A Procedural Method for Automatic Generation of Spelunky Levels

Walaa Baghdadi[1], Fawzya Shams Eddin[1], Rawan Al-Omari[1], Zeina Alhalawani[1], Mohammad Shaker[2] and Noor Shaker[3]

[1]Information Technology Engineering of Damascus, Damascus, Syria
[2]Joseph Fourier University, Grenoble, France
[3]IT University of Copenhagen, Copenhagen, Denmark
walaabaghdadi, fawziashamseddin91, rawan.alomari91, zeina.helwani,
mohammadshakergtr@gmail.com, nosh@itu.dk

Abstract. Spelunky is a game that combines characteristics from 2D platform and rogue-like genres. In this paper, we propose an evolutionary search-based approach for the automatic generation of levels for such games. A genetic algorithm is used to generate new levels according to aesthetic and design requirements. A graph is used as a genetic representation in the evolution process to describe the structure of the levels and the connections between the rooms while an agent-based method is employed to specify the interior design of the rooms. The results show that endless variations of playable content satisfying predefined difficulty requirements can be efficiently generated. The results obtained are investigated through an expressivity analysis framework defined to provide thorough insights of the generator's capabilities.

1 Introduction

Procedural Content Generation (PCG) is receiving increasing attention due to the advantages it provides in terms of speeding up the content generation process, enabling on-line generation, reducing the development budget and facilitating the creation of endless content variations [1]. Furthermore, since most PCG methods are based on extensive search in the content space [2], it is likely that utilising PCG approaches for generating content will yield novel solutions that can be directly used in the game [3, 4] or employed as a form of inspiration for human designers.

Different techniques have been explored to automatically generate different aspects of content for different game genres [5, 3, 4, 6] and some of them achieved remarkable results in commercial games [7–10]. *Rogue* is one of the early games where PCG is successfully employed to generate infinite variations of content as the game is being played. The game inspired many others and the automatic generation of dungeons is well investigated and used in several well-known games such as *Diablo* and *Dungeon Siege*. Most of the techniques used so far for dungeon generation however suffer from the lack of controllability as it is usually hard to specify design constraints or requirements and they mostly tend to produce neat structures [1].

One of the recent well-known commercial game that combines characteristics from the dungeon and platform games is *Spelunky*. The game successfully employes PCG

techniques to generate variations of structures that are unique with every replay. Such as most rough-like and dungeon generation methods, randomness forms the basis of diversity and hand crafted templates are used to control the level structures.

In this paper we present a procedural approach that allows the generation of variant content for a Spelunky-like game while permitting control over important content aspects. We analyse the game aspects and we employ the search-based approach of PCG [2] to generate game content. More specifically, a Genetic Algorithm (GA) is used to evolve game content where levels are represented using an indirect representation in the form of graphs that specify the navigation order of the rooms and the connections between them. A separated agent-based approach is then implemented to define the inner structure of each room. Rooms are then filled with different items according to a distribution scheme. We define a difficulty measure that scores levels according to their final structure and the presence of certain items and their placement. We show that infinite playable levels of varying difficulties can be generated and we present a thorough analysis of the results obtained.

2 Spelunky

Spelunky is an action adventure indie game that combines the characteristics of two genres: rogue-like and 2D platform games. The game was created by Derek Yu in 2008 as an open source game for PCs. An updated version of the game was released for Xbox Live Arcade which attracted millions of players.

The main game mechanic in Spelunky, similar to platformers, is jumping to collect items and to kill enemies. Much the same as in dungeon crawl games, the game is structured in rooms filled with collectable items and monsters. A common feature between Spelunky and most rough-like games is the presence of randomness which is the key element in generating the structure of the levels and placing enemies and decoration items (more details about the method used can be found in [1, 11]).

The player controls *Spelunker*, the main character of the game. To win the game, the player should possess good playing skills as well as being able to efficiently manage different types of resources such as ropes, bumps and money.

