The 2010 Mario AI Championship: Level Generation Track

Noor Shaker, Julian Togelius, Georgios N. Yannakakis, *Member, IEEE*, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, *Member, IEEE*, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, Gillian Smith, *Student Member, IEEE*, and Robin Baumgarten

***Abstract*—The Level Generation Competition, part of the IEEE Computational Intelligence Society (CIS)-sponsored 2010 Mario AI Championship, was to our knowledge the world's first procedural content generation competition. Competitors participated by submitting level generators—software that generates new levels for a version of *Super Mario Bros* tailored to individual players' playing style. This paper presents the rules of the competition, the software used, the scoring procedure, the submitted level generators, and the results of the competition. We also discuss what can be learned from this competition, both about organizing procedural content generation competitions and about automatically generating levels for platform games. The paper is coauthored by the organizers of the competition (the first three authors) and the competitors.**

***Index Terms*—Computational and artificial intelligence, computational intelligence, computer science education, evolutionary computation, hybrid intelligent systems, neural networks education.**

I. Introduction

In the last few years, a number of game AI competitions have been run in association with major international conferences, several of them sponsored by the IEEE Computational Intelligence Society (CIS). These competitions are based either on classical board games (such as *Othello* and *Go*) or video games (such as *Pac-Man*, *Super Mario Bros*, and *Unreal Tournament*). In most of these competitions, competitors submit controllers that interface to the game through an application programming interface (API) built by the organizers of the competition. The competition is won by the person or team that submitted the controller that played the game best, either on its own (for single-player games such as *Pac-Man*) or against others (in adversarial games such as *Go*). One interesting variation on this formula is the 2k BotPrize, where the submitted entries are not supposed to play the game as well as possible, but in an as human-like manner as possible [1]. Several of these competitions have spurred valuable research contributions as reported in [2] and [3] (among others).

However, nonplayer character (NPC) behavior is not the only use for computational intelligence (CI) and artificial intelligence (AI) in games. In fact, according to some game developers [4], it might not even be the area where new advances in AI are needed the most. Another very interesting area, in which there is growing interest both from the CI and AI research communities and from game developers, is procedural content generation (PCG).

PCG refers to any method which creates game content algorithmically, with or without the involvement of a human designer. There are several reasons one might want to create game content automatically: saving development costs, saving storage or main memory (e.g., in creating "infinite" games), or adapting the game to players and augmenting human creativity. The field has a fairly long history [see, for example, the early 1980s games *Rogue* (AI Design 1983) and *Elite* (Acornsoft 1984)], but only recently have approaches from artificial and computational intelligence begun to be explored in the context of creating central game elements such as levels, maps, items, and rules. In particular, "search-based" approaches to PCG, building on evolutionary algorithms or other stochastic search/optimization algorithms, have recently been the subject of some interest in the computational intelligence and games community [5]–[7]; recent overviews of such techniques can be found in [8] and [9], along with a taxonomy of PCG in general. The coupling of player experience and PCG under a common framework named "experience-driven PGC" is introduced in [10].

A key concern for many commercial game developers is the spiraling cost of creating high-quality content (levels, maps, tracks, missions, characters, weapons, vehicles, artwork, etc.) for games. As the graphics and other technical capabilities of game hardware have increased exponentially, so have the demands on game content. However, the most common use of PCG in commercial games today is offline creation of trees and

Manuscript received October 25, 2010; revised March 10, 2011 and June 02, 2011; accepted August 14, 2011. Date of publication August 30, 2011; date of current version December 14, 2011. This work was supported in part by the European Union FP7 ICT project SIREN (Project 258453) and by the Danish Research Agency project AGameComIn (Project 274-09-0083).

N. Shaker, J. Togelius, and G. N. Yannakakis are with the Center for Computer Games Research, IT University of Copenhagen, Copenhagen 2300, Denmark (e-mail: nosh@itu.dk; juto@itu.dk; yannakakis@itu.dk).

B. Weber, P. Mawhorter, G. Takahashi, and G. Smith are with the Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: bwebersoe.ucsc.edu; pmawhort@soe.ucsc.edu; glen.takahashi@gmail.com; gsmith@soe.ucsc.edu).

T. Shimizu was with the University of Electro-Communications, Tokyo 182-8585, Japan. He is now with Fuji Xerox Co., Ltd., Tokyo 107-0052, Japan (e-mail: tomoyuki@media.is.uec.ac.jp).

T. Hashiyama is with the University of Electro-Communications, Tokyo 182-8585, Japan (e-mail: hashiyama@is.uec.ac.jp).

N. Sorenson and P. Pasquier are with Simon Frasier University, Burnaby, BC V5A 1S6 Canada (e-mail: nds6@sfu.ca; pasquier@sfu.ca).

R. Baumgarten is with Imperial College, London SW7 2AZ, U.K. (e-mail: robin.baumgarten06@doc.ic.ac.uk).

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.2011.2166267

vegetation.[1] Even though there are a few examples of level generation in commercial games, e.g., *Rogue* and games inspired by it such as *Diablo* (Blizzard 1996), PCG algorithms are still rarely used for the creation of central game elements, or for online creation of game content during gameplay. This is because available PCG techniques are not seen, by many game developers, as efficiently and reliably producing content of sufficient quality to be used in such roles. Therefore, given the need for making content creation faster and more reliable, the development of better PCG techniques is an important research direction for industrially relevant game AI research and beyond. As there are many different types of game content that could potentially be generated (levels, maps, weapons, rules, stories, etc.), and several different roles that could be imagined for PCG within a game, different content generation problems are expected to require different approaches [11].

Apart from being fast, reliable, and producing high-quality content, another desirable characteristic of PCG algorithms in many contexts is that they are *controllable*. A controllable PCG algorithm can take parameters that describe desired features of the generated content, and produce content that complies to these specifications. Such features can be defined on different levels of abstraction, from the geometric aspects (e.g., the length of a race track or the ruggedness of a landscape) to gameplay aspects (e.g., how hard a level would be to clear for a particular player). This is useful when content is produced collaboratively by human designers and algorithms, so that the human designer can request content with particular features suitable for further human editing or content that fits into already human-authored content [12]–[14]. It is also important when using PCG to automatically adapt a game to the human player (e.g., producing more challenging levels for better players or more fun levels for particular player types) [5], [10], [15], [16]. Such personalization becomes increasingly important as the game-playing population gets more diverse [17], [18].

With the importance of research on effective and controllable PCG in mind, we created the level generation track within the Mario AI Championship to spur and benchmark development of PCG algorithms. To the best of our knowledge, this is the first PCG competition within an academic research community, and the first competition about adaptive or controllable PCG.

Competitors participated in the competition by submitting controllable content generation algorithms, which would create game content intended to maximize enjoyment for individual players. In order to ensure the relevance of the competition, we set ourselves the goal of addressing an important content generation problem with considerable generality, within a complex and well-known game context. We then evaluate the generated content in a fair and accurate manner. This goal was addressed by using *Infinite Mario Bros* (Persson 2008), an all-Java clone of the classic platform game *Super Mario Bros* (Nintendo 1985). For that game the content type is specified to be complete levels which yields a particularly complex content generation task with room for diverse strategies. The submitted level generators were evaluated by letting human players play levels generated to suit their particular playing style, and ranking them in order of enjoyment.

Our hope is that this competition will spur research in methods of creating levels for platform games, and also in modeling players of such games and adapting levels to individual players. The competition is also expected to advance the study on computational gameplay aesthetics, playing experience modeling, and experience-driven PGC [10]. Many concerns relevant to designing platform game levels recur in the design of levels and maps for other games, for example, rhythm and variation may be as important in, e.g., first-person shooter (FPS) levels and role-playing game (RPG) dungeons as in platform games, and it is likely that principles for generating levels that include these features carry over to other game genres. Appropriate challenge balancing is an important concern in the design of almost all game content.

The paper is structured as follows. First, a brief introduction is given to Infinite Mario Bros and the Mario AI Championship, a series of AI competitions built around this game. This is followed by a description of the level generation track (part of the Championship), including the Java interface between the game and the generators, the rules of the competition, and the scoring procedure. The section after this describes the level generators that were submitted to the competition. To ensure that the descriptions of the generators are both accurate and allow for meaningful comparison, the subsection about each level generator is written by the authors of the corresponding generator. However, all authors were asked to answer a specific set of questions about their level generator within their text. After the presentation of the submitted generators, the results of the competition are presented. Moreover, a concluding section discusses what we can learn from this competition, both in terms of generating levels for platform games and in terms of organizing a PCG competition.

II. Infinite Mario Bros

Infinite Mario Bros (Markus Persson 2008) is a public domain clone of Nintendo's classic platform game *Super Mario Bros* (1985). The original *Infinite Mario Bros* is playable on the web, where Java source code is also available.[2]

The gameplay in *Super Mario Bros* consists in moving the player-controlled character, Mario, through 2-D levels, which are viewed sideways. Mario can walk and run to the right and left, jump, and (depending on which state he is in) shoot fireballs. Gravity acts on Mario, making it necessary to jump over holes to get past them. Mario can be in one of three states: *Small, Big* (can crush some objects by jumping into them from below), and *Fire* (can shoot fireballs).

The main goal of each level is to get to the end of the level, which means traversing it from left to right. Auxiliary goals include collecting as many as possible of the coins that are scattered around the level, finishing the level as fast as possible, and collecting the highest score, which in part depends on the number of collected coins and killed enemies.

