A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels

Nathan Sorenson, Philippe Pasquier, and Steve DiPaola

***Abstract*—This paper presents an approach to automatic video game level design consisting of a computational model of player enjoyment and a generative system based on evolutionary computing. The model estimates the entertainment value of game levels according to the presence of "rhythm groups," which are defined as alternating periods of high and low challenge. The generative system represents a novel combination of genetic algorithms (GAs) and constraint satisfaction (CS) methods and uses the model as a fitness function for the generation of fun levels for two different games. This top–down approach improves upon typical bottom–up techniques in providing semantically meaningful parameters such as difficulty and player skill, in giving human designers considerable control over the output of the generative system, and in offering the ability to create levels for different types of games.**

***Index Terms*—Challenge modeling, fun, player enjoyment, procedural content creation, video games.**

I. Introduction

As video games exhibit progressively expansive game environments, there has been a growing interest in employing generative computational algorithms to mitigate the cost of authoring game content [1]. Broadly referred to as procedural content generation, these techniques can be seen in early games such as *Rogue* [2] and *Nethack* [3], as well as more recent games such as *Far Cry 2* [4]. These computational techniques promise to reduce the involvement of a human designer, thereby enabling smaller development teams to create much more content than would otherwise be possible and, because algorithmically generated content is not as fixed as content authored by hand, create content that is more readily adapted to the unique preferences of individual players.

This paper presents a procedural content generation system that is able to create game environments (levels) for a variety of games. Our system differs from existing systems in its adherence to a top–down approach, as opposed to a bottom–up, rule-based approach. These bottom–up systems typically create levels through an iterative execution of a number of production rules or through an *ad hoc* assortment of deeply nested conditional branches and lengthy switch statements with no principled, overarching design.[1] This lack of modularity contributes to their reputation for being difficult to debug [6]. As well, their idiosyncratic design ties these generative techniques to a single game; it is not straightforward to extract generalizable functionality from the highly specific procedural code. Finally, bottom–up approaches often provide little artistic control over the final output.

The present work adheres to a top–down orientation by adopting an evolutionary computation approach. Level designs are considered to be individuals in a population and levels of high-quality pass on their characteristic genetic information to future generations. Quality is determined by high-level design goals specified as a fitness function. This fitness function operates solely on the final generated output and is ignorant of the specific manner in which the content is generated, allowing for much more generality than bottom–up approaches.

We present, first, our model of player enjoyment, which serves as the foundation for the rest of the system. We describe the scope of the model, justify its design by drawing from video game research as well as analysis of existing commercial game levels. We evaluate this model in its ability to identify commercial-quality levels among arbitrarily generated levels. We then show how this model is used as a fitness function in an evolutionary system. The system is applied to two different game contexts, demonstrating its general applicability. We conclude with discussion of the benefits of this approach and discuss future work.

A. Previous Work

1) Video Game Generative Systems: Procedural content creation is an active area of research, and many different approaches exist. A particularly ambitious project has been the automatic generation of entire games. Browne [7] employs genetic algorithms (GAs) to construct combinatorial abstract games similar to *Chess* and *Go*. He successfully produces entertaining games using a fitness function that consists of a weighted sum of 57 design criteria, drawn from a wide array of sources, including psychology, subjective aesthetics, and personal communication with game designers. We also use an evolutionary approach but strive to make the underlying model more parsimonious and transparent.

Togelius and Schmidhuber also explore the evolution of game designs [8]. They consider designs to be fun insofar as a neural net is able to learn to play that game, arguing that it is the process of learning to master a task that ultimately provides pleasure. However, the effectiveness of this approach could be limited

Manuscript received November 01, 2010; revised March 09, 2011; accepted June 20, 2011. Date of publication July 07, 2011; date of current version September 14, 2011. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

N. Sorenson is with the School of Interactive Arts and Technology, Simon Fraser University, Vancouver, BC V5N4L3 Canada (e-mail: nds6@sfu.ca).

P. Pasquier and S. DiPaola are with the School of Interactive Arts and Technology, Simon Fraser University, Surrey, BC V3A0A3 Canada.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2011.2161310

[1]Though most commercial titles are closed source, we can see *ad hoc*, bottom–up systems in open-source titles such as *Freeciv* [5] and *Nethack* [3].

by the degree to which neural networks are able to mirror the experience of human players.

Hastings *et al.* [9] evolve weapons for a space-themed video game. The fitness of a given weapon design is inferred from the behavior of the player as the game progresses; if the player uses a weapon frequently, similar weapons are made available. Conversely, if a weapon is left unused, it appears less frequently. This approach is an example of an interactive fitness function, where evolution is guided through human choice. Our work differs as it provides an automatic fitness function which does not require direct human involvement.

Smith *et al.* [10] introduce a mixed-initiative design tool for the creation of 2-D platformer games. Their tool allows certain portions to be specified by hand, with the rest constructed according to a rule-based system. This system produces levels that conform to a rhythm-group structure, with contiguous periods of challenge interspersed with moments of rest. Although our work shares the goal of permitting human designers a great deal of influence over the system's output, we express rhythm groups explicitly through an objective model instead of implicitly through the behavior of a collection of production rules. Shaker *et al.* [11] also generate *Mario* levels to optimize a model of player enjoyment. Instead of explicitly constructing a minimal model with author-adjustable parameters, as is the case with our work, their model is inferred from player behavior using a statistical approach; players must explicitly evaluate how much fun they had during a play session before the model can predict what future levels will be most enjoyable. As well, their generative technique is restricted to adjusting four parameters of a rule-based random level generator that is specific to *Mario*, whereas our generative approach seeks to offer greater control to level designers in a manner which can be generalized to different types of games.

2) Characterizations of Fun: Fun is a broad term and any attempts at a precise definition would certainly require a limitation of scope, as no single statement is likely to encompass the entire concept. Even restricting the discussion to the domain of video games does not permit a simple characterization; Hunicke *et al.* [12] even argue that the term "fun" should be discarded entirely, as it is too vague to be considered a practical unit of analysis. They suggest several more precise terms, such as "problem solving," "competition," and "discovery." Similarly, the works of Malone [13], Apter [14], and Garneau [15] outline various major components of fun. Although these taxonomies provide useful terminology, it is doubtful they allow for deeper analysis of the nature of fun in video games. While they identify broad categories of fun often encountered in games, they say little regarding precisely what structures and dynamics are responsible for creating pleasure.

3) Flow: A common thread among analyses of game performance and player enjoyment is the notion of "flow," as expounded in the work of Csikszentmihalyi [16]. The psychological state of flow is brought about when a number of prerequisite conditions are satisfied, such as one feeling in control of a situation, losing awareness of the passage of time, and executing a task that is neither too easy nor too difficult for one's skill level. Flow is characterized by intense focus and heightened task performance and is often referred to as a state of "optimal experience." Sweetser and Wyeth have adapted the principles of this concept into a framework, called "GameFlow" [17] that identifies properties of game designs which especially facilitate the creation of a sense of flow.

The importance of flow in understanding certain gaming experiences, particularly the connection between challenge and fun, is noted by several authors. In *A Theory of Fun*, Koster states that "fun is the act of mastering a problem mentally" [18] and that the process of overcoming difficult tasks is the source of pleasure in games, whether it is through identifying patterns in the behavior of enemy characters or developing the muscle-memory necessary to execute a sequence of button presses in a fighting game. If the task is too difficult, the player does not experience a sense of mastery. Conversely, if the task is too easy, the player does not need to develop any skills to succeed. Salen and Zimmerman confirm the central role of difficulty in providing fun experiences, and though they emphasize that flow is not synonymous with fun, they do claim that challenge and frustration are "essential to game pleasure" [19].

There are definite affinities between the challenge-based dimension of flow and the Yerkes–Dodson law [20], which states that performance reaches its maximum when the arousal felt during the completion of a task is neither too little, nor too great. Piselli *et al.* [21] have shown that this phenomenon is equally applicable to the context of video games, even when considering pleasure, instead of task performance, as a function of difficulty; players have the most fun when presented with challenges that are difficult, but not impossible to overcome.

II. Model of Fun

One of the central contributions of this work is a computational model of fun as experienced in challenge-based video games. This model is based on the notion of *rhythm groups* as introduced by Smith *et al.* [22], which are design structures consisting of repeating oscillations between periods of high and low difficulty. The relationship between the degree of challenge posed by a rhythm group and the resulting fun as experienced by the player is informed by the Yerkes–Dodson law and aspects of the concept of flow.

Our previous attempt to model this phenomenon [23] was intended to provide a clear account for the experience of fun as a result of rhythm groups, and offer meaningful parameters to allow the model to be adapted to a wide variety of game context and player abilities. The currently presented model has the same purpose, but improves on the earlier formulation significantly. Whereas the previous model could reproduce level structures that appeared visually similar to those existing in commercial games, the current model is able to be trained as a classifier on actual level data, and is able to effectively distinguish between examples of good and poor level design, providing much stronger evidence of its effectiveness.

A. Model Scope

Certainly, the notion of fun is a nebulous concept, and the specification of the model's characterization of quality requires particular attention. We emphasize that the purpose of our rhythm-group model is to serve as a fitness function in a generative process, evaluating the relative quality of generated levels.

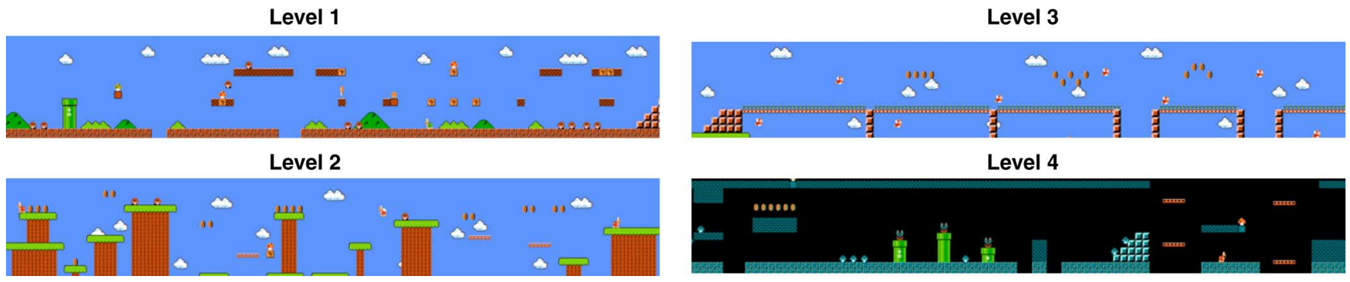


Fig. 1. Segments of four of the 28 levels used to inform the design of the model of fun.