Spelunky exhibits a number of properties that motivates exploration of the applicability of PCG and AI methods. It is a 2D game that combines properties from platform games in terms of gameplay with the graphical representation and layout of rogue-like games. Automatic generation of content for such a game is therefore an interesting problem as one should consider the characteristics of both genres. The game is also receiving an increasing interest in research as a benchmark for computational and artificial intelligence algorithms was recently proposed around this game [11].

3 Modifications to The Original Game

We used the source code available for the first published version of the game as a base for our implementation. We made several modifications to the original code to allow complete automatic level generation with desired difficulty. Our modifications include adding an extra goal to the game, namely, Spelunker should save the princess; this

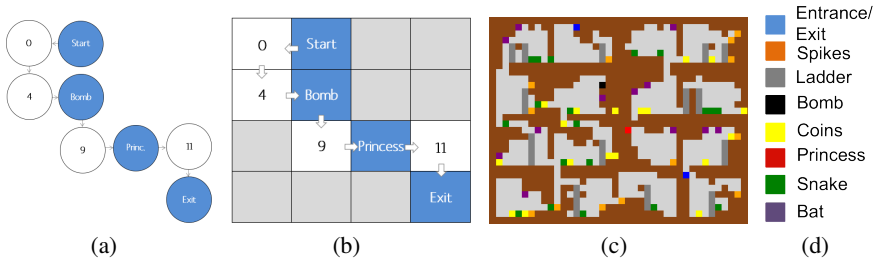


Fig. 1. (a) An example of an evolved graphical representation of a level and (c) the actual corresponding level map. The key rooms are presented in blue in (b) and the path is presented as direct links between the rooms. A gray room can contain anything. The different colours in the resultant level stands for different types of items as presented in the legend.

requires finding and carrying her along while navigating safely to the exit. We also chose to lock the princess in a closed room to make the game more interesting since this necessitate searching for bombs to break the walls and enter the room. The bombs are placed in hidden places in one of the 16 rooms of the level.

In short, we are using the same art assets used in the original game but the gameplay in our version, called *InfiSpel*, consists of starting the journey in the entrance room, navigating through the level searching for the bomb, locating the princess and using the bomb to enter her room, carrying the princess and heading to the exit while overcoming challenges such as monsters and traps. These new requirements entailed heavily modifying the original source code.

4 Evolving content in InfiSpel

In our level generation approach, a Genetic Algorithm (GA) method is employed to evolve content. When using GA, there are two main factors that are essential to the quality and performance: content representation and content quality.

4.1 Level Representation

The phenotype (level structure) is represented as an integer vector where each level consists of 16 rooms organised in a 4×4 matrix (such as in the original game). This representation is used to visualise the generated levels and to measure their quality.

Since we are interested in the placement of items and the presence of paths between the rooms and from the entrance to the exit, the genotype is represented indirectly where the complete level is represented as a graph in which each room is a node. The links between nodes are translated into direct connections between rooms while a missing link indicates a wall. This means that starting in the entrance room and following the links should lead to the exit in a playable level. More specifically, the genotype is a vector of codons carrying the following information: five integers identifying the start room, the bomb room, the princess room, and the exit room, the length of the path from

the entrance to the exit (measured as the number of rooms passed). These are followed by fields for storing the total number of monsters, the total number of collectable items and a list of connections. The total number of connections between the rooms is 28 (notice that we don't allow loops) and therefore the list of connections consists of 28 binary slots. Fig. 1 presents an example level with its graph representation.

4.2 The Interior Design of Rooms

To construct the layout of each room, a digger algorithm is used and applied on each room separately. This method has been previously used to construct maps for First-Person Shooter games and showed fast performance and interesting results [12, 13]. The method is used in a similar way in this paper. Initially, each room is filled with bricks (walls) with all cells having the value 1. The digger moves in the room switching the value of some of the cells from 1 to 0 hence generating walkable areas.

