 200 levels as representations of our population and each genotype is initialized as a uniformly random string of symbols drawn from the 24-character alphabet of vertical slices. We used a simple $\mu + \lambda$ evolution strategy with $\mu = \lambda = 50$ with a combination of mutation and one-point crossover as genetic operators[10].

Before any evolution operation is performed on the population it is evaluated according to a fitness function (see section 5). After that the population of 200 members are ranked according to its fitness value. The top 50 percent of the population are kept and the weakest 50 percent are discarded, thus leaving 100 level positions for evolutionary purposes.

We then let the top 50 percent breed with each other and so utilizing the "empty" positions in our population. The breeding is executed as a one-point crossover between pairs in ranking order, in such way that the best ranked is breed with the second in ranking, resulting in two new offspring, and so on. Our implementation of the one-point crossover has a fixed place for the crossover point in the middle of the parents' strings and from this point the strings are simply swapped with each other. In order to certify that we do not get stuck in a local maximum of the search-space we apply a simple mutation operation to the offspring by inject a new random character from our alphabet in a random position. Since we have the opportunity to run the level generator in offline mode our evolutionary search runs for 10.000 generations in the current version of the implementation.

7. EXAMPLES OF GENERATED LEVELS

[9]A rhythmic variation between exciting parts and calm parts where the player can regain energy to tackle the next exciting section.

[10]The one-point crossover is illustrated in figure 7.

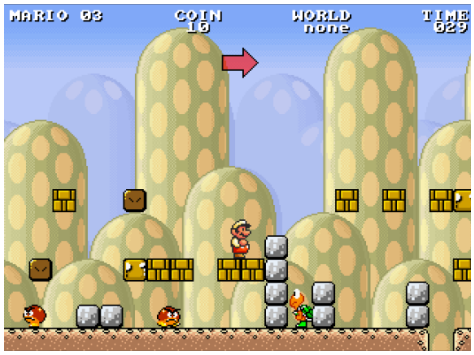

Figure 8: α-level showing tendencies to overfill levels.

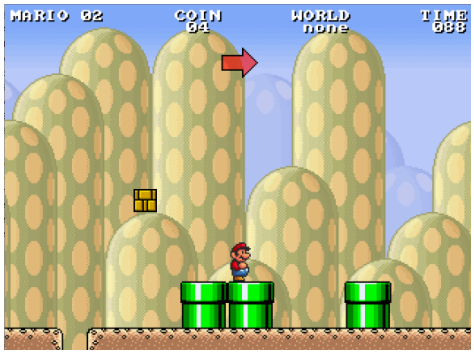

Figure 9: β-level showing tendencies to stack patterns.

The evolutionary approach together with vertical slices result in levels with both patterns, stacking of patterns and similarity to the original game (compare figure 8, 9, 10, and 11). However, our current implementation does not support the concept of beats *well enough*[11] where the content alternate between high-intense and low-intense parts of the levels. Adding better support to this might solve the tendencies to tightly stack patterns and overfilling game space as in figure 10 and 11. The reverse version of our approach α-level (see section 8) in figure 8 is overfilling the game space but at least our level generator does not do as bad.

8. EVALUATION

In order to get some feedback on our prototype we devised a simple play-test with three different levels generated from three different stand-points. We refer to the different levels as α, β and γ. Level α was using a reversed version of our fitness function that in principle, punished any form of pattern or beginning or end of a pattern. Level β was generated using our actual fitness function and level γ used a combination of our pattern-based prototype and imitated original content from *SMB*. The play-testers consisted of 24 experienced players (23 male, 1 female) in their twenties. The test platform consisted of ordinary but high-end PC:s with keyboards as control unit (UI). Our player feedback was gathered through a simple survey. In order to limit

[11]According to some comments by the play-testers. See section 8 for more play-test feedback.

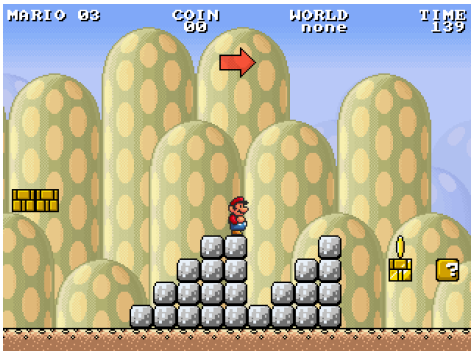

Figure 10: β sometimes stack patterns too close.