Because our goal is not primarily to model a psychological state of fun, we use the term "fun" in a more pragmatic sense: as a measure of the quality of a level's design. According to this sense of the term, validation for the model does not come from subjective studies with human participants but, rather, from observations of the game industry's classic, enduring examples of good level design. We do not entirely distance ourselves from the notion of fun as a pleasurable mental state; certainly, a level of high quality is ultimately a level which gives pleasure to the player. However, for our purposes, we assume that the properties that elicit this state of pleasure are sufficiently manifest in the level designs and that we can, therefore, understand important qualities of challenge-based fun by restricting our analysis to the level designs alone. In other words, the model is considered successful insofar as it is able to attribute high fitness values to levels which exhibit structures characteristic of those found in well-designed levels. The important task of validating this assumption through user studies is reserved for future work.

Our model of fun, and our generative process as a whole, applies to challenge-based action games. These are games where the predominant form of pleasure does not arise from exploration, online social interaction, logical puzzle solving, or narrative, but rather through reflex-based tests of skill. Indeed, there are so many aspects to the concept of fun in general that it becomes necessary to restrict our focus to a single area; we claim that reflex-based challenge is more amenable to formal analysis than, for example, narrative or aesthetic pleasure.

To define our scope even more precisely, we are interested in games in which the challenge is delivered primarily through the level design. This requirement includes genres such as platformers and action–adventure games, but notably excludes genres such as fighting games and sports games. In these games, the nature of the challenge, and ultimately the fun experienced by the players, is primarily a function of the skill of the opponents (artificial or human) or of the rules and controls governing the game mechanics. The levels of such games serve merely as an aesthetic backdrop to frame the game and do not serve as a promising target for automatic generation. This distinction justifies the use of our model as a *direct* fitness function for level evaluation, a notion which Togelius *et al.* [1] defines in contrast to *simulation-based* fitness functions, which depend on a dynamic observation of actual gameplay to assess quality.

B. Model Design

The model's purpose is to estimate the amount of fun provided by a particular level, based on that level's configuration of challenge. The model depends on the notion of rhythm groups, similar to those described by Smith *et al.* [24], but with important distinctions. Whereas Smith *et al.* describe rhythm groups in terms of timed sequences of button presses, analogous to rhythmic beats in musical compositions, we emphasize the more general notion of oscillating challenge over time.

Our model attributes high values of fun to levels containing rhythm groups that resemble those found in actual games, and its ultimate purpose is to evaluate and guide the output of our generative system. For the model to be deemed useful, it must reward levels which exhibit a number of important properties that are noticeably present in real-world levels.

1) Level Test Cases: Twenty-eight levels taken from *Super Mario Bros.* were used to inform the design of the model. These levels were chosen according to the degree to which their design contributed to the ultimate challenge dynamics of the game. This criteria excluded "boss levels," as the difficulty of these levels is primarily determined by the patterns of movement of the final enemy. Some other levels were excluded, such as the ones containing a "Cloud Koopa" enemy, who follows the player, hurling new enemies for the duration of the level. Underwater levels were also excluded due to their significant difference from the rest of the game. Excerpts from some of the 28 chosen levels are shown in Fig. 1.

The levels were reconstructed by Ian Albert from recorded screen captures of play sessions, which are made available on his website [25]. We convert these screen captures into a format more amenable to analysis through computer vision techniques; the basic sprites corresponding to the various types of blocks and enemies are identified, and the location and distribution of each object type is found through template matching. We determine enemy locations from their sprite coordinates and find hole locations by detecting gaps in the blocks located at the bottom of the level. This information is converted into a time-series representation of the "challenge events," a signal that is zero everywhere, except for unit impulses at the places where enemies or holes are located. The particular distribution of these challenge events is what we hypothesize as predominantly affecting the amount of fun had by players. Examples of some of the resulting time series are shown in Fig. 2.

From examining the 28 test cases, we draw the following generalizations, which we will later use as evaluative criteria for our model.

1) Too much continuous difficulty is undesirable. Though each rhythm group presents the player with a high degree of challenge, it is not so great as to cause frustration and a reduced sense of pleasure for the player.

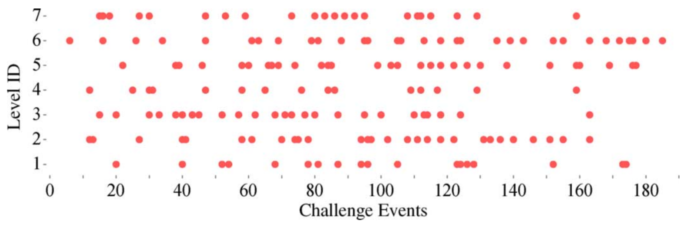


Fig. 2. Depiction of the challenge events of seven of the 28 levels used to inform the design of the model of fun. Horizontal scale is in 16 × 16 block units. Note the visible clustering of challenge events into rhythm groups.

2) Rhythm groups are not necessarily strictly periodic. They certainly exhibit a cyclical pattern; however, there is no evidence that there is a predominant frequency to the amplitude of challenge over time. Levels with variously spaced rhythm groups should not be penalized.
3) The model should account for players of different skill levels. The model should not be fragile in the sense that it applies only to a single, idealized player but should rather be adaptable to a wide variety of players through the manipulation of a few, semantically meaningful parameters.
4) Rhythm groups should conform to a reasonable scale on the order of seconds; it should not be possible for a generative system to exploit the model with obvious degenerate constructions, such as levels containing a single rhythm group lasting for the entire duration of the level. The model is most useful for the purposes of level generation if it is sensitive on a small scale and is able to identify even minute improvements in a level's layout.

C. Model Formalization

1) Challenge: Because rhythm groups are defined in terms of challenge dynamics, our model presupposes the existence of a suitable technique to measure the change of challenge over time. It is outside the scope of this work to address the notion of challenge generally; thus, we must assume that the model is provided with a challenge metric $c(t)$, which returns, for a given level, the degree of challenge at the time t. Certainly, the manner in which this value is calculated will vary depending on the game context. For example, with *Mario* we use a challenge function that identifies a portion of a level with any given value of t, and associates difficulty values based on the design of the level at that given point. In particular, a large gap with a small margin of error for mistakes will be attributed a relatively high value of difficulty, whereas a large, straight segment with no enemies will be given a challenge value of zero.

The construction of the challenge metric entails certain simplifications. For one, we must treat challenge as a single dimensional value. We also assume that the challenge metric is always nonnegative. While the particular values do not matter (all we are concerned with is relative ordering), we take zero to represent the lowest possible challenge experienced in the game. We also depict challenge as a value that can be sampled at a single point t. Sections IV and V provide example formulations for $c(t)$ in the context of two different games and demonstrate the modeling of challenge as instantaneous impulses, that is, formulating $c(t)$ as a sum of Dirac delta functions (unit impulses), as illustrated in Fig. 3. Finally, we take the value of challenge to represent the difficulty of a certain level segment absolutely, that is, irrespective of player skill. Thus, if we were to suggest a possible unit of measure for the challenge metric, these units would be constant for everyone, not relative to a particular personal experience of that level. Again, this does not prove to be a problem as the model provides threshold parameters which account for the fact that skilled players would be capable of enjoying higher degrees of challenge than a player with less skill. In other words, skilled players could be exposed to a higher amount of these hypothetical "challenge units" before becoming frustrated.

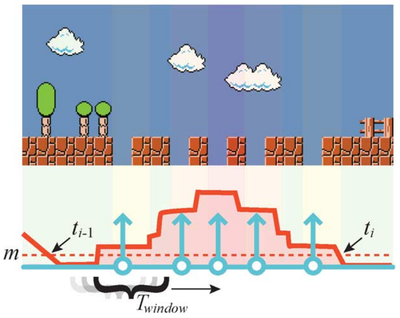


Fig. 3. Illustration of rhythm group i in the context of *Mario*. Vertical arrows represent the challenging events (holes) as unit impulses, and the curve represents the amount of accumulated challenge in the time window. Rhythm-group boundaries are located at points t_{i-1} and t_i, because the windowed challenge temporarily decreases below the threshold m. The accumulated challenge in the entire rhythm group c_i corresponds to the integration of the impulses located between boundaries t_{i-1} and t_i.

It must be emphasized that because $c(t)$ is ultimately used as a component of an automatic fitness function, it needs to be calculated without any human input; it must be possible to find a reasonable estimate of a level's configuration of difficulty over time simply by analyzing the level's layout. This is the primary reason why we restrict our discussion to challenge-based games in which the difficulty is mostly a function of the level design.

2) Modeling Fun: The determination of a level's quality consists of a two-pass process. First, the level is partitioned into a set of n rhythm groups with boundaries located at times $t_0, t_1, \ldots, t_n$. The identification of rhythm-group boundaries is governed by a greedy algorithm that identifies periods of sufficiently low challenge, which are identified as periods of relaxation. A window of size T_{window} is shifted along the challenge function, and positions the boundaries at points where the total amount of challenge in the window is less than the threshold m. More precisely, boundaries are located at positions t where $\int_{t-T_{\text{window}}}^{t} c(t)dt \leq m$. After a boundary is associated at a given point in time, the window does not place any more boundary points until after it has witnessed a period where the challenge temporarily exceeds m. Otherwise, extended periods of low challenge would be identified with a dense interval of infinitely many rhythm groups. Put more simply, the greedy process only identifies a new period of relaxation until after an intervening period of challenge has elapsed. A rhythm-group boundary is always placed at the beginning and end points of a level, and because the process is greedily run from the beginning to the end, it produces a unique segmentation.