To dig one room, the digger performs the following steps:

1. The digger agent is placed in the center of a room.
2. The agent randomly choses one of the following directions for his next move: right, left, up and down. The agent moves in the direction chosen and change the value of the destination cell from 1 to 0.
3. The above process is repeated until a maximum number of moves is reached (35 moves is used in our implementation).
4. Walls are then digged, if necessary, to ensure a path between the start and the end rooms.

This method is repeated for each of the 16 rooms in each level resulting in rooms with various structures. Corridors between rooms are added later in the process according to the structure of the level evolved by GA.

4.3 Content Quality

Designing an interesting level that is fun to play is the ultimate goal when generating game content. Therefore, measuring the quality of the generated content is vitally important. This task is not obvious given that there is no universal agreement of what makes a good level or how to measure the "goodness" of a piece of content. Several attempts can be found in the literature on identifying the properties that should be present in a level to make it fun [14–16]. Several researches have employed these theory to generate content that is fun to play [5, 17]. In our system, we based our definition of an interesting level design on two factors: the first contributes to a set of design requirements that we found to be important, while the second factor relates to the difficulty of a level which affects the challenge presented to the player which proven to be an important aspect for an optimal experience [16, 18]. Generated levels are scored according to these two factors as follows:

$$fitness = 10\% * G_{score} + 90\% * D_{score}$$

where G_{score} is a measure of the quality of the level design while D_{score} assigns a fitness to the level according to its difficulty. The weights of these two factors are assigned experimentally after generating a number of levels, visualising them and tuning the values.

The following paragraphs explain how the G_{score} and D_{score} values are calculated.

Design Constraints, G_{score}: A score value is assigned to each level according to its final design. We identified a number of requirements, most of which contribute to playability and/or aesthetics, that should be satisfied in a level. These are the followings:

- Placement of the entrance room, P_s: in Spelunky, the entrance room should always be one of the rooms in the first or second row.
- Placement of the exit room, P_e: the exit room should be one of the rooms in the last two rows.
- Connections between mandatory rooms, C: a player in our game should be able to navigate to the exit going through the bomb room then the princess room. Therefore, for a level to be playable, there should be a path connecting these rooms directly or indirectly.
- Uniqueness of mandatory rooms, U: there shouldn't be more than one instance of each of the four mandatory rooms in each level. These include the start, exit, princess and bomb rooms. A level can not also contain any room that combines two of these features, for example, the princess can't be placed in the exit room.

The total design score of a level is calculated as follows:

$$G_{score} = 20 * P_s + 20 * P_e + 35 * C + 25 * U$$

If a level passes a certain predefined threshold (G_{score} is higher than 90%), the process continues to evaluating its difficulty. A high threshold is used since a level that breaks any of the above conditions is very likely to be either unplayable or uninteresting. The weights assigned to each of the above conditions are chosen experimentally.

Spicing up the Level: Enemies and Items Distribution: After generating the physical structure of a level and before calculating its difficulty, several auxiliary items are added to complete the level design. These include: enemies (including bats and snakes that are placed randomly or around gaps), traps such as spikes, resources such as bombs and ropes, ladders which are used to connect vertically adjacent rooms and coins and rubies.

In order to maintain a fair distribution of items over the whole map, the map is divided into 2×2 areas each containing 2×2 rooms. The items of each type are distributed in all areas equally. This is guaranteed by generating a list of possible positions in each region where an item of each type can be placed. An item is then placed at a randomly chosen position from this list.

Level Difficulty: The final phase after generating the level and adding different items is to assess its difficulty through evaluating the following conditions:

(a) Difficulty = 10% (b) Difficulty = 50% (c) Difficulty = 90%

Fig. 2. Three sample levels of increasing difficulty.

