# Experience-Driven Procedural Music Generation for Games

David Plans and Davide Morelli

***Abstract*—As video games have grown from crude and simple circuit-based artefacts to a multibillion dollar worldwide industry, video-game music has become increasingly adaptive. Composers have had to use new techniques to avoid the traditional, event-based approach where music is composed mostly of looped audio tracks, which can lead to music that is too repetitive. In addition, these cannot scale well in the design of today's games, which have become increasingly complex and nonlinear in narrative. This paper outlines the use of experience-driven procedural music generation, to outline possible ways forward in the dynamic generation of music and audio according to user gameplay metrics.**

***Index Terms*—Adaptive algorithm, genetic algorithms.**

## I. Introduction

TODAY, game music reacts adaptively to player experience, foreshadowing upcoming events in gameplay, and adding impact to major events. Instead of composing music to specific game cues, composers are now asked to compose not only different versions of the same cue (varying in intensity or instrumentation), but also versions that adapt to changes in player style.

In recent games, which usually contain very large open worlds with open-ended narratives, it is often impossible to know what exact events will occur in the game during the length of a particular audio segment. If a segment is simply looped, it leads to tedium for the player, a problem acknowledged by Tomas Dvorak, composer of the soundtrack to *Machinarium* [1]: "Soundtrack music has to be more abstract to give space for the image and also to not be annoying if it repeats" [2]. As one of the most important roles of music in game design is to immerse the player in the gameplay through emotional induction, avoiding tedium is paramount. This is particularly true in mobile applications, which live and die by gameplay length (time between launch and app termination), where emotional engagement is therefore crucial, as mobile operating systems typically kill the application right away when a "home" button is used.

Manuscript received November 03, 2011; revised April 27, 2012; accepted August 01, 2012. Date of publication August 13, 2012; date of current version September 11, 2012.

D. Plans is with the School of Arts and New Media, University of Hull, Scarborough, North Yorkshire YO11 3AZ, U.K. (e-mail: d.plans@hull.ac.uk).

D. Morelli is with the Computer Science Department, University of Pisa, Pisa I-56127, Italy (e-mail: info@davidemorelli.it).

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.2012.2212899

Procedural content generation (PCG) refers to the programmatic (algorithmic) generation of game content. So far, it has been used to produce dynamic, as opposed to precomputed, content such as light maps and levels. As a set of techniques, it offers great advantages toward creating music that adapts more granularly to player experience, avoiding needless repetition and providing an evolving, more emotionally intelligent soundtrack. Repetition itself, in terms of composed music for games, should, for the context of this paper, be understood to be inherently counterproductive. Indeed, repetition in music is a known device, and has been proven to have an effect on liking [3]. Studies locating repetition within an overall theory of musical syntax to better understand its role in popular music, for example, have usefully outlined its success as a mechanism of musical production [4]. However, there is often a direct disconnect between the near nonlinearity of recent games, which can contain open worlds and narratives, and the use of repetitive music: often, users find the tail ends of composed pieces of music simply "restarting," having run out of timed gameplay. This has prompted new games to have an increased number of cues, for composers to work to. But as Collins examines, "composers now commonly re-use cues in other areas of a game, to reduce the amount of unique cues needed, but without creating a repetitive sounding score" [5]. We believe that some of the techniques outlined in this paper, while obviously not meant to replace the work of a composer, or to label all repetition erroneous or facile, could help in bridging the gap between the increasingly large scale of games and their music, the composer's dilemma, and what Collins terms "listener fatigue" [5, p. 269].

Procedural music can be defined as "composition that evolves in real time according to a specific set of rules or control logics" [6, p. 13]. More specifically, procedural generation of audio can mean that audio events are stored as code and unpacked when triggered to synthesize audio in real time, requiring only text-based storage as opposed to sampled audio loops, offering significant memory and storage savings.

Notwithstanding this advantage, PCG is not yet common in game design. Primarily, this is because the complex control logics that PCG demands are at odds with traditional budget allocation for music, which must compete with graphics. Hiring traditional composers is simply cheaper than hiring audio programmers. Also, procedural audio tends to be central processing unit (CPU)-intensive and difficult to tie logistically to meaningful game elements [6, p. 12].

