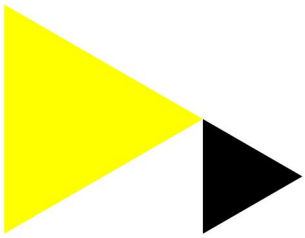
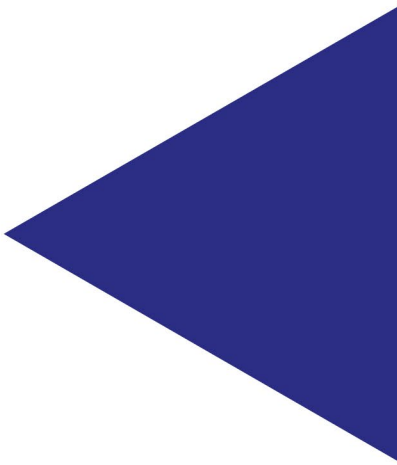
Procedural generation of collaborative puzzle-platform game levels





Procedural generation of collaborative puzzle-platform game levels

Benjamin van Arkel, Daniël Karavolos, and Anders Bouwer
Amsterdam University of Applied Sciences
Wibautstraat 2-4
1091 GM Amsterdam
The Netherlands

KEYWORDS
Game Design, Procedural Content Generation, Generative Grammars, Automated Game Design, Level Generation, Game Mechanics for Two-Player Collaboration

ABSTRACT

This paper addresses the procedural generation of levels for collaborative puzzle-platform games. To address this issue, we distinguish types of multiplayer interaction, focusing on two-player collaboration, and identify relevant game mechanics for a puzzle-platform game, addressing player movement, interaction with moving game objects, and physical interaction involving both players. These are further formalized as game design patterns. To test the feasibility of the approach, a level generator has been implemented based on a rule-based approach, using the existing tool called Ludoscope and a prototype game developed in the Unity game engine. The level generation procedure results in over 3.7 million possible playable level variations that can be generated automatically. Each of these levels encourages or even requires both players to engage in collaborative gameplay.

1. INTRODUCTION

Procedural content generation (PCG) is the algorithmic creation of content for games, such as assets, levels, worlds, and even whole games. PCG has been part of published computer games since the eighties. Prominent examples include Rogue, Diablo, and Minecraft, among others. Recently, PCG is receiving increasingly more academic attention (Togelius et al. 2011, Hendrikx et al. 2013, Shaker et al. 2014). PCG has been used for various reasons, including working around memory constraints in past decades when hardware was less powerful, increasing replayability by generating variation, making the game development process more efficient, exploring - and perhaps enlarging - the game design space, and formalizing the rules of game design.

Research on the application of PCG for platform games has focused on creating levels for the game Super Mario Bros (Shaker et al. 2011), with its typical obstacles, enemies, and straight levels. However, little work has been done on generating platformer levels for multiple players.

In this paper, we explore the potential of PCG to generate levels for puzzle-platform games that involve two-player interaction, in particular, collaboration. We will define game design patterns, which we will then translate to level segments which create gameplay situations featuring certain game mechanics. Similar to Dahlskog and Togelius (2013), we will let a generator combine these level segments to create a level. However, we will use generative grammars to generate the levels instead of evolutionary algorithms.

In section 2, we consider related work on procedural level generation. Section 3 identifies game mechanics that specifically address collaboration, and section 4 describes relevant game design patterns for these. Section 5 gives an overview of the implementation of the whole generation process. Section 6 offers a number of points for discussion and section 7 presents our conclusions and ideas for future work.

2. RELATED WORK

Although many approaches to procedural level generation are tied to specific games, p.e. (Shaker et al. 2011, Smith et al. 2011, Shaker et al. 2013, Ferreira and Toledo 2014), there are some tools available that are more general in their approach. For example, LudoScope (Dormans 2011; 2012, Dormans and Leijnen 2013), is a tool that allows the transformation of design concepts about missions and space to concrete game levels for playable games, automatically, or semi-automatically in interaction with a human designer (Karavolos et al. 2015). It is based on the principles of model driven engineering and generative grammars. In order to generate levels for a specific game, the designer makes a model of the generation process, breaking it down into steps (modules) that can be executed separately. We will use this tool to generate the levels for our puzzle-platformer prototype.