- Path length, P_l: the longer the path Spelunker should navigate to successfully reach the exit, the more difficult the level is since this requires facing more enemies and overcoming more obstacles. The length of a path should be at least 4 rooms (navigating only the four mandatory rooms).
- Vertical corridors, V_c: the presence of vertical corridors of a long length to connect two or more rooms makes the level harder to navigate since this necessitates the use of ropes or the existence of ladders. Otherwise, it is very likely that Speluker will lose a life due to falling a large distance.
- Number of Spikes, N_{sp}: spikes are special complication items that Spelunker can walk safely through but falling on them leads to a lose of life. The more the spikes the harder the level is.
- Number and type of enemies, E: as the the number of enemies increases, the level becomes more difficult. Bats, N_b, are given the highest weight since they are the most dangerous as they move around and follow Spelunker when he is in a close distance. A lower weight is given to the placement of snakes around gaps, N_{sg}. The lowest weight is given to the presence of snakes, N_s.

The difficulty score is measured as a weighted sum of the normalised values (using min-max normalisation) of all of the above factors according to the equation:

$$D_{score} = 20 * P_l + 15 * V_c + 15 * N_{sp} + 20 * N_b + 20 * N_{sg} + 10 * N_s$$

The weights are chosen experimentally according to how the elements affect the difficulty of a level (the weight of the number of snake around gabs, N_{sg}, for instance, is higher than the weight assigned to the total number of snakes, N_s).

Three levels of increasing difficulty can be seen in Fig. 2. The figure clearly shows that the increase in difficulty is associated with the presence of more enemies, more vertical corridors, and paths of longer length.

5 Implementation Setup

The GA experimental parameters used are the following: 100 runs of 200 generations with a population size of 100 individuals. The mutation probability is 0.05, and we used

Table 1. The average time and the number of generations required to evolve levels of different difficulties.

Difficulty	$Time(sec)$	$\#Generations$
10%	16.3 ± 5.32	117.13 ± 46.37
50%	0.57 ± 0.39	6.76 ± 4.49
90%	18.45 ± 1.62	110.58 ± 49.48

two-point crossover with probability equals to 1. Tournament selection is used to reproduction. The stopping condition is to reach a predefined fitness, otherwise evolution continues for 200 generations.

6 Results and Evaluation

An experiment is conducted to evolve levels using the framework proposed. Evolution is repeated for 100 runs starting from a random population each time. Different levels of difficulty are specified and levels are evolved accordingly. This is done by scoring the evolved levels according to the equation:

$$fitness = 10\% * G_{score} + 90\% * (1 - Dis_{diffScore})$$

where $Dis_{diffScore}$ is the difference between the difficulty of the evolved levels and a target difficulty value.

The analysis showed that only 46% of the total individuals passed the threshold specified on the design (having a G_{score} higher than 90%). Those levels were further evolved and evaluated for difficulty.

The amount of processing time and the number of generations required varies significantly when evolving levels of different predefined difficulty values. Table. 1 presents the average time required and the total number of generation reached when evolving levels that are 10%, 50% and 90% difficult. The results show that generating levels of medium difficulty ($diff = 50\%$) is the easiest. On the other hand, levels that are easy or hard ($diff = 10\%$ or $diff = 90\%$) require a longer evolution process and therefore significantly more time.

7 Expressivity Analysis

To evaluate our content generator and explore its capabilities, we run an expressivity analysis that helps us better understand how our system works and elaborate on its strengths and weakness. The expressive range of a generator is the space of all levels it can generate [19]. It can be measured by generating a large number of representatives of the generator's output, defining expressive measures that capture the variations in the outputs along different dimensions, scoring the content according to the defined measures and visualising the results. In what follows, we describe several expressivity measures that we defined to analyse our generator. Some of the measures are inspired by previous work on expressivity analysis [19].