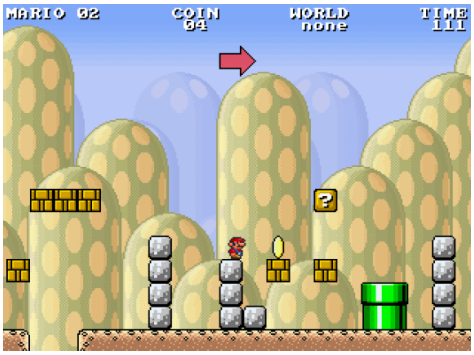

Figure 11: β almost overfill game space as α does.

bias on previous play-through of other versions three different groups were created with 8 individuals each playing the levels in different order (Group 1: α, β, γ, Group 2: β, α, γ, Group 3: γ, β, α.). Apart from an open-ended question regarding the overall experience with the level and some questions covering general information the play-testers supplied level-specific information on three questions on a 6-value scale. The level-specific questions were on; 1) Boring–Fun, 2) Not similar–Similar (to the original game) and 3) Easy–Hard (to complete).

The results show a marginal difference between the approaches our search-based approach (β) seemed to be slightly more fun than the others, more similar to the original than α but easier to beat than γ (see table 3).

9. DISCUSSION

Table 3: Results by level.

Version	Avr.	Median	Standard deviation
α Fun	3.75	4	1.041
α Similar	4.125	4	1.062
α Hard	3.5	3	0.791
β	3.792	4	1.159
β	4.625	5	0.906
β	2.375	2	0.989
γ	3.708	4	0.923
γ	4.833	5	0.875
γ	3.292	3	1.006

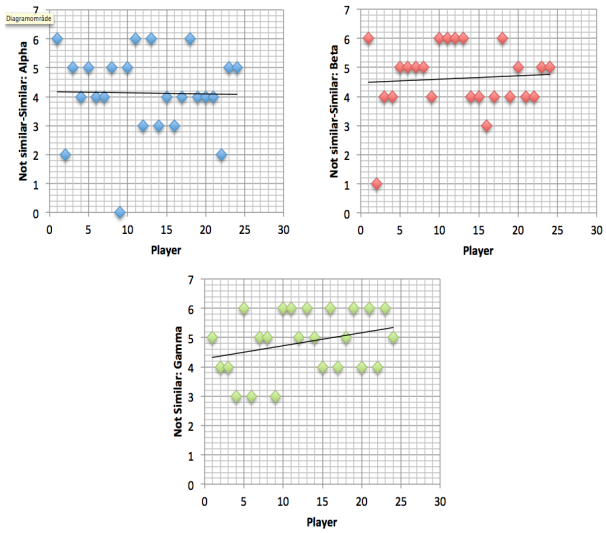

Figure 12: Not similar–Similar, Blue = α, Red = β and Green = γ.

Search-based optimization solutions like evolutionary approaches work well with patterns in regards to variation and can in our implementation solve the issue of being able to stack patterns. However, the fine-tuning of the fitness function may be problematic when introducing new patterns since the values for rewarding or punish the level can affect previous configuration. We suggest that other content analysis methods are applied when utilizing building-blocks smaller than beats or patterns because this can give an appropriate frequency of reward, enemies and geometry.

Since *SMB* has several levels with distinct *look–and–feel* in the different worlds and levels, we intend to implement a set of fitness functions that allow for generating world and level specific content. We have planned and prepared the next phase of the prototyping projects which will include a frequency analysis of game content in order to fine tune the fitness functions according to the different worlds and levels of the original *SMB*.

10. CONCLUSION

This paper has discussed how patterns can be used in procedural level generation, and in particular how they can be used as objectives rather than building blocks to start from. An implementation of the idea of patterns as objectives for generating levels for Super Mario Bros was presented. In this implementation, the original levels of *SMB* reoccur in two ways: as the fine-grained "vertical slices" that are recombined in the evolvable level representation, and the higher-level patterns that serve as objectives. Thus, the evolved levels retain much of the look and feel of original Mario levels, yet the generator can output a large range of diverse levels. An exploratory user study comparing levels that were generated with different fitness functions gave some indication that those that were evolved to maximize the number of patterns appear more similar to the original game than the others.