However, new PCG techniques beyond the field of audio design and music composition are emerging, modeling player experience in order to create adaptive content [7]. While experience-driven PCG (EDPCG) is now primarily used to gen-

erate terrain, maps, levels, and weapons, "there is room for approaches other than those that have already been tried; both the type of content generated and the algorithmic approach to generating it may change in the future" [7, p. 13].

In the context of PCG research, content is most often generated as a result of a stochastic search process, such as an evolutionary algorithm, which will use a fitness function to evaluate whether created content is appropriate. This is done over many generations, where candidate content is ranked, some discarded, some mutated, and further generations created.

The combination of procedural audio generation and experience modeling could provide ways forward in the design and composition of both diegetic sound and music in large, virtual environments such as open-world games, where traditional composition and sampled audio techniques would not scale. At the same time, it would provide more engaging player experiences in traditional linear narratives, as well as other environments currently influenced by game design, such as therapeutic systems, on-demand and in-flight entertainment systems, and web 2.0 services.

This paper investigates whether recent findings in search-based PCG (SBPCG) [8] and EDPCG could be applied to sound design and music composition. SBPCG refers to a special case of the generate-and-test approach to PCG, whereby the test function does not just accept or reject candidate content, but grades it according to a fitness function [8, p. 4], and EDPCG expects to give this search-based approach a model of the user experience that it can use to "generate content that optimizes the experience for the player" [7, p. 3]. Both can be deployed in order to help avoid tedium in large open-world and open-ended game environments, as well as game-influenced aspects of industrial and medical design. We will primarily focus on biologically inspired computational intelligence, embeddable synthesis, and player experience modeling to provide new pathways for music composition in open narrative media.

### A. The Origins—Algorithmic Composition

From Markov models, generative grammars, transition networks, evolutionary algorithms, and chaos theory, to agent, neural, and cellular automata-based systems, algorithmic composition has a long history, which is now well documented [9]. Algorithmic composition's relationship to other environments, such as game design, is more recent. Entire generative models have been developed for the automated composition of music drawing on theories of emotion, perception, and cognition [10], and models from computational intelligence and stochastic computation in general [11]. Much work has been done in the area of mood tagging and effect for adaptive music, almost always using western-notational representations of music through the Musical Instrument Digital Interface (MIDI) standard to compose segments of music that match particular "moods" [12]. However, while there is previous work in wholly synthesized, generated audio (as opposed to MIDI) in the algorithmic composition domain [13], little work has been done on adaptive game audio, synthesized in real time, that uses player experience modeling to drive adaptive content creation.

### B. Precedents and Future of PCG in Game Development

Early game developers attempted to provide unique gameplay based on personal player style [14, p. 12], despite severely limited storage space [15]. This adaptive process was mirrored by music composers, who used repetition, transposition, and other techniques to offer variation to the player. Algorithmic sound design, which generates audio and music based on rules and behaviors, has been used since the beginning of games development. Usually combining sampled audio with synthesis, it allows for flexible layering and real-time effects instead of just a stereo mixdown of precomposed diegetic audio or music. Reactive ambiences, for example, can be made of collections of sampled audio loops; a country ambience made of bird song, insects, and wind might die down in reaction to a gunshot fired by a player or AI entity.

As early as 1987, Iwai's *Otocky* [16] used the shooting controls in the game to generate different harmonic palettes in the background music, thus allowing the player to affect the game's music through gameplay itself, a concept he later developed in *SimTunes*, a predecessor to many of today's musical video games.

Some LucasArts games, such as *Monkey Island 2: LeChuck's Revenge* [17], use iMuse, a software system that allows musical changes to occur at very small intervals, approximately every four measures, between a large number of branching musical segments [15], depending on the player's in-game exploration choices and nonlinear dialog selections. The music–player interaction afforded by iMuse has continued in many contemporary games, such as *Tron 2.0* (Monolith Productions, 2003), *XIII* (Ubisoft Paris, 2003), and *Heavy Rain* (Cage, 2010), to offer players direct structural control over the cinematic soundtrack without giving them explicit clues as to how their choices affect this music.

Very recent developments in embeddable audio engines such as libpd [18], an adaptation of the Puredata (PD) dataflow language as a signal processing library, mean that it is now possible to use PD for the creation of procedural music and game audio at an industrial level, independent of game engine or programming language. *Spore* (Electronic Arts, 2008) is one of the first examples to implement this in the core design process, with Electronic Arts and Brian Eno creating their own version of PD (EApd) to embed within their core engine.

