Characteristics of Generatable Games

Julian Togelius, Mark J. Nelson, Antonios Liapis
Center for Computer Games Research
IT University of Copenhagen
Copenhagen, Denmark
julian@togelius.com, mjas@itu.dk, anli@itu.dk

ABSTRACT

We address the problem of generating complete games, rather than content for existing games. In particular, we try to answer the question which types of games it would be realistic or even feasible to generate. To begin to answer the question, we first list the different ways we see that games could be generated, and then try to discuss what characterises games that would be comparatively easy or hard to generate. The discussion is structured according to a subset of the characteristics discussed in the book *Characteristics of Games* by Elias, Garfield and Gutschera.

1. INTRODUCTION

In recent years, it has been rather conclusively shown that artficial intelligence systems can satisfactorily generate game content of a number of different types: textures, plants, levels, maps, creatures and so on [21]. There is currently intense interest in research and development in procedural content generation within both academia and the game industry, as answers are sought to technological challenges such as how to design more replayable games and how to cut game development costs. It is also being explored to what extent design principles for game content can be formalised.

But could an AI system generate a complete game? By this we mean generating the core parts of the game, such that it if they were different, it would be a different game—not just the same game with new content. Therefore, generating a complete game would typically mean generating at least the rules of the game. Of course, the demarcation line between rules and other types of content is never going to be absolutely sharp. In some trading card games, the rules are rather basic and the form of the gameplay is mostly defined by the cards. But the general class of systems we're interested in here are those that generate new game rules and mechanics, rather than new levels or models to be used in the context of a fixed set of rules and mechanics, as is so far the most successful application of procedural content generation.

Why would we want to attempt to generate games? Some of the reasons are the same as for generating game content in general: to save human effort (and thus money), or to create new types of games that rely on constant creation of new sub-games. Game-generation would, for example, make it possible to create the world's first truly single-player games, in the sense of games unique to a given player and/or moment in time [16]. Perhaps game-generation research may illuminate the intellectual exploration of computational creativity [5] more than other types of procedural content generation. Besides fully automated design, automating parts of game design has a strong connection to mixed-initiative game design and development tools, which help a human designer in designing games according to some vision or criteria [17, 27]. Mechanizing the game-design process (or attempting to do so) can also rigorously test theories of game design and lead to a better understanding of the design space [22, 8]. Last but not least, there is a strong connection to general game playing as a testbed for artificial general intelligence, where the generation of unseen games is crucial for testing the generalisation capacity of artificial intelligence algorithms [9, 20].

In fact, there are a number of recent efforts to generate complete games. They are based on different types of algorithms, and have different genres or families of games as their domain. They also differ in their expressive ranges, i.e. the size of the space of games they could potentially generate. These have met with various success, but generally it seems to be very hard to generate complete games—much harder than generating content for an existing game. There is only one known example of a program generating a new game which is good enough to be sold in a box: Yavalath [1], a relatively simple board game. It is likely that some kinds of games are easier to generate than others, and at this point it would make sense to concentrate research efforts on the type of games we might have success in generating. In this paper, we try to identify promising domains for automatic game generation.

A recent book by several well-known game designers, *Characteristics of Games* [10] (hereafter *CoG*), aims to identify concrete characteristics that impact the design of games. We take this design analysis as our starting point for investigating the characteristics of *generatable* games. Our method is simply to take these proposed characteristics of games in turn, and consider what insights they suggest for the design of game generators.

2. HOW COULD GAMES BE GENERATED?

To lay the groundwork from a technical perspective, we see three ways in which complete games could be generated: constructive methods, solver-based methods, and search-based methods.[1]

Constructive methods are those where properties of the game are identified ahead of time, and the generator constructs individual games by choosing combinations of the identified properties. The process does not include constrained or objective-driven search of the game space. The earliest known example of game generation, Metagame [18], the earliest game-rule generator we know of, uses a constructive method. Metagame generates variants of chess by encoding potential rule variations in a grammar of chess variants, and then selecting specific games by choosing a production from the grammar. One of the main issues with constructive methods is that the method of construction must only construct valid games. Metagame is deliberately designed with a fairly narrow generative space as a result of this demand, so that games sampled from its grammar are almost always valid chess variants (it does also include a lightweight validation process to reject some kinds of design errors in a generate-and-test fashion).

