Computational Intelligence Techniques for Music-Based Level Generation in One-Button Platform Games

Nikolas Tzimoulis and Anastasios Tefas

Abstract

Procedural content generation is a promising area of research, not only for its potential in automating design work, but also for the artistic possibilities it opens up. In this paper, a method based on computational intelligence techniques is presented for generating levels for a platform game so that they are synchronized to a set piece of music. The levels are created in two distinct stages: First, the music is analyzed and machine learning is used to extract the optimal timing for the player-controlled character's jumps. Then, evolutionary optimization is used to add gameplay elements to the level so that the player is rewarded for following the aforementioned jumps and punished for straying off the path. Experimental results so far have been rather encouraging for using computational intelligence techniques in procedural content generation for games.

Index Terms

procedural generation, machine learning, evolutionary optimization, music, video game.

- *N. Tzimoulis and A. Tefas are with the Department of Informatics, Aristotle University, Box 451, 54124, Thessaloniki, Greece.*
  E-mail: ntzimoul@csd.auth.gr, tefas@aiia.csd.auth.gr

Manuscript revised February 11, 2011.

1 INTRODUCTION

1.1 A short history

From the time of *Rogue* [1] to *Dwarf Fortress* [3] and the present day, video games have been very fertile ground for procedural content generation. In *Rogue*, the primary motivation for developing a self-assembling game was that the creators intended to play the game themselves and did not want to know what happens next beforehand, as described in [2]. During the same period, early video games used procedural generation to overcome the memory restrictions of the relatively small mediums available at the time [17]. For example, in [19], the game *Elite* [18] is praised for including a whole universe of procedurally generated planetary positions, names, politics, and descriptions. In recent years, procedural content generation has been noted as a way for game studios to bring down the high development costs of content creation or to vastly increase the scope of a game, as argued in [17].

1.2 Video game content that can be generated procedurally

Procedural content generation refers to the practice of automating the design of content that is traditionally produced manually by humans. In video games, procedural content generation is predominantly used to generate game spaces [30], like levels [1], terrain [31] and environments [32]. Procedural generation of art assets, like textures [33] and models [34] is also commonplace. Other types of content have been procedurally generated successfully too; e.g., items [35], creatures [36], dramatic pacing [37] and soundtrack [36].

Although there is no specific set of algorithms that are utilized in procedural content generation, there are some guiding principles: The aim is, starting with a small number of parameters or rules, to have a large number of possible outcomes. A collection of algorithms that adhere to that principle is listed in [38].

1.3 Input-based level generation

An underappreciated aspect of procedural content generation is the input parameters. The generation rules use that data as a seed to constrain and specialize the resulting

content's properties, much like the DNA is used to express a living organism's structure. Designing content generation algorithms with expansive input parameters may give procedurally generated content the touch of human creativity and ingenuity it is missing.

In general, it is difficult to replicate high-level design ideas in procedurally generated content. One way to achieve this is by adapting high-level parameters of the level generation process, as was done in [49]. Instead of extrapolating from data collected from player experience, we propose using a piece of music as the basis of the process. A piece of music can be seen as a form of narrative. It has pace, style, passion, subtlety, sometimes even twists and catharsis. All these elements can be transferred to some degree to the final product. In other words, why follow models of rhythmic structure to generate a game level, as was done in [20], when you can use a real song?

This idea was first applied in *Audiosurf* [14]. In this game, songs are used as a seed to generate a virtual roller coaster, the pace of which is synchronized with that of the music. However, there is a number of ways it could be improved. In [20], it was suggested that platform games rely heavily on rhythm. This fact probably makes them the best candidate for music-based level generation. In addition, while the flow of music is represented in *Audiosurf*'s levels, the core way the player interacts with the game is mostly irrelevant to the music. Finally, its creator described *Audiosurf*'s algorithm as "alchemy" in [21]. Like anything else, it would probably benefit from a more systematic approach.