Kent Jolly and Aaron McLeran, the audio programmers in *Spore*'s team who worked with Eno, described creating adaptive/procedural music as "a different way to compose," where you are actually "composing in probabilities" [19] using game events to trigger musical variations.

In other spheres, sound generation is starting to play a key role in the gameplay itself, in games such as *Papa Sangre* (Somethin' Else Studio, 2010), which use binaural real-time sound to give its users a sense of a 3-D soundworld.

However, as Collins points out, most algorithms used for music control are "transformational, rather than truly generative," due to the "difficulties in composing effective procedural music for games" [6, p. 8].

Namely, this is because procedural music and sound has to be bound by strict control logics in order to function adequately within the constraints of game functions, such as action anticipation, leitmotif (within and across games), reward signals, and emotional induction.

These control logics must enable the music to adapt quickly to player input and in-game parameters, as well as be able to change adaptively when situated in long gameplay. As most modern games offer 40–60 h of scripted scenarios [6, p. 6], and players can spend hours in particularly difficult parts of a game, listening to music that is not adaptive can become tiring and, crucially, boring. Designers and composers have reacted by incorporating timing cues that will fade music out instead of looping endlessly when in long periods of gameplay in the same scenario, or in *Spore*'s case, the "density of the instrumentation in the procedural music is reduced over time." Procedural music may thus "offer some interesting possibilities that may solve some of these complications of composing for games" [6, p. 7].

When algorithms used in procedural music are not just transformational, and are instead generative (according to the terms discussed in [20]), they fit abstract game narratives better than set narratives with traditional plot points. More recently, both the *Creatures* series and *Spore*, relying heavily on procedural generation for the creation of their game world, have opted for more organic approaches, using very short samples of recorded music (note units) to generate the soundtrack in real time according to in-game parameters such as mood, environment, and actions. Even here, however, the algorithmic strategy is mostly transformational and simply stochastic (relying on rule-based grammars), as opposed to purely generative.

An approach to generative algorithms in the procedural generation of audio and music that would not only take into account in-game rule systems, but also search-based, experience-driven parameters, would provide both better variety of transformational potential, and a more relevant emotional connection to the player. We chose to take inspiration from the EDPCG framework outlined by Pedersen [21], and closely modeled player behavior in order to derive musical rules and sets of variables we could then use within our generative algorithms, which procedurally generate MIDI music in real time. Some of the musical structure is generated by a simple iterative process, and some generated by classical genetic algorithms. Below, we outline this process.

### C. Frustration, Challenge, and Fun

Following the EDPCG framework, we encoded *frustration*, *challenge*, and *fun* as functions of the metrics of the player's gameplay, as contained within the MarioAI Championship engine [22], with some modifications. MarioAI is a modified version of Markus Persson's Infinite Mario Bros, which is a public domain clone of Nintendo's Super Mario Bros, and was created for the Mario AI Competition. The focus of the competition was "on developing controllers that could play a version of Super Mario Bros as well as possible" using computational intelligence [23]. We have adapted the Java source code of the Level Generation Track of this competition in order to generate adaptive music for MarioAI following the EDPCG model.

TABLE I
VALUES CHOSEN AS MULTIPLIERS

| Multiplier | Frustration | Challenge | Fun |
|---|---|---|---|
| Shells kicked | 0 | 0 | 1 |
| Running time | 0 | 0 | 1 |
| Coins collected | 0 | 0 | 1 |
| Coin blocks destroyed | 0 | -1 | 1 |
| Monsters killed | 0 | 0 | 1 |
| Fell into gap | 1 | 1 | 0 |
| Time ducked | 0 | 1 | 0 |
| Power blocks destroyed | 0 | -1 | 0 |
| Alive time | -1 | -1 | 0 |
| Time standing still | 1 | 0 | 0 |

As our purpose was to generate adaptive music that would take EDPCG into account, we decided to evaluate frustration, challenge, and fun in real time in order to implement a musical engine capable of reacting to the player's mood as it changed during gameplay.