Solver-based methods use constraint solving or logical methods to generate games. A prominent example is Smith and Mateas' use of answer-set programming to generate complete games [23, 24]. In this work, individual game mechanics were specified in the logical language AnsProlog, and an ASP solver used to generate complete games based on constraints—for example, the user could specify that the game should be winnable by indirect pushing.

Search-based methods use evolutionary computation or other stochastic search/optimisation algorithms to generate games by searching the space of possible games [26]. This requires an objective or evaluation function that assigns a number reflecting how "good" the game is. As it seems almost certain that to properly evaluate a game you need to play it, evaluation functions for complete games are almost always *simulation-based*. Playing an unknown game ("general game playing") is in itself a hard challenge. In an early example of search-based generation of complete games, Togelius and Schmidhuber evolved Pac-Man-like games in a two-dimensional grid world. The evaluation function is meant to judge how learnable the games are by humans [25]. The probably most successful game generation system so far, *Ludi* by Browne, generates simple board games similar to Checkers, Go and Othello, through evolutionary computation. The rules are represented in a Lisp-like language and games are evaluated using a weighted sum of several heuristics [3]. Other examples of search-based game generation include Cook and Colton's *Angelina* project [6] which generates platformers (among other things) and Font et al's project to generate card games [11].

[1]There are conceivably other ways in which a computer could generate a game, e.g. by inventing human-level AI and teaching the AI to behave like a professional game designer. However, the three approaches discussed in this section have the merit of being concrete and not AI-complete.

3. CHARACTERISTICS OF GENERATABLE GAMES

Below, we discuss game generation from the vantage point of some of the characteristics presented in *CoG*. The characteristics can shape the generation in at least two possible ways: by constraining the generation space and by acting as objectives. In the first case, the generator is limited in some way so that it cannot generate games that do not have the characteristic, either because it cannot express them in its representation or because it throws any such games away without evaluating them. In the second case, we are capable of expressing and evaluating games with or without some characteristic, but we choose to search for games with more or less of the characteristic. This way of directing the generation is particularly useful when some characteristic is closely tied to the qualities we want in our generated games. Note that both solver-based and search-based methods are capable of both optimising for objectives and enforcing constraints.

3.1 Basics

CoG starts with some basic characteristics that define what kind of game is being played, and how players learn to play it.

3.1.1 Length of Playtime

One of the most elementary aspects of a game is how long it takes to play. For game generation, playing time is particularly relevant for simulated play-throughs using search-based methods. Games with longer playing times clearly add a computational burden whenever simulation is necessary. The relevant measure of playing time here is probably the time needed to simulate the game rather than the number of decisions that need to be taken. Mahlmann et al's work on generating strategy games found that the major bottleneck was the time taken to simulate playouts [14].

Note that the important measure here is the playing time for a single game atom, or the shortest unit of meaningful play, rather than the playing time for a complete game. In most cases, it will be sufficient to simulate a single round of a card game or a single level of a platform game, decomposing the generation process accordingly.

3.1.2 Number of players

CoG categorises games into zero-player games (too trivial to consider in this paper), "pure" one-player games playing against the system (sudoku, Tetris, Myst, Asteroids), "one and a half" player games where a player plays against simulated opponents (Civilization, Diablo), two-player games, two-sided team games, one-sided team games (Left 4 Dead, Arkham Horror), multiplayer games (three or more players) and massively multiplayer games. Among these categories, solver-based methods are ideal for generating "pure" one-player games, provided that the system is tightly defined and deterministic. The success of ASP at generating puzzle games [24] and arcade-like [23] games—both "pure" one-player games—lends credibility to this claim. However, one-player games with semi-continuous space, stochasticity and/or frequent player-environment interactions , such as Asteroids, are difficult to solve completely due to the explosion of the game state space.