There are several ways to characterize multiplayer games. Zagal et al. (2006) distinguish three different types of multiplayer games, based on the interaction between players: competition, cooperation and collabo-

ration. Competitive games require a player to confront other players in the game. In such games, players have opposing goals. In cooperative games, opportunities exist for players to work together, from which both players could benefit. However, "a cooperative game does not always guarantee that cooperating players will benefit equally or even benefit at all" (Zagal et al. 2006; p. 2). Collaborative games differ from cooperative games in that the players have the same goal, and share the rewards or penalties of their decisions, whereas in cooperative games players may have different goals, and achieve these goals independently from each other. So,"the challenge for players in a collaborative game is working together to maximize the team's utility"(Zagal et al. 2006; p. 3), while in cooperative games players only have to consider their own utility. Considering non-competitive multiplayer games, generating levels for collaboration is the more challenging type, because the forced mutual benefits of the cooperation puts an extra constraint on the design space.
The generation process is based on the idea of generating a path within a space and subsequently filling this with rooms containing more concrete gameplay sections. This method partially draws inspiration from methods used for level generation in the game *Spelunky* (Kazemi n.d.), and Karavolos et al. (2015). Design choices like using numbers to define certain types of rooms is something that we have applied to our project as well. However, while Spelunky does not pay much attention to the layout of the individual rooms (entrances and exits being the only important element), we want our level generator to generate levels with a set path in mind that we want the players to follow.

3. GAME MECHANICS FOR COLLABORATIVE PLATFORMERS

A platform game requires the players to be able to jump, so they can traverse platforms. To create a puzzle-platformer, this requirement must be paired with a certain mechanic that creates interactive puzzles that the players can solve. For basic gameplay, the players must be able to run, jump and interact with objects in the game space in different ways, p.e. activating a lever to move platforms or being able to pick up a player or object to change the required jump height or width. Other typical means to challenge the player's skills and add variation to the gameplay are the addition of enemies and dangerous obstacles.
The prototype in this paper incorporates additional objects that affect player movement, i.e. trampolines, moving platforms and grabbable elevators. These objects affect the pacing of the game, and serve to make the platforming aspect of the gameplay more interesting for the players. The prototype also contains a typical implementation of puzzle mechanics, players will be required to work together to solve lock-and-key puzzles. However, the lock-and-key mechanism is implemented as a gate that is opened by pressing a lever.

Most of these game elements require only one player to achieve their effect on the gameplay. The collaborative mechanics involve the lock-and-key mechanism and using each other's head as a platform to reach a higher place within the level, and will be described by the patterns in the next section.

4. GAME DESIGN PATTERNS

There have been several attemps to formalize game design ideas into patterns. Some of these patterns focus on gameplay sections of a level (Reuter et al. 2014, Hullett and Whitehead 2010), others focus on game mechanics (Rocha et al. 2008, Seif El-Nasr et al. 2010). Dahlskog and Togelius (2012) even extract patterns in the form of level segments from hand crafted game levels in order to generate levels with the same style.
The design patterns in this study are defined manually, and inspired by the templates of Reuter et al. (2014) and Hullett and Whitehead (2010). However, they are related to the other patterns as well. For example, the upsy-daisy and the timed lever/gate pattern are a type of 'Shared Puzzles' as defined by Seif El-Nasr et al. (2010).