First, we modified the DataRecorder class within the MarioAI engine in order to derive information about the timing of events: every metric was transformed from a scalar value to an array of values, representing the number of events regarding that particular metric that occurred during a particular timespan. For example, the first scalar of array *coinsCollected* contains the number of coins the player collected during the first second of the game (timespan $=$ 1 s).

Pedersen [21] showed that each of the frustration, challenge, and fun functions has certain weighted correlations with each of the gameplay metrics. Following their findings, we then assigned a $c_{i,j}$ as the multiplier for the $i$th metrics. Thus, for the $j$th function, $c_{i,j}$ is positive if the $i$th metric has a positive correlation with the $j$th function, negative if it has a negative correlation, and zero if it is unrelated

$$f_j = \sum_{i}^{n} c_{i,j} m_i \tag{1}$$

where $f_j$ is one in frustration, challenge, or fun; $m_i$ is the $i$th metrics; and $n$ is the number of metrics.

Table I shows the values that were chosen as multipliers. All the values are normalized on the time span to take into account, i.e., *shells kicked*: the number of shells kicked per second. Values with a positive correlation with the metric taken into account have a positive multiplier (i.e., *shells kicked* and *fun*); values with a negative correlation have a negative multiplier

(i.e., *alive time* and *challenge*); values not correlated have a multiplier equal to zero (i.e., *running time* and *challenge*).

### D. Mapping of Frustration, Challenge, and Fun to Excitement

To keep the musical generation and its rules as simple and obvious as possible, and to offer the player discernible musical changes according to his mood, we decided to map frustration, challenge, and fun to a single metric—target excitement—expressing the mood we try to induce in the player through the music. In our context, we define "target excitement" following this process: we want to excite and encourage the player if: 1) he is not having a frustrating game (the contrast between the gaming experience and the music would be disturbing); 2) he is having fun; and 3) he is not challenged too far (to avoid rising frustration). In this sense, fun is the most important metric in the equation. All our variables (frustration, fun, and challenge) are EDPCG metrics, implemented following its core literature.

To implement target excitement, we assigned weights to each metric

$$\text{target excitement} = \text{fun} - 0.5 \times \text{challenge} - 0.5 \times \text{frustration}. \tag{2}$$

Fun was then positively correlated to excitement as its most important indicator. Frustration and challenge should be negatively related to excitement: the more difficult the game is for the player, the more calming the music should be. We decided to use the pentatonic mode as the calming target. The pentatonic and minor scale is used to achieve a calming effect on listeners because it is the most common scale worldwide, primarily because it lacks any dissonant intervals between its pitches. This makes the pentatonic scale unique in that any of its pitch members can be combined without harmonic clashes, and therefore offer the listener no acoustic challenges. We focused on brightness and major modes to react to excitement, in order to celebrate flow by rewarding the user with obviously joyous-sounding music. By traveling from exciting, bright, major-mode musical structure to pentatonic, soft-filtered structures, we hope to both reduce the frustration of playing a difficult game, and contribute to a state of flow when challenge and skill are matched.

### E. Generative Music Engine

Music is generated at runtime using frustration, challenge, and fun as generative parameters. The generative engine has been conceived to be as simple as possible, yet able to produce simple tonal music. It is composed of the following parts, both of which follow a classic genetic algorithm (GA) structure of (Fitness)Selection–¿Crossover–¿Mutation–¿Offspring:

- harmonic sequence generator;
- period builder for the melodic line.

Every generator (chord sequence, melody) breeds new candidates every time the current item is ready (i.e., the last chord of the current sequence has ended). The fitness function is then used to select the winner from the candidates. Both the harmonic sequence generator and the period builder work in real time, creating musical structures reacting to the excitement metric. The music is generated as a stream of MIDI events.

The generators react in real time to the excitement metric with the following rules:

- beats per minute: 120 beat/min for minimum excitement (relaxed) and 135 beat/min for full excitement;
- scale: pentatonic scale is preferred for minimum excitement, and full major scale is preferred for excitement;
- novelty: repetition of already heard material is preferred for minimum excitement, and introduction of new material is preferred for excitement;
- sparseness: phrases with sparse notes are preferred for minimum excitement, and dense phrases are preferred for excitement;
- resonant filter: low resonant filter is preferred for minimum excitement, and high resonant filter is preferred for excitement.