[1]See http://www.speedtree.com

[2]http://www.mojang.com/notch/mario/

Complicating matters is the presence of holes and moving enemies. If Mario falls down a hole, he loses a life. If he touches an enemy, he gets hurt; this means losing a life if he is currently in the Small state. Otherwise, his state degrades from Fire to Big or from Big to Small. However, if he jumps and lands on an enemy, different things could happen. Most enemies (e.g., goombas, cannon balls) die from this treatment; others (e.g., piranha plants) are not vulnerable to this and proceed to hurt Mario; finally, turtles withdraw into their shells if jumped on, and these shells can then be picked up by Mario and thrown at other enemies to kill them.

Certain items are scattered around the levels, either out in the open, or hidden inside blocks of brick and only appearing when Mario jumps at these blocks from below so that he smashes his head into them. Available items include coins, mushrooms which make Mario grow Big, and flowers which make Mario turn into the Fire state if he is already Big.

No textual description can fully convey the gameplay of a particular game. Only some of the main rules and elements of *Super Mario Bros* are explained above; the original game is one of the world's best selling games, and still very playable more than two decades after its release in the mid-1980s. Its game design has been enormously influential and inspired countless other games.

The original *Super Mario Bros* game does not introduce any new game mechanics after the first level, and only a few new level elements (enemies and other obstacles). There is also very little in the way of story. Instead, the player's interest is kept through rearranging the same well-known elements throughout several dozens of levels, which nevertheless differ significantly in character and difficulty. This testifies to the great importance of level design in this game (and many others in the same genre), and to the richness of the standard *Super Mario Bros* vocabulary for level design.

III. The Mario AI Championship

The Mario AI Championship was set up as a series of linked competitions based on *Infinite Mario Bros*. In 2009, the first iteration of the Championship (then called the Mario AI Competition) was run as a competition focusing on AI for playing *Infinite Mario Bros* as well as possible. A writeup of the organization and results of this competition can be found in [3].

The 2010 Mario AI Championship was a direct successor of this competition, but with a wider scope. It consisted of three competition tracks (the Gameplay Track, the Learning Track, and the Level Generation Track) that were run in association with three international conferences (EvoStar, IEEE Congress on Evolutionary Computation, and IEEE Conference on Computational Intelligence and Games). While the championship was open to participants from all over the world, the cash prizes (sponsored by the IEEE CIS) could only be awarded to competitors that were physically present at the relevant competition event.

IV. The Level Generation Track

While the Gameplay and Learning tracks, which will be discussed at length in a separate paper, focused on controllers that could play *Infinite Mario Bros* as well as possible, the Level Generation track focused on software that could design levels for human players. For this track, special software was designed that allowed the game to connect with the submitted level generators, and that partly automated the scoring procedure. The competition also required inventing a scoring system, as well as laying down general rules for what was and was not allowed.

A. Rules

The competition was open to individuals or teams from all over the world without any limitations, e.g., in terms of academic affiliation. (In practice, all competing teams in the Level Generation Track included at least one graduate student, but this is incidental; the other tracks of the championship had several entrants without academic affiliation.) While the highest scoring competitor would be the overall winner of the competition and receive the certificate, in case no representative of the winning team was present at the competition event, the IEEE CIS-sponsored prize money would be awarded to the highest scoring competitor who was actually present. The competition event was held August 19, 2010 in Copenhagen (during the IEEE Conference on Computational Intelligence and Games), and final entries had to be submitted by a deadline a week before that date. The final submissions were expected to already fulfill the technical requirements, but technical assistance was available from the organizers up until the deadline.

The main technical requirement was that the software should be able to interface to an unmodified version of the Java framework built by the organizers around the *Infinite Mario Bros* game. It was not a requirement that the submissions be written in Java, though no particular assistance was given for non-Java development. Another key requirement was that the call to the level generation routine should return within one minute on a standard MacBook from 2009—in other words, that a level should always be generated in under a minute.

In what was probably the most controversial rule, which was later relaxed, the organizers decided to impose certain arbitrary and unpredictable requirements on the generated levels. The interface was extended so that in addition to data about how the human judge played the test level, the required number of coin blocks, turtles, and gaps in the ground was passed to the level generator (the final numbers were not revealed to the competitors until the competition event). Originally, it was intended that any level generator which generated levels with numbers of gaps, turtles, and coin blocks that differed from those specified would be disqualified. The motivation for this rule was to prevent competitors from bypassing the purpose of the competition by entering "level generators" that only generated a single, human-designed (and presumably well-designed) level at each method call, or one that simply generated minor variations on a single level. However, some competitors complained that the rule overly restricted the level generators, and after some deliberation the organizers decided to not disqualify any level generator that was deemed to generate sufficiently dissimilar levels each time.

All important information regarding the Mario AI Championship, including rules, and software was posted on a dedicated

website.[3] Prospective participants and other interested parties were encouraged to join a Google Group devoted to the competition.[4] All technical questions were supposed to be posted and answered publicly within the group, so that the archive of the group could function as a searchable repository of technical knowledge regarding the championship.

B. Scoring Procedure

The rationale behind the scoring was that the level generator which generated levels that were preferred by most players should win. As mentioned earlier, the primary aim of the competition was the generation of personalized *Super Mario Bros* levels for particular players. For this purpose, we used human judges as Mario players to assess the quality of each submitted competitor; everyone who was present at the competition event was encouraged to participate in the judging. Each human judge was given a test level to play, and his or her performance on that level was recorded and passed on to the level generators. The judge then played two generated levels from two competing generators, and ranked them according to how much fun they were to play.

A two-alternative forced-choice questionnaire was used according to which each judge expressed a pairwise preference of fun after completing the two levels (i.e., "which game of the two was more fun to play"). (The concept of "fun" was deliberately not defined further, so as not to bias judges more than what is unavoidable.) The adoption of this experimental procedure was inspired by earlier attempts to capture player experience via pairwise preference self-reports which were introduced by the competition organizers (see [19]–[21] among others). For all competition entries to be treated fairly, all generators had to be played an equal number of times by the judges and compared against all other generators submitted. On that basis, the required minimum number of judges was 15 given that there were six competitors (i.e., all possible combinations of two games out of six competitors). To control for order of play effects, each pair was played by the same judge in both orders.

To make sure that each pair of competitors were judged at least once in both orders we set up an online structured query language (SQL) database that initially contained all possible pairs marked as "unplayed." Whenever a game session started, the software connected to the database and asked for an unplayed pair to load. Once the two level generators in the pair had been chosen from the database, the levels were generated according to the judge's gameplay behavioral statistics and the judge was set to play the generated two levels in both orders. The level generators had access to player metrics such as numbers of player jumps and coins collected (see Section IV-C for more details about those data).

When the two games and the questionnaire were completed, the judge's preferences and gameplay statistics were stored to the database and the pair was marked as "played." The experiment was reset if there were no more pairs available in the database to play (all pairs were marked as "played").

[3]http://www.marioai.org

[4]http://groups.google.com/mariocompetition

C. Software and Interface

An interface was designed to pass information between the game and the level generator. In the main loop, the level generator was called by the competition software with information on the human player's playing style and expected to return a complete level, expressed as a 2-D array of level elements.

Gameplay metrics were collected and statistical features were extracted from these data. Features included number of jumps, time spent running, time spent moving left, number of opponents of each type killed, and many others; for a complete list of the data collected, see [16]. The selection of features was based on the organizers' understanding of what differentiates players in this particular game, and were all features that could be extracted with a minimum of processing from the game engine. These data about the player's behavior were available to each competitor at the end of each level.

The resulting software is a single threaded Java application that can run on any major hardware architecture and operating system, with the methods that the generators need to implement specified in Java interface files. Level generators had to implement the *LevelInterface* which specifies how the level is constructed and how different type of elements are scattered around the level:

```
public byte[][] getMap();
public SpriteTemplate[][] getSpriteTemplates()
```

The size of the level was constrained to be the same for all competitors: 320 $\times$ 15 level cells. Different levels can be generated by placing different types of elements in each cell of the level map. The type of elements that can be placed in each cell may vary from basic level elements like a block, a ground, a specific background, and a coin to different enemy types like a goomba, a turtle, a cannon, and a flower. The total number of elements that can be used is 29.