Defining game design patterns and their instantions as level segments allows us to view the level as a sequence of patterns, or combinations of patterns. Then, the generator can combine these patterns in order to design gameplay situations. Because of space constraints, we will only describe the collaborative patterns we found. For each pattern we give a description of what the pattern entails, as well as possible affordances for the pattern and the resulting consequences for the player experience. Listed below are a three of these game design patterns:

- **The Upsy-Daisy**
  Description: An obstacle that requires two players to time their jumps together to get onto a platform. After this, the player can push an object down to the other player so they can both get up and proceed further through the level (See figure 1).
  Affordances:
  - Obstacles near the platform that needs to be reached.
  - Distance of platform above the ground.
  Consequences: Causes a sharp increase in challenge and coordination for the players. Both players must time their jumps together so that one of the players can use the other players head to get high enough to reach the platform.

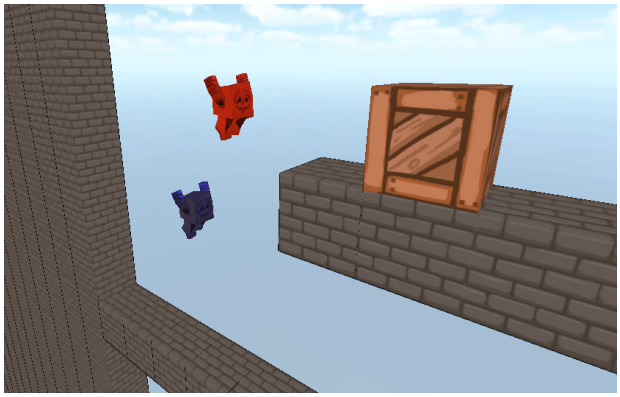


Figure 1: An example of the upsy-daisy pattern.

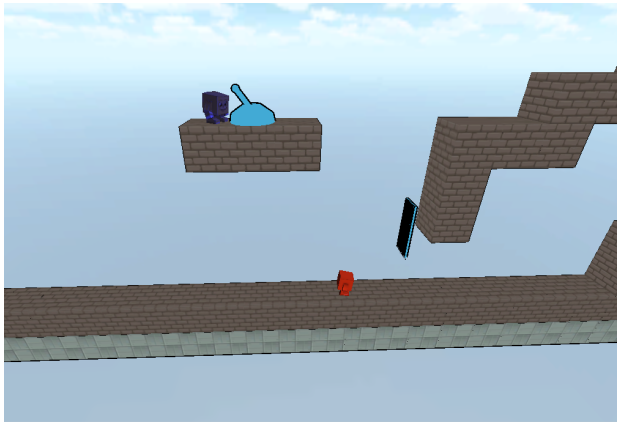


Figure 2: An example of the timed lever-and-gate pattern.

- **Timed Lever-and-Gate**
  Description: A lever that, when pulled, opens the related gate and starts a timer. When the timer runs out, the lever is reset and the gate closes (As seen in figure 2).
  Affordances:
  - Amount of time before lever resets
  - Number of obstacles between the lever and the gate
  Consequences: An increase in challenge and coordination because of the fact that the players must communicate about which player will go through the gate and which is going to pull the lever.

- **Common Enemy**
  Description: An enemy that cannot be destroyed if the player confronts it alone, since the sides that are facing the player are invulnerable (See figure 3).
  Affordances:
  - Number of enemies placed
  - Amount of health of the enemy
  - Damage done when hit by the enemy
  - Speed of the enemy.
  Consequences: Confronting a common enemy offers a greater challenge than a regular enemy, as it requires more skill to draw the enemy to one player, while the other player hits its blind spot. This also results in a greater sense of competence when the players manage to defeat the enemy. Furthermore, this pattern demands coordination between the two players, since they will need to communicate which player lures the enemy and which one attacks it.