With this segmentation in place, it is possible to identify the level with a fitness value. A level is rewarded for each rhythm group that contains the appropriate amount of total accumulated

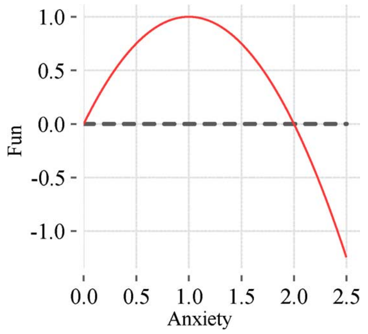


Fig. 4. Fun f as a response to increasing anxiety (accumulated challenge) c_i when $M = 1.0$, in the context of a single rhythm group. The response is defined by the function $f = (2c_i/M) - (c_i^2/M^2)$.

challenge, which we refer to as "anxiety," specified by the upper threshold M. Formally, if the amount of anxiety contained in rhythm group i is given by $c_i = \int_{t_{i-1}}^{t_i} c(t)dt$, then the amount of fun f attributed to the level as a whole is defined by

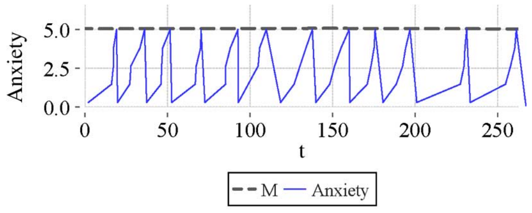
$$f = \sum_{i=1}^{n} \frac{2c_i}{M} - \frac{c_i^2}{M^2}. \quad (1)$$

D. Model Characteristics

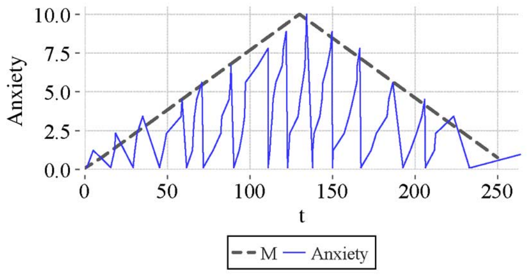
The numerical response of the model is demonstrated in Fig. 4, which illustrates the amount of fun in a particular rhythm group as a function of the accumulated challenge in that rhythm group. Recall that accumulated challenge—that is, challenge integrated over a period of time—is referred to as "anxiety." In other words, where $c(t)$ represents the amount of challenge present at the instantaneous point t, c_i represents the total amount of challenge integrated over the duration of rhythm group i, which constitutes a quantity of anxiety. The rhythm group attains its maximal fun potential when the amount of anxiety present is exactly M. The fun provided by a rhythm group decreases if the amount of anxiety experienced in that group is greater or lesser than this critical point. This function is evaluated independently for each rhythm group, and the fun for the entire level is the sum of each independent evaluation.

We define the formula depicted in Fig. 4 in order to mimic the famous "inverted U" shape described by the Yerkes–Dodson law [20], which essentially states that task performance is optimal if arousal is neither too high nor too low. Piselli *et al.* [21] have shown that this phenomenon is equally applicable to the context of video games, even when considering pleasure as a function of difficulty instead of task performance. Our model, then, idealizes this phenomenon within a computational setting, and applies it at the scale of individual rhythm groups.

To illustrate the effect of using this model to guide a generative system, we employ the model as a fitness function in a GA that evolves challenge time series directly, with no reference to an actual game context. The genotype is a variable-length list of challenge locations, which corresponds exactly to our representation of the 28 levels drawn from *Mario*. We use typical GA settings, with crossover at 0.9 and mutation at 0.05, and stop the evolutionary run when progress has stalled for ten generations. The results are shown in Fig. 5.



(a)



(b)

Fig. 5. Challenge time series with high fitness, as induced by the model with $T_{\text{window}} = 10$, and $m = 1$. The curve depicts accumulated challenge in each rhythm group (c_i). Notice that the anxiety in each rhythm group attains M. (a) Constant M. (b) Varying M.

Our model satisfies the four motivating properties outlined in Section II-B. First, it is evident that the model penalizes levels with excessively high difficulty values; the reward for a given rhythm group attains its maximum at the value M and decreases quickly after that threshold has been surpassed. Second, there is nothing in the formulation that encourages strict periodicity; fitness is rewarded solely based on the amount of challenge present in a rhythm group and is not predicated on the rhythm group conforming to any specific width.

As well, the model can easily and intuitively be adjusted to account for players of different skill levels. The parameter M corresponds to the skill of the player, and can be raised or lowered to create levels with higher or lower levels of difficulty. As Fig. 5(b) shows, this parameter can even be adjusted over the course of a single level, providing the designer with the ability to control the overall arc of challenge at a high level, in this case creating a level with a very challenging midpoint with easier portions at the beginning and end. It would similarly be possible to adjust m dynamically in order to raise or lower the lower bound of anxiety required to trigger rhythm-group boundaries.

Finally, the above model does not lead to degenerate cases, as rhythm groups that are too long impose a low upper bound on the total amount of fun that can be attributed to a level. If a level were to be identified as consisting of a single rhythm group, then the amount of fun attributed to that level would be, at most, 1. Any level with more numerous (and smaller) rhythm groups will certainly be able to exceed that value and be favored for selection. On the other hand, rhythm groups must be at least a width of T_{window}, which places a lower bound on their size. In this way, these two bounds ensure that rhythm groups exist at a scale that can allow a level design to be analyzed at a meaningful resolution—on the order of seconds, not minutes.

E. Learning Parameters

Provided that a suitable challenge metric $c(t)$ is defined and that values are specified for the parameters M, m, and T_{window}, the model is able to estimate a level's entertainment value. However, it is possible to perform the reverse operation, beginning with a set of levels of a known entertainment value and working backward to infer the specific model parameters that reproduce the observed values. This task is an instance of an expectation maximization problem, with the goal of finding model parameters that account for an observed set of example levels. By following this procedure, it becomes possible to mimic the particular challenge characteristics of an existing game; given a set of levels that are considered to be well designed, the model can be trained to reward levels with similar challenge configurations. This could be an effective way to expand the content of a game, as any automatically generated levels would exhibit the same difficulty patterns as the human-designed levels.

The inference of the model parameters also serves as a means of validating the model itself. If certain parameters allow the rhythm-group model to successfully distinguish between well-designed and poorly designed levels, it stands to reason that the model is sensitive to relevant characteristics of high-quality level design. Indeed, this validation is of critical importance when asserting the usefulness of analyzing levels in terms of rhythm groups. We do not currently rely on qualitative, subjective evaluations of fun (such as questionnaires) to evaluate the model; instead, we can assess the model's effectiveness by treating it as a classifier of existing, real-world levels, judging its performance in the same way many other machine learning techniques are judged. Furthermore, this approach provides an avenue of testing the model independently of any particular generative context. Evaluating the output of a generative system that is based on the rhythm-group model does not necessarily demonstrate that the model, itself, is responsible for the quality of that output; it could be the case that the generated levels are fun only because of some felicitous property of the generative system. However, in evaluating the model in isolation, we can more confidently defend the use of rhythm groups as a meaningful analytical tool.

1) Classification Results: It proves to be the case that the model is, indeed, able to successfully distinguish between the challenge time series of the 28 levels taken from the original *Super Mario Bros.* game, which are considered to be examples of good level design, and 30 time series that were crafted to represent examples of poor level design. We constructed the 30 negative examples to represent properties that would be obviously undesirable in well-designed levels, arguing that the model would certainly need to be able to identify these levels as poorly designed before it could be seen as effective. In that sense, this experiment establishes baseline functionality and serves as a "sanity check" of the model's usefulness.

Because we have seen that actual levels have no regular periodicity, we have generated negative examples that do exhibit a regular, periodic structure. We have 14 negative examples intended to represent levels that are clearly too difficult. Some contain challenge impulses located 1–3 units apart for the entire duration of the level, while others contain bursts of 20 contiguous challenge impulses located between 20 and 30 units apart. Similarly, we have 16 examples of levels that would be too easy, with single challenge impulses separated by 20–30 units of space.

TABLE I
MEAN AND VARIANCE FOR THE TEN OPTIMAL PARAMETER SETTINGS FOUND THROUGH CROSS VALIDATION

Parameter	Mean	Variance
T_{window}	9.5	6.6
M	7.3	3.5
m	1.5	0.2

The model can be converted to a classifier through the addition of an extra parameter θ, which represents the threshold of fun above which a level is considered to belong to the class of well-designed levels. For the sake of convenience, we train the classifier using the evolutionary system we have in place, which was, indeed, able to find effective model parameters for this classification task. We treat model parameters as individuals in a population and define a fixed-length genotype defined by the tuple $(T_{\text{window}}, M, m, \theta)$. The fitness of a set of parameters is given by the proportion of correct classifications of the training data. Optimal values were routinely found within the span of a few generations, and evolution was stopped if there was no fitness improvement after five generations.

The experiment was conducted as a tenfold stratified cross validation by randomly partitioning the 58 training examples (consisting of the 28 real *Mario* levels and the 30 hand-designed negative examples) into ten groups of five or six examples, with roughly three positive and three negative examples per group. The model was then trained on each of the ten different groups formed by removing one of the sets for validation purposes (so that the same data points were never used for both training and validation). The model was quite successful at distinguishing our real levels from the poorly designed levels; with only two false negatives, it achieved precision score of 1.0 and a recall score of 0.93.

Also encouraging is the fact that the optimal parameters conformed to their intuitive roles in the function. As shown by their mean values in Table I, rhythm groups corresponded to periods containing an average of seven challenge impulses, and inspection of the 28 example levels reveals visible clusters of challenge events containing roughly that many items. It is reasonable that a period of about ten blocks with only a single hole or enemy (according to T_{window} and m) would constitute a rhythm-group boundary. In other words, it is a reassuring result that the parameters which correspond with an intuitive, visual inspection of actual *Mario* levels are precisely the parameters that lead to successful automatic classification under the proposed model.

III. IMPLEMENTATION

The generative process is ultimately a search through the space of possible designs. The system attempts to find a particular level that demonstrates a good configuration of rhythm groups and that possesses, therefore, a high fun value. This section goes into more detail regarding the particular techniques used to traverse this space successfully.

At the core of the approach lies a GA, for which each potential level design is represented by a genetic encoding. We extend the basic algorithm with new features that help to overcome some difficulties associated with evolutionary search. Constraint satisfaction (CS) methods are employed to form a hybrid system that effectively optimizes the value of fun for levels while simultaneously observing the strict constraints inherent to level design.

A. Problem Domain

Level design remains a challenging AI search problem for two primary reasons. First, it is a task characterized by high dimensionality; a single level design in our system contains hundreds of degrees of freedom. GAs are an effective tool when approaching this kind of problem, as they are well suited to such high-dimensional search spaces. However, level design is also a highly constrained task. Level elements must be arranged in such a way as to ensure that the player is able to traverse the level. For example, in *Super Mario Bros*, if a platform was placed too far from a ledge for a player to reach, the entire level would be rendered unplayable. An objective function would typically associate completely broken levels such as this with a fitness value of zero. Because a small change in the positioning of a single element of a level can drastically change the objective quality of the level, the problem is said to have a highly discontinuous fitness landscape, suggesting the problem is poorly suited to an evolutionary approach. In cases where the constraints between solution elements are critical, CS methods are more appropriate.