Generators implement the *LevelGenerator* interface—that is used to communicate with the simulator—and are bound to respond to the *GenerateLevel* method call with a new level:

```
public LevelInterface generateLevel
(GamePlay playerMetrics);
```

The *GamePlay* interface provides information about the player experience and might be useful to construct a personalized level. An example of five statistical features (as captured by the *GamePlay* interface) that contain information about level design parameters and gameplay characteristics is as follows:

```
//total number of enemies
public int totalEnemies;
//total number of empty blocks
public int totalEmptyBlocks;
//total number of coins
public int totalCoins;
//number of Green Turtle Mario killed
public int GreenTurtlesKilled;
//total time spent running to the left
public int timeRunningLeft;
//number of empty blocks destroyed
public int emptyBlocksDestroyed;
```

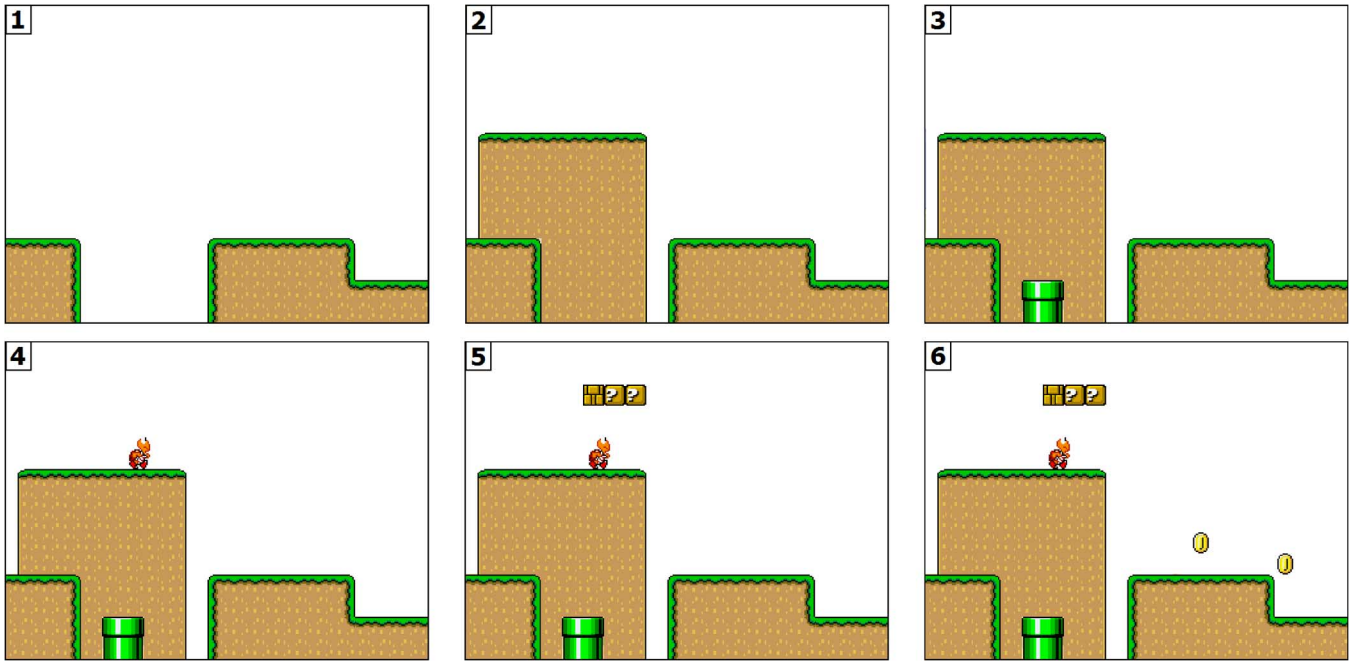

Fig. 1. Passes applied by Ben Weber's ProMP generator: (1) ground, (2) hills, (3) pipes, (4) enemies, (5) blocks, and (6) coins.

Keeping with the tradition from previous IEEE CIS-sponsored competitions, the competition software was open source and full source code was published on the competition web page.

V. The Competitors

In this section, the five level generators that took part in the competition are presented. Each section is written by the author(s) of the level generator. In order to facilitate comparison of the level generators, and make sure that information about key features was present, a certain structure was imposed on these descriptions. The competitors were asked to answer the following questions about their generator, if possible in the indicated order.

1) What is the main idea behind, and general architecture of, the level generator?
2) Were any CI/AI techniques used for offline training? If so, which?
3) Does the level generator adapt to the recorded playing style of the human player? If so, how?
4) How much of the generated levels are actually designed by a human designer? Conversely, what level of creative control would a human designer have when using the generator?
5) What are the main strengths and weaknesses of the level generator?
6) Could the underlying principles be generalized to work for other games, or other types of content?

A. Ben Weber: Probabilistic Multipass Generator

1) Idea and Architecture: The probabilistic multipass (ProMP) generator creates a base level and then iterates through it several times, each pass placing a new component type. The generation process consists of six passes, where each pass places a different component type by traversing the level from left to right. At each generation step, a set of events specific to the current pass can occur based on weighted probabilities. For example, during the initial pass events can occur that change the ground height, begin a gap, or end a gap. Events are selected using a uniform probability distribution. In total, the system includes 14 event types with author-specified weights. An overview of the level generation process is shown in Fig. 1.

The system enforces two types of constraints. Playability constraints are used to constrain the range of values that can be selected by the generator, such as limiting the maximum height of pipes to ensure that players can traverse the levels. Competition constraints are enforced by limiting the number of objects placed each pass. For example, if a generated level contains the maximum number of gaps, the probability for new gap placement is set to zero.

2) Offline Training: No offline training is performed.

3) Creative Control: The authorial control provided by the ProMP generator is limited to parameter selection. The author can manipulate weights of specific events in order to change the frequency of gaps, enemies, and hills. However, creating noticeably different levels requires modifying the algorithm.

4) Adaptation: The initial ProMP algorithm did not adapt based on the player log. Since the competition, the algorithm has been modified to adapt event probabilities based on the skill of a player. Level completion causes an increase in the enemy and gap placement probabilities, while deaths cause a decrease in these probabilities.

5) Strengths and Weaknesses: While the generator is capable of building levels in real time, it outputs levels of limited variation. One of the main disadvantages of the ProMP algorithm

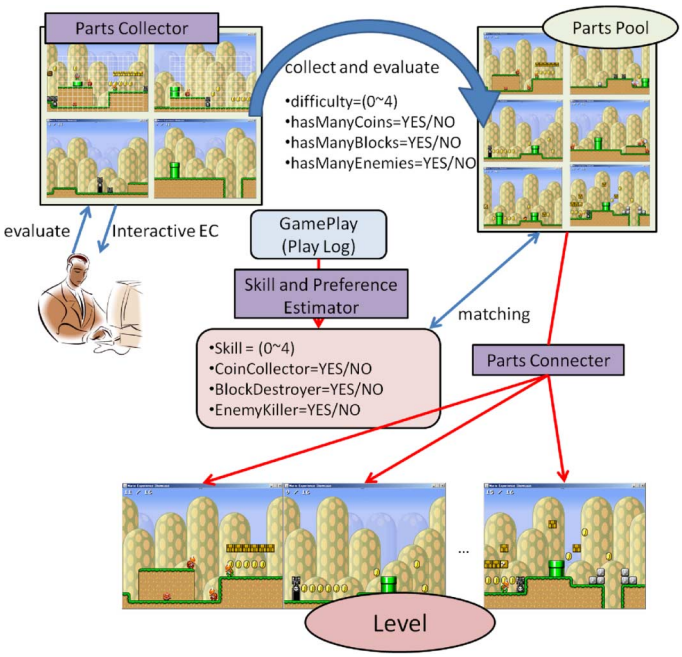

Fig. 2. This figure shows the overall architecture of the Tomoyuki Shimizu and Tomonori Hashiyama's level generator. The Parts collector runs offline using interactive evolutionary computation. The Skill and preference estimator derive players' characteristics. Based on the outputs of these modules, the parts connector arranges the corresponding parts sequentially.

is that scaling up the range of the generator is nonintuitive, because adding new event types or additional passes may break previously playable levels.

6) Generalizability: The ProMP algorithm was designed specifically for platformer level generation and has limited application outside this domain. However, the concept of creating a base level and then applying procedural decoration [22] may translate well to other genres.

B. Tomoyuki Shimizu and Tomonori Hashiyama

1) Idea and Architecture: The main idea behind our level generator is to make players experience *flow*, according to the theory of Csikszentmihalyi [23]. A key element of the theory of flow implies a linear relationship between challenge and skill as an important factor of enjoyment. To realize this relationship, we have implemented and combined three separate modules: 1) the *skill and preference estimator*, 2) the *parts collector*, and 3) the *parts connector* (see Fig. 2).

2) Offline Training: Players' skills and preferences are evaluated by the *skill and preference estimator* with GamePlay logs. Based on the player's log from a test level, this module carries out the inference using heuristic rules given by the designers *a priori*. The premises of these rules include parameters such as number of deaths, time spent running, numbers of enemies killed by stomping, time spent in each mode, and numbers of mode switches. Players' skills are classified into five degrees from 4 (excellent) to 0 (below average). Players' preferences are represented as three values, each corresponding to a distinct playing behavior: 1) CoinCollector, 2) BlockDestroyer, and 3) EnemyKiller. Each preference is represented by a real number between 0 and 100, which denotes the percentages of: 1) coins collected, 2) blocks destroyed, and 3) enemies killed by the player in the test level.

The *parts collector* is a tool for the designers to collect the appropriate parts corresponding to a set of sprites and environments through interactive evolutionary computation (IEC) [24]. This module works offline. Parts are generated randomly at initialization, and their difficulty and features are evaluated by the designer (collector). The difficulty of these parts is classified into five degrees. Features of these parts are classified into three categories depending on their number of 1) coins, 2) blocks, and 3) enemies. Five degrees of difficulty and three categories of features correspond to those of players' skills and players' preferences, respectively. The parts used in this competition were evolved by us in advance and saved into the parts pool.

The *parts connector* is a module which generates a level as serial connection of evolved parts. Some parts which match best to the player's skill and preferences as derived from the *skill and preference estimator* are selected as candidates. This module connects these candidates from left to right horizontally.

3) Adaptation: Our level generator estimates players' skills and preferences through a *skill and preference estimator*. Those parts which match the player's skill and preference best are selected and connected with a level by the *parts connector*.

