Towards a Generic Framework for Automated Video Game Level Creation

Conference Paper *in* Lecture Notes in Computer Science · April 2010

DOI: 10.1007/978-3-642-12239-2_14 · Source: DBLP

2 authors:

Nathan Sorenson
Simon Fraser University
9 PUBLICATIONS **356** CITATIONS
SEE PROFILE

Philippe Pasquier
Simon Fraser University
221 PUBLICATIONS **2,895** CITATIONS
SEE PROFILE

Towards a Generic Framework for Automated Video Game Level Creation

Nathan Sorenson and Philippe Pasquier

School of Interactive Arts and Technology,
Simon Fraser University Surrey, 250 -13450 102 Avenue, Surrey, BC
{nds6, pasquier}@sfu.ca

Abstract. This paper presents a generative system for the automatic creation of video game levels. Our approach is novel in that it allows high-level design goals to be expressed in a top-down manner, while existing bottom-up techniques do not. We use the FI-2Pop genetic algorithm as a natural way to express both constraints and optimization goals for potential level designs. We develop a genetic encoding technique specific to level design, which proves to be extremely flexible. Example levels are generated for two different genres of game, demonstrating the system's broad applicability.

Key words: video games, level design, procedural content, genetic algorithms

1 Introduction

Procedural content creation, which is the algorithmic generation of game assets normally created by artists, is becoming increasingly important in the games industry [1]. Not only does this automation provide a way to produce much more content than would be possible in the typical game development process, but it could also allow for game environments to be adapted to individual players, providing a more personalized and entertaining experience.

Current level generation techniques [2, 3] tend to be bottom-up, rule-based systems which iterate over a set of production rules to construct an environment. These rules must be carefully crafted to create playable levels, and design goals are restricted to the emergent behaviour of the rule set. Furthermore, these techniques are extremely idiosyncratic; no standard toolbox or library exists for game content generation, and every game requires the construction of a specialized algorithm. These factors result in level generation systems that can take more effort to construct than the levels themselves [1].

We describe an approach that avoids these issues by allowing designers to specify the desired properties of the generated level, instead of requiring them to specify the details of how levels are assembled. This is done using a Feasible-Infeasible Two-Population (FI-2Pop) genetic algorithm [4], which is an evolutionary technique that proves to be well suited to the present domain. Level designers specify a set of constraints, which determine the basic requirements for a level to be considered playable. The "infeasible population" consists solely of levels which do not yet satisfy all these constraints, and these individuals are evolved towards minimizing the number of constraints violated. When individuals are found to satisfy all the constraints, they are

moved to the "feasible population" where they are subjected to a fitness function that rewards levels based on any criteria specified by the level designers. Essentially, this population is for generating levels that are not only playable, but also fun.

A simple, yet expressive genetic encoding for levels is presented, based on the specification of simple *design elements*. These design elements are the building blocks used by the GA to construct the level. We argue for the flexibility of this encoding by providing encodings for two different types of games: one is the 2D platformer *Super Mario Bros* [5], and the other is an exploration-adventure game similar to *The Legend of Zelda* [6]. With these encodings, we are able to evolve levels that satisfy a number of game-specific constraints and, furthermore, are fun. To measure how enjoyable a level is, we use a generic model of challenge-based fun described in previous work [7], and we apply this model to the generated levels of both types of games. Early results indicate that our approach is indeed useful and likely to be widely applicable.

2 Previous Work

Generative systems are often used to as a means to theoretically analyze the nature of games, as opposed to simply providing a way to create more game content. Togelius and Schmidhuber use GAs to generate novel game designs [8] that are subsequently evaluated with a neural net to test how fun they are, relying on the hypothesis that machine-learnable games are more entertaining. Similarly, Yannakakis and Hallam [9] explore the relationship between difficulty and fun by generating computer opponents of varying degrees of skill. Smith et. al. [10] share our goal of automatically generating game levels for 2D platformers; however, they use a generative grammar, which is a bottom-up, rules-based approach to constructing game levels which is tied to this specific genre. Pedersen et al. [11] generate levels for 2D platformers with the goal of modeling player experience. Their model suggests how player behaviour and player fun are related, however it does not provide any details on how specific level configurations (at the local level) influence player enjoyment and it is unclear how this model would effectively inform an automated level generation process. As well, their generative technique is restricted to the genre of 2D platformers.