The generative approach presented in this paper is constructed to address both concerns simultaneously, and is an example of a hybrid constraint solver and evolutionary system [26]. Such systems strive to observe the constraints of the problem domain while exploring a high-dimensional search space in order to maximize an objective fitness function.

B. Genetic Representation

The genetic representation of a level is a variable-sized, unordered set of design elements (DEs). Design elements are atomic units that combine to form a game level. Intuitively, DEs represent the components a human level designer would arrange when manually constructing a level for a game. For example, a single enemy in *Mario* is represented as a DE. A given game will include a number of different types of DEs, which together express the breadth of elements available to the game designer. More detailed examples of constructions of DEs for specific games are offered in Sections IV and V.

Each type of DE is defined by a number of parameters and is essentially a tuple containing floats, integers, and Booleans that represent the characteristics of that DE. For example, in an adventure game, an enemy DE might have two dimensions representing its horizontal and vertical position, one dimension representing its strength, and a Boolean dimension determining if it is armed or unarmed.

1) Genetic Operators: Mutation is accomplished with respect to a single DE. A single mutation operation can either be the addition of a new random DE to the genotype, the deletion of a DE from the genotype, or the modification of one of a DE's constituent property dimensions. A new DE can be created by selecting from one of the game's basic types of DE and setting its dimensions to random values in their respective domains. Mutation of a DE is achieved by selecting a new value for a random parameter. If the parameter is a real-valued number or an integer, the parameter is perturbed to a degree defined by a Gaussian distribution. If it is an unordered categorical variable (including Boolean parameters), it is set to a new allowable value with uniform probability. Each DE dimension can also be associated with a scaling parameter that affects both the scale of the Gaussian distribution and the variance of the mutation that is applied to a particular dimension.

The crossover operator is similar to variable-point crossover but modified slightly to be compatible with our representation. Standard variable-point crossover is achieved by picking a random cut point in the two parent genotypes and swapping two halves of each split parent genotype to create two new offspring. Our genotype representation consists of an unordered set of DEs, and typical crossover operators are defined in terms of ordered, linear genotypes, so standard variable-point crossover cannot be applied directly. However, because our DEs represent substructures with spatial position, we can impose a linear order by sorting along a spatial dimension. This approach is applicable to any n-dimensional space; every crossover involves picking a dimension at random, sorting by that dimension, and behaving exactly as a variable-point crossover on the now-linear representation. For example, in a 2-D context, the parents will be split by a random horizontal or vertical plane, and the offspring will be formed by taking all the DEs that lie to one side of the plane from the first parent, as well as all the DEs that lie on the other side of the plane from the other parent.

This approach serves to draw together within the genotype DEs that represent level structures that are in close proximity, providing the property known as gene linkage [27]. An important aspect of any genetic representation is the strength of the gene linkage, which determines the efficacy of the crossover operation in preserving useful modular substructures. In the worst case, when the DEs have an arbitrary ordering, the GA degrades into regular hill-climbing (albeit with large, random, and disruptive changes interspersed with smaller mutations). Strong gene linkage, however, is what enables GAs to naturally preserve high-fitness substructures throughout a population, which would be otherwise destroyed through small-step mutations.

C. Constraint System

CS methods are added to the typical GA structure in order to address the challenges of a highly constrained solution space with a discontinuous fitness landscape. We use CS to repair the genotypes that are subjected to breaking changes. We use two distinct forms of CS, which address two particularly relevant forms of level design constraints. Constraints can be formulated as simple, local, spatial relations such as "the object X must not overlap the object Y." These constraints can be solved with the "Tier 1" CS system, which immediately alters the genetic representation to directly satisfy the constraints. Not all constraints can be easily expressed in terms of local, spatial relations, however. For example, the constraint "there must be an unblocked path between the points A and B" cannot be easily expressed as a geometric constraint between two elements. These more complex constraints are handed by the "Tier 2" constraint system.

1) Tier 1 System: Tier 1 is an example of a typical constraint solving algorithm that employs variable selection, domain pruning, and backtracking. We use, specifically, the JaCoP open source Java constraint solving library [28] as the foundation of our approach and modify it to better suit our use of it as a reparation step in a GA. JaCoP is particularly useful as it features a geometric constraints module which allows many constraints typical of spatial arrangements to be straightforwardly expressed and efficiently solved.

Because of JaCoP's role as a genotype reparation step in a larger process, we are concerned with more than simply finding a set of values that satisfy the problem constraints. We want to ensure that the reparation process modifies an existing genotype as little as possible in its attempt to provide a viable solution. The benefits of this are twofold. First, because the GA uses the rhythm-group model to evaluate designs, the more we alter a given level design to satisfy game-specific constraints, the more likely we are to disrupt the rhythm-group structure and reduce the effectiveness of the model in producing fun levels. A second benefit to altering the genotype as little as possible is the possibility of allowing human designers direct control over portions of the system's output. We can use the same machinery that minimizes genotype modifications to ensure that the system respects the designer's adjustments to the level and alters the level in such a way to minimize disruptions to content made by human designers.

Our CS problem can then be framed as a search for values for the DE dimensions that minimize a cost function that represents how much they are altered. This is achieved by extending the JaCoP CS library with alternate variable selection and value selection processes. Constraint solving requires picking both a variable to alter and a new value for that variable and our approach is to prioritize the choice of values so as to reduce the potential negative impact. In other words, if the original value of a certain DE dimension, as set by the GA, is 3, the values are chosen in the order of increasing distance, e.g.: $[4, 2, 5, 1, 6, 0, 7, -1, \ldots]$. We do not claim that this approach always produces the absolute global minimum disruption, but it is more effective at approaching this goal than an arbitrary value assignment.

Every individual that can be successfully repaired through the Tier 1 system is fixed and placed back into the population.

2) Tier 2 System: The Tier 2 system handles individuals that cannot be repaired through the Tier 1 process, and can satisfy certain constraints which cannot be easily expressed using the primitives provided by the JaCoP system. The subsystem, which is described in earlier work [29], is modeled after the feasible/infeasible 2-population GA (FI-2pop), developed by Kimbrough *et al.* [30]. The FI-2pop consists of two populations which are evolved in parallel, one labeled the "feasible population," which contains all the individuals that satisfy the constraints of the problem domain, while the other is referred to as the "infeasible population," which contains those individuals that do not satisfy the constraints. In our case, the feasible population contains all the levels that satisfy the constraints of the game in question as well as those individuals that can be repaired by the Tier 1 constraint solver so that they do not violate any constraints. The individuals which cannot be repaired, or which violate constraints that cannot be expressed in the terms of the Tier 1 subsystem, are placed in the infeasible population.

Whereas the feasible population is evolved according to our primary fitness function, that is, by the rhythm-group model of fun, the infeasible population is evolved according to a fitness function which seeks only to satisfy the still-violated constraints. This is done with a measurement of the degree to which a given level violates the set of constraints. By minimizing this function, levels ultimately reach a state where they violate no constraints, at which point they can be moved back into the feasible population. Because Tier 2 constraints are enforced through a fitness function, they can express any arbitrary, global constraint on a level design; they do not need to be limited to spatial relationships between individual level elements. An example of a global property that is difficult to express in terms of local constraints is connectivity—ensuring there is a traversable path from the beginning to the end of a level.

D. Summary

This architecture maintains all the advantages of the top–down approach. It allows the level design criteria to be described declaratively, irrespective of the actual generative implementation. Procedural generative processes are restricted to the reification of the DEs (through the genotype to phenotype mapping process) and our architecture provides a clear separation between the mechanics of the level creation and the evaluation process. Furthermore, this system is capable of respecting the designs provided by human content creators; the DEs could be provided by a designer through their usual level design editor and given special status in the system. The GA treats these DEs as fixed and immutable and does not alter them under mutation or crossover. Likewise, the CS system gives a higher weighting to the variables corresponding to the DEs that are provided by the human designer when altering their values, preferring to mutate automatically generated DEs provided by the GA than to change the DEs specifically placed by the human. Essentially, this amounts to a system that works around and with a human designer to "fill in the blanks," upsetting as little as possible both the guidance of the rhythm-group model and, more importantly, the designs provided by the human.

IV. Mario

In this section, we outline a concrete application of the model and generative framework in the context of an actual game, namely the 2-D platformer *Super Mario Bros*. First, we describe a challenge function for this game in order to apply our rhythm-group model. We present a way to model this design task in terms of DEs and constraints, and discuss the results.

The system produces levels that are directly playable. The implementation used is an open source clone of *Super Mario Bros.* called *Infinite Mario*. It is written by Markus Persson in the Java language and is currently used in several video game research problems, including the Mario AI Championship, where it functions as a platform for testing the performance of various AI character controllers, learning agents, and generative systems, such as our own. *Infinite Mario* implements many elements of the original game, including Super Mushrooms, Fire Flowers, Goombas, Koopas, Spiked Koopas, Bullet Bills, and Piranha Plants. It is not a completely faithful replication, though, as it does not include some game elements, such as moving platforms or the Cloud Koopa character, who hurls Spiked Koopas from the sky. However, a large proportion of the gameplay is still intact, and, judging by its popularity on online gaming sites such as NewGrounds [31], where it has been played by over 11 000 people, it can be considered a legitimate representative of the platformer genre.

A. Challenge Metric

One should recall that the rhythm-group model serves as the core of the GA, and, since the rhythm-group model is defined in terms of challenge, a method for estimating the challenge of a given level is required. In challenge-based games, difficulty arises from the precision required to execute a properly timed sequence of button presses. In *Mario*, difficult segments correspond to the locations where enemies are densely located and where the platforms are narrow. Easy segments are, conversely, the areas where there is very little precision required to successfully traverse the section. More technically, if we consider the set of all possible sequences and timings of button presses, a challenging section can be defined as sections where the ratio between sequences of button presses that result in successful traversals to sequences that result in unsuccessful traversals is relatively small. It is this notion which motivates the metric of Compton and Mateas [32], in which the challenge of a particular jump is defined as the ratio between the number of trajectories that traverse the gap to the number of unsuccessful trajectories which result in falling into the gap. Our challenge metric is similar, and is shown in the following equation, where $d(p_1, p_2)$ is the Manhattan distance between the platforms p_1 and p_2 minus the sum of the two "landing footprints," fp, of both platforms plus a constant:

$$c(t) = d(p_1, p_2) - (fp(p_1) + fp(p_2)) + 2fp_{\max}. \quad (2)$$

The landing footprint is a measurement of the length of a platform, bounded to the maximum distance a player can jump $fp_{\max}$. This measure is important, as there is a much greater margin of error when jumping to a wide platform than to a narrow platform, and it is a less challenging maneuver. The constant $2fp_{\max}$ is added to ensure that this difficulty measure is nonnegative.