At first, this module selects some candidate parts whose difficulty matches the player's skill. These parts are then examined for whether they match the player's preference. The selected parts are connected sequentially by the level, growing it from left to right. This selection–connection procedure is repeated until the length of generated level meets the requirement of the competition.

4) Creative Control: The designer can control the generator in at least two important ways. The estimation of players' skills and preferences is done through human-authored rules, based on our domain knowledge. Also, the parts are evolved using IEC, and their difficulty and features evaluated by human designers.

5) Strengths and Weaknesses: Our approach has two main advantages. 1) We generate levels that correspond to players' skills and preferences. 2) Designers can affect the composition of levels directly through IEC. No formula needs to be derived for the fitness function of the evolutionary algorithm, because the level parts are evaluated by the designers themselves.

The main weakness of our approach is that the variety of levels depends on the evolved parts. If there is not enough variety in the parts pool, the generated levels may be monotonous. The variety of levels also depends on that of the evolutionary mechanism used in IEC. IEC relies on interaction with humans; it becomes a bottleneck for evolution, because of the (human) time required for evaluation.

6) Generalizability: Our approach is capable of applying to various types of game content. The approach simply consists of two main modules: 1) collecting parts of game content through IEC, and 2) connecting these parts. Moreover, the propriety of rules for players' skills and preferences estimation could improve by tuning rules [25].

C. Nathan Sorenson and Philippe Pasquier

1) Idea and Architecture: Our system combines an evolutionary algorithm and a constraint satisfaction solver to generate

levels in a top–down manner. It is a generic approach which is able to create levels for a variety of games, and Mario is one of its primary applications. As opposed to bottom-up techniques characterized by low-level production rules that can be inflexible and difficult to debug, our system is ultimately driven by a high-level fitness function that specifies desirable design goals independent of any particular generative procedures. This fitness function, which we use to guide the evolution of a population of potential level designs, is based on the observation that certain configurations of challenge are vital to a player's experience of fun [26], [27]. Specifically, levels which present the player with alternating periods of high and low difficulty, known as rhythm groups [28], are often considered examples of good design.

The fitness function used for the competition is a modified form of one previously discussed [29], and is used to estimate the entertainment value of a given level. Essentially, the function infers the location of a number of rhythm groups, according to threshold parameters which identify periods of low challenge. Each of these rhythm groups is then assessed to ensure it presents an appropriate amount of difficulty to the player. The underlying model is described in (1), where c_i is a heuristic estimation of the challenge of rhythm group i, and M represents the ideal amount of challenge a player can experience while still having fun. This formulation rewards levels that have a large number of rhythm groups with appropriate degrees of difficulty. Because rhythm groups boundaries are located at periods of low difficulty, levels that alternate between challenging and relaxing segments will be rated the highest and be favored for selection by the genetic algorithm

$$\sum_{i=0}^{n} \frac{2c_i}{M} - \frac{c_i^2}{M^2}. \tag{1}$$

A challenge presented by the evolutionary approach is that the crossover and mutation operations often yield infeasible offspring which contain gaps that are too wide to leap across or walls too high to jump over. A constraint satisfaction subsystem is used to repair these unplayable designs, and is detailed in previous work [30]. This subsystem is also used to enforce the contest regulations that dictate the specific number of various design elements that must be present in a valid level.

2) Offline Training: Offline training is used to find values for the constant terms in the fitness function. Our approach attempts to find parameter values which assign high values to well designed levels, and low values to poorly designed levels. A number of actual levels from the original *Super Mario Bros* form the set of positive examples and a number of levels randomly generated with no regard for player enjoyment form the negative set. The optimal parameter settings are those which best discriminate between the two sets.

3) Adaptation: Currently, the generative process is guided only by the fitness function, which results in challenge configurations that resemble those of the original *Super Mario Bros* game. However, adaptive design could certainly be considered in future work. By adjusting the model parameters based on player feedback, levels could be generated that have different challenge configurations. An example of this would be generating easier levels by reducing the value of M if the player is found to be failing more than expected.

4) Creative Control: One of the advantages of a top–down generative approach is that it provides a human designer with a small number of high-level parameters to manipulate. The simplest way to influence the design of a level is through the manipulation of the model parameters. By varying the value of M over time, one can create levels with a specific difficulty profile. For example, one could strategically inflate M to produce levels that have a particularly difficult portion at the halfway point, with another challenging section near the end. Another approach to influence the generated levels is to anchor any manually created elements of a design. The evolutionary algorithm is then not permitted to alter these human-created portions of the level. Because the fitness function is applied to levels as a whole, this procedure results in the algorithm selecting for designs that best incorporate these fixed elements into a cohesive experience. In other words, if a designer creates a very challenging segment of a level by hand, the algorithm will naturally create easier segments on either side of this section.

5) Strengths and Weaknesses: An advantage to the evolutionary approach is the ability to influence the designs at a high level by manipulating the fitness function. However, genetic algorithms and constraint solvers are both computationally intensive, and, therefore, only offline generation of levels is practical; it is not yet possible to generate a level on the fly as a player is playing. Search time is not prohibitive, however: if the original population of level designs is seeded with existing well-designed levels, new viable designs can be found quickly, even within the one minute time limit dictated by the contest.

6) Generalizability: The system's top–down design is motivated by the goal of devising a general approach to level generation which is not bound to a single, specific game. For example, the genetic encoding of the levels is not only applicable to Mario, but can describe any spatial arrangement of components; thus, it is suited to describing many different types of game levels. More importantly, the fitness function is defined only in terms of the configuration of challenge over time, and is likely applicable to any game where this dynamic is fundamental to player enjoyment, such as action or arcade games. We are currently exploring the possibility of using our system to create levels for top–down adventure games such as *The Legend of Zelda* (Nintendo 1986). Though this has proven to be a much more difficult task, our initial results are promising.

D. Peter Mawhorter: Occupancy-Regulated Extension

1) Idea and Architecture: The occupancy-regulated extension (ORE) algorithm [31] builds a level by fitting together small hand-authored pieces. Each piece (called a "chunk") is annotated with anchor points, which represent positions that the player might occupy relative to that chunk during gameplay. These anchor points are used to align chunks as they are being placed, and once used, each anchor point will not be reused (unless all anchor points get used up). The chunks, which come from a hand-authored library, are annotated with various properties, and generation is customized by defining rules for probabilistic chunk selection that depend on these properties

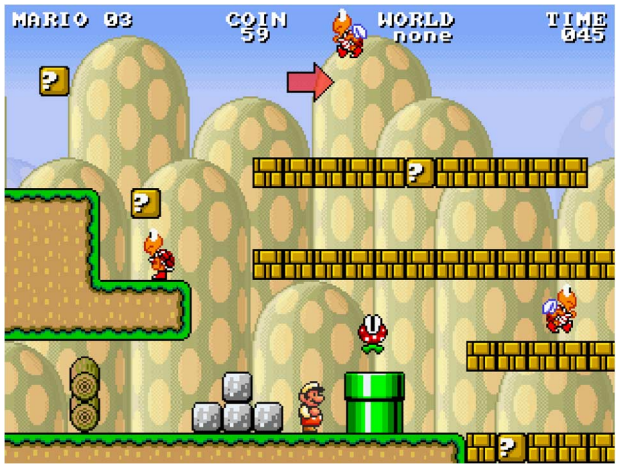

Fig. 3. A screenshot from a particularly complex level generated by Peter Mawhorter's level generator.

(in this way, the algorithm bears some similarity to case-based reasoning [32]). Once the level is constructed using chunks, there is a final postprocessing step that enforces some global constraints and maintains a specified distribution of enemies and powerups. An example of generator output is shown in Fig. 3.

2) Offline Techniques: The ORE algorithm does not use any AI techniques to optimize offline parameters, instead relying on a human to build a chunk library and define both the properties of each chunk and the biases with which chunks are selected during generation. However, future work on automatic extraction of chunks from existing levels would change this, adding intelligent techniques for chunk extraction and labeling, and more fully automating the level-design process.

3) Adaption: For the purposes of the competition, and to demonstrate the customizability of the basic ORE algorithm, some basic adaption techniques were implemented. From the given data, a very rough player model is constructed, focusing mostly on how often the player used the run button (more often being taken to imply higher skill) and how often and how the player died. This player model is then used to alter generated levels, both by altering the default rules for chunk selection (such as by making chunks with a particular label less common) and by altering the distributions of enemies and powerups maintained by the postprocessing step. The adaption parameters were hand-tuned; more robust methods would use some form of optimization, although getting enough data to do so might be time consuming. Of course, because ORE is iterative, it should also be possible to use it for dynamic difficulty adjustment. There would be some additional challenges to overcome (such as finding a way to run the postprocessing online), but dynamic difficulty adjustment has been shown to be a promising application of procedural content generation [33].

4) Creative Control: Because the chunk library is hand-authored, the human designer has quite a bit of control over the types of levels generated, albeit in an awkward manner. Since in this case the chunk library author is the system designer, it is easy to use knowledge of the specifics of the system to author chunks that would result in certain kinds of output (e.g., adding chunks to make levels that had more height variance, for example). The ability to tune the chunk library to achieve desired results does depend on a thorough understanding of the algorithm, however, and so in general, chunk authoring is not an interface that provides much leverage on level design. On the other hand, the ORE algorithm is almost purely incremental, so it is in theory possible to hand-author part of a level and have ORE generate the rest. Given the right interface, and combined with library manipulations, this would offer a rich interface for mixed-initiative level design, which is a topic that has already received some study [13].