1.4 Computational intelligence in level generation

In that light, a new game has been designed. It is a platformer, to capitalize the connection between the genre and music. In addition, since music always moves forward, the player-controlled character is set to always run forward with constant velocity. That has the effect of simplifying the control scheme down to a single action: jumping. Turning the experience into that of an one-button game makes designing the level generation process more manageable.

To systematize the idea of transferring the features of music to a game level, concepts, ideas and techniques from the field of computational intelligence are used to a great

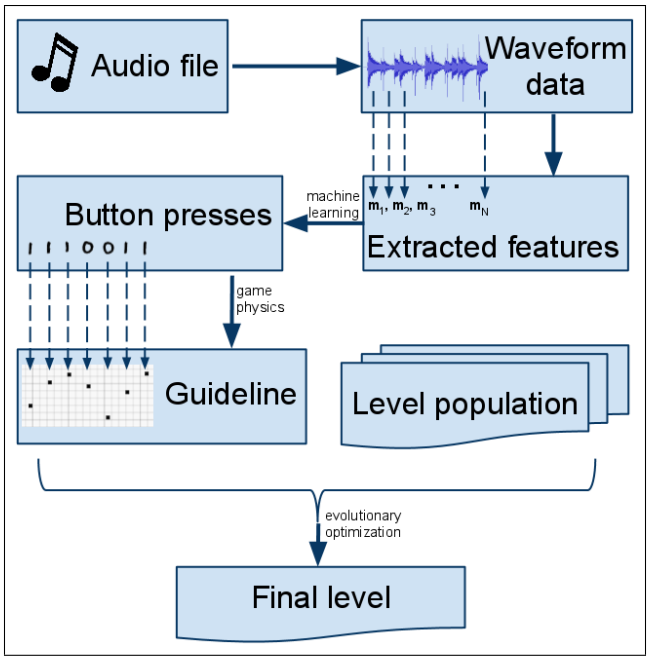

Fig. 1. An overview of the level generation process.

extent. The novelty of the proposed approach in procedural content generation lies in the use of machine learning and evolutionary optimization in order to adjust and specify the generated content.

More specifically, as illustrated in Figure 1, the process begins with some audio features being extracted from the song in regular intervals. Then, a learning machine classifies each of these samples into two categories: button pressed or button released. From this sequence of button presses and following the game's rules of physics and jumping mechanics, the player-controlled character's optimal trajectory is calculated. Finally, a population of game levels is generated, each containing gameplay elements in random placement. Using evolutionary optimization, the levels are adapted to the optimal trajectory through several epochs and the best level is chosen and presented to the player.

The rest of the paper is organized as follows. Terminology from video games and computational intelligence is presented in Section 2. The problem that we attempt to

solve is described in mathematical terms in Section 3. The two stages of the level generation process are detailed in Sections 4 and 5. The implementation of the level generation method is discussed in Section 6. Finally, conclusions are drawn in Section 7.

2 NOMENCLATURE

2.1 Gameplay elements

Gameplay elements are the building blocks for any game's levels. They are visible entities that inhabit the game world and interact with the player. In twitch games, that is games in which player success depends to a large degree on reacting quickly to stimuli [4], gameplay elements tend to be invariable in terms of appearance and designed so that they can be easily identified visually, as presented in [5]. Their behavior and interaction with the player is usually simple and predictable [6]. In other types of games, they are sometimes more subtle, complex or random [7]. For example, in the iconic *Super Mario Bros* [6], goombas are one of the gameplay elements. They all look the same, they always march forward with constant velocity, disregarding platform edges and falling down if they meet one. Colliding with any obstacle makes them march in the opposite direction. Their interaction with the player is equally simple: When the player character lands on a goomba from above, then it is 'stomped'. If the character and the goomba collide in any other way, then it is the character that is hit. An example of more subtle usage is deathclaws, from *Fallout 2* [7], which are originally portrayed as ferocious monsters, i.e., enemies. Player expectations are subverted when the player encounters a pack of these creatures that have human-like intelligence and a highly developed sense of morality.