"One and a half" player games rely, by definition, on AI-controlled opponents, and can be simulated in a straightforward manner. In such simulations, the only assumption is about the player's behavior; in cases where simulated opponents follow the same rules as the player, using those controllers to simulate the player is a fast and relatively realistic shortcut. Search-based methods with simulation-based evaluation can be used to generate such games; however, lengthy simulated playthroughs (e.g. Civilization) or a large degree of stochasticity on the part of the controllers (e.g. random monster damage in Diablo) can make the use of search-based methods unrealistic. For games involving two or more players (including one-sided team games), the interaction between players and different combinations of personality, skill or strategy (e.g. veteran versus newbie, defensive tactics versus rushing tactics) make simulations even more cumbersome. While for clearly defined, deterministic games such as card games [11] and board games [3], two players can be simulated to a satisfactory degree, it is difficult to envision fully generatable multiplayer or massively multiplayer games with current methods. One advantage of symmetric two-player games is that various forms of balance between players are useful objectives [12].

3.1.3 Heuristics

CoG (p. 29) argues that a core feature that makes games interesting is the presence of *heuristics* developed as players gain skill in a game:

> Players typically gain skill by developing heuristics: rules of thumb that help them play the game. Some of these rules might be quite concrete ("never draw to an inside straight" in poker) and some might be fairly vague ("develop your pieces" in chess) ... if someone asks "how do you play that game?" and they already know something about the rules, chances are they are looking, not for even more detailed rules, but for some basic heuristics.

This characteristic bears a certain resemblance to the idea that players' learning curves are an important property to take into account when designing a game. Several existing PCG systems generate games by aiming to produce games whose learning curves are neither too shallow nor too steep. Togelius and Schmidhuber [25] use learning curves as their primary objective, targeting a "medium" learning curve by running a machine-learning agent on generated games, and rejecting games that the agent learns either too quickly or too slowly.

However, heuristics are not precisely learning curves, and the difference points to some interesting venues for future research in automated game generation. Learning curves in existing systems typically are estimated by taking a fixed game-playing model (such as a neural-network controller) and iteratively improving its weights or parameters. This "continuous" approach to improving skill is akin to how humans improve their performance in a racing simulator, but doesn't as strongly resemble the idea of players gaining new heuristic information about a game. In particular, players use the gained heuristic information not only to *play* the game, but also to conceptualise it and talk about it with others. This scaffolding, where players build simple models of how to play a game, and name those models' parts, is part of what distinguishes interesting from uninteresting games. Methods for determining whether a game supports such scaffolding are, we propose, a key challenge for automated game generation.

Expicitly modelling players' acquisition of discrete knowledge about a game has been investigated in specific contexts. Educational games sometimes *start* from pre-specified heuristics they hope users will develop while playing the game: the educational outcomes. The game is then designed around attempting to ensure that playing the game scaffolds the acquisition of precisely this set of heuristics. To our knowledge, such games have not yet been fully generated, with the intended learning outcomes taken as input and a fully generated game produced as output. However, the idea has been used used by level generators to produce a desired skill progression through sequencing levels. The mathematics game *Refraction* starts out by requiring only basic operations to solve early levels, and then progressively requires more complex operations at the higher levels [4]. This is intended to result in players developing more and more complex "heuristics" about arithmetic through the process of learning the heuristics of the game. The extent to which acquiring specific heuristics impacts gameplay has also been used as an analysis tool to provide feedback on game-design prototypes [15, 13], a form of analysis that may be useful as a component of a game-generation system that explicitly targets the existence of such heuristic progressions.

3.2 Infrastructure

The next set of properties involves the basic infrastructure on which games are built: their rules, standards, outcomes, ending conditions, and other kinds of feedback to the player.

3.2.1 Rules

Rules are arguably the core feature of a game, and for an algorithm to generate a game it needs to generate the rules in some form. Such rules can be predefined hand-crafted rules which can be inserted into (or removed from) the game or parameterisable functions where their arguments can be swapped with current game elements.