*1) Harmonic Sequence Generator:* The harmonic sequence generator works on a basic idea of harmony: chords. It is responsible for deciding the next group of chords with respect to a history of all already played chords. A chord is expressed as a main pitch over a particular scale. For example, the C major chord in the C major key is a couple (main pitch, scale) where main pitch $= 36$ (where 36 is the central C note in MIDI), and scale $= (36, 38, 40, 41, 43, 45, 47, 48, \ldots)$, the C major scale in MIDI notes; the fifth grade of the C major key would then have the same scale as the previous example but 43 as the main pitch (a G note). We give the engine the chord sequences (harmonic functions) that we want to be used by the generator, for example, $I-V-I$, $I-IV-V-I$. For simplicity, we only insert chord sequences in a single key (C major).

At runtime, every time a new chord sequence is needed, all existing chord sequences are evaluated by the fitness function: the chords contained in the sequence are matched against the history of already played chords. An index of novelty is then assigned to each sequence. Sequence $s$ has a high historic novelty if it has not been heard before and a high inner novelty if it contains many different chords

$$\text{historic novelty} = \sum_{i=0}^{n} \text{eq}(s, s_i) \tag{3}$$

where

$$\text{eq}(s, s_i) = \begin{cases} 1, & \text{if } s = s_i \\ 0, & \text{if } s \neq s_i \end{cases} \tag{4}$$

and $n$ is the number of sequences in the history; and

$$\text{inner novelty} = \frac{1}{m} \sum_{i=0}^{m} \sum_{j=0}^{m} \text{neq}(c_i, c_j) \tag{5}$$

where $c_i$ is the $i$th chord of the sequence and $m$ is the number of chords in the sequence and

$$\text{neq}(s, s_i) = \begin{cases} 0, & \text{if } s = s_i \\ 1, & \text{if } s \neq s_i \end{cases} \tag{6}$$

$$\text{novelty} = \text{historic novelty} + \text{inner novelty}. \tag{7}$$

For every sequence $s$, we calculate its error $\epsilon_s$ as

$$\epsilon_s = |\text{excitement} - (\text{novelty} + \text{scale type})| \tag{8}$$

where

$$\text{scale type} = \begin{cases} 0.5, & \text{if full major scales} \\ -0.5, & \text{if pentatonic scales.} \end{cases} \tag{9}$$

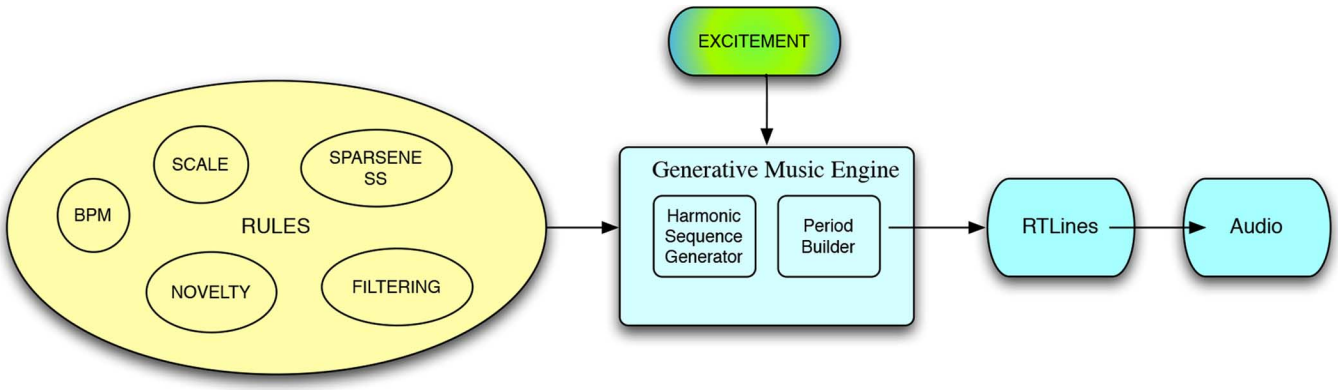


Fig. 1. The Period and Harmonic generators at work.

Sequence $s$ with the minimum $\epsilon_s$ is chosen.

*2) Period Builder:* As well as chords, phrases are another basic compositional notion we use in our engine. For our purposes, phrase $p$ is a sequence of single MIDI notes.