In general, these applications of generative systems employ specific algorithms that are tailored to address precisely defined research questions in a single domain, whereas our goal is to present a widely applicable framework, usable in a variety of contexts.

3 Implementation

3.1 Feasible-Infeasible Two-Population Genetic Algorithm

Optimization is clearly an aspect of level design; game levels are constructed in order to maximize the amount of fun a player experiences. However, levels generally consist of a spatial arrangement of rooms, platforms, doors, and other components, and if these elements do not align properly, the entire level can become unplayable. For these reasons, the process of game design must be seen as both and optimization problem and a constraint satisfaction problem. GAs are effective in solving high-dimensional optimization problems, but are, in general, ineffective when tasked with solving constraint

satisfaction problems [12]. Gradual, incremental improvement becomes impossible if there are too many constraints on the feasibility of the levels, and this causes the evolutionary algorithm's performance to suffer. Conversely, the nuanced and complex notion of fun does not lend itself readily to the simple finite-domain predicates required by most constraint solvers.

Kimbrough et al. [4] present the Feasible-Infeasible Two-Population genetic algorithm (FI-2Pop) as an effective way to address these problems. FI-2Pop maintains two separate populations, one containing only feasible solutions and the other containing only infeasible solutions. Any solution that satisfies each specified constraint is moved into the feasible population, and any solution that violates a constraint as a result of mutation or crossover is moved into the infeasible population. FI-2Pop therefore requires two different fitness functions: the first is a typical optimization function that drives incremental changes towards a global optimum; the second fitness function is specifically for generating individuals that satisfy a set of constraints. Kimbrough et al. suggest that even a simple operation, such as counting the number of constraints violated, serves as an effective way to guide the population towards feasibility. We essentially follow this approach by summing up a set of individual constraint functions, which are each responsible for measuring a specific kind of constraint violation.

The two populations exert evolutionary pressure on one another through frequent migration, and because infeasible individuals are not simply killed off, a degree of genetic diversity can be maintained. Because level design criteria generally consist of a number of hard constraints and a number of soft, high-level goals, we find the FI-2Pop GA to be well suited to our domain. Though a multitude of evolutionary techniques have been developed to address similar concerns [13], FI-2Pop is a technically straightforward and conceptually simple approach to handling heavily constrained optimization problems. The choice of this particular algorithm is not the central contribution of this work; it is important to emphasize that, with our generic genotype encoding, we will be able to employ a more sophisticated and efficient algorithm should it prove necessary.

3.2 Genetic Representation

Because we desire this system to maintain as much generality as possible, care must be taken in the design of the genotype. Our genotype is therefore based on the notion of a *design element* (DE). DEs are small, composable building blocks that represent logical units of levels. For example, the DE for a game like *Breakout* [14] would be an individual block. DEs can be parameterized; the *Breakout* block DE could have parameters for both x and y coordinates, as well as for its physical properties. DEs do not have to be atomic, and can be specified at any level of abstraction. For example, a DE might be defined to create a star pattern of blocks, which would be parameterized by the size of the star. Essentially, DEs constitute a one-to-one genotype-to-phenotype mapping, as they literally describe a physical level element. They are therefore simple to define, and their behaviour is easy to predict.

The genotype representation consists of a variable-size set of these design elements. To allow for the use of typical variable-point crossover, we must impose an order onto the set to treat it as a linear genotype. The DEs can be sorted by an arbitrary function of their parameters, but crossover is most effective when genes exhibit strong genetic

linkage [15]. It is therefore best to sort the genotype in such a way that will tend to keep mutually influential genes together, maximizing the probability that beneficial arrangements of genes are not disrupted by crossover. Because level design is largely spatial in nature, we sort based on the coordinate parameter of the design elements, with the expectation that mutually influential genes will predominately represent level elements that are in close proximity. The sorting decision is especially straight-forward in the case of games such as 2D platformers; because all level elements are principally arranged along the horizontal axis, we sort the genotype based on the x coordinate. However, the second example we provide demonstrates that our approach is not restricted to linear games and indeed works in two-dimensional situations as well.