2.2 Platform games

The platform game (or platformer) is a video game genre characterized by the player-controlled character jumping from suspended platform to platform, avoiding obstacles and collecting items [8]. Success or failure depends on the timing and duration of each jump. The most common form of player-game interaction in platformers is the jump action. Ultimately, the goal is to reach the end of each level in the game. Visually, most platform games are presented from a third-person perspective, to provide the player

with the most helpful view of the jumps. In 2D platformers, the perspective is from the side of the character [6], while in 3D games it is from the back [39].

2.3 One-button games

One-button games, as the name implies, are games that are controlled solely by a single button. This means that at any time the player is faced only with the binary decision of pressing or releasing the button. The button may either be tied to a single action which defines all the interaction there is to the game (e.g., *Canabalt* [9]) or it may change actions depending on context (e.g., *One Button Bob* [10]). A more in-depth exploration of the nuances of one-button interaction has been conducted in [11].

2.4 Music games

"Music games" is an umbrella term used to describe any game that is related to music. For the purposes of this paper, we will focus on a branch of the evolution of music games. In rhythm games, the basic premise is that of the "quick time event". A button name is displayed on the screen and the player must respond by pressing it in a specific timeframe. This is coupled with a piece of music so that the timing of the button presses coincides with the flow of music. For example, in *Dance Dance Revolution* [12] the player moves rhythmically on a stage that contains pressure sensors, following the sequence of 'correct' steps displayed on the screen. In *Guitar Hero* [13] the player has to press the 'correct' combination of buttons on a miniature guitar accessory, following the sequence displayed on the screen.

This 'correct' sequence of button presses has to be programmed in the games manually, by a human designer. The answer to the question "*Can this be automated?*" has given birth to a new category of music games. The first of these games, titled *Audiosurf* [14], aims to create a virtual roller coaster out of a song's features. For example, if the song's pace is slow, the track is uphill and the speed is slow, while if it is fast-paced, the track is downhill and the speed is fast.

2.5 Support Vector Machines

Suppose we are given N observations, each consisting of a w-dimensional feature vector $\mathbf{m}_i \in \Re^w$ and a binary class categorization $b_i \in \{0, 1\}$. If we wanted to build a classifier

that learns the mapping $\mathbf{m}_i \mapsto b_i$ without explicitly storing each observation, we would require a $(w-1)$-dimensional hyperplane in $\Re^w$. All feature vectors that lie on one side of the plane are classified as belonging to one class while the feature vectors that lie on the other side of the plane are classified as belonging to the other class. A *linear classifier* is s *learning machine* that, given such observations, can calculate a separation hyperplane as described above.

Now, suppose that the N observations are the training set and we want the learning machine to classify correctly all potential observations, not only the ones given, that is to have the generalization ability. A possible strategy to generalize the classifier is to maximize the margin between any feature vector of one class and any feature vector of the other class.

Support Vector Machines implement this strategy by transcribing it into the quadratic programming problem of maximizing the margin subject to a number of constraints, each stating that a particular feature vector should be on a specific side of the hyperplane. A number of solutions to the quadratic programming problem can be found in [41].

The above assumes that the observations are linearly separable. What if they're not? There have been two major extensions to SVMs that address this. Soft margins [26] allow for some feature vectors to be ignored when it's impossible to have a hyperplane that separates them clearly. The kernel trick [42] is used to non-linearly transform the feature space to a higher dimension where the transformed feature vectors are more likely to be linearly separable. An extensive introduction to Support Vector Machines has been conducted in [43].

2.6 Genetic Algorithm

The *Genetic Algorithm* is an optimization algorithm inspired by the processes of natural selection. First and foremost, the structure of a solution is abstracted and encoded to binary data form. That makes it possible to generate a random population of candidate solutions.