3.2.2 Standards

CoG defines standards as "commonly accepted patterns that many players are already familiar with". Standards include low-level details (e.g. a discard pile for card games, or the W,A,S,D controls for a first person shooter) or high-level ideas (e.g. the concept of jumping on enemies for platformers, or quick-time events for modern games). Regarding the lower-detail aspect of standards, one could consider standards to be akin to rules. Generatable games can theoretically include the use of such standards among their generated rules. Arguably, most game generators currently choose between rules hand-crafted according to their programmers' expert knowledge on genre standards, which means that most generatable games are more likely to follow genre standards than not. On that account, even if rule generation becomes more freeform in the future (using e.g. dynamic programming), evaluating these generated rules (or combi-

nation thereof) according to their conformity to genre standards can be another measure of the learnability of the game by human players.

3.2.3 Outcomes

Outcomes of a game include ranking in a race, number of points scored, levels cleared, time survived etc. When generating a game, however, it is not a priori knowable what the outcomes are. In previous work, researchers have solved this problem by simply stipulating that there is such a thing as score and that having much of it is good [25] or that there are things such as winning and losing and the rule set needs to include actions to lead to these states [3]. Some such stipulation seems to be necessary for any game generation method.

3.2.4 Ending conditions

An ending condition determines when the game ends; at least in one-player or "one and a half" player games, which are the focus of this paper, a game ends when a player "wins". This ending condition is important for the purposes of simulations, as a game is evaluated after the simulation ends. Although the ending condition can also be generated, it is important to ensure that the simulation does not go on forever e.g. via a secondary timeout ending condition. Evaluation of the ending condition is incorporated in the evaluation of the game: for instance if the game ends before a game arc is formed will result in a low objective score due to the ending condition. Solver-based generation also requires ending conditions, as the entirety of a game needs to be checked for constraint satisfaction (e.g. that the winning condition can be reached). Although theoretically the ending condition could be fully genaratable (through dynamic programming), game generators so far either have a predefined ending condition or are allowed to select from a range of hand-crafted ending conditions. It is not clear how games without ending conditions could be generated: in theory, endless games could be generated if they consist of repetitions of a game atom of playtime (see 3.1.1) which can then be simulated and evaluated.

3.2.5 Sensory Feedback

While an important aspect of games, generating or even assigning sensory feedback for generated game mechanics is arguably an even larger problem than game generation, as it touches upon sensory perception, musical appreciation, cognition and visual taste. Granted that entire research fields are attempting to understand these concepts as well as computationally generate music and art pieces, the topic of a generatable game's sensory feedback falls outside the scope of this paper. However, it is worth noting that some early steps towards purposefully accomodating sensory feedback in generated games are being taken by Cook, Colton and Pease [7].

3.3 Games as Systems

At the core of games are the abstract systems of mechanics that define their dynamics and possibility spaces. While they do not directly control gameplay, designing the systems level is an important aspect of producing interesting games.

3.3.1 Abstract subgames

While many contemporary games, both analog (e.g. Android) or digital (e.g. Grand Theft Auto) include subgames of varying levels of abstraction, it is somewhat early to consider having generatable subgames within fully generatable games. On the other hand, there is considerable potential for the creation of fully generatable subgames within existing (authored) games; the simpler nature of subgames lends itself better to the current state of game generation than creating, for instance, a fully generatable AAA game. It is also worth considering including templates for abstract subgames in the vocabulary of the game generator (see Section 3.2.2).

3.3.2 Snowball and catch-up

"Snowball" is the *CoG* term for what in cybernetic language would be called positive feedback: the rich get richer, and the poor poorer. "Catch-up" is the opposite. The presence of a certain amount of catch-up could be used as an objective, as a moderate amount of this characteristic is generally seen as good. It is also relatively easy to measure, using Monte Carlo-based metrics such as Browne's "outcome uncertainty" metric [3].

3.3.3 Complexity tree growth and game arc

The branching factor of a game is how many different actions a player can take from a given state. The branching factor along with other measures of complexity [2] of a generated game can also infer the heuristics developed by players to conceptualise it (see 3.1.3). Moreover, a game's branching factor largely determines whether a game is playable by a given algorithm. As a rule of thumb, the higher a branching factor, the less effective any given algorithm is at playing an unknown game — although certain algorithms such as Monte Carlo Tree Search are more capable of handling games with larger branching factors such as Go. As search-based generators require a simulation with an adequate AI controller in order to evaluate a generated game's quality, games with exceedingly high branching factors are impossible to generate at the current state of AI research. Although not tied to AI-controlled agents, solver-based methods also suffer from high branching factors as the number of possible action combinations that need to be tested for feasibility becomes excessive.