Our mutation operator adjusts an arbitrary parameter of a random design element gene in a genotype. Continuous values, such as height or width, are displaced using a normal distribution with variance derived from their permissible range, and categorical values are simply given a new permissible value.

3.3 Fitness Function

To generate enjoyable levels, we employ a fitness function that is able to identify how fun a given level is. Certainly, the notion of fun is exceedingly complex and difficult to define precisely. However, as a starting point, we can identify aspects of fun that are more tractable for computational analysis. For example, a large number of theorists [16–19] identify the presence of an appropriate level of challenge as integral to nature of fun in games. This is particularly true for a skills-based action game such as *Super Mario Bros.* Broadly speaking, a player has the most fun when presented with challenges that are neither too easy nor too difficult. We currently use a generic model of player enjoyment that is not restricted to a particular game, and is discussed in more detail in [7]. The model does not require any genre-specific information, instead it relies only a simple challenge measurement, $c(t)$ to determine the perceived difficulty a player experiences at time t, and rewards levels for matching a desired challenge configuration. In other words, this model is used to characterize the amount of fun, f, that is acquired by a player throughout the course of a level. The model is summarized in Equation (1).

$$\frac{df}{dt} = m * c(t) \tag{1}$$

The variable m can take the value of $+1$ or -1, and represents two important states of the model at a given time. When $m = 1$, the amount of fun measured increases with the challenge measurement. However, when $m = -1$, challenge serves to reduce the amount of fun. We specify threshold values that determine when the value m changes. When the amount of challenge in a given time period has exceeded the upper threshold, m becomes negative. Conversely, if not enough challenge has been measured , as determined by the lower threshold, m becomes positive. This model, in practice, tends to reward level designs that interpose periods of high difficulty with segments of low difficulty, and even though the challenge metric and model of fun are rough approximations to reality, they have been constructed in a principled manner and appear to

produce acceptable results. Devising and characterizing such fitness functions is certainly a difficult question and will continue to be a topic for future study. We must emphasize, however, that our framework does not necessarily depend on any particular characterization of level quality. Indeed, any fitness function can be created to express the subjective design goals of the game developer.

4 Validation Results

4.1 Super Mario Bros.

Our first example of this genetic encoding is based on the original *Super Mario Bros.* (SMB) [5]. SMB is a 2D platformer game, in which levels consist of an arrangement of platforms and enemies. Inspecting existing levels from the original game, we identify a number of design elements occur frequently, which are shown in Figure 1:

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

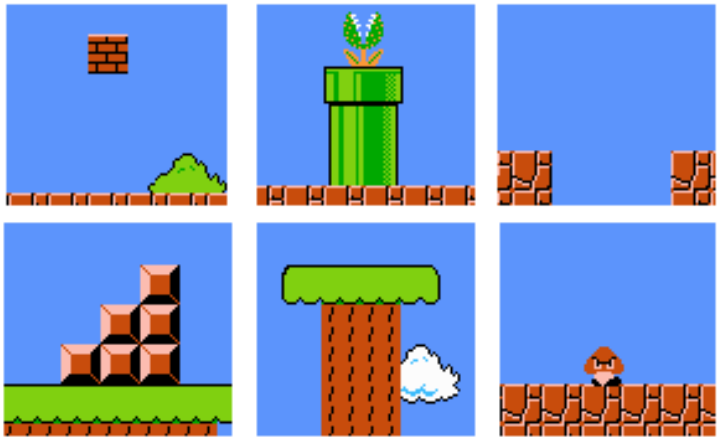

Fig. 1. The 6 DEs for SMB level design. Clockwise from top-left: block, pipe, hole, enemy, platform, and staircase DE.