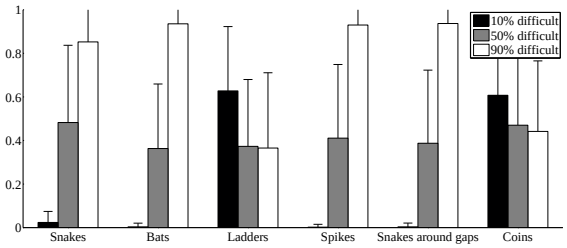
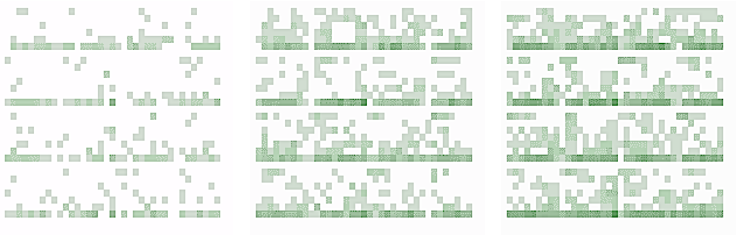


Fig. 3. The average values of generated items for 100 levels of different difficulties.



(a) Difficulty = 10% (b) Difficulty = 50% (c) Difficulty = 90%

Fig. 4. Colour maps for the distribution of snakes in 100 levels of increasing difficulties.

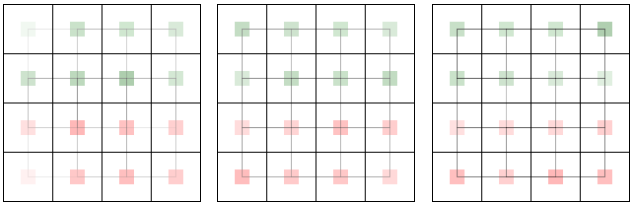
7.1 Frequency Analysis

The simplest and most compact method of showing the generator's characteristics is through calculating simple statistics about the components' frequency. Fig. 3 presents a comparison between the average and the standard deviation normalised values (using min-max normalisation) of key items in 100 playable levels evolved for three difficulty scales. As can be seen, as the levels become more difficult an increasing number of enemies of different types with less ladders and coins are generated.

7.2 Color Map

To facilitate a more in-depth insight on the differences between the generated levels, we converted them into colour maps and projected them on one image. The resultant colour map is an image where the value of each pixel is the average colour value of all pixels at the same position in the full set of levels generated.

Analysis of the colour map can be done more clearly if they are generated for each item separately since this permits illustration of the distribution of the different items independently. Fig. 4 presents three maps for the distribution of snakes in 100 levels of different difficulties. The two main interesting observations are the increase in the number of snakes as the difficulty increases and the fair distribution of snakes along all rooms.



(a) Difficult = 10% (b) Difficult = 50% (c) Difficult = 90%

Fig. 5. The colour maps for the generated paths for 100 levels of different difficulties. The entrance room is presented in green, the exit room in red and the black lines corresponds to the connections between the rooms.

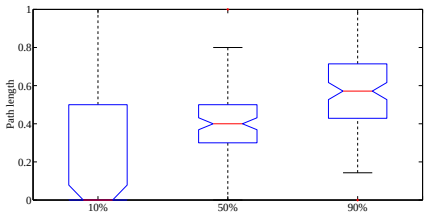


Fig. 6. The boxplot for generated path lengths for levels of increasing difficulty.

7.3 Visited Rooms and Generated Paths:

In order to plot the generate paths in a colour map, the entrance and exit rooms are given district colours (green and red, respectively) and the connections between the rooms are represented as solid lines.

The paths generated for multiple levels can be viewed in one image illustrating the variations in the designs according to the difficulty. Fig. 5 presents the resultant path maps of 100 levels for various difficulty configurations. The figure shows that as the difficulty increases more rooms become part of the path as indicated by a higher frequency (darker colour) of visited rooms and darker links between the rooms. For further analysis, we calculated the average and standard deviation values of the lengths of the paths generated for different difficulty setups and the results are presented in Fig. 6. The figure clearly illustrates longer path lengths as the difficulty increases. Notice that a path of length 0 means that only the minimum number of rooms (in this case 4) should be traversed to reach the exit. Fig. 7 presents the histogram for the number of rooms in the paths generated in 100 levels for three difficulty scale. The figure shows a clear bias towards generating longer paths as the difficulty increases.