One of the characteristics of games is the game arc, which is the graph of the branching factor of a game at different stages of gameplay. Many games exhibit an upwards convex arc, where there are few options at the beginning of the game, many in the middle stage of the game, and fewer again towards the end. Games as different as Monopoly, Chess and Civilization arguably exhibit such a game arc, although arcade games such as Tetris and Pac-Man have less of a game arc (or an arc with a different shape). Having an upwards convex game arc, together with a minimum and maximum branching factor at each stage of the game, can be a useful addition to both search-based and solver-based game generation. This is related to (but not the same as) the Monte Carlo-based tension and uncertainty metrics proposed by Browne [3].

3.4 Indeterminacy

Indeterminancy is a common feature of both analog games (dice, random card draws) and digital games (randomized

damage in Diablo or Starcraft, randomized opponent behavior in Ms. Pacman). The more actions with uncertain outcomes and the more uncertain outcomes per action, the more difficult it is to assess a full playthrough. Solver-based methods need to account for all possible outcomes and thus can only handle the most basic indeterminancy. Search-based methods can use an objective score based on the average of multiple simulations, thus limiting the effects of indeterminacy. However, the higher the degree of indeterminacy, the more simulations need to be averaged; granted that simulations already are the bottleneck of search-based generation, it may be unrealistic to search for fully generatable games with high indeterminacy.

3.5 Player Effort

Players invest substantial *effort* into games in a number of ways, and good design is distinguished from bad design in part by whether it makes good use of this effort, directing it in a way that is interesting rather than merely tedious.

3.5.1 Costs

CoG enumerates several kinds of *costs* that players incur when playing a game. These include monetary cost of the game and the equipment required to play it (e.g. gaming console for console games, ball and glove for baseball), as well as the prerequisite skills necessary to play the game (e.g. physical strength for American football or hand-eye coordination for first-person shooters). Within generatable games, skill prerequisites could realistically be factored as constraints or evaluations. Generated games can require that the reaction time necessary to complete a game is above the minimal reaction time of humans.

3.5.2 Rewards

Within a generatable game, identifying rewards can be quite challenging; however, a common reward in all generated games would be winning (or completing) the game. Granted that generatable games, be they generated via solvers or search, will have an ending condition, it is generally easy to evaluate such a reward. More low-level rewards can also be generated, such as incrementing the score value for performing an action such as eating a pill or killing a monster. Teasing apart, however, how the different rewards interact with each other and with the winning condition can be quite challenging and likely requires a measure of reward/effort ratio.

3.5.3 Downtime & Busywork

Evaluating downtime in simulated playthroughs or expanded solutions of a game could look into the amount of game ticks where no action was performed. However, this will greatly depend on the game genre, the type of feedback, and the goal of the downtime; waiting for a moving platform to return to the player is different than waiting for a ball to drop in Peggle (which is accompanied with very satisfying visual and aural feedback). Moreover, simulating the downtime of a reluctant player is much more complex than creating an efficient AI for solving the game, and could well be beyond the current capabilities of research in the field of agent control.

Busywork, i.e. the parts of the game which are not considered fun or do not require any skill (e.g. setting up a board game or the initial build sequence in Starcraft), is even more complicated to evaluate without an accurate model of a player's fun. Despite research into modeling a player's experience in existing games [28], assessing a player's entertainment in completely unseen games is much less straightforward. If a measure of player reward can be estimated, however, an assumption could be made that busytime consists of actions which do not provide a short-term reward or do not bring the player closer to a long-term reward.

3.6 Superstructure

The Superstructure section of *CoG* elaborates on elements of a game which do not pertain to gameplay and occur "outside of the game", such as the metagame, the game's theme or lifetime. Evaluating the superstructure of fully generatable games would require a quite elaborate AI system encompassing higher-level knowledge on game genres, existing games and even market or public opinion data. Such an AI would then act as a game producer [19] able to cluster games within existing genres, and predict a game's potential lifetime or popularity based on data of previous games of the same cluster. Developing such an AI is more ambitious in scope than the examples laid out in this paper, and would require a very different approach (such as machine learning from big data on game ratings, user feedback or sales figures).