In addition to the DEs, we provide the following constraint functions for the infeasible population:

1. $require\text{-}exactly(n, type)$. This function allows designers to specify the desired number of certain types of design elements to be present in individuals. As a penalty, it returns the absolute difference between the counted number of instances of $type$ and the desired amount n.
2. $require\text{-}at\text{-}least(n, type)$. This function penalizes levels that contain less than n of a given $type$, returning 0 if $n \geq type$ and returning $type - n$ otherwise.
3. $require\text{-}at\text{-}most(n, type)$. This function penalizes levels that contain more than n of a given $type$, returning 0 if $n \leq type$ and returning $n - type$ otherwise.
4. $require\text{-}no\text{-}overlap(type_1, type_2, ...)$. 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 example, we specify that pipes, stairs, enemies, and holes should not overlap one another. As a penalty, the number of overlapping elements is returned.
5. $require\text{-}overlap(type_1, type_2)$. Though similar to function 4, 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.
6. $traversible()$. 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 simple 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 a player could get stuck.

All the previous functions are specified such that a value of 0 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 0 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 2D 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 [4], 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 twenty generations. Figure 2 depicts some resulting levels.

A significant advantage of the evolutionary approach is the adaptability of the solution. For example, it is possible for an artist to hand-craft certain portions of a level, and have the GA automatically fill in the gaps according to the specified constraints. Consider the manually-created arrangement and the resulting evolved level in Figure 3. No extra functionality was needed to provide this behaviour; the genotype was simply hard-coded to always include the user-specified arrangement of DEs.

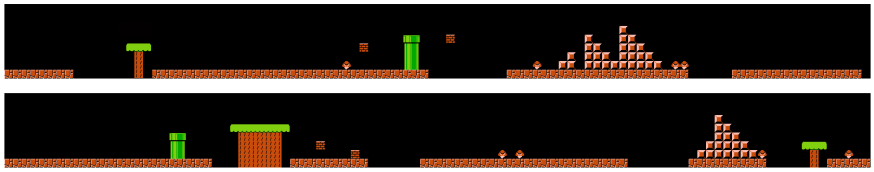

Fig. 2. Two different levels, created in 892 and 3119 generations, respectively. The number of staircases, platforms, and enemies are specified through constraints. On a mid-range dual-core laptop, the running time was for each was less than 30 minutes.

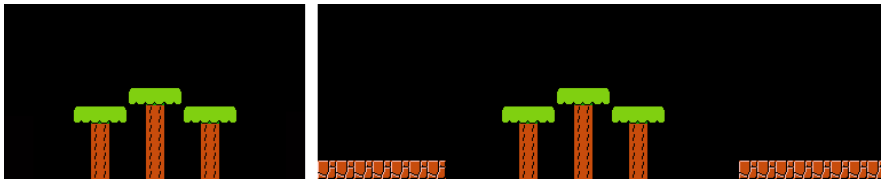

Fig. 3. An explicitly specified arrangement of platform DEs (left) is automatically wrapped in a surrounding hole (right).

4.2 2D Adventure Game

A major disadvantage of typical generative systems is that they are restricted to a single application. Any improvements to the generative technique will benefit only that particular application. We claim that our evolutionary framework provides the ability to factor out some of the generative logic from any game-specific context. For this to be the case, it must be relatively simple to express a variety of different game design goals without requiring fundamental changes to the underlying system. As a proof-of-concept example in support of this claim, we present a set of constraints for the evolution of levels for a simple top-down 2D adventure game similar to *The Legend of Zelda* [6].

The levels for this game will be constructed from three design elements:

1. $Hallway(x, y, length, direction)$. This codes for a hallway of a given length, whose direction can either be vertical or horizontal.
2. $Room(x, y, width, breadth)$. This creates a rectangular room of the specified size.
3. $Monster(x, y)$. This creates a monster at a given coordinate.

The genotype and the mutation and crossover operators are the same as in the previous example. Even though the coordinates of the design elements must now be expressed as (x, y) pairs instead of as a single x coordinate, we find that sorting by x to linearize the genotype produces acceptable results.

We specify two constraints for this simple game:

1. $connected(start, end)$. Returns 0 if there is a 4-connected path between the start and end points. Otherwise, penalizes levels for the minimum distance between the two areas reachable from the start and end points. In other words, levels that are far from being connected are penalized more than areas that are nearly connected.

2. $require\text{-}overlap(type_1, type_2)$. We use this constraint, introduced in the 2D platformer experiment, to ensure monsters are located in rooms or hallways. The number of monsters that do not overlap hallways or rooms is returned.