7.4 Shortest versus Winning Path Length

The shortest path is the number of rooms connecting the entrance to the exit (without necessarily passing through the bomb and princess rooms). Notice that following this path means reaching the end of the level but not winning the game. It is interesting to

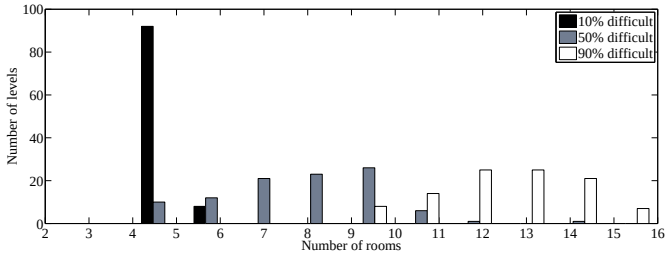


Fig. 7. The number of rooms in the paths generated in 100 levels for different difficulty scales.

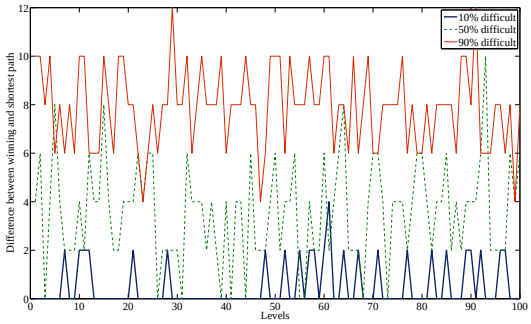


Fig. 8. The differences between the length of the winning paths generated and the corresponding shortest paths in 100 evolved levels of different difficulties.

compare whether and by how much this path differs from the actual path the player has to follow to win the game, which we call the *winning path*, since this gives an indication of the efficiency of the evolution algorithm and its success in generating levels that deviate from being obvious.

Fig. 8 presents the differences between the shortest and winning paths generated for 300 levels of increasing difficulty. The figure shows that for easy levels (10% difficult), most of the generated winning paths are of the same length or slightly longer than the shortest possible paths. The difference however becomes larger as the levels become harder. The average observed differences between the shortest and the winning path lengths are 0.5, 3.56 and 7.88 rooms for levels that are 10%, 50% and 90% difficult, respectively.

8 Conclusions

The paper presents a methodology for procedurally generating complete playable levels in games similar to Spelunky. A genetic algorithm is implemented to evolve game content. Level maps are represented as graphs where the nodes, the connections and the other design-specific properties are evolved. Rooms' inner structure is constructed by an agent-based approach and a distribution method is employed to place collectable

items and enemies. Content quality is measured through a fitness function that scores levels according to how well they match predefined design and difficulty requirements. Finally, the evaluation of the system consists of defining and running a number of expressivity measures on 100 levels evolved for different settings. The results show that interesting playable content that satisfies our design requirements can be evolved and that using the proposed approach, levels of desired difficulty can be efficiently generated.

The suggested approach can easily scale to other games from the rouge-like and dungeon crawl genres that exhibit similar representation where levels are structured in rooms filled with monsters and rewarding items and connected via corridors.

An issue that we did not investigate in this paper, and is important in game design, is the amount of variations between the content evolved for the same setting. We focused in our expressivity analysis on the differences between the levels evolved for various experimental setups with minor analysis on the dissimilarity between the levels within each category. An interesting future work will be defining new measures that capture the diversity in the designs generated along more than one dimension to draw more robust conclusions and to improve the generator.