The fitness function is used to evaluate each of these solutions. In essence it calculates how close each candidate solution is to the unknown ideal solution. Candidate

solutions with high fitness are preserved for the next epochs and have a higher chance of reproduction while the ones with low fitness are discarded.

The Genetic Algorithm includes two mechanisms for promoting the diversification of the population. In each epoch, a number of new candidate solutions is produced equal to the number of low-fitness solutions that are discarded. Each of these new candidate solutions is the result of the reproduction of two high-fitness solutions. This reproduction is achieved by combining parts of the binary representation of the parents. Mutation usually occurs at a constant rate, reversing bits randomly from candidate solutions. An in-depth overview of the Genetic Algorithm can be found in [44].

3 PROBLEM STATEMENT

Let $\boldsymbol{a} = (a_1, \dots a_M)$, where $a_j \in [-1, 1]$, be a vector of unprocessed waveform data from an audio file. Let E be the set of available gameplay elements. Let $\boldsymbol{z} = (z_1 \dots z_N)$, where $z_i \in E$, be a vector of gameplay elements that compose a game level. In its most basic form, the problem we are facing requires us to construct a function $h : \boldsymbol{a} \mapsto \boldsymbol{z}$ that given an audio file it can produce the corresponding game level.

3.1 Guideline generation

To make things more manageable, h needs to be broken down to semantically sensible stages. As it was mentioned in Section 2.4, a rhythm game level consists of a sequence of quick time events which are synchronized to the tune of a piece of music. By applying the same principle in the context of an one-button platform game, the notion of the guideline is reached. The guideline $\boldsymbol{g} = (\boldsymbol{g}_1, \dots \boldsymbol{g}_N)$, where $\boldsymbol{g}_i \in \Re^2$, is a sequence of points in the two-dimensional euclidean space of a platformer's level, restrained by the game's rules for the player-controlled character's movements. As can be seen in Figure 2, it is the 'correct' path that must be followed, much like the 'correct' sequence of button presses that is featured in rhythm games. The reasons why the size N of $\boldsymbol{g}$ is equal to the size of $\boldsymbol{z}$ will be discussed in Section 5.2. Constructing $h_1 : \boldsymbol{a} \mapsto \boldsymbol{g}$ is the first stage of the problem.

This can be further broken down. Assuming that the character moves with constant velocity, calculating the x-axis part of $\boldsymbol{g}$ becomes trivial. In turn, the y-axis part of $\boldsymbol{g}$

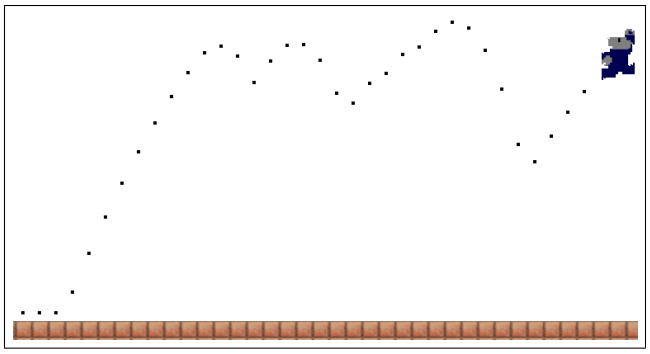

Fig. 2. An example of a guideline in an one-button platform game.

depends solely on the presses of the one button. Assuming consistent game physics and jumping mechanics, a sequence of button presses $\boldsymbol{b} = (b_1 \dots b_N)$, where $b_i \in \{0, 1\}$, is interchangeable with the guideline. To illustrate this, we make one final assumption, that $N = N_0$, where N_0 is the number of frames the game calculates character movement for in the course of a level. In that case, each $g_y(i)$, i.e., the value of $\boldsymbol{g}_i$ on the y-axis, depends on the frame before it and the character's velocity v_y on the y-axis.

$$g_y(i) = g_y(i-1) + v_y(i) \tag{1}$$

In turn, v_y depends on gravity acceleration α_g and whether the button is pressed or released, where v_b is the upwards velocity added to the character while the button is pressed.

$$v_y(i) = v_y(i-1) + \alpha_g + b_i \cdot v_b \tag{2}$$

The reason why we will use the button presses sequence as an intermediary will be explained in Section 4.3.