To evaluate the challenge of a level, we simulate a player's traversal of the level from the start to the end point, using a standard A* search algorithm. Challenge is determined to be the number of monsters within a given radius at a given point along this traversal path. With this measurement in place, we are able to employ the same fitness function as specified in Section 3.3. Several runs of the algorithm are presented in Figure 4.

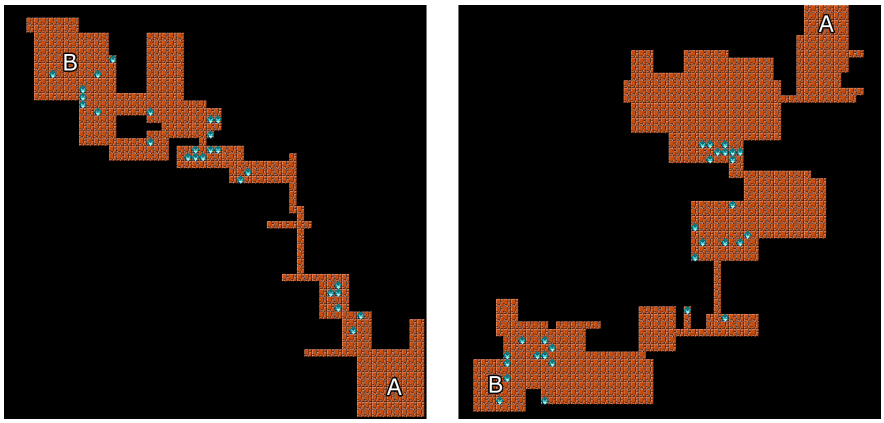

Fig. 4. A small and a large level, created in 1629 and 2330 generations, respectively. The starting point is labeled A and the goal is labeled B. On a mid-range dual-core laptop, the running time was for each was less than an hour.

With very little extra effort, one can see levels generated for an entirely different type of video game. The generic fitness function has resulted in creating rooms containing clusters of enemies, interspersed with areas containing none, in a manner that could be expected from a human-designed level. A bottom-up, rules-based approach would have necessitated an entirely new set of production rules, but our approach has allowed us to re-use the same genetic algorithm, fitness function, and even some constraints.

5 Discussion

The apparent simplicity of the two examples provided should not obscure the fact that this approach represents a promising alternative to current generative techniques. Firstly, the optimization fitness function allows the intended player experience to be represented explicitly. In other words, instead of specifying how levels are assembled, game designers may simply indicate the particular properties that levels should have. In this way, levels are described declaratively rather than procedurally; instead of treating player experience as an incidental side-effect of the level creation process, the fitness function provides an effective means of handling it directly.

Furthermore, this approach does not exclude the possibility of other, complementary design techniques. Where rule-based generative systems tend to operate in isolation, GAs can work well when used in conjunction with other techniques. As we have shown, game designers are able to hand-craft particular portions of a level without being required to make any changes to the system. This same idea could be used to interface with other generative systems, either to glue together elements generated elsewhere, or to directly manipulate and optimize the systems themselves.

Another advantage is that this approach is quite modular; constraints, optimizations, and design elements can be added or removed individually. This makes it easier to adjust and test the behaviour of the genetic search.

In a broad sense, this approach factors out the generative algorithm from a particular game artefact. The drive to abstract and generalize implementation details is essential to modern software development, and this practice is used heavily in game development. Features such as 3d animation, physics, and artificial intelligence tend to be handled by third party game engines and are no longer developed from scratch. In the same way that game designers are now able to declaratively specify the specific graphical or physical properties of a given aspect of a game, expecting these properties to be properly handled by the underlying engine, we argue that our approach models a way in which this could be done for level generation. Simple constraints, fun optimization functions, and design elements can be defined for a particular game with no real concern for exactly how these constraints are to be satisfied. Since it is more than likely that games will have many constraints in common (for example, connectivity is a concern in many different types of games), it is possible that these units can be shared among games.