A small set of phrases ($A$, $B$, $C$, and $D$, called seeds) is defined at compile time, which are hardcoded. A large number of variations of each seed is then generated ($A', A'', \ldots, B', B'', \ldots$, etc.). A variation is a small change in a phrase where a random note is either deleted, added, or modified, and we use this as a basic transformative mechanism in crossover.

A period $P$ is a sequence of phrases: $P = p_0, p_1, p_2, \ldots, p_n$. A set of phrases is defined as a compile time ($ABA, AABB, ABAC, ABCA, BDB, BDCB$, etc.) and a large number of variations is generated. A variation is the substitution of a phrase with one of its variations ($AB'A$ is a variation of $ABA$), which we use as the basic process of mutation.

Similarly to what is done with the harmonic sequence generator, every time a new period is needed, all the existing periods are evaluated by fitness functions: the period is matched against the history of already played periods, using the historic novelty formula defined in Section I-E1

$$\text{historic novelty}(P) = \sum_{i=0}^{n} \text{eq}(P, P_i). \tag{10}$$

Also, the density of a period is defined as the average value of the density of the phrases. The density of a phrase is defined as the number of notes in the phrase divided by 16 (the maximum number of notes a phrase can hold in its genotype)

$$\text{density}(P) = \frac{1}{n} \sum_{i=0}^{n} \text{density}(p_i) \tag{11}$$

where $n$ is the number of phrases in the period $P$ and density

$$\epsilon_P = |\text{excitement} - (\text{historic novelty} + \text{density})|. \tag{12}$$

The period with the minimum $\epsilon_P$ is chosen.

### F. JMusic RTLines

Throughout the definition of genotypical material such as phrases and chords, we use JMusic [24], a Java environment for music composition designed to assist the compositional process by providing an open but partially structured environment for musical exploration. JMusic was chosen because it fit with MarioAI's Java environment, and because it already contained musical notions that we could use to model variables and game metrics from player experience into audible events, contained in Java classes such as Note and Phrase. While JMusic is primarily designed to create music offline, a few classes for real-time generation of music, notably RTLine, can be used to generate music on the fly. Our Harmonic and Period generator helps to build such a stream using an RTComposition, a class that amalgamates the different generative streams within our engine, and composes RTLines using a synthesized instrument, SawLPFInstRT2, which is then used to transform the note-by-note information from our stream into synthesized audio.

Fig. 1 outlines the audio engine's process.

## II. Testing

We carried out a small perceptual study that embedded the MarioAI Applet in a Java-WebStart webapp running Node.JS and MongoDB. We modified the Applet to send JavaScript Object Notation (JSON) data directly from our embedded gameplay metrics. The study ran for two rounds each game, and two games per tester. After the first game, a small form was presented to the user to ask pertinent questions (Are you a musician? Have you played Mario before? Did you enjoy this game?). After the second game, only enjoyment was measured through the questionnaire. Question results and metrics were then bundled into JSON packages and sent to our server.

In all, we ran it through 31 testers, eight of which were non-musicians, 17 played an instrument but were not musicians, and six were actual musicians. Our results, from a small userbase, were inconclusive, although interesting. We would like to expand this next, with adequate time and social advertising.

### A. Test Data

All our metrics were expressed in a floating point scale from 0 to 1. In all two-round games, the frustration during the first round was generally higher than during the second round: 90th percentile in the first round was 0.48 (average value 0.11 with 0.29 standard deviation), while in the second round, it was 0 (average value 0.00 with 0.01 standard deviation). Frustration was

0 on 77% of the time in the first round and 0 on 93% of the time in the second; this indicates that the second round was almost always easier for all the players while the first round was sometimes frustrating. This was to be expected, as less experienced users get used to controls and game dynamics during their first round. We built a randomization system that presented either a game round with generative music, or a precomposed MIDI track alternatively, each time a test was requested. Of all tests, the first round was generative 0.61 of the time. Contrary to our expectations, average frustration during the generative round was equal to the MIDI round, with 0.08 (with 0.25 standard deviation) for generative and 0.03 (with 0.15 standard deviation) for MIDI rounds. Generative rounds were prevalent during the first round, which we think probably biased bad results for generative rounds. However, the frustration average during generative rounds was consistently lower than the frustration average during first rounds overall, which could be taken to mean that generative music helped in lowering player stress during the first round of play; however, without adequate measurement systems such as galvanic skin response and heart rate, and an adequate clinical protocol with physically present users, as opposed to remote ones, we cannot offer proof of this.