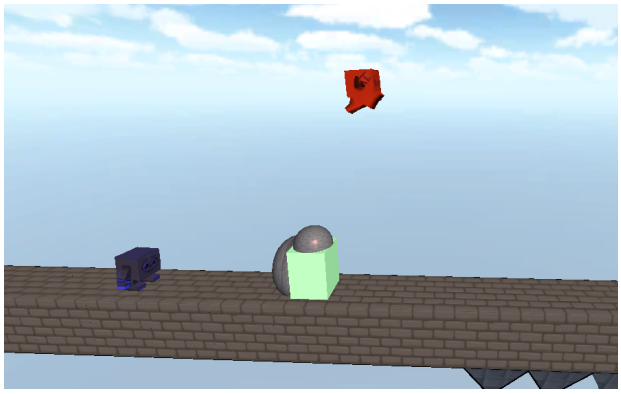


Figure 3: An example of the common enemy.

5. THE LEVEL GENERATION PROCESS

The level generation process is based on a sequence of generative grammars. Each grammar receives input, performs its transformations and sends the output to the next grammar, until the level is finished. The first input is a 6x3 grid of tiles, which are all of the type undefined, except for the leftmost column, which are of the type 'start'. These tiles define the possible positions of the level segment with the initial spawn location of the player. The first grammar transforms two of the three tiles into an undefined tile and transforms one of the tiles adjacent to the remaining tile into an 'end' tile. The next steps of the process, which will be described in more detail in the following subsections, are as follows:

- Path Generation
- Define Level Segments
- Apply Design Patterns
- Final Adjustments

Path generation

We generate a path from left to right in two passes. First from left to right, then from right to left. We split this process into two steps, because grammars are context-free systems. By marking the orientation of each new tile that is generated we add context to a context-free system and can guarantee a connecting path.

Figure 4: During the first pass, the generator creates a path from the left column to the right column. This path is marked, so the grammar will not attach level segments to these segments during the second pass. The abbreviation of each tiles signifies the orientation that the tile will have. 1H stands for a horizontal segment and 1V a vertical one. A tile such as LCD stands for Left-Corner-Down, meaning the player would enter the segment from the left side of the tile and move through a corner, exiting on the right. UCR stands for Up-Corner-Right, DCR for Down-Corner-Right, et cetera.

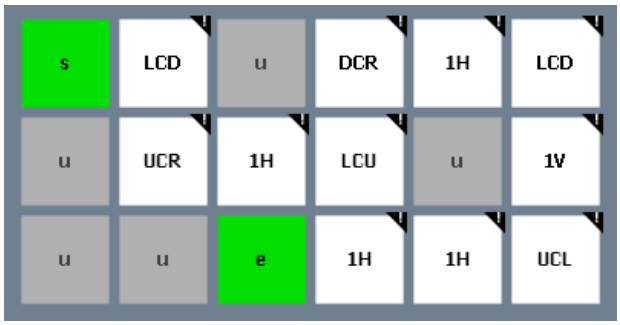


Figure 5: During the second pass, a path to the left is generated. In contrast to the first pass, the length varies due to stochastic rules. The marks of the highlighted level segment indicates that the player moves from right to left in this segment, and that it is the final level segment before the end tile.

Define Level Segments

After the generation of the path is completed, each tile is expanded into a 20x20 tile "segment". These segments depict the first step in visualizing what the final level will look like. Each level segment has several templates, which are chosen on the basis of what the segment had been marked as in the previous step. While generating these segments, some might contain encounters. Encounters are the term used to describe instances of design patterns within the level generation process.

Apply Design Patterns

Applying the design patterns to our level generation process is done through the use of what we call encounters. Encounters are used to apply instances of design patterns within the level generator, as shown in figure 6. The larger level segments contain locations for encounters. These encounters are smaller level segments which contain challenges involving certain mechanics. To create an added layer of depth to the generation of the design patterns, it is also possible for an encounter to contain other encounters. So a movement-based jump encounter could contain smaller danger encounters within it. Thus, the generator can create level segments that pose both a movement-based challenge, as well as a danger-based challenge.