11. REFERENCES

[1] Acornsoft. Elite. [Digital game], 1984.

[2] E. Adams and J. Dormans. *Game Mechanics: Advanced Game Design*. Voices That Matter. Pearson Education, Limited, 2012.

[3] T. Adams. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. [Digital game], August 2006.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language – Towns, Buildings, Construction*. Oxford University Press, New York, U.S.A., 1977.

[5] Avalanche Studios. Just Cause. [Digital game], March 2006.

[6] S. Björk. Game Design Patterns 2.0. Web page, March 2013.

[7] S. Björk and J. Holopainen. *Patterns in Game Design*. Cengage Learning, 2005.

[8] Blizzard Entertainment. StarCraft. [Digital game], March 1998.

[9] D. Cermak-Sassenrath. Experiences with design patterns for oldschool action games. In *Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System*, IE '12, pages 14:1–14:9, New York, NY, USA, 2012. ACM.

[10] K. Compton and M. Mateas. Procedural Level Design for Platform Games. In *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference*, 2006.

[11] S. Dahlskog and J. Togelius. Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, DPG '12, pages 1:1–1:8, New York, NY, USA, 2012. ACM.

[12] H. Desurvire, M. Caplan, and J. Toth. Using Heuristics to Evaluate the Playability of Games. In *CHI 2004 Extended Abstracts on Human Factors in Computing Systems*, April 2004.

[13] Evolutionary Games. Galactic Arms Race. [Digital game], 2010.

[14] Firebird. The Sentinel. [Digital game], 1986.

[15] T. Fullerton. *Game Design Workshop - A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, New York, U.S.A., second edition, 2008.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, U.S.A., 1994.

[17] Gearbox Software. Borderlands. [Digital game], 2009.

[18] K. Hullett and J. Whitehead. Design Patterns in FPS Levels. In *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 78–85, New York, NY, USA, 2010. ACM.

[19] Introversion Software. Darwinia. [Digital game], March 2005.

[20] A. Kerr. *The Business and Culture of Digital Games: Gamework and Gameplay*. SAGE Publications, 2006.

[21] R. Khaled, M. J. Nelson, and P. Barr. Design Metaphors for Procedural Content Generation in Games. In *Proceedings of the 2013 ACM SIGCHI Conference on Human Factors in Computing Systems*, 2013.

[22] C. Lewis, N. Wardrip-Fruin, and J. Whitehead. Motivational game design patterns of 'ville games. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 172–179, New York, NY, USA, 2012. ACM.

[23] P. Mawhorter and M. Mateas. Procedural Level Generation Using Occupancy-Regulated Extension. In *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG)*, 2010.

[24] MicroProse. Civilization. [Digital game], 1991.

[25] Mojang. Minecraft. [Digital game], May 2009.

[26] N. Montfort and I. Bogost. *Racing the Beam: The Atari Video Computer System*. Platform Studies. MIT Press, 2009.

[27] Nintendo. Super Mario Bros. [Digital game], 1985.

[28] Re-Logic. Terraria. [Digital game], 2011.

[29] A. M. Smith and M. Mateas. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):187–200, 2011.

[30] G. Smith, R. Anderson, B. Kopleck, Z. Lindblad, L. Scott, A. Wardell, J. Whitehead, and M. Mateas. Situating quests: design patterns for quest and level design in role-playing games. In *Proceedings of the 4th international conference on Interactive Digital Storytelling*, ICIDS'11, pages 326–329, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] G. Smith, M. Cha, and J. Whitehead. A Framework for Analysis of 2D Platformer Levels. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 75–80, New York, NY, USA, 2008. ACM.

[32] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):201–215, 2011.

[33] Spectrum Strategy Consultants. From exuberant youth to sustainable maturity - Competitiveness analysis of the UK games software sector. Consultant report, DTI - Department of Trade and Industry, U.K., 2002.

[34] F. Spufford. *Backroom Boys – The Secret Return of the British Boffin*. Faber and Faber Limited, Croydon, U.K., 2003.

[35] Subset Games. FTL: Faster Than Light. [Digital game], September 2012.

[36] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 Mario AI Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2010.

[37] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis. Multiobjective Exploration of the StarCraft map space. In G. N. Yannakakis and J. Togelius, editors, *CIG*, pages 265–272. IEEE, 2010.

[38] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and Games*, 3:172–186, 2011.