In addition, instead of using $\boldsymbol{a}$ directly, a vector $\boldsymbol{m} = (\boldsymbol{m}_1, \dots \boldsymbol{m}_N)$, of w-dimensional feature vectors $\boldsymbol{m}_i \in \Re^w$ is extracted, for reasons that will become apparent in Section 4.2. Summarizing, the composition of the first stage is:

$$h_1 : \boldsymbol{a} \mapsto \boldsymbol{m} \mapsto \boldsymbol{b} \mapsto \boldsymbol{g} \tag{3}$$

A solution for calculating h_1 will be presented in Section 4.

3.2 Gameplay element placement

The second stage of the content generation procedure aims at defining $h_2 : \boldsymbol{g} \mapsto \boldsymbol{z}$. That is, to find a method for placing the appropriate gameplay element $e \in E$ will to the corresponding position z_i in the level. We define an evaluation function $f : E^N \times \Re^{2N} \rightarrow \Re$ which given a game level and the corresponding guideline as an input can produce a fitness score that shows how well the level follows the guideline.

$$f(\boldsymbol{z}, \boldsymbol{g}) = c_d \cdot \sum_{i=1}^{N} f_{z_i}(\boldsymbol{g}_i) \tag{4}$$

In (4), $f_{z_i} : \Re^2 \rightarrow \Re$ represents a local fitness function which is different for each type of gameplay element while $c_d \in [0, 1]$, which is treated as a constant for the time being, is a term that is maximized when the frequency of appearance for each gameplay element in the level is consistent with an ideal pre-defined distribution of elements, as will be elaborated in Section 5.3.2.

To maximize f means to find the best possible level for a given guideline. A solution for maximizing f and therefore calculating h_2 is presented in Section 5.

4 MACHINE LEARNING IN GUIDELINE GENERATION

4.1 Dataset creation

We need to build a system that calculates the function h_1, i.e., given a piece of music $\boldsymbol{a}$ it can produce a fitting guideline $\boldsymbol{g}$ as its output. One way to achieve this is by using supervised learning. In order to train such a learning machine, a set of input-output pairs is necessary, input in this case being the piece of music and output being the corresponding guideline. Obviously, this dataset must be produced manually by a human designer.

Since the processing cost of audio is rather high and this work needs to be done manually from scratch, it is clear that it is important to choose the songs carefully, in order to include a variety of genres. That would contribute towards making the machine as general as possible. In addition, unless the available toolset gives powerful editing options, it stands to reason to keep the size of each song to a minimum, in order to minimize mistakes on the part of the human guideline designer.

4.2 Audio feature extraction

At this point, we need to extract features of the music that correspond with the binary decision of whether the button is pressed or released at any given moment. This can be done by finding the moment in the song that corresponds to the point in the guideline and extracting the waveform data using a window around that moment. The appropriate window size s_w should be $s_w \geq \frac{T}{N}$, where T is the song's length and N is the size of the guideline; otherwise, parts of the audio will never be taken into account.

The issue to be answered is what are the features of the music that a human would use to decide when to press or release the button. This is a difficult question to answer, although certain suggestions can be given. One idea is to split the window further into n_{sw} sub-windows. Then, the signal from each sub-window is separated into multiple frequency bands. This would allow for the learning machine to base its decision not just on a single state, but on a progression of states which can be compared with each other.