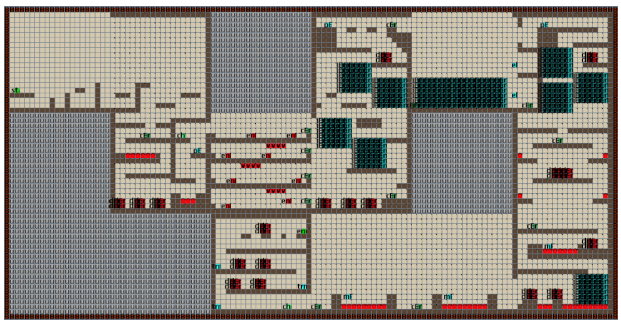


Figure 6: The grammar expands the level into 20x20 segments and gives a first impression on how the final level will be shaped. Note the blue-colored encounters, indicating that these will be translated to specific gameplay situations.

Final adjustments

The level generator proceeds with doing small adjustments regarding object rotation and removing any variables which are not nescessary for the parser. Variables that are used by the gate and lever mechanisms, however, are left as these are important for the parser to identify which gate is linked to which lever. This leaves us with the final level (as shown in figure 7) that can be exported as a text file and used by the parser.

Parsing in Unity

To test the playability of the levels that are generated, we have created a prototype in the Unity3D engine that utilizes all of the mechanics described in section 3. This prototype contains a parser that is able to read the text file from Ludoscope and puts every tile into a 2D array. Tiles are placed accordingly through the use of a switch-case programming statement, allowing the parser to instantiate game objects based on the associated tile.

Figure 7: Final representation of a level, ready for parsing.

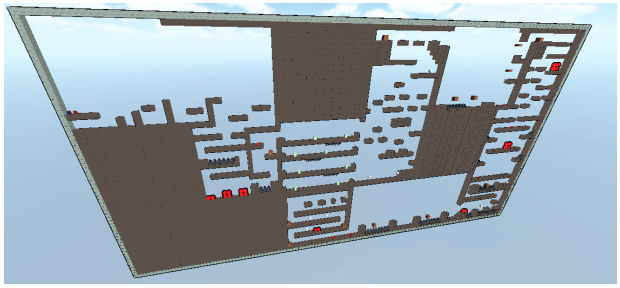


Figure 8: A parsed level within the Unity prototype.

6. DISCUSSION

A good level generator should increase replayability, and be efficient in terms of development costs, i.e. creating the generator should weigh out the cost of hand crafting the levels. Both of these characteristics depend on the variation of the possible levels that can be generated. The generation process described allows for a lot of variation with a limited set of handcrafted rules. To calculate the minimum amount of different levels that could be generated, we compose an equation which takes the number of variations of the shortest path possible, from left to right with one corner segment (as seen in figure 9). The variables within this equation are dictated by the the possible variations for each segment. For this path, it means that the startSegment only has four different possible variations, straightSegment has six (but is executed four times) and finally the corner and endSegment both have three possible variations. Note that the result produced by this equation is excluding the generation of encounters entirely. If we would multiply this answer with the amount of encounters to only the straightSegment, which has three different variations of encounters and is applied four times within the level, this amount would increase significantly.
We can accurately calculate a minimum amount of possible level variations with the equations below. We divide the equation in two parts so we can present the difference in possibilities when encounters are included to the equation. The equation below shows the possible variations when excluding encounters from the generation process:

$$excludingEncounters = startSegment \times straightSegment^4 \times cornerSegment \times endSegment = 4 \times 6^4 \times 3 \times 3 = 46,656$$

The following equation shows the possible variations when including encounters found in horizontal segments:

$$includingEncounters = excludingEncounters \times encounters^4 = 46,656 \times 3^4 = 3,779,136$$

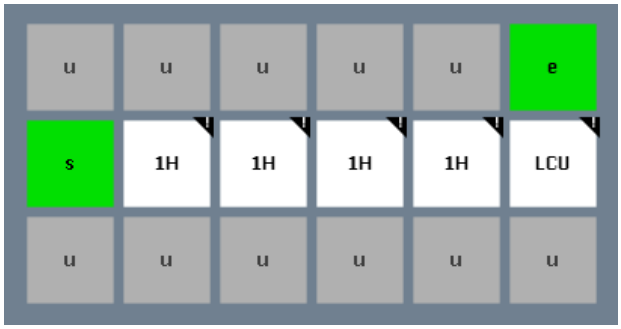


Figure 9: The shortest possible path that can be generated. See the caption of figure 4 for an explanation of the abbreviations.