*1) Enjoyment and Fun—Similar Results:* We found that fun levels in our metrics looked approximately the same during gameplay (average fun during generative round: 0.61 with 0.29 standard deviation; average fun during MIDI round: 0.64 with 0.29 standard deviation). Enjoyment also looked the same during generative and MIDI rounds (average enjoyment during generative rounds was 0.66 with 0.35 standard deviation and average enjoyment during MIDI rounds was 0.67 with 0.33 standard deviation). Users reported the same level of enjoyment 77% of the time, which gave us inconclusive results as to whether generative music, which may have helped lower stress during first rounds, contributed at all to overall levels of enjoyment. Nonetheless, such a high percentage of users selecting the same level of enjoyment could indicate a poor test design: users may have given the same answer because we offered only three choices ("I enjoyed this round," "I almost enjoyed this round," and "I didn't enjoy this round") leading to users giving the same answer for both rounds. When the users reported a different level of enjoyment in the two rounds, the generative round was preferred 57% of the time. More testing and better test design is required to understand if the slightly higher number of preferred generative rounds is statistically relevant.

### B. Future Testing and Work

While the user testing group was too small to give us enough data to successfully prove generative music helped gameplay, we hope to test with larger groups and using a better test design later on. We are encouraged that frustration averages seemed consistently lower during generative rounds, and we hope that on much larger groups, we could gather conclusive data to illustrate this effect. We hope to leave the experiment running at its base URL, and to point to it from ongoing Facebook advert campaigns, over a period of a few months.

## III. Conclusion

While algorithmic composition and procedural music both have a long history as academic disciplines and within the game industry, player behavior modeling, and in particular, adaptive audio engines that listen to such a model are still rare. While games like *Spore* go a long way toward embedding normally academic music and synthesis engines like Puredata into studio-produced games, the state of the art still relies on engines such as iMuse, technology from 1991, for seamless musical component building that adapts to player decisions, trying to connect pieces of music as the player moves from area to area, using the idea of seamless musical "bridges" to connect major pieces of music.

This paper proposes that paradigms such as EDPCG, and ideas from computational intelligence in use in player experience modeling, could transfer into adaptive musical composition for games. It proposes the use of tried-and-tested evolutionary algorithms subsuming the idea of EDPCG as one example of using computational intelligence to achieve this. We think that such an approach could help build adaptive audio engines to supersede tools such as iMuse, pointing the way forward for music in games that is not just linear and scene reactive, without necessarily demanding large investment. While libraries like JMusic are practical for the purposes of engaging with a competition-driven environment such as MarioAI, embedded composition environments such as libpd can now be used and embedded in most major programming languages. This makes them accessible to popular game engines such as Unity and indeed iOS/Android and HTML5 environments, which puts decades of research into algorithmic composition and real-time audio synthesis at the fingertips of not just AAA studios, but also amateur and indie developers, from whom so often come gameplay ideas that change the field.

## References

[1] Machinarium [Online]. Available: http://machinarium.net/
[2] Game-OST [Online]. Available: http://game-ost.com/articles.php?id=76action=view
[3] D. J. Hargreaves, "The effects of repetition on liking for music," *J. Res. Music Edu.*, vol. 32, no. 1, pp. 35–47, 1984.
[4] R. Middleton, "'Play it again Sam': Some notes on the productivity of repetition in popular music," *Popular Music*, vol. 3, pp. 235–270, 1983.
[5] K. Collins, "An introduction to the participatory and non-linear aspects of video games audio," in *Essays on Sound and Vision*, S. Hawkins and J. Richardson, Eds. Helsinki, Sweden: Helsinki Univ. Press, 2007, pp. 263–298.
[6] K. Collins, "An introduction to procedural music in video games," *Contemporary Music Rev.*, vol. 28, no. 1, pp. 5–15, 2009.
[7] 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.
[8] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation," in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2010, vol. 6024, pp. 141–150.
[9] G. Nierhaus, *Algorithmic Composition: Paradigms of Automated Music Generation*. New York: Springer-Verlag, 2009.
[10] D. Birchfield, "Generative model for the creation of musical emotion, meaning, and form," in *Proc. ACM SIGMM Workshop Exp. Telepresence*, 2003, pp. 99–104 [Online]. Available: http://doi.acm.org/10.1145/982484.982504
[11] W. Hsu, "Strategies for managing timbre and interaction in automatic improvisation systems," *Leonardo Music J.*, vol. 20, pp. 33–39, 2010.