5) Strengths and Weaknesses: The main strengths of the ORE algorithm lie in the variety and unpredictability of possible output (it is a generator that regularly surprises even its author) and in the possibilities for customization. Combinations of low-level chunks result in emergent structures that can be quite complex, which means that even after playing many levels generated from the same chunk library, one will still encounter surprising new constructs. The ability to manipulate the chunk library and the fact that the algorithm is iterative mean that ORE has lots of potential for customization to different purposes. Unfortunately, the iterative model means that certain constraints (including playability constraints) are difficult to implement. In this respect, ORE is unlike many other generators [16], [29], [34], which take advantage of more constrained generation to achieve a particular goal.

6) Generalizability: As written, ORE could generalize to another grid-based game quite easily, and in theory any spatial (and even some nonspatial) content could be generated using it. As long as there are concepts of anchor points and chunks, ORE can generate content in a space. The strength of the algorithm depends on the specifics of the anchors and chunks, however. ORE works well in *Super Mario Bros* in part because using potential positions as anchor points naturally results in coherent levels.

E. Glen Takahashi and Gillian Smith: The Hopper Level Generator

1) Idea and Architecture: Hopper was designed to create levels that imitate the style of *Super Mario World* levels. These levels are customized according to the style of player and their skill at playing, both of which are inferred from player metrics.

Hopper uses a rule-based approach to place level terrain, enemies, coins, and coin blocks on a tile-by-tile basis. Levels are built from left to right, with probabilities governing which tile will be placed next. These probabilities are manually tuned according to the inferred player types and difficulty described below, and control the variance in terrain height, occurrence and width of gaps, and frequency of enemy placement. For example, an "easy" level will have a low probability of gap placement, and a level generated for a speed run play style will be flatter than one created for a player who jumps a lot. Obstacle placement is also influenced by the number of times a player died on the particular obstacle: for example, even in a medium difficulty level, there is a lower probability of gaps appearing if the player has previously died by falling down a gap. In order to ensure a reasonable distribution of gaps and

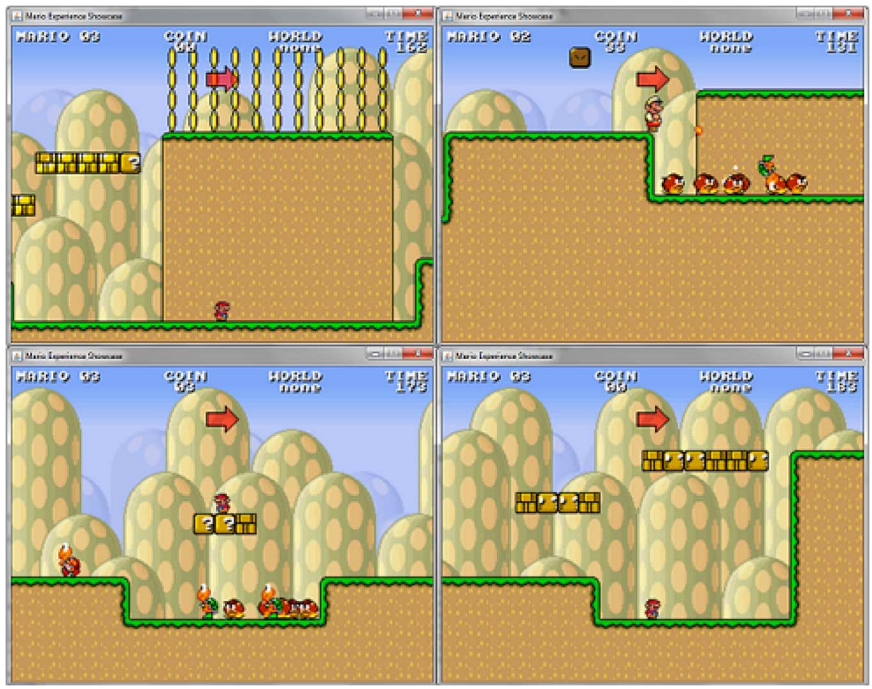

Fig. 4. Examples of the hidden coin zone (top left), fire zone (top right), shell zone (bottom left), and super jump zone (bottom right), as used in Glen Takahashi and Gillian Smith's level generator.

enemies, the probability of placing these increases with the distance from the last such feature.

2) Offline Training: No offline training was performed.

3) Adaptation: Based on metrics from the initial test level, players are classified in two ways: by the type of behavior they exhibit, and their skill level. These classifications drive the level generation process by influencing generation parameters. Hopper infers three different special styles of player behavior: a speed run style, an enemy-kill style, and a discovery style. A player is categorized as a "speed runner" if they take very little time to complete a level and do not engage in collecting coins or killing enemies. The enemy-kill style is applied to players who spend a lot of time killing enemies. Players are placed into the discovery style category if they collect a large number of coins, powerups, and coin boxes. These categories are not mutually exclusive; i.e., it is possible for a player to have none of these traits, or more than one of them. There are three discrete difficulty levels—easy, medium, and hard—which are determined by the number of times the player died in the test level, and how long it took the player to complete it. Player styles, difficulty levels, and the thresholds used to calculate them are based on informal observation of a number of players with differing skill levels.

4) Creative Control: This base level generation algorithm creates approximately 85% of a given level. The remainder is taken up with "special zones" that are built from human-authored patterns. The four special zone patterns are: fire zone, shell zone, super jump, and hidden coin area. A given level may contain a small number of each type of zone, depending on the inferred player behavior and difficulty level. Each zone has a varying length. Fire and shell zones are more likely to appear for players who spend a lot of their time killing enemies, the super jump zone appears for speed run players, and the hidden coins appear for discovery style players. Fig. 4 shows an example of each zone.

5) Strengths and Weaknesses: Hopper is capable of creating a wide variety of levels for different player types; however, only the first level it creates is given to the player. Future incarnations of this generator will incorporate a generate-and-test structure similar to that found in an author's prior rule-based level generator [34]. Generate-and-test allows a designer to exert additional control over created levels by specifying global qualities of the level that they wish to see; it would also be possible to choose levels that are similar to others that the player has enjoyed.

The incorporation of special zones gives a designer direct influence over the generator. These patterns and the probabilities for their appearance are quite simple to specify. They reflect a desire expressed by some 2-D platformer designers [35] for procedural level generation to support designers by building a level around preauthored sections.

Hopper's parameters for adaptation are currently tuned based on informal testing with friends and colleagues. A formal study of different player behavior in platformer levels would improve Hopper's adaptation and be a useful contribution to the field. Incorporating a model of the difficulty of certain combinations of geometry [33] is also a potential way to improve adaptation. More information from player metrics would be helpful in categorizing player behavior; for example, time-stamped player behavior would allow us to determine the purpose of a jump, or understand if the player confidently killed enemies or made multiple attempts before being successful.

6) Generalizability: Hopper's level generation technique is not particularly extensible to other genres; while rule-based approaches in general have shown promise in content generation

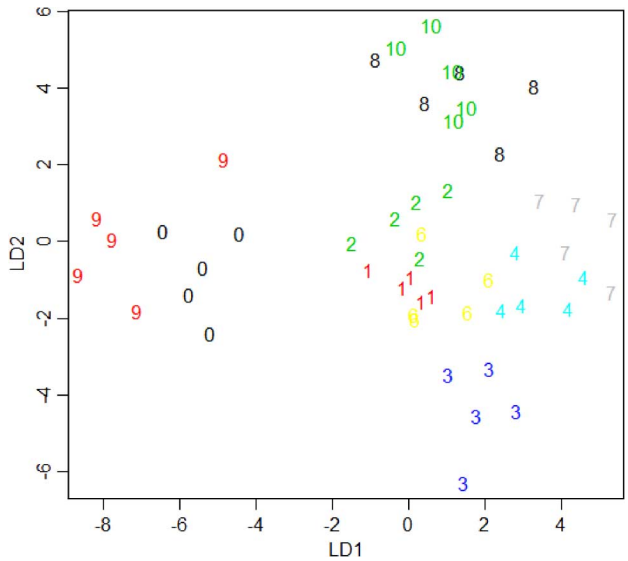

Fig. 5. Linear discriminant analysis of 11 players with five sessions each, projected onto the first two dimensions. Data used by Robin Baumgarten's level generator.

[36], [37], they require a great deal of domain specific information to be built into the rules. However, the general approach of creating levels based on a formal understanding of play styles and associated behavior is an interesting future direction for research in procedural level generation.

F. Robin Baumgarten: LDA-Based Level Generator

1) Idea and Architecture: This level generator uses linear discriminant analysis (LDA) to analyze the data provided after the initial play-through of a player. The new data vector is projected into an LDA space created by playing data gathered in a prior survey. This LDA representation provides us with a single value that we interpret as skill and use to create a level based on handcrafted level chunks with varying difficulty.

Discriminant analysis is used in statistics and machine learning to characterize or separate classes of objects based on a set of measurable features and class information of these objects. LDA utilizes a linear combination of these features to separate the groups of objects. This combination can be used as a linear classifier or for dimensionality reduction. LDA has previously been used to estimate feature weights for heuristics in an Othello game tree [38], and to automatically analyze logged game data to identify the most significant metrics for player classification in *Pac-Man* [39].