To separate a sub-window into frequency bands, we can use the *Discrete Fourier Transform* [47]. We assume that the subwindow starts at the n_a-th element of $\boldsymbol{a}$ and ends at its n_z-th element.

$$A_k = \sum_{n=n_a}^{n_z} a_n \cdot e^{-\frac{2\pi i}{n_z+1}k(n-n_a)}, k = n_a, \ldots n_z, i = \sqrt{-1} \tag{5}$$

A is then segmented to n_{fb} frequency bands of increasing size in elements, each denoted as $A_{k_a}^{k_z} = (A_{k_a}, \ldots A_{k_z})$, where k_a is its first element and k_z its last. At this point, a single value must be chosen to represent each band. One option is to select the element with the maximum magnitude while another is to calculate the band's spectral energy density.

$$E_s = \left|A_{k_a}^{k_z}\right|^2 \tag{6}$$

From the above, it is evident that at each time instance we can extract from the audio signal a feature vector of size $n_{sw} \cdot n_{fb}$.

Finally, it should be noted that models used for onset detection [15] seem to be promising grounds in the search for the best feature extraction technique.

4.3 Button presses sequences

A guideline is composed of a sequence of points. Therefore, having the machine output the guideline one point at a time seems like the obvious choice. However, the guideline includes information on game rules that concern jumping mechanics and physics. The machine would have to learn those too, implicitly, to make accurate predictions about individual points in the guideline. As was shown in Section 3, the guideline and the button presses sequence are interchangeable. Therefore, the button presses sequence could be used as target output in the place of the guideline. That would have the effect of relieving the machine from having to internalize these inner workings of the game. An additional advantage of button presses is that it is just a binary choice. This simplifies things further, changing the learning machine from a function estimator to a binary classifier. While there are powerful tools to conduct function estimation, e.g., Support Vector Regression [45], given the choice, a binary classification problem is usually more desirable for its increased tractability.

4.4 Training support vector machines to generate button presses sequences

First of all, a learning machine needs to be chosen. Support Vector Machines [26] are widely used as classifiers with great success [40] and are therefore chosen. The search for the best kernel to use has not been conclusive, however.

To achieve the best results, a systematic method to try the SVM's parameters is needed. With grid search [16], a p-dimensional grid is generated featuring multiple values for each of the p parameters of the SVM. Then, for each cell of the grid, the machine is trained using k-fold cross-validation [48]. When dividing the samples into folds, samples from the same song should not be present both in train and test folds, to prevent the introduction of dependencies that could compromise the integrity of the test accuracy results. Finally, it is important to note that the machine needs to be tested not only against the dataset target outputs but also by a human judging the generated guidelines. Test accuracy is significant, but it does not tell the whole story on its own.

5 EVOLUTIONARY OPTIMIZATION OF GAMEPLAY ELEMENT PLACEMENT

5.1 Composing gameplay elements to generate a game level

Given a guideline which is based on a piece of music, the corresponding game level's purpose is to make the player try to follow the guideline. In rhythm games, this is usually reflected with the player's score corresponding to the percentage of 'correct' button presses. If we were to transfer this idea directly to a platform game, scoring would work by comparing the trajectory of the character's jumps with the guideline. The bigger the distance between the character and the corresponding point in the guideline, the lower the score would get. Obviously, this approach lacks player involvement, in the sense that the player is not expected to be creative with his play at all.

This is an almost paradoxical situation. If following the guideline is what the player needs to do, how is it possible to produce gameplay that rewards creativity? According to the authors' opinion, the answer lies in player mistakes: As long as the player is following the guideline, no creativity is required. However, things could be arranged so that with each mistake, the system descends into a state of imbalance, which the player must amend before going back to the task of following the guideline. It is in that state of imbalance where the player is expected to improvise.

The means by which this can be achieved is a system of rewards and punishments that is realized in-game by a set of gameplay elements. In their interaction with the player, these elements either reward him for following the guideline, helping him regain the balance if it is lost, or punish him for straying off course by pushing the game into the aforementioned state of imbalance, as will be explained in detail in Section 6.