[12] M. Eladhari, R. Nieuwdorp, and M. Fridenfalk, "The soundtrack of your mind: Mind music—Adaptive audio for game characters," in *Proc. ACM SIGCHI Int. Conf. Adv. Comput. Entertain. Technol.*, 2006 [Online]. Available: http://doi.acm.org/10.1145/1178823.1178887
[13] M. J. Yee-King, "An automated music improviser using a genetic algorithm driven synthesis engine," in *Proc. EvoWorkshops EvoCoMnet, EvoFIN, EvoIASP, EvoINTERACTION, EvoMUSART, EvoSTOC and EvoTransLog: Appl. Evol. Comput.*, 2007, pp. 567–576.
[14] R. DeMaria and J. Wilson, *High Score!: The Illustrated History of Electronic Games*, ser. Computer Games. New York: McGraw-Hill/Osborne, 2003.
[15] K. Collins, *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. Cambridge, MA: MIT Press, 2008.
[16] T. Iwai, *Otocky*, 2011 [Online]. Available: http://www.youtube.com/watch?v=S9niZfECTPk
[17] LucasArts, *Monkey Island 2*, 2011 [Online]. Available: http://www.lucasarts.com/games/monkeyisland2/
[18] Girorius, libpd, 2011 [Online]. Available: http://gitorious.org/pdlib
[19] D. Kosak, "The bet goes on: Dynamic music in *Spore*," *GameSpy*, Feb. 20, 2008 [Online]. Available: http://uk.pc.gamespy.com/pc/spore/853810p1.html
[20] R. Wooller, A. R. Brown, E. Miranda, J. Diederich, and R. Berry, "A framework for comparison of process in algorithmic music systems," in *Generative Arts Practice*, B. David and E. Ernest, Eds. Sydney, Australia: Creativity and Cognition Studios, 2005, pp. 109–124.
[21] C. Pedersen, "Modeling player experience for content creation," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 1, pp. 54–67, Mar. 2010.
[22] J. Togelius, "2011 Mario AI Championship," [Online]. Available: http://www.marioai.org/
[23] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 Mario AI competition," in *Proc. IEEE Congr. Evol. Comput.*, Jul. 2010, DOI: 10.1109/CEC.2010.5586133.
[24] A. R. Brown and A. C. Sorensen, "Introducing jmusic," in *Proc. Australasian Comput. Music Conf.*, A. R. Brown and R. Wilding, Eds., Brisbane, Qld., Australia, 2000, pp. 68–76.

**David Plans** is a Lecturer in Music at the School of Arts and New Media, University of Hull, Scarborough, North Yorkshire, U.K. He has given papers and performances at the International Computer Music Conference, the European Conference on Artificial Life, the Darwin Symposium, and the Computer Arts Society in London. His early studies were in classical music in several world Conservatories; he then pursued BaHONS and Ph.D. studies in music and computer science at the University of East Anglia. His doctoral research used evolutionary computation techniques, in particular genetic coevolution, as applied to MPEG7, in order to create reflexive music systems. His current research focuses on adaptive media algorithms for therapeutic uses.

**Davide Morelli** is currently working toward the Ph.D. degree in computer science at the University of Pisa, Pisa, Italy, where his research focuses on energetic models for algorithms.

He is a Computer Scientist, a musician, and an entrepreneur. He has run a software consultancy (Parser s.r.l.) since 2001, focusing on customized business solutions, distributed software, and SharePoint consultancy. He is also an expert High Performance Computing Engineer, serving as the cluster manager of ITCenter, the first Acer HPC Competence Centre, and teaches mobile applications programming at the Graduate School, University of Pisa. As a musician, he achieved a Diploma in Saxophone performance at Livorno Conservatory and is an active composer, having written music for ballet, theater, and short films in both experimental electronic and orchestral forms.