2) Offline Training: In our case, we first gathered data in a small survey, which comprised the playing data of 11 players playing five different levels each. The levels were randomly generated (but the same across players) and had an increasing difficulty. For data analysis, we use LDA to both perform a dimensionality reduction and extract information about player behavior from the resulting transformed space, which is shown in Fig. 5. We treat each set of five sessions of a player as one class.

The weights of the features in the first dimensions of the LDA transformed space indicate the most important features that determine the behavior of a player and how it differs from other players. A positive side effect of this method is that unimportant or highly correlated features are eliminated automatically.

3) Adaptation: As the LDA space automatically highlights variables that were especially helpful in separating players from each other, we can use the first few dimensions of the feature vectors in LDA space to guide the level generator in order to tailor a level suitable to the player.

In our initial survey, we found that the first LDA dimension (LD1 from now on) gave us a fairly accurate indication of player skill; as players we (subjectively and manually) judged as good (bad) players had a high (low) LD1 value. Thus, in this initial version of our algorithm, we only used the LD1 value of each player to guide level generation.

4) Creative Control: Our level generator builds levels by concatenating chunks of predesigned level parts, each with a length of slightly more than one screen (25 blocks). The human designer manually annotates the expected difficulty of each chunk, allowing a selection based on the LD1 skill level. The proportion of easy, medium, and hard chunks is directly based on the estimated skill level, with a slight randomization and repetition avoidance to increase level diversity.

Thus, in this first version of the level generator, the human designer still plays a big role in creating individual parts of the level and annotating their difficulty.

5) Strengths and Weaknesses: The process of judging the skill of a player has been fully automated with the help of LDA using existing playing data of other players, with the possible exception of interpreting the first dimension of the LDA space as the skill level. However, our previous work indicates that a combination of the first two or three dimensions should give an accurate representation of player behavior.

Weaknesses of our current implementation are the dependency on a human designer to create the building blocks of our level, and annotating their difficulty. Furthermore, there was a programming error in the generator that was submitted to the contest, which disabled the proper selection of level chunks and always led to the selection of the most difficult piece first, which led to a low ranking in the competition. This issue has been fixed for following studies.

The described version of the level generator leaves a lot of room for further automatization, especially in selecting appropriate dimensions of the LDA space for level generation, and annotating the difficulty of level chunks, where our A* playing bot could be used (described in [3]).

6) Generalizability: The approach of using LDA to generate a semiautomatic classification of players can easily be generalized to at least some other games, as we have shown with our *Pac-Man* study [39]. It could conceivably be generalized further.

G. Taxonomic Classification of Competition Entries

According to the taxonomical classification in [8], Ben Weber's, Robin Baumgarten's, Peter Mawhorter's and Glen Takahashi and Gillian Smith's level generators can all be classified as *constructive* generators, as they construct their levels in one or a fixed number of sweeps, without backtracking. Nathan Sorenson and Phillipe Pasquier's level generator is *search based*, as it uses a search/optimization algorithm (in this case a

TABLE I
THE RESULT OF THE LEVEL GENERATION TRACK FOR THE 2010 MARIO AI CHAMPIONSHIP, TURING TEST TRACK

Name	Affiliation	Score
Weber	University of California Santa Cruz	9
Shimizu and Hashiyama	Uni. of Electro-Communications Tokyo	8
Sorenson and Pasquier	Simon Fraser University Canada	6
Mawhorter	University of California Santa Cruz	4
Takahashi and Smith	University of California Santa Cruz	2
Baumgarten	Imperial College London	1

genetic algorithm) to search a space of possible content (levels) in order. Tomoyuki Shimizu and Tomonori Hashiyama's level generator is a combination, which performs a search-based generation of level segments (using an interactive fitness function) offline, whereas the online generation of complete levels is constructive.

Only three of the level generators attempted any kind of adaptation to the playing style and/or inferred preferences of the judge. Shimizu and Hashiyama's and Takahashi and Smith's generators adapt the levels using *theory-driven* player models, i.e., the algorithms sort players into categories (e.g., CoinCollector, speed run style) based on thresholds explicitly specified by the human designers. Baumgarten's generator, on the other hand, uses a *data-driven* player model where the classification is based on data collected from a number of players.

VI. RESULTS

Following the scoring procedure presented in Section IV-B, we needed to have at least 15 participants for a fair competition result (with 15 participants we guarantee that each pair of competitor submissions is played at least once in both orders). Since we encouraged everyone present at the competition event to participate as a judge, we ended up having more than 15 participants but fewer than 30. Thus, for the sake of fairness, the winner was decided by taking into consideration the first complete set with all pairs played by the first 15 judges only. The results presented in Table I are also taken from the first 15 participants.

The numbers presented in the score column in Table I refer to the number of times the particular generator scores higher than another generator when played in a pair. The maximum value of the score is 10: the competitor is preferred to any other of the five competitors in both orders. As can be seen from the table, the winner of the competition was Ben Weber with a difference of only one vote from Tomoyuki Shimizu and Tomonori Hashiyama who came in second, with the other competitors relatively evenly spread out in the score table.

A. Level Features and Pairwise Preferences

During the competition, all levels that were generated by the generators were stored on the competition server together with the reported preferences of the players. This has given us an opportunity to extract statistical features from the levels, and attempt to correlate these with player reported preferences. Note that, in the first implementation of the competition server–client system (used in the CIG 2010 competition), data related to player actions were not collected. Thus, any attempt to relate level features generated with player characteristics and furthermore with reported fun preferences is not possible at this stage. On that basis, reported pairwise preferences cannot be linked to individual players' playing styles (as done in [19] among others) but only associated to level attributes. Any model learned from these data will therefore be a generic model that does not take the differences between players into account.

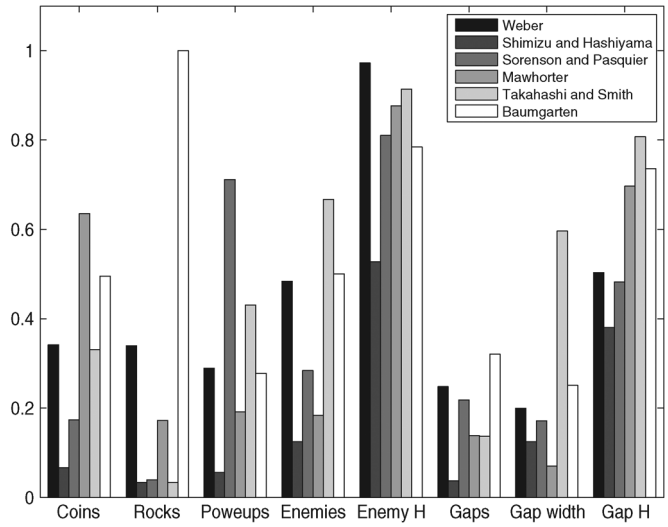

Fig. 6. Average values of eight statistical features that have been extracted from all generated levels of each competitor.

Fig. 6 presents a comparison between the average values of eight key statistical features that have been extracted from the data of all competitors: numbers of coins, rocks, powerups, enemies and gaps, the average gap width, as well as the spatial diversity of gaps (gap H) and enemy placements (enemy H) which is measured by the entropy of the number of gaps and enemies, respectively, appearing in a number of 10 equally spaced segments of the level (see [16] for more details on the calculation of entropy). All feature values are uniformly normalized to the range $[0, 1]$ using max–min normalization. As clearly seen from Fig. 6, the winner's entry (Weber) generates, on average, more gaps than most competitors and the most enemies placed in a rather unpredictable manner. The aforementioned characteristics contribute to more challenging levels which might be one of the criteria that this level generator was preferred more than any other entry. The levels generated by Shimizu and Hashiyama's generator reached second place in the competition with level features that are inverse to those of Weber: the levels have, on average, fewer coins, enemies, and gaps while enemies are more evenly distributed across the level. Results from these two very different levels indicate that the relationship between level characteristics and fun is most likely not a simple linear function. They also reflect upon the highly subjective notion of level aesthetics and gameplay attributes.

TABLE II
CORRELATION COEFFICIENT VALUES BETWEEN EIGHT KEY STATISTICAL FEATURES EXTRACTED FROM GENERATED LEVELS AND FUN PAIRWISE PREFERENCES. SIGNIFICANT VALUES APPEAR IN BOLD —SIGNIFICANCE EQUALS 5% IN THIS PAPER

Extracted level feature	$c(\mathbf{z})$
Coins	**−0.466**
Rocks	**−0.466**
Powerups	−0.333
Enemies	−0.333
Enemy H	**−0.466**
Gaps	−0.333
Gap width	**−0.466**
Gap H	−0.333

At the bottom of the score board, the entry of Baumgarten generates way too many rocks and gaps which result in highly challenging levels that were not preferred by most judges. It is also worth noticing that Takahashi and Smith's entry (which received two votes) generates, on average, challenging levels with very wide gaps which are placed in a rather unpredictable manner. The levels generated by Mawhorter's entry are characterized by many coins while the entry of Sorenson and Pasquier seems to generate the most powerups among all competitors. These level features appear to be valued by some judges and brought these entries in the middle of the score board.