We extend this basic formula to account for the challenge posed by the enemies in the level. Because all enemies in *Super Mario* can be defeated by jumping on, or over them, we can regard each enemy as a gap in the level. We measure challenge in the same way as with platforms, save for an extra constant C_e, which is added to account for the fact that since the enemies are moving, there is a slight increase in difficulty as opposed to a static hole of the same size. These parameters are all assigned values through the learning process outlined in Section II-E.

B. Design Elements

A *Mario* level is a grid of 250 × 15 block units with a floor consisting of blocks occupying the entirety of the 14th and 15th rows of the level array. The beginning and end level points, which are required by *Infinite Mario Bros.* to determine the spawn position and victory criterion, are set to be (0,13) and (230,13), respectively. Notice that, because levels contain a floor by default, holes must be created explicitly in the environment through DEs.

1) Basic DEs: The system is composed of the following basic DEs.

- *Block(x, y)*. This DE is a single block, parameterized by its *x* and *y* coordinate.
- *Pipe(x, height, piranha)*. A pipe serves as both a platform and a possible container of a dangerous piranha plant.
- *Hole(x, width)*. This specifies a hole of a given width in the ground plane.
- *Enemy(x)*. This specifies an enemy at the given horizontal location.
- *Platform(x, width, height)*. This specifies a raised platform of a given width and height.
- *Staircase(x, height, direction)*. Staircases are common enough to warrant a dedicated DE. The direction specifies whether the stairs are ascending or descending.

2) Compound DEs: Ten additional DEs are defined, which are used in conjunction to the basic set. These elements drawn from common elements seen in both the original *Super Mario Bros.* as well as one of its sequels *Super Mario World* [33], and represent compound arrangements of the basic DEs.

- *Blocks(x, height, width)*. Several blocks in a horizontal row.
- *Hill(x, width)*. A "background" hill that can be either jumped upon or bypassed.
- *Hill-with-enemies(x, height, n, type)*. A Hill with *n* enemies of the *type* variety.
- *Cannon(x, height)*. A cannon fires "Bullet Bill" enemies toward the player.
- *Steps(x, width, height, dir, n)*. This DE represents when the ground itself rises or declines (specified by *dir*) to *height* in *n* distinct steps.
- *Enemy-pit(x, width, type, n)*. This specifies a group of enemies at the given horizontal location, situated in a depression in the ground.
- *Enemy-pit-above(x, width, type, n, left, right)*. This specifies a group of enemies at the given horizontal location, bounded by rocks, pipes, or cannons, as specified by *left* and *right*.
- *Enemy-row(x, type, n)*. This specifies a group of enemies at the given horizontal location.
- *Coin-arc(x, height, n)*. This specifies a number of coins at the given horizontal location.
- *Impediment(x, type)*. A high pipe or wall that can only be mounted by jumping from another pipe, a row of blocks, or a background hill. This DE constructs both the obstacle and the helper object needed to cross it.

Though there is certainly additional complexity involved in specifying more DEs, each element is completely independent

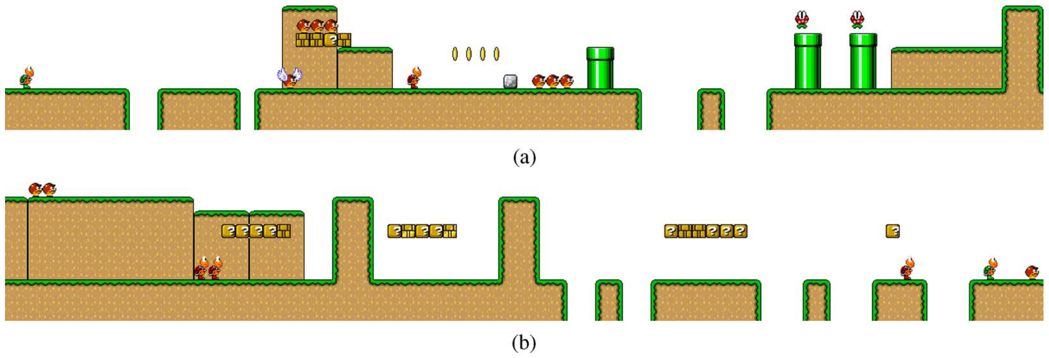


Fig. 6. Segments from final levels with extended set of DEs. (a) Generation: 151. (b) Generation: 138.

and can therefore be developed and tested independently. As well, though the number of constraints between DEs could increase exponentially as more are specified, many of the compound DEs are subject to identical constraints. For example, most of the DEs form a set of elements whose members cannot overlap with each other. Thus, it is not necessary to manually specify pairwise constraints between each DE.

3) Constraints: We define, in addition to the DEs, a number of constraints that express the requirements for a playable *Infinite Mario* level. As previously noted, each constraint in our system can specify a method for penalizing levels proportionally to how greatly they violate the constraint.

- *require-exactly(n, type)*. This constraint specifies the desired number of certain types of design elements to be present in the levels. As a penalty, it returns the absolute difference between the counted number of instances of *type* and the desired amount *n*.
- *require-at-least(n, type)*. This function penalizes levels that contain less than *n* of a given *type*, returning 0 if $n \geq$ *type* and returning *type-n* otherwise.
- *require-at-most(n, type)*. This function penalizes levels that contain more than *n* of a given *type*, returning 0 if $n \leq$ *type* and returning *n-type* otherwise.
- *require-no-overlap*($type_1$, $type_2, \ldots$). This function states that the specified types are not to overlap in the phenotype. It is, therefore, only relevant for design elements that contain a notion of location and extent. In the present application, we specify that pipes, stairs, enemies, and holes should not overlap one another. As a penalty, the number of overlapping elements is returned.
- *require-overlap*($type_1$, $type_2$). This function specifies that $type_1$ must overlap $type_2$, though $type_2$ need not necessarily overlap $type_1$. We use this function to require that platforms must be positioned above holes. The number of $type_1$ elements that do not overlap with a $type_2$ element is returned as a penalty.
- *traversable()*. This function is to ensure that a player can successfully traverse the level, meaning that there are no jumps that are too high or too far for the player to reach. This is determined using a greedy search between level elements. The penalty is the number of elements from which there is no subsequent platform within a specified range, that is, the number of places at which a player could get stuck.

All the previous functions are specified such that a value of zero reflects a satisfied constraint and a positive value denotes how severely a constraint is violated. Therefore, any individual level that is given a score of zero by all of the above functions is considered a feasible solution and is moved into the feasible population for further optimization. The feasible population is evaluated using our generic model of challenge-based fun. We adapt this model to 2-D platformers by providing a method for estimating challenge at any given point in a level. This is done by a function that returns a challenge value for each jump required between platforms, with difficult jumps being rated higher, and a set constant for each enemy in the level.

With no pressing concern for efficiency, we choose to set the mutation rate to 10% of individuals per generation and to generate the rest via crossover, using tournament selection of size 3. Finally, following the convention of Kimbrough [30], we limit the sizes of the infeasible and feasible populations to 50. Our stopping criterion is reached if the fitness of the levels does not improve for 20 generations. The evolutionary runs took between two to ten minutes on a midrange dual-core PC.

Fig. 6 depicts segments from some of the resulting levels. Figs. 7 and 8 depict levels generated with the model parameter M fixed at 6.0. These levels were the result of contiguous runs of the system (i.e., they were not singled out according to any subjective criteria). Fig. 9 demonstrates the effect of varying M. By doing so, levels are created such that the most difficult portions are located where M is the highest, in this case, in the center of the level. However, some challenge is still present throughout the level, and the player is provided with constant engagement. This ability to alter the model's parameters offers designers a unique, high-level way to influence the system's output, and illustrates some of the variety of design allowed by the system and the control that is afforded to the user.

Fig. 10 depicts another avenue for high-level control over the system output. In this case, a constraint was specified that there should be no *Hole* DEs in the level genotypes. The system is able to produce rhythm-group structures while observing this externally imposed design requirement. By restricting the presence of certain level elements in this manner, we can ensure that levels do not all contain the same proportion of level elements and can therefore maintain diversity in the system's output.

Fig. 11 demonstrates the potential for our system to allow for certain portions of the level to be directly authored by human designers. A small segment of a level is designed by hand and

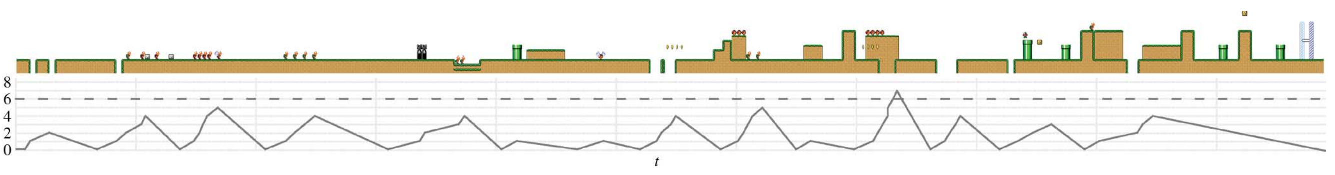


Fig. 7. Fixed M. Generation 270, fitness 11.09.

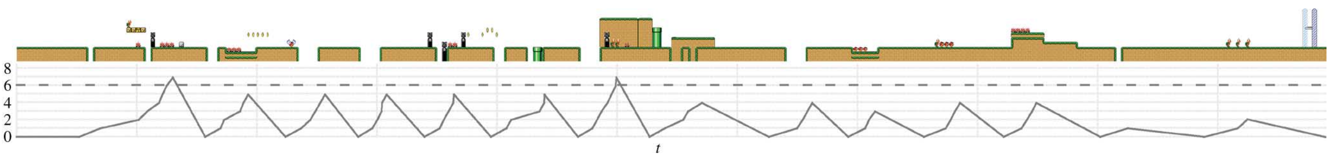


Fig. 8. Fixed M. Generation 365, fitness 12.17.