4. CONCLUSION

In this paper we have identified three different ways in which complete games could be generated. We have also discussed which types of games would be more or less feasible to generate, and what properties of games to look for when generating them. To structure the space of games, we utilised a subset of the characteristics outlined in *CoG*. The reader can see this as a set of hypotheses regarding the what and how of game generation, hypotheses which can be tested by implementing game generators and seeing how well they work. Even if we are wrong in some specifics, we believe that this paper can help structure the discussion around game generation. Following this methodology can also be seen as an empirical test of the material in *CoG*: using these identified characteristics of games to implement game generators is a sort of acid test of whether we have *really* identified the important features that characterise games with enough specificity to use them constructively.

5. REFERENCES

[1] C. Browne. Yavalath, 2007.

[2] C. Browne. Elegance in game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):229–240, 2012.

[3] C. Browne and F. Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[4] E. Butler, A. M. Smith, Y.-E. Liu, and Z. Popović. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th Symposium on User Interface Software and Technology*, pages 377–386, 2013.

[5] S. Colton and G. A. Wiggins. Computational creativity: The final frontier? In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 21–26, 2012.

[6] M. Cook and S. Colton. Multi-faceted evolution of simple arcade games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 289–296, 2011.

[7] M. Cook, S. Colton, and A. Pease. Aesthetic considerations for automated platformer design. In *Proceedings of the 8th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 124–129, 2012.

[8] J. Dormans. Simulating mechanics to study emergence in games. In *Proceedings of the AIIDE Workshop on Artificial Intelligence in the Game Design Process*, pages 2–7, 2011.

[9] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. In *Artificial and Computational Intelligence in Games*, pages 85–100. Dagstuhl Publishing, 2013.

[10] G. S. Elias, R. Garfield, and K. R. Gutschera. *Characteristics of Games*. MIT Press, 2012.

[11] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius. Towards the automatic generation of card games through grammar-guided genetic programming. In *Proceedings of the 8th International Conference on the Foundations of Digital Games*, pages 360–363, 2013.

[12] V. Hom and J. Marks. Automatic design of balanced board games. In *Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 25–30, 2007.

[13] A. Jaffe, A. Miller, E. Andersen, Y.-E. Liu, A. Karlin, and Z. Popović. Evaluating competitive game balance with restricted play. In *Procedings of the 8th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 26–31, 2012.

[14] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Modelling and evaluation of complex scenarios with the strategy game description language. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 174–181, 2011.

[15] M. J. Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Proceedings of the AIIDE Workshop on Artificial Intelligence in the Game Design Process*, pages 14–18, 2011.

[16] M. J. Nelson and M. Mateas. Towards automated game design. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637. Springer, 2007. Lecture Notes in Computer Science 4733.

[17] M. J. Nelson and M. Mateas. An interactive game-design assistant. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, pages 90–98, 2008.

[18] B. Pell. Metagame in symmetric, chess-like games. In *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*. Ellis Horwood, 1992.

[19] M. O. Riedl and A. Zook. AI for game production. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, 2013.

[20] T. Schaul, J. Togelius, and J. Schmidhuber. Measuring intelligence through games. *arXiv preprint 1109.1314*, 2011.

[21] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014. (To appear).

[22] A. M. Smith. *Mechanizing Exploratory Game Design*. PhD thesis, University of California, Santa Cruz, December 2012.

[23] A. M. Smith and M. Mateas. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 273–280, 2010.

[24] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[25] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 111–118, 2008.

[26] 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 AI in Games*, 3(3):172–186, 2011.

[27] M. Treanor, B. Blackford, M. Mateas, and I. Bogost. Game-O-Matic: Generating videogames that represent ideas. In *Proceedings of the 3rd Workshop on Procedural Content Generation*, 2012.

[28] G. N. Yannakakis, P. Spronck, D. Loiacono, and E. Andre. Player modeling. In *Artificial and Computational Intelligence in Games*. Dagstuhl Publishing, 2013.