These calculations show that for the smallest possible level, we can have over approximately 3,7 million different versions when including the encounters. It is important to note that differences in the levels are much more subtle when including encounters in the calculation as this can be as simple as one platform being moved a few tiles. When excluding these encounters from the calculation, the variation is much more prominent within the level as it involves changing the layout of at least one segment of the level.

The approach described in this paper allows game designers to generate a multitude of levels containing collaborative gameplay. Advantages of our approach are the fact that it is rule-based in a way that allows intuitive inspection by human game designers, that it is generic enough to be applied to different games, and that it can, in principle, be applied in a mixed-initiative setting (Karavolos et al. 2015).

In our system, collaboration is enforced by requiring encounters for collaboration in specific parts of the level, e.g. in the patterns for the corners. However, the approach taken in this project is very modular, with forms of local collaboration designed especially for

small segments within a level. Except for those areas of the level, there might not be interaction between the players. Collaboration could of course also occur outside of the collaboration segments, and even outside the game. More varied and meaningful ways to control and guarantee collaboration could include adding more cooperative game mechanics, such as the ones described in (Rocha et al. 2008) or adding more types of design patterns, such as the ones in (Seif El-Nasr et al. 2010). Perhaps this creates enough variation to compose a level completely of collaborative patterns. To find out to what extent people actually like or prefer these different kinds of pattern-based levels it is necessary to study how players actually play the game, and where, how, and how much they collaborate.

While designers will need to spend time designing their grammars for the level generator, this can be worth the effort when compared to the amount of time saved when the generator is functioning properly. A functional generator can create levels in a matter of seconds and even then can be tweaked should the designer feel the need to do so. Furthermore, spending time on finding out how all of these grammars and mechanics are going to interact with each other gives the designer an idea on whether or not some mechanics will fit the game as they are meant to. Finally, any system created by the designer can also be applied to other future projects that contain similar mechanics and/or gameplay, as opposed to games which have a level generator centred around it's core mechanics.

Although we have chosen to implement this method of level design to the specfic genre of puzzle-platform games, we believe the approach can be useful for other genres as well. For example, in the shooter genre, a similar method could be used to create levels containing areas fit for different styles of combat. In the roguelike genre, it could be used to generate dungeons, using encounters to generate various kinds of rooms.

7. CONCLUSIONS

In this paper we have shown how design patterns can be used to generate levels for collaborative puzzle-platformers. We have used a method based on generative grammars to create a path in space, and transform this path into level segments with variable elements. These variable elements can be transformed into instances of game design patterns. We have identified several game design patterns that incorporate collaboration between players, including the upsy-daisy, the timed level/gate and the common enemy patterns. The variation in the levels this generator can create derives from the combination of variable path length, number of possible level segments, and the number of encounters that each template can contain.

Limitations of the approach were discussed, as well as possible directions for further work, which include more systematic ways to control and enforce levels of collaboration, study empirically how players actually collaborate within and around the game, giving the designer more control over the generation process in a mixed-initiative setting, and the application to other genres of games.