It is evident that an optimization algorithm needs to be selected to carry out the task of placing the gameplay elements in the level in such manner as to ensure the above-stated goal. Since gameplay elements can be combined in many ways and their behavior is subject to change during game development, we conclude that this is a very complex task which includes much uncertainty. Solving it analytically is out of the question, however a more appropriate choice for fast and accurate optimization is to use evolutionary optimization. In this paper, the Genetic Algorithm has been used to that end.

5.2 Mapping the level to the genome

For the Genetic Algorithm to work, each possible level needs to be encoded to a genome consisting of a sequence of genes. For the purposes of this section, we will be using the terms 'genome' 'and game level' interchangeably and both will be denoted as $\boldsymbol{z}$. The obvious way to represent a level is for each gene z_i to represent a game element type and its position in the level.

$$z_i = \{e, x, y\},\ e \in E,\ x, y \in \Re \tag{7}$$

That would introduce the constraint that all solutions must have the same total number of gameplay elements N_G. This can be avoided by adding the 'empty element' $\emptyset$ to E, meaning that no gameplay element is placed in the position.

In order to simplify things and at the same time promote a more even distribution of the gameplay elements across the level, we suggest using a different encoding. In this approach, there is exactly one gameplay element for each point in the guideline. The elegance of this lies in the fact that $\boldsymbol{g}$ and $\boldsymbol{z}$ have the same size N. This is achieved by setting the position of each element on the x-axis to its corresponding point in the guideline. As for the position on the y-axis, that could either be chosen by a deterministic algorithm, so that $z_i \in E$, or be part of the gene, which would mean $z_i = \{e, y\}$. The former would promote stability as it reduces the complexity of the search space while the latter gives more expressiveness to the level generation process. In either case, experimentation has shown that the constraint of having at most one gameplay element in the vertical space above and below each point in the guideline does not significantly impact level expressiveness.

This one-to-one relation between the guideline and the genome is illustrated in Figure 3, where the guideline is depicted above and the genome below. If the gameplay elements of that genome were to be placed in the level, they would coincide with their corresponding points in the guideline on the x-axis but they would not necessarily coincide on the y-axis, as their position would be decided by a deterministic algorithm.

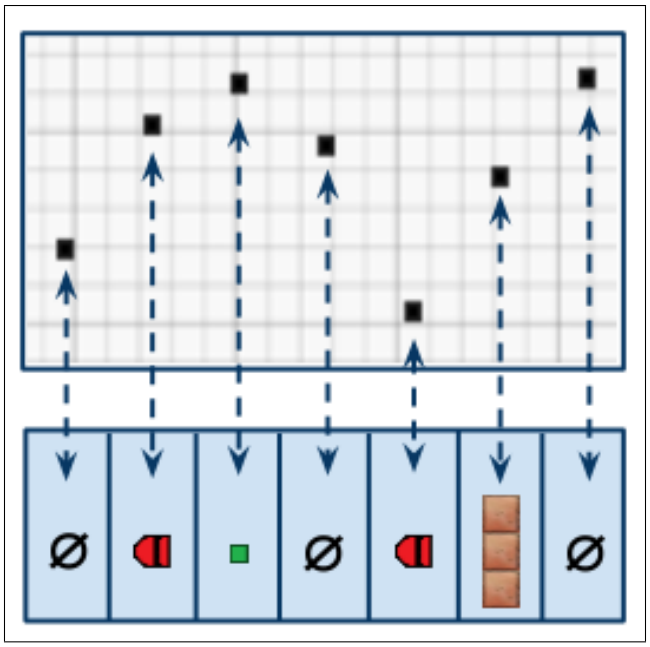

Fig. 3. A guideline and a corresponding genome.

5.3 Fitness function

5.3.1 Local fitness

The first step towards evaluating a generated game level's fitness is to evaluate the local fitness of each gene in as much isolation as possible. The only relevant piece of information that there is available for each gene is the position of the corresponding point of the guideline. That and the gameplay element type e are the only factors that affect the local fitness value. Thus, we define one local fitness function $f_e : \Re^2 \rightarrow \Re$ for each $e \in E$.