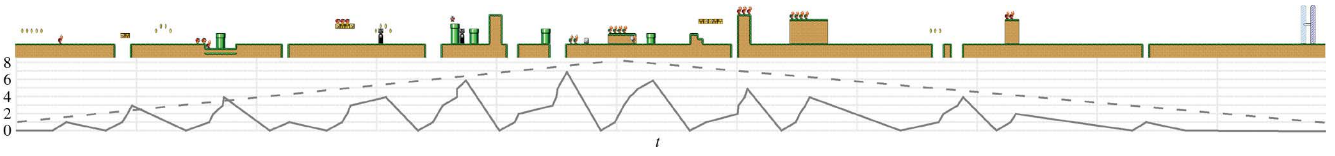


Fig. 9. Varying M. Generation 629, fitness 10.55.

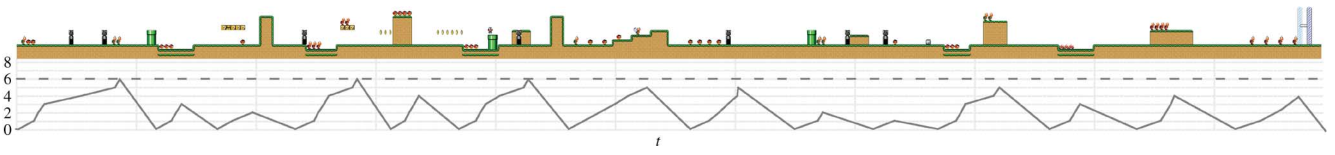


Fig. 10. Fixed M, with no *Hole* DEs permitted. Generation 202, fitness 11.50.

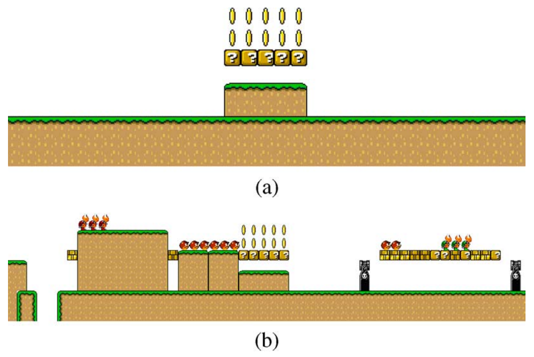


Fig. 11. The evolutionary system adds content surrounding the human-specified portion. Notice the highly challenging portions on either end of the relatively simple middle section. (a) Hand-specified design. (b) Automatically generated content added.

converted to its DE representation. This translation is always possible for *Infinite Mario*, as every basic level component has a corresponding DE. This translated portion is fixed in every genotype of the evolutionary population and cannot be altered by any crossover or mutation operators. Otherwise, the manually created portion is internally treated the same as the generated segment of the level and is subjected to the same constraints and fitness function. For this reason, the system is able to integrate human and artificial designs together in a manner that exhibits rhythm groups.

C. *Mario AI Championship*

This system was entered into the 2010 Mario AI Championship Level Generation Track, which was held at the 2010 IEEE Conference on Computational Intelligence and Games, Copenhagen, Denmark. Conference participants were invited to play levels generated by the various systems and to evaluate them according to how fun they were. The contest was arranged so that the systems had to produce levels that adhered to certain compositional requirements. For example, levels might be required to have four gaps and three Shelled Koopas. This requirement was put in place to discourage cheating through the use of systems that could merely return levels that were pre-designed by hand. To observe this rule, our system translated the composition requirements into constraints.

Fifteen participants took part in the event, and our system placed third out of six. This is an encouraging result, as our system, guided by a generic fitness function, was able to rank competitively with systems designed specifically for this particular game.

D. *Discussion*

One possible shortcoming of our approach is that the rhythm-group model cannot express diversity; there is no evolutionary advantage to producing rhythm groups that contain a mixture of different elements, as opposed to creating levels consisting of a single kind of element. For example, it is possible for a level containing only Koopa enemies to have the same rhythm-group configuration, and thus, the same fitness value, as a level containing a mixture of enemies and holes. This type of monotonous design does not seem to occur in practice because of the stochastic nature of the GA. On the other hand, this restricted diversity in a level's design might be considered desirable. In this case, the constraint system can be used to influence the variety of the level designs by enforcing a maximum or minimum amount of certain game elements. For example, it is common to introduce certain enemies only in later levels of a game; in this case the designer can set a constraint specifying that levels contain no DEs of this type. Fig. 10 depicts how this high-level control can be used to create a level with no holes. This kind of high-level control can ensure that the system produces levels that do not resemble each other but instead differ greatly in terms of composition and appearance.

Another potential criticism might be placed against the complexity of some of the design elements. Indeed, reifying some structures, such as staircases, requires an imperative set of construction instructions to be specified. This type of bottom–up, procedural approach may seem out of place in a framework that purportedly minimizes such low-level, game-specific code. It is important to emphasize that though the method to build a staircase is, in itself, a bottom–up, rule-based procedure, the instructions for doing so are parameterized: the overarching system can manipulate it on a high level by altering its height, width, and position, in the same manner as any other level element. In this sense, each DE is treated as a black box. This policy ensures that all the procedural knowledge contained within the DEs is insulated not only from the high-level system, but also from other DEs; each basic level element can be developed and tested independently. Ultimately, we do not claim to completely eliminate all traces of imperative generative techniques; rather we recognize that a certain amount of procedural specification is necessary but restrict the scope of such techniques, and subject them to a easy manipulation by the high-level system. Complex components can be seamlessly used in conjunction with the rhythm-group fitness function and constraint solver in exactly the same way as any other DE. This flexibility is not afforded by monolithic, rule-based production systems.

V. Zelda

To support our claims of generality, we present an application of the generative system to a different game genre. In this section, we target game levels that consist of rooms and doors arranged in a 2-D space and are viewed from an overhead perspective, as opposed to a side perspective as was the case in the *Infinite Mario* levels. We do not develop the construction process in as much detail as the previous example; instead of attempting to generate levels of comparable quality to commercial levels, we focus on generating levels that demonstrate a simple yet essential aspect of the level design task. The purpose of this simplification is to focus on the core problem of designing 2-D levels, as opposed to 1-D designs. It proves significantly more difficult to generate levels for this domain, but the fact that our approach is still able to efficiently create feasible solutions that exhibit a rhythm-group structure illustrates both our system's generality and its promise as a practically usable technique.

A. Game Background

Zelda is an action-adventure series developed and produced by Nintendo, which centers on the adventures of Link in the kingdom of Hyrule. In the course of a typical *Zelda* game, Link must successfully overcome the challenges of several dungeons. Each dungeon adheres to a recognizable pattern and consists of an arrangement of rooms filled with enemies, collectible items, and puzzles. For our purposes, we consider the top–down, 2-D gameplay characteristic of the earlier *Zelda* games, most specifically the original game: *The Legend of Zelda* [34]. Because finding an optimized arrangement of rooms is considered a difficult challenge for heuristic searches, our initial attempts to model this game involve significant simplifications: at this point we only attempt to produce dungeon layouts and monster placement and do not consider the puzzles, keys, and other aspects of the game. However, promising results in the simplified domain justify further experimentation.

B. Design Elements

The following DEs are sufficient to examine the problem of 2-D room layout as it relates to challenge dynamics.

- *Room(x, y, w, h)*. A room of dimension $w \times h$ with its origin at the point *(x, y)*.
- *Door(x, y)*. A door located at the point *(x, y)*.
- *Enemy(x, y)*. An enemy located at the point *(x, y)*.

C. Challenge Metric

The challenge function is defined in terms of the enemies the player faces as they move from room to room. Rooms can be viewed as sets of the entities they contain, so given a player who enters *Room(t)* at time t, the challenge at that point is defined as the number of enemies in that room, or, more formally, $c(t) = |\{e \in \mathit{Room(t)}\}|$.

This formulation inherently presupposes that the player's path through the rooms is fixed and that *Room(t)* represents a single value for each point t. We pick the shortest path movement between the entrance point and the exit point as the canonical path of the player through a given level. The entrance and exit points are explicitly defined before evolution. Certainly, this is a large simplification of the behavior a human would actually exhibit when moving through a complex virtual environment. However, our choice of shortest path can be justified by ensuring that the levels contain only a single, unique sequence of connected rooms from start to finish. In levels such as these, any path which does not double-back on itself will be the shortest path. It proves to be simple to ensure that the generated levels contain no multiple paths or dead-ends and thus adhere to this linear topology.

D. Constraints

The Tier 1 constraints that can be solved in terms of a CS formulation, are as follows.

- *Room location and dimension values must be multiples of 30*. This constraint is equivalent to snapping the rooms to a coarse grid with cell sizes of 30 units, which simplifies detecting the property of room adjacency.
- *Doors must exist on room edges*. This constraint ensures doors do not exist, for example, in the center of a room. Enforcing this constraint involves snapping each door to its nearest wall segment.
- *No overlap between rooms*. Enforcing that rooms cannot intersect one other is solved through the manipulation of the shape and position of the rooms. This is an example

of a 2-D geometric packing problem, which is more difficult to solve than the 1-D constraints used in the *Mario* application. However, the constraint solving library JaCoP provides geometric extensions that are able to efficiently solve this type of problem.

There is only a single Tier 2 constraint that cannot be expressed as a simple CS problem.

- *Connected path from the start point to the end point.* Two rooms are considered connected if they share a common edge and there is a door located on that shared edge.

To guide infeasible levels toward connectivity, we use the heuristics listed below. The heuristics are listed in order of the priority they are given when sorting the level designs. In other words, any level which receives a higher value under heuristic 1 will be given a higher fitness than any individual, irrespective of their heuristic 2 valuation. Similarly, any individual that maximizes heuristic 2 will be favored irrespective of heuristic 3. Note that, in this application, it is more convenient to specify the heuristics such that positive values are desirable, as opposed to the convention of positive values representing penalties.

- *Room on start or end points?* This heuristic detects if a room exists at the beginning and ending locations of the level, returning 0 if neither is the case, 1 if one point is covered, and 2 if there is a room on both points.
- *Max path length from start and end.* This heuristic measures the maximum distance that can be traveled from the start point and the end point.
- *Max path length from other rooms.* This heuristic sums together the maximum distance that can be traveled starting from every room in the level, other than the start and end point.