Table II presents a correlation analysis between the judges's expressed fun preferences and the eight key level features examined earlier. Correlation coefficients are obtained through $c(\mathbf{z}) = \sum_{i=1}^{N}\{z_i/N\}$, following the statistical analysis procedure for pairwise preference data introduced in [19], where N is the total number of game pairs (N is 15 in this paper) and $z_i = 1$, if the judge preferred the game with the larger value of the examined feature and $z_i = -1$, if the judge chose the other game in the game pair i. The p-values of $c(\mathbf{z})$ are obtained via the binomial distribution. A high positive correlation value indicates that levels with a high value of the examined level feature are in general preferred over levels with lower values of that feature. On the contrary, features which are highly but negatively correlated to fun preferences characterize levels which are not preferred. A correlation value close to zero suggests that there is no apparent linear relationship between the examined feature and fun preferences. From the significant correlations of Table II it can be inferred that levels with fewer coins and rocks, smaller gaps, and even distribution of enemies are, in general, preferred (or generate more fun). There appears to be a relationship between level fun preference and game challenge showcased through these statistical effects: the lower the challenge in a level the higher the preference for that level. The clear relationship of the two can only be obtained if the sample size of the judges is larger and, in addition to fun preferences, the judges are asked to report the level that generated the most challenging gameplay. Previous work on the relationship between reported fun and reported challenge in *Super Mario Bros* has demonstrated that they are highly and positively correlated [16] (in contrast to what is observed here), at least for a more restricted class of levels.

The correlation values obtained suggest that the relationship between content characteristics and game preference is most likely nonlinear (as also found in [16]) since the linear relationships are far from being exact—i.e., the correlation values are significant but not close to 1 or −1. Moreover, studies have shown that player behavioral characteristics are key towards the prediction of player preferences (see [16] among others) which further implies that level personalization would most likely yield more successful generators.

In order to further validate the results of the competition with more participants/judges, we are currently performing an additional round of data collection online. A Java applet has been created and placed on a web page,[5] which has been advertised over social networks, mailing lists and blogs.

VII. DISCUSSION

This section discusses what we can learn from this round of the Level Generation Track (which was also the first academic PCG competition and the first competition about adaptive or controllable PCG), both about organizing a PCG competition and about generating levels for platform games.

A. Organizing a PCG Competition

Compared to other game AI competitions the PCG competition attracted a reasonably large set of competitors, representing a considerable diversity geographically and, in particular, in terms of algorithmic approaches to the particular content generation problem. All of the entries submitted contain novel elements, most of the approaches are sophisticated, and some of them are connected to the competitors' ongoing research programs. The number and quality of submissions indicate a fairly strong interest in the field of procedural content generation, forming a subcommunity devoted to PCG that lies within the broader game AI and computational intelligence and games communities. Therefore, it seems very plausible that given a simple enough interface and an interesting enough content generation problem, future PCG competitions will attract good attention.

In organizing this competition, the organizers drew on experience of organizing several previous game AI-related competitions, as well as a set of "best practices" that have been accumulated within the computational intelligence and games community over the past few years. One core principle is that the competition should be as open as possible in every sense, both in terms of source code, rules, procedures, and participation. Another key principle is that the software interface should be so simple that a prospective competitor is able to download the software and hack together a simple entry in five minutes. Limitations in terms of operating systems and programming languages should be avoided wherever possible. It has also become customary to provide a cash prize in the range of a few hundred dollars, along with a certificate, to the winner. We believe that these principles have served us well.

This is not to say that the current competition has been without its fair share of problems, actual as well as potential. It was until the last moment unknown how many members of the audience would be willing and able to participate in the judging, and it would in general be desirable to have a

[5]http://noorshaker.com/participate_in_experiments.htm

larger number of votes cast in order to increase the statistical validity of the scores. One of the key limitations of the existing survey protocol is that all entries need to be played against each other; ideally multiple times from different judges. That generates a large number of judges—which is combinatorial with respect to the number of entries—required to sufficiently assess the entries. This problem can be solved, in part, with a fair sampling of the pairs and an adaptive protocol which is adjusted according to the number of judges existent in the competition room. It is also questionable how representative of the general game-playing population an audience of game AI researchers is. As already mentioned, an Internet-based survey is currently running, where the software is included on a public web page and judges are solicited through mailing lists and social networking sites; this approach would undoubtedly come with its own set of limitations, such as preventing the competitors from gaming the system by voting multiple times themselves.

Additional minor problems include the short time period given for the presentation of the competition; the competitors agree that it would have been very useful to have on-spot presentations of their submissions as well. Moreover, one of the entries included a trivial but severe bug which was only discovered during the scoring, and which was arguably responsible for the very low score of that entry. The competition software repeatedly locked up on several of the judges' laptops during level generation for as yet unknown reasons.

A potential problem which was briefly discussed in Section IV-A is that someone could submit a "level generator" that essentially outputs the same human-designed level each time and, if that level is good enough, it could win the competition. As we have abandoned the idea of forcing additional constraints on the level generators for fear of restricting them too much, such a case would probably have to be decided by the organizers of the competition based on some fairly fuzzy guidelines. The deeper problem is the distinction between a level and a level generator and it is not clear. It should rather be thought of as a continuum with intermediate forms possible, e.g., a fixed level design that varies the number and distribution of enemies according to the player's skill level. (Bear in mind that several of the submitted level generators included complete human-designed level chunks of different sizes.)

A possible solution to the above problem would be to let the judge play not one but several levels generated by the same level generator with the same player profile as parameters. In such a setting, a generator that always outputs the same level would probably come across as boring. This solution would also ensure that the judges rate the actual design capacity of the generator rather than just the novelty value of a single generated level. If this is done, the player metrics might be updated as the player plays, allowing the generators to continuously adapt to a player's changing playing style. It would require that each judge spends more time on judging, which might lead to a shortage of willing judges, but given the considerable advantages it seems like a good idea that the next level generation competition lets judges play several levels from each generator.

There are certainly aspects of the questionnaire protocol used that could be improved on the next iteration of the competition. A four-alternative forced-choice questionnaire scheme [40] could be adopted to improve the quality of self-reported preferences. Such a questionnaire scheme would include two more options for equal preferences (i.e., "both levels were equally fun" and "neither level was fun") and thereby eliminate experimental data noise caused by judges who do not have a clear preference for one of the two levels.

In the future, we might consider including hand-authored levels (e.g., original *Super Mario Bros* levels) among the generated levels; a litmus test for whether the (personalized or other) level generators are really successful would be whether they were generally preferred over professionally hand-authored levels. We would also like to try to answer not only the "which" question about fun levels, but also the "why" question; asking judges why they prefer a particular level over another would be interesting, but would require significant human effort in interpreting the data. Another method would be to ask not only which level was more fun, but also which was more challenging, interesting, etc., similar to the questionnaires used in [16].

Another takeaway from previous CIG competitions is that competitions usually benefit from repetition. When basically the same competition is run a second or third time, competitors get a chance to perfect their entries and learn from each other, meaning that much better entries are submitted. Refining individual entries also means that techniques that are more appropriate for the problem stand out from initially interesting ideas that fail to deliver on their promise. In other words, the scientific value of a competition in general increases with the number of times it is run.

B. Generating Levels for Platform Games

The main point to note about the competition results is that the simplest solution won. Ben Weber's ProMP level generator does not search and backtrack while constructing the level, does not include any human-designed level chunks, and does not in any way adapt to the judge's playing style. Above all, it does not attempt any form of large-scale level structure, pacing or anything similar, but simply places individual level elements in a context-free manner.

It would be premature to conclude that the aforementioned features (adaptation, human-designed chunks, search in level space and macrostructure), which were attempted by the other generators, cannot in principle add to the quality of generated levels. Rather, we believe that imperfect implementation and a lack of fine-tuning were responsible for the relative failure of the more complex level generators. It is clear that the entrants need more time to perfect their entries, and possibly recombine ideas from different approaches. In addition, player behavioral information could assist the generation of more personalized, and thereby preferred, levels (as in [41]). While level generation studies in *Super Mario Bros* indicate features that are responsible for a level's high aesthetical value [16] we are still far from identifying the complete set of features—which could be represented computationally—that would yield a highly engaging platform game. Earlier findings suggest that this feature set needs to be individualized for each player behavioral

type [16]. In other words, the competition needs to run again to give the competitors further opportunities to improve their level generators.

While Ben Weber's level generator did not generate any macrostructure, it can be argued that it generates more microstructure than several of the other level generators. Individual images of levels generated by Ben Weber's generator tend to be densely filled with items, creatures, and landscape features and frequently give the false appearance of macrostructure, such as there being multiple paths through the level. This suggests that the current evaluation mechanism incentivizes judges to make judgements on level quality early or based only on local features.

On a positive note, all the entries produced levels that were, at least once, judged to be more entertaining than some level generated by another entry. Also, the score difference between the winner and the runner-up was very small, despite the level generators being very dissimilar. This suggests that widely differing approaches can successfully be used to generate fun levels for *Super Mario Bros*. This particular content generation problem is still very much an open problem.

We have also attempted to see how much of the preference for certain levels over others, and therefore the quality of level generators, can be explained by simple extracted features using linear correlations. The analysis showed that there are particular key level attributes, such as the number of coins and rocks as well as the average gap width and the even placement of enemies, that affect the fun preference of judges. These features are all negatively correlated; more items and more irregularly distributed items are associated with less fun. The most succinct summary of the statistical analysis would be that the less clutter, the more fun level.