8. AUTHOR BIOGRAPHY

BENJAMIN VAN ARKEL completed his BSc degree in Game Design at the Amsterdam University of Applied Sciences in 2015, on the topic of this paper. In the past, he has worked on visualizing motion capture data of soccer players for Dutch soccer club Ajax. His research interests currently involve procedural content generation and modular level design, subjects that he hopes to apply during his future career as game developer.
benjamin.v.arkel@hotmail.com

DANIËL KARAVOLOS is a researcher at the Amsterdam University of Applied Sciences, and teaches various courses at the Game Development department. He graduated from the University of Amsterdam with a Masters degree in Artificial Intelligence in 2013. His main research interests are computational intelligence, intelligent agents, procedural content generation, and automated game design.
k.d.karavolos@hva.nl

ANDERS BOUWER is a researcher and lecturer at the Amsterdam University of Applied Sciences. He studied Artificial Intelligence at the VU Amsterdam and the University of Edinburgh, and has a PhD from the University of Amsterdam since 2005. His research interests include interactive learning environments, music interaction & cognition, multimodal mobile systems, and automated game design.
a.j.bouwer@hva.nl

REFERENCES

Dahlskog S. and Togelius J., 2012. *Patterns and procedural content generation: revisiting Mario in world 1 level 1.* In *Proceedings of the First Workshop on Design Patterns in Games*. ACM, 1.

Dahlskog S. and Togelius J., 2013. *Patterns as objectives for level generation.* In *Proceedings of the Second Workshop on Design Patterns in Games*. ACM.

Dormans J., 2011. *Level design as model transformation: A strategy for automated content generation.* In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, 2.

Dormans J., 2012. *Engineering emergence: Applied the-*

ory for game design. Ph.D. thesis, University of Amsterdam.

Dormans J. and Leijnen S., 2013. *Combinatorial and Exploratory Creativity in Procedural Content Generation*. In *Proceedings of the 4th International Workshop on Procedural Content Generation in Games*.

Ferreira L. and Toledo C., 2014. *A search-based approach for generating Angry Birds levels*. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 1–8.

Hendrikx M.; Meijer S.; Van Der Velden J.; and Iosup A., 2013. *Procedural content generation for games: A survey. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9, no. 1, 1.

Hullett K. and Whitehead J., 2010. *Design patterns in FPS levels*. In *proceedings of the Fifth International Conference on the Foundations of Digital Games*. ACM, 78–85.

Karavolos D.; Bouwer A.; and Bidarra R., 2015. *Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation*. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

Kazemi D., n.d. *Spelunky Generator Lessons*. http://tinysubversions.com/spelunkyGen. URL `tinysubversions.com/spelunkyGen`.

Reuter C.; Wendel V.; Göbel S.; and Steinmetz R., 2014. *Game Design Patterns for Collaborative Player Interactions. DiGRA 2014 (accepted for publication)*.

Rocha J.B.; Mascarenhas S.; and Prada R., 2008. *Game mechanics for cooperative games. ZON Digital Games 2008*, 72–80.

Seif El-Nasr M.; Aghabeigi B.; Milam D.; Erfani M.; Lameman B.; Maygoli H.; and Mah S., 2010. *Understanding and evaluating cooperative games*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 253–262.

Shaker N.; Shaker M.; and Togelius J., 2013. *Ropossum: An Authoring Tool for Designing, Optimizing and Solving Cut the Rope Levels*. In *Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Shaker N.; Togelius J.; and Nelson M.J., 2014. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Procedural Content Generation in Games: A Textbook and an Overview of Current Research*.

Shaker N.; Togelius J.; Yannakakis G.N.; Weber B.; Shimizu T.; Hashiyama T.; Sorenson N.; Pasquier P.; Mawhorter P.; Takahashi G.; et al., 2011. *The 2010 Mario AI championship: Level generation track. Computational Intelligence and AI in Games, IEEE Transactions on*, 3, no. 4, 332–347.

Smith G.; Whitehead J.; and Mateas M., 2011. *Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. IEEE Transactions on Computational Intelligence and AI in Games*, 3, no. 3, 201–215.

Togelius J.; Yannakakis G.N.; Stanley K.O.; and Browne C., 2011. *Search-based procedural content generation: A taxonomy and survey. Computational Intelligence and AI in Games, IEEE Transactions on*, 3, no. 3, 172–186.

Zagal J.P.; Rick J.; and Hsi I., 2006. *Collaborative games: Lessons learned from board games. Simulation and Gaming*, 37, no. 1, 24–40.