6 Future Work and Conclusion

There are many promising avenues for future work in the area of automated game level generation. Simply continuing to devise constraints and optimization functions for various types of games would likely prove fruitful in evaluating the general applicability of this approach. For example, our current operational definition of fun only accounts for challenge dynamics, which is certainly only one component of fun. A more comprehensive model of fun would need to be employed to account for the presence of game elements not relating directly to challenge, such as collectible rewards. Also, though this paper focuses on proof-of-concept examples rather than on efficiency, a more rigorous comparison of the performance characteristics of the FI-2Pop GA to other possible evolutionary techniques would certainly be worthwhile.

It is also our hope that our method will serve as a useful environment in which to experiment with theoretical conceptualizations of game design. We believe that the ability to explicitly realize models of enjoyment in games will contribute to furthering knowledge in that field. In the same way that simple computational models can serve to elucidate the dynamics of otherwise complex natural phenomena [20], it is possible that models of fun will serve to illustrate fundamental principals of game design. Preliminary work of this nature is introduced in [7], where a model of challenge-based fun in games is explored in more detail. Our generic level generation framework would certainly contribute to this ongoing research.

Even though this work is presently in an exploratory stage, it already exhibits encouraging results and can be viewed as a prototype for a practical tool to assist level designers. Much work is yet to be done, and we anticipate that our general top-down approach to level generation will offer much to the practice and theory of game design.

References

1. Remo, C.: MIGS: Far Cry 2's Guay on the importance of procedural content. Gamasutra (11 2008) http://www.gamasutra.com/php-bin/news_index.php?story=21165.
2. Meier, S.: Civilization. MicroProse (1991)
3. The NetHack DevTeam: Nethack. (2009) http://www.nethack.org/.
4. Kimbrough, S.O., Lu, M., Wood, D.H., Wu, D.J.: Exploring a two-market genetic algorithm. In: GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2002) 415–422
5. Miyamoto, S., Yamauchi, H., Tezuka, T.: Super Mario Bros. Nintendo (1987)
6. Miyamoto, S., Nakago, T., Tezuka, T.: The Legend of Zelda. Nintendo (1986)
7. Sorenson, N., Pasquier, P.: The evolution of fun: Towards a challenge-based model of pleasure in video games. In: ICCC-X: First International Conference on Computational Creativity, Lisbon, Portugal (2010) 258–267
8. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: IEEE Symposium on Computational Intelligence and Games. (2008) 111–118
9. Yannakakis, G., Hallam, J.: Towards capturing and enhancing entertainment in computer games. Advances in Artificial Intelligence (2006) 432–442
10. Smith, G., Treanor, M., Whitehead, J., Mateas, M.: Rhythm-based level generation for 2d platformers. In: FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games, New York, NY, USA, ACM (2009) 175–182
11. Pedersen, C., Togelius, J., Yannakakis, G.: Modeling player experience in Super Mario Bros. In: IEEE Symposium on Computational Intelligence and Games. (September 2009)
12. Hemert, J.I.: Comparing classical methods for solving binary constraint satisfaction problems with state of the art evolutionary computation. In: Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002, London, UK, Springer-Verlag (2002) 82–91
13. Coello Coello, C.A.: Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Computer Methods in Applied Mechanics and Engineering **191**(11-12) (January 2002) 1245–1287
14. Bushnell, N., Bristow, S., Wozniak, S.: Breakout. Atari (1976)
15. Harik, G.R.: Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms. PhD thesis, Ann Arbor, MI, USA (1997)
16. Sweetser, P., Wyeth, P.: Gameflow: a model for evaluating player enjoyment in games. Comput. Entertain. **3**(3) (2005) 3
17. Salen, K., Zimmerman, E.: Rules of Play : Game Design Fundamentals. The MIT Press (October 2003)
18. Koster, R.: Theory of Fun for Game Design. Paraglyph Press (November 2004)
19. Juul, J.: Fear of failing? the many meanings of difficulty in video games. In Yao, X., Burke, E., Lozano, J.A., Smith, J., Merelo-Guervs, J.J., Bullinaria, J.A., Rowe, J., Tino, P., Kabn, A., Schwefel, H.P., eds.: The Video Game Theory Reader 2. Routledge, New York (2009) 237–252
20. Humphreys, P.: Mathematical modeling in the social sciences. In Turner, S.P., Roth, P.A., eds.: The Blackwell guide to the philosophy of the social sciences. Wiley-Blackwell, New Jersey, USA (2003) 166–184