These heuristics serve to guide the infeasible population toward feasibility. By encouraging the maximum possible travel distance, the GA favors designs that attempt to connect multiple rooms together, increasing the probability that a design will be found that connects the beginning and end points together. This extra guidance is necessary, due primarily to the significant increase in the size of search space that the extra spatial dimension implies; every DE has two significant dimensions x and y instead of just x. Furthermore, a typical *Mario* genotype had approximately 40 DEs, whereas the *Zelda* levels we considered contained between 60 and 100 DEs. These two factors lead to a drastic increase of dimensionality, resulting in levels that would not achieve connectivity without some form of heuristic guidance. In a sense, these heuristics serve as an approximation of fitness on partial levels; instead of assigning a value of 0 to every unplayable level, we determine how much of the level is traversable (by measuring how far one is able to walk from a given starting point). Therefore, these heuristics are not entirely *ad hoc* and unmotivated, but serve as a kind of adaptation of the fitness function to the case of broken levels.

It should be mentioned that there is no explicit constraint that enforces the existence of a unique linear path through the level. Instead, this property is achieved through a postprocessing step. We can create a minimal genotype by attempting to remove each gene in turn, and replacing that gene if its removal affects the level's fitness value. Because we only represent the player's idealized movement in the $c(t)$ calculation as a single path, branches and dead-ends cannot possibly contribute anything to the fitness of a level, and are therefore always removed by this operation. Doors that lead nowhere are also removed from the genotype for this reason.

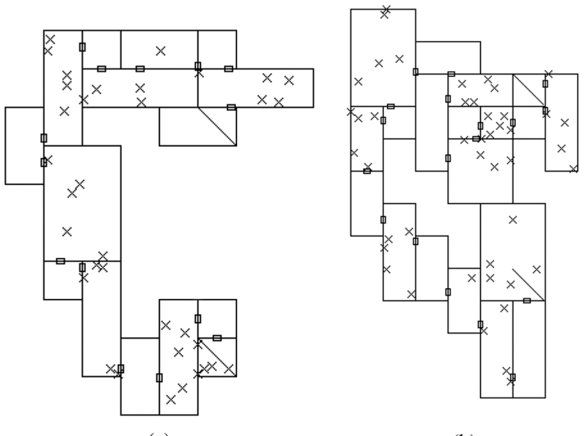


Fig. 12. Sample generated *Zelda* designs from two different runs. Small rectangles denote doors, crosses denote enemies, and diagonal lines denote start and end points. Note, in particular, the higher number of generations required to find feasible designs. (a) Generation: 1142. (b) Generation: 1471.

E. Results

Because it is difficult to arrive at feasible levels in this context, care must be taken when selecting the parameters for the evolutionary algorithm. For generating *Mario* levels, standard GA parameters were sufficient to enable reasonable evolutionary progress. However, these same parameter settings prove ineffective in the present case. To find useful values, several evolutionary runs were conducted under a number of different parameter settings, and the parameters which consistently produced the highest average rate of fitness improvement were selected. Evolution was conducted with five islands,[2] each containing 20 feasible individuals and 20 infeasible individuals. Crossover was found to be optimal at a rate of 0.7, with mutation at a probability of 0.03 per gene. There was elitism of a single individual, and interisland migration was conducted every 50 generations. The stopping criterion is a lack of progress in all of the populations for 100 generations, as it was rare to witness any improvement after this point.

Two resulting levels are depicted in Fig. 12. The rhythm pattern structure is evident, with highly challenging, enemy-filled rooms separated by empty rooms. It is also obvious that all the levels satisfy the connectivity constraint; as expected there is a single, unbroken path between the start and end points. The evolutionary runs took between 15 and 45 min on a midrange dual-core PC.

1) Comparison to Mario Results: Though the generated levels appear simpler in appearance than the generated *Mario* levels and the formulation contains fewer DEs and employs a simpler challenge function, it is, in reality, far more difficult to generate a feasible level for this context than for *Mario*. The combined factors of the higher dimensionality and the

[2]The island model involves evolving several independent populations in parallel with occasional migrations between the populations, increasing the chances of escaping local minima in the search space. For more background, see [35].

connectivity constraint render it virtually impossible to produce a feasible level by chance alone. Indeed, there are a vastly greater number of possible broken levels than feasible levels; Compared to the *Mario* application, where 195 out of 1000 randomly generated levels prove to be feasible with no need for further CS, the *Zelda* application produces no feasible individuals at all from chance alone.

The minimal representation adopted in this context was likely responsible for the high computational cost of evolving levels. Level connectivity had to be attained through the use of two different types of elements (doors and rooms, corresponding to edges and nodes in graph terminology) aligned in a fragile configuration. These difficulties were not present at all in the case of *Mario*, where an empty level (consisting only of the floor, from beginning to end) was still considered feasible. A *Zelda* representation which treated doors as a dimension of a room DE (for instance, as Boolean flags determining if there are doors on certain sides of a room) would reduce the number of different types of elements that would need to be aligned to achieve connectivity, and therefore reduce the search space significantly. Choices of representation greatly affect the scalability of generative processes, and future work must consider these ramifications in more detail.

However, it is not necessarily the number of different DEs and the complexity of the challenge metric which predominantly influence the difficulty of generating levels for a given game; it appears that the nature of the constraints influences the difficulty of generation to a far greater degree. For this reason, it is probable that more DEs and a more precise challenge metric could be developed to allow the system to generate more interesting Zelda levels, so long as the connectivity constraints do not become more difficult to satisfy. There is likely a reasonable flexibility within the design space provided by the existing design constraints of single-path connectivity. It is even possible that single-path connectedness is a more fragile, and hence a more difficult, constraint to satisfy than constraints over multiply connected spaces, and levels that admit multiple paths might prove to be easier to generate.

In any case, the results of this intentionally simple design context provide evidence that this generative system can be successfully applied to contexts quite different than 2-D platformers. This application is not intended to demonstrate the maximal amount of detail and complexity that can be produced by the system (which was demonstrated in the *Mario* application) but rather to determine the system's ability to satisfy difficult design constraints in a different game genre. Indeed, by using exactly the same fitness function and genetic operators (with some parameter adjustments) viable levels were generated in a significantly different context while maintaining all the advantages of the top–down approach: the model offers high-level parameters which serve to alter the layout of the rooms to provide higher or lower levels of difficulty, and the system is still able to incorporate human-authored content by adapting the output surrounding fixed components specified by a human designer.

VI. Future Work

Many avenues for future research are readily apparent. Due to the initial success of the *Zelda* context, it would seem promising that a more comprehensive set of DEs, such as the ones defined for *Mario*, could lead to levels more closely resembling those from the actual game. More constraints could be considered to model the puzzle aspects of the game. For example, keys must be attained to access various parts of the dungeons, which provide the player with various items needed to successfully defeat the dungeon's final boss. These complex constraints seem well suited to being specified as Tier 2 constraints, in a manner similar to the connectivity constraint. As well, it would be interesting to consider levels containing multiple paths, instead of requiring singly-connected paths. It might be possible to allow multiple paths by aggregating together the results of running each possible path independently, via the mean, minimum, or maximum output from the model. As well, many of these constraints can be expressed in terms of graph grammars, and generative grammars have been applied to adventure game mission creation by Dormans [36]. As generative grammars can be evolved with GAs, it would be interesting to see if such an approach could be combined with the model of player enjoyment presented in our work.

It is also possible that this approach could be generalized to even more types of games. Arcade games such as *Breakout* [37] and *Space Invaders* [38] could have the arrangement of blocks and enemies determined by the rhythm-group model. As well, layouts for first-person shooters could be generated in a manner very similar to the dungeon layouts as seen in the *Zelda* application. This could prove especially lucrative for large, open world games where a large amount of content is required.

It would also be desirable to further test the model. Though we have had the system's output indirectly evaluated at the Mario AI Challenge, it would be interesting to see if there was any discernible correlation between what players found fun, and what the model predicted as fun. Similarly, one could attempt to train the model not on positive examples taken from commercial games, as was the case in this paper, but rather to train the model to learn a particular player's preference, as witnessed by their subjective evaluation of play. There is also research in the automatic detection of player frustration in games [39], as well as on statistical methods for modeling player preferences [11], [40]–[42], and it would seem that those efforts could be fruitfully combined with the current model.

Finally, because level generation takes on the order of tens of minutes, it is not currently well suited to online level generation, where content is created in real time as the game is being played. It is possible the generation time could be reduced by creating smaller portions of game levels at a time instead of creating entire levels, as is currently the case. Even so, it may be feasible to generate levels on the client's system, even if they are not immediately available. For instance, should a player be found to be failing too often, the system could begin generating a new level in the background, and switch the current level out for the easier one when it is finished. If these background generation processes were seeded with initial populations of levels with relatively high quality, it is quite possible that acceptable variations for different model parameters could be found much more quickly than the current method of constructing new levels from scratch.

VII. Conclusion

We have demonstrated an approach to the generation of video game levels that is founded on an explicit model of the relationship between challenge and fun. The model is based on the notion of rhythm groups, which are alternating periods of high and low challenge that present a player with an engaging gameplay experience. It identifies fun as a function of challenge with an "inverted-U" shape inspired by the Yerkes–Dodson law, where a particular rhythm group is deemed fun if it is neither too difficult nor too easy. The model's effectiveness was evaluated by employing it in a classification task, where it was able to identify levels from the original *Super Mario Bros.* with high accuracy.

We have also demonstrated the generality of the approach in applying the model to two different games. It was relatively straightforward in both cases to represent the design problem in terms of a set basic design building blocks, referred to as DEs, and a collection of geometric constraints. We also noted that the size and complexity of the DE set does not necessarily mean that levels will be more difficult to generate. Although the *Zelda* formulation had a much simpler representation, it proved, by far, more challenging to produce feasible levels in this context.

Finally, our model provides parameters that correspond directly to intuitive concepts. In particular, the parameter M corresponds to the skill of the player, and can be adjusted by the designer to create levels with various challenge profiles. This high-level control is not typically offered by bottom–up, rule-based approaches where the relationship between the generative parameters and the final output is not always clear. Furthermore, humans can directly specify certain portions of the level by hand, which are then evaluated by the model in the same manner as the automatically generated content. This results in the rhythm-group structure adapting itself around the manually created portion of the level; easy portions are surrounded by difficult sections, whereas simple stretches are surrounded by areas of high challenge. This natural adaptation to externally provided content is afforded by the top–down design of the system. It is our hope that by modeling challenge dynamics in a high-level, explicit manner will not only improve the quality of procedurally generated content, but also contribute toward further research in the analysis and understanding of the nature of fun in video games.