At the same time, the correlations are far from strong enough to explain all of the expressed preferences, suggesting that the relationship between level features and quality is too complex to be captured by linear correlations. We also know from previous research that level preferences are highly subjective. It is likely that an analysis of more extracted features, including playing style metrics, from a larger set of levels played by a larger set of judges could help us understand the complex interplay of the different aspects of level design better.

REFERENCES

[1] P. Hingston, "A new design for a turing test for bots," in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 345–350.

[2] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lönneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, and J. Quadflieg, "The 2009 simulated car racing championship," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 2, pp. 131–147, Jun. 2010.

[3] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 Mario AI competition," in *Proc. IEEE Congr. Evol. Comput.*, 2010, DOI: 10.1109/CEC.2010.5586133.

[4] A. J. Champandard, *AI Game Development*. Berkeley, CA: New Riders Publishing, 2004.

[5] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *Proc. IEEE Symp. Comput. Intell. Games*, 2007, pp. 252–259.

[6] E. Hastings, R. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Proc. IEEE Symp. Comput. Intell. Games*, 2009, pp. 241–248.

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

[8] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in *Proceedings of EvoApplications*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2010, vol. 2024, pp. 141–150.

[9] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 3, pp. 172–186, Sep. 2011.

[10] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Trans. Affective Comput.*, vol. 2, no. 3, pp. 147–161, Jul.-Sep. 2011.

[11] 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

[12] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 2, pp. 111–119, Jun. 2010.

[13] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proc. Int. Conf. Found. Digit. Games*, 2010, DOI: 10.1145/1822348.1822376.

[14] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "Integrating procedural generation and manual editing of virtual worlds," in *Proc. ACM Found. Digit. Games*, Jun. 2010, DOI: 10.1145/1814256.1814258.

[15] N. Shaker, J. Togelius, and G. N. Yannakakis, "Towards automatic personalized content generation for platform games," in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertain.*, Oct. 2010.

[16] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling player experience for content creation," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 1, pp. 54–67, Mar. 2010.

[17] T. L. Taylor, *Play Between Worlds*. Cambridge, MA: MIT Press, 2006.

[18] J. Juul, *A Casual Revolution*. Cambridge, MA: MIT Press, 2009.

[19] G. N. Yannakakis and J. Hallam, "Towards optimizing entertainment in computer games," *Appl. Artif. Intell.*, vol. 21, pp. 933–971, 2007.

[20] G. N. Yannakakis, H. P. Martínez, and A. Jhala, "Towards affective camera control in games," *User Model. User-Adapted Interaction*, vol. 20, no. 4, pp. 313–340, 2010.

[21] G. N. Yannakakis and J. Hallam, "Real-time game adaptation for optimizing player satisfaction," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 2, pp. 121–133, Jun. 2009.

[22] J. Whitehead, "Toward procedural decorative ornamentation in games," in *Proc. Workshop Procedural Content Generat. Games*, 2010, DOI: 10.1145/1814256.1814265.

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

[24] H. Takagi, "Interactive evolutionary computation: Fusion of the capacities of EC optimization and human evaluation," *Proc. IEEE*, vol. 89, no. 9, pp. 1275–1296, Sep. 2001.

[25] J. Jang, "ANFIS: Adaptive-network-based fuzzy inference system," *IEEE Trans. Syst. Man Cybern.*, vol. 23, no. 3, pp. 665–685, May./Jun. 1993.

[26] J. Juul, "Fear of failing? the many meanings of difficulty in video games," in *The Video Game Theory Reader 2*. New York: Routledge, 2009, pp. 237–252.

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

[28] 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.

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

[30] N. Sorenson and P. Pasquier, "Towards a generic framework for automated video game level creation," in *Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2010, vol. 6024, pp. 130–139.

[31] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy-regulated extension," in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 351–358.

[32] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Commun.*, vol. 7, no. 1, pp. 39–59, 1994.
[33] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: Dynamic difficulty adjustment through level generation," in *Proc. Workshop Procedural Content Generat. Games*, 2010, DOI: 10.1145/1814256.1814267.
[34] 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.
[35] A. Neuse*, Personal Communication to Gillian Smith.* May 2010.
[36] A. Smith, M. Romero, Z. Pousman, and M. Mateas, "Tableau machine: A creative alien presence," in *Proc. AAAI Spring Symp. Creative Intell. Syst.*, Mar. 2008.
[37] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," in *Proc. ACM SIGGRAPH*, 2006, pp. 614–623.
[38] M. Buro, "Statistical feature combination for the evaluation of game positions," *J. Artif. Intell. Res.*, vol. 3, no. 1, pp. 373–382, 1995.
[39] R. Baumgarten, "Towards automatic player behaviour characterisation using multiclass linear discriminant analysis," in *Proc. AISB Symp., AI Games*, 2010.
[40] G. N. Yannakakis, "How to model and augment player satisfaction: A review," in *Proc. 1st Workshop Child Comput. Interaction*, Chania, Crete, Oct. 2008.
[41] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling player experience in Super Mario Bros," in *Proc. IEEE Symp. Comput. Intell. Games*, Milan, Italy, Sep. 2009, pp. 132–139.

Noor Shaker received the five-year B.A. degree in IT engineering from Damascus University, Damascus, Syria, in 2007 and the M.Sc. degree in artificial intelligence from Katholieke Universiteit Leuven, Leuven, Belgium, in 2009. Currently, she is working towards the Ph.D. degree at the IT University of Copenhagen, Copenhagen, Denmark.

Her research interests include player modeling, procedural content generation, affective computing, and player behavior imitation.

Julian Togelius received the B.A. degree in philosophy from Lund University, Lund, Sweden, in 2002, the M.Sc. degree in evolutionary and adaptive systems from University of Sussex, Sussex, U.K., in 2003, and the Ph.D. degree in computer science from University of Essex, Essex, U.K., in 2007.

He is an Assistant Professor at the IT University of Copenhagen, Copenhagen, Denmark. Today, he does game adaptivity, procedural content generation, player modeling, reinforcement learning in games, etc.

Georgios N. Yannakakis (S'04–M'05) received the M.Sc. degree in financial engineering from the Technical University of Crete, Crete, Greece, in 2001 and the Ph.D. degree in informatics from the University of Edinburgh, Edinburgh, U.K., in 2005

He is an Associate Professor at the IT University of Copenhagen, Copenhagen, Denmark. His research interests include user modeling, neuroevolution, computational intelligence in computer games, cognitive modeling and affective computing, emergent cooperation, and artificial life.

Ben Weber is currently working towards the Ph.D. degree in computer science with M. Mateas in the Expressive Intelligence Studio, University of California Santa Cruz, Santa Cruz.

His research focuses on the application of planning, machine learning, and case-based reasoning to game AI.

Tomoyuki Shimizu received the B.Eng. and M.Eng. degrees from The University of Electro-Communications, Tokyo, Japan, in 2009 and 2011, respectively.

He has been working with Fuji Xerox co., Ltd., Tokyo, Japan, since 2011. His research interests include computational intelligence for game applications.

Tomonori Hashiyama (M'96) received the B.Eng., M.Eng., and Dr.Eng. degrees in information electronics from Nagoya University, Nagoya, Japan, in 1991, 1993, and 1996, respectively.

He joined Nagoya University in 1996 and Nagoya City University in 2000. Since 2007, he has been with The University of Electro-Communications, Tokyo, Japan. His research interests include computational intelligence for human–computer interactions.

Nathan Sorenson received the M.S. degree in interactive arts and technology from the School of Interactive Arts and Technology, Simon Fraser University, Burnaby, BC, Canada, in 2011.

With his background in mathematics and computer science, he researches the application of computational intelligence to problems that typically demand human creativity. His thesis focused on formal models of fun in video games and automated level design.

Philippe Pasquier received the B.Sc. degree from the Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium, in 1998, the M.Sc. degree from Nantes Science University, Nantes, France, in 1999, and the Ph.D. degree from Laval University, Quebec City, QC, Canada, in 2005, all in computer science.

He is an Assistant Professor at the School of Interactive Arts and Technology, Simon Fraser University, Burnaby, BC, Canada. His scientific research focuses on the development of models and tools for endowing machines with autonomous, intelligent or creative behavior. His contributions vary from theoretical research in artificial agent theories to applied research in computational creativity and generative processes.

Peter Mawhorter received the B.S. degree in computer science from Harvey Mudd College, Claremont, CA, in 2008. He is currently working towards the Ph.D. degree studying games and AI with M. Mateas at the University of California Santa Cruz, Santa Cruz, focusing on procedural generation and storytelling.

Glen Takahashi is currently working towards the B.S. degree in computer science at the University of California—Los Angeles, Los Angeles.

He also works at an education company where he writes programs to aid in the tutoring of children.

Gillian Smith (S'10) received the B.S. degree in computer science from the University of Virginia, Charlottesville, in 2006 and the M.S. degree in computer science from the University of California Santa Cruz, Santa Cruz, in 2009, where she is currently working towards the Ph.D. degree in computer science.

Her research interests include procedural content generation and mixed-initiative design tools.

Robin Baumgarten received the M.Sc. degree in advanced computing from Imperial College, London, U.K., in 2007, where he is currently working towards the Ph.D. degree within the Computational Creativity Group, supervised by S. Colton.

His research interests are applying AI methods to game design and automatically adapting video games.