One way of taking this work one step further and rewarding content diversity is to explore the use of novelty search methods [20]. This approach has recently been used with promising results in many domains including computer games [21, 22]. An enhancement of this approach was recently suggested through combining it with a two-population feasible-infeasible [23]. The method can be employed in the framework proposed where individuals that satisfy a set of requirements are placed in the feasible population while the infeasible population contains those that break some of the constraints (in our case this might be invalidating the design constrains or being judge as unplayable). Evolution can then be performed on the two populations towards generating novel, yet playable solutions.

Another interesting future direction is to incorporate the player in the evolution process. In the current approach, players' preferences, how they perceive the evolved content and how the content affects their experience are not considered. Given that the ultimate goal of game design is to please players, indicators about the "goodness" of content to specific players are essential for generating high quality content. Therefore, future directions will also investigate ways of including the player in the content generation loop so that, for example, the difficulty of the next generated level is set according to its performance in a previous level or through an interactive approach where content are presented to players and evolved based on their reported, or measured, preferences.

Acknowledgments

The research was supported in part by the Danish Research Agency, Ministry of Science, Technology and Innovation; project "PlayGALe" (1337-00172).

References

1. Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.

2. J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation. In *Proceedings of EvoApplications*. Springer LNCS, 2010.
3. E. J. Hastings, R.K. Guha, and K.O. Stanley. Evolving content in the galactic arms race video game. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, pages 241–248. IEEE, 2009.
4. N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. ONeill. Evolving levels for super mario bros using grammatical evolution. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, 2012.
5. J. Togelius, R. D. Nardi, and S. M. Lucas. Making racing fun through player modeling and track evolution. In *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.
6. Mohammad Shaker, Noor Shaker, and Julian Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.
7. Blizzard North, 1997. Diablo, Blizzard Entertainment, Ubisoft and Electronic Arts.
8. Mojang, 2011. Minecraft, Mojang and Microsoft Studios.
9. Maxis, 2008. Spore, Electronic Arts.
10. D. Yu and A. Hull, 2009. Spelunky, Independent.
11. Daniel Scales and Tommy Thompson. Spelunkbots api-an ai toolset for spelunky. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
12. Luigi Cardamone, Georgios N Yannakakis, Julian Togelius, and Pier Luca Lanzi. Evolving interesting maps for a first person shooter. In *Applications of Evolutionary Computation*, pages 63–72. Springer, 2011.
13. Noor Shaker, Mohammad Shaker, Ismaeel Abuabdallah, Mehdi Zonjy, and Mhd Hasan Sarhan. A quantitative approach for modeling and personalizing player experience in first-person shooter games. 2013.
14. R. Koster. *A theory of fun for game design*. Paraglyph press, 2004.
15. T. Malone. What makes computer games fun? ACM, 1981.
16. J. Chen. Flow in games (and everything else). *Communications of the ACM*, (4):31–34, 2007.
17. Nathan Sorenson and Philippe Pasquier. The evolution of fun: Automatic level design through challenge modeling. In *Proceedings of the First International Conference on Computational Creativity (ICCCX). Lisbon, Portugal: ACM*, pages 258–267, 2010.
18. P. Rani, N. Sarkar, and C. Liu. Maintaining optimal challenge in computer games through real-time physiological feedback. In *Proceedings of the 1st International Conference on Augmented Cognition, Las Vegas, NV*, pages 184–192, 2005.
19. Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the mario ai framework. 2014.
20. Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.
21. J-B Mouret and Stéphane Doncieux. Encouraging behavioral diversity in evolutionary robotics: an empirical study. *Evolutionary computation*, 20(1):91–133, 2012.
22. Brian G Woolley and Kenneth O Stanley. Exploring promising stepping stones by combining novelty search with interactive evolution. *arXiv preprint arXiv:1207.6682*, 2012.
23. Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Enhancements to constrained novelty search: Two-population novelty search for generating game content. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 343–350. ACM, 2013.