References

[1] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation," in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2010, vol. 6024, pp. 141–150 [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12239-2_15

[2] M. Toy, G. Wichman, K. Arnold, and J. Lane, "Artificial intelligence design," 1983, Rogue.

[3] The NetHack DevTeam, Nethack, 2009 [Online]. Available: http://www.nethack.org/

[4] P. Rivest, *Far Cry 2*, Ubisoft, 2008.

[5] The Freeciv Developers, Freeciv, 2010 [Online]. Available: http://freeciv.wikia.com/wiki/Main_Page

[6] C. Remo, "MIGS: Far cry 2's guay on the importance of procedural content," *Gamasutra*, Nov. 2008 [Online]. Available: http://www.gamasutra.com/php-bin/news_index.php?story=21165

[7] C. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Comput. Sci. Dept., Queensland Univ. Technol., Brisbane, Qld., Australia, 2008.

[8] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. IEEE Symp. Comput. Intell. Games*, 2008, pp. 111–118 [Online]. Available: http://togelius.blogspot.com/2008/12/automatic-game-design.html

[9] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Interactive evolution of particle systems for computer graphics and animation," *IEEE Trans. Evol. Comput.*, vol. 13, no. 2, pp. 418–432, Apr. 2009.

[10] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proc. 5th Int. Conf. Found. Digit. Games*, New York, 2010, pp. 209–216.

[11] N. Shaker, G. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proc. AI Interactive Digit. Entertain. Conf.*, 2010, pp. 63–68.

[12] R. Hunicke, M. LeBlanc, and R. Zubek, "MDA: A formal approach to game design and game research," in *Proc. 19th Nat. Conf. Artif. Intell. Challenges Game AI Workshop*, 2004, pp. 1–5.

[13] T. W. Malone, "What makes things fun to learn? Heuristics for designing instructional computer games," in *Proc. 3rd ACM SIGSMALL Symp./1st SIGPC Symp. Small Syst.*, New York, 1980, pp. 162–169 [Online]. Available: http://dx.doi.org/10.1145/800088.802839

[14] M. J. Apter, "A structural-phenomenology of play," in *Adult Play: A Reversal Theory Approach*, J. H. Kerr and M. J. Apter, Eds. Amsterdam, The Netherlands: Swets and Zeitlinger, 1991, pp. 18–20.

[15] P.-A. Garneau, "Fourteen forms of fun," *Gamasutra*, Oct. 12, 2001.

[16] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. New York: Harper Perennial, March 1991.

[17] P. Sweetser and P. Wyeth, "Gameflow: A model for evaluating player enjoyment in games," *Comput. Entertain.*, vol. 3, no. 3, p. 3, 2005.

[18] R. Koster, *Theory of Fun for Game Design*. Sebastopol, CA: Paraglyph Press, Nov. 2004.

[19] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. Cambridge, MA: MIT Press, Oct. 2003.

[20] R. M. Yerkes and J. D. Dodson, "The relation of strength of stimulus to rapidity of habit-formation," *J. Comparat. Neurol. Psychol.*, vol. 18, pp. 459–482, 1908.

[21] P. Piselli, M. Claypool, and J. Doyle, "Relating cognitive models of computer games to user evaluations of entertainment," in *Proc. 4th Int. Conf. Found. Digit. Games*, 2009, pp. 153–160.

[22] G. Smith, M. Cha, and J. Whitehead, "A framework for analysis of 2D platformer levels," in *Proc. ACM SIGGRAPH Symp. Video Games*, 2008, pp. 75–80.

[23] N. Sorenson and P. Pasquier, "The evolution of fun: Automatic level design through challenge modeling," in *Proc. 1st Int. Conf. Comput. Creativity*, 2010, pp. 258–267.

[24] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2d platformers," in *Proc. 4th Int. Conf. Found. Digit. Games*, 2009, pp. 175–182.

[25] I. Albert, *Video Game Maps*, Sep. 2010 [Online]. Available: http://ian-albert.com/misc/gamemaps.php

[26] C. A. Coello Coello, "Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art," *Comput. Methods Appl. Mech. Eng.*, vol. 191, no. 11–12, pp. 1245–1287, Jan. 2002.

[27] G. R. Harik, "Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms," Ph.D. dissertation, Dept. Comput. Sci. Eng., Univ. Michigan, Ann Arbor, MI, 1997.

[28] K. Kuchcinski and R. Szymanek, JaCoP: Java Constraint Programming Library, 2010 [Online]. Available: http://www.jacop.eu/

[29] N. Sorenson and P. Pasquier, "Towards a generic framework for automated video game level creation," in *Applications of Evolutionary Computation*. Berlin, Germany: Springer-Verlag, 2010, pp. 131–140.

[30] S. O. Kimbrough, M. Lu, D. H. Wood, and D.-J. Wu, "Exploring a two-market genetic algorithm," in *Proc. Genetic Evol. Comput. Conf.*, 2002, pp. 415–422.

[31] Newgrounds Inc., "Everything, by everyone," 2010 [Online]. Available: http://www.newgrounds.com/

[32] K. Compton and M. Mateas, "Procedural level design for platform games," in *Proc. 2nd Artif. Intell. Interactive Digit. Entertain. Conf.*, 2006, pp. 109–111.

[33] S. Miyamoto, S. Hino, and T. Tezuka, *Super Mario World*, Nintendo, 1990.

[34] S. Miyamoto, T. Nakago, and T. Tezuka, *The Legend of Zelda*, Nintendo, 1986.

[35] M. Tomassini, "Island models," in *Spatially Structured Evolutionary Algorithms*, ser. Natural Computing, G. Rozenberg, T. Bäck, J. N. Kok, H. P. Spaink, and A. E. Eiben, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 11–18.

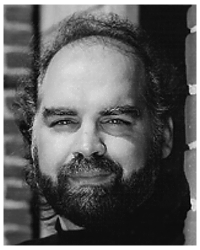
[36] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proc. Workshop Procedural Content Generat. Games*, 2010, pp. 1:1–1:8 [Online]. Available: http://doi.acm.org/10.1145/1814256.1814257
[37] N. Bushnell, S. Bristow, and S. Wozniak, *Breakout*, Atari, 1976.
[38] T. Nishikado, *Space Invaders*, Midway, 1978.
[39] R. Hunicke and V. Chapman, "AI for dynamic difficulty adjustment in games," in *Proc. Challenges in Game AI Workshop/19th Nat. Conf. Artif. Intell.*, 2004, pp. 91–96 [Online]. Available: http://www.cs.northwestern.edu/~hunicke/pubs/Hamlet.pdf
[40] G. Yannakakis and J. Hallam, "Towards capturing and enhancing entertainment in computer games," in *Proceedings of the 4th Hellenic Conference on Artificial Intelligence*, ser. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer-Verlag, 2006, vol. 3955, pp. 432–442.
[41] C. Pedersen, J. Togelius, and G. Yannakakis, "Modeling player experience in Super Mario Bros," in *Proc. IEEE Symp. Comput. Intell. Games*, Sep. 2009, pp. 132–139.
[42] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: Dynamic difficulty adjustment through level generation," in *Proc. Workshop Procedural Content Generat. Games*, 2010, pp. 11:1–11:4 [Online]. Available: http://doi.acm.org/10.1145/1814256.1814267

Nathan Sorenson is currently working towards the M.S. degree at the School of Interactive Arts and Technology, Simon Fraser University, Vancouver, BC, Canada.

With his background in mathematics and computer science, he researches the application of computational intelligence to problems that typically demand human creativity. Having worked as a research programmer on educational "serious games" in conjunction with the University of Calgary and as an independent video game developer, he is interested in advancing procedural content creation systems to enable smaller development groups to produce rich, interactive environments. His thesis focuses on formal models of fun in video games and automated level design. In addition to his work on content creation for games, he also works to employ genetic algorithms in the creation of art by developing tools for graphic designers to interactively "breed" procedural, vector-based designs.

Philippe Pasquier studied computer science and cognitive sciences in Europe, receiving the B.Sc. degree from the Université catholique de Louvain (UCL), Belgium, in 1998 and the M.Sc. degree from Nantes Science University, Nantes, France, in 1999. He then received the Ph.D. in artificial intelligence from Laval University, Sillery, QC, Canada, in 2005.

He joined the School of Interactive Arts and Technology, Simon Fraser University, Surrey, BC, Canada, in January 2008, as an Assistant Professor. He is both a scientist specialized in artificial intelligence and a multidisciplinary artist. As a scientist, his work has focused on the development of models and tools for endowing machines with autonomous, intelligent, or creative behavior. Contributions vary from theoretical research on agents and multiagent systems to applied research in computational creativity. As an artist, he has been acting as a performer, sound designer, composer, producer, jury, committee member, and teacher in many different contexts. He is serving or has served as a member or administrator of several artistic collectives (Robonom, Phylm, MIJI), art centers (Avatar, Bus Gallery) and artistic organizations (P: Media art, Machines, Vancouver New Music). His work has been shown internationally and supported by more than 20 scientific or cultural institutions including the National Sciences and Engineering Research Council, the Canadian Council for the Arts, the French Ministère de la Culture et de la Communication, the Australian Research Council and the Australian Council for the Arts.



Steve DiPaola received the M.A. degree from New York Institute of Technology (NYIT), New York, in 1991 and the Ph.D. degree from the University of British Columbia, Vancouver, BC, Canada, in 2011.

Active as an artist and a scientist, he is the Director of the Cognitive Science Program, Simon Fraser University (SFU), Surrey, BC, Canada, and leads the iVizLab, a research lab that strives to make computational systems bend more to the human experience by incorporating biological, cognitive, and behavior knowledge models. Much of the labs work is creating computation models of very human ideals such as expression, emotion, behavior, and creativity. He is most known for his AI-based computational creativity and 3-D facial expression systems. He came to SFU from Stanford University and before that NYIT Computer Graphics Lab, an early pioneering lab in high-end graphics techniques. He has held leadership positions at Electronic Arts, Saatchi Innovation and consulted for HP, Macromedia, and the Institute for the Future. His computer-based art has been exhibited internationally including the AIR and Tibor de Nagy galleries in New York City, Tenderpixel Gallery in London, U.K., and Cambridge University's Kings Art Centre, Cambridge, U.K. The work has also been exhibited in major museums, including the Whitney Museum, the MIT Museum, and the Smithsonian. His science work has been published in over 50 peer-reviewed science publications and showcased in the journal *Nature*.