For example, a reward-type gameplay element could be defined to have a higher local fitness score at the points in the guideline where the character is jumping high, to signify its importance. Likewise, a different punishment-type gameplay element could be used for when the character jumps too high than for when the character fails to jump at all. The analytical expressions of a few specific local fitness functions are presented in Section 6.

5.3.2 Global fitness

When a new gene is generated, the probability distribution for the gameplay element it is chosen to represent is not uniform. Let p_e be the probability that a newly created gene is the gameplay element e. The conservation of this initial distribution over the course of the selection process is both desirable and non-trivial. To showcase this, consider the problem of choosing an appropriate $f_\emptyset$, the local fitness function for the 'empty element'. Because the empty element's placement fitness is dependent on the existence of better positions for the non-empty gameplay elements, it is difficult to estimate a local fitness function that works in isolation. On the other hand, if we choose $f_\emptyset = 0$ for simplicity, then the selection process would gradually but methodically eliminate any instances of the empty element from the genome. That would lead to the undesirable outcome of an overpopulated level.

To rectify this problem, a central mechanism for controlling the gameplay elements' frequency of appearance is proposed: Let q_e be the number of appearances for the gameplay element e in a genome. We define r to be the total number of distribution errors in a genome.

$$r = \sum_{e \in E} |q_e - N \cdot p_e| \tag{8}$$

Instead of setting the genome's total fitness to the sum of local gene fitness values, it becomes a function of both the sum of local fitness values and the total number of distribution errors, as follows.

$$f(\boldsymbol{z}, \boldsymbol{g}) = c_d\left(\frac{r}{N}\right) \cdot \sum_{i=1}^{N} f_{z_i}(g_i) \tag{9}$$

$c_d : \Re \rightarrow [0, 1]$ is defined so that $c_d(0) = 1$, meaning that if the total number of distribution errors is zero, then the global fitness equals to the sum of local fitness values. Else, the more errors, the more the total fitness is decreased.

5.4 Level refinement

Obviously, the level produced by the selection process cannot be perfect as is. In the end, it is expected that the placement of some gameplay elements will be less than ideal. Therefore, the level goes through a refinement process before it is presented to

the player. The specifics of this process are dependent on the gameplay elements chosen. As a general principle, the least that this process must do is, remove any punishment-type gameplay elements that are too close to the guideline. This can be achieved by traversing the level using a small window around the guideline. Any punishment-type gameplay elements inside that window are removed.

6 IMPLEMENTATION ISSUES

6.1 Gameplay

To demonstrate the methodology presented, a new one-button platform game was developed in *Java* [22]. The *JGame* library [23] was used to render the graphics while the *JLayer* library [24] was used to play the music.

The gameplay features the player-controlled character running forward with constant velocity. The player can use the jump action to evade contact with some damaging gameplay elements and to collect beneficial gameplay elements. In order to allow as much flexibility as possible for the jump action, it is not only made available to the player when the character is touching the ground, but also when he is in mid-air, as can be seen clearly in Figure 2.

To avoid trivializing the challenge of the game, the *energy* variable was introduced. It is decreased when the jump action is used and increased when the character is falling. The rate by which *energy* is expended is greater than the rate it is replenished, which has the effect of reducing the jumps' potency over time, until it is reset when the character returns to the ground. When the *energy* is reduced to zero and the jump action is still active, the character 'floats', meaning he falls with constant velocity instead of accelerating like in free fall. That allows for particularly long musical notes to be represented in the guideline.

A number of gameplay elements are used to motivate the player to follow the guideline. *Bullets*, β, are placed above the guideline and *blocks*, λ_k, of various shapes beneath it. *Holes*, ϖ, are placed when the character should be jumping high. All of these reduce the character's velocity temporarily, putting him out-of-sync with the music. By collecting *greens*, γ, the character accelerates to catch up and restore the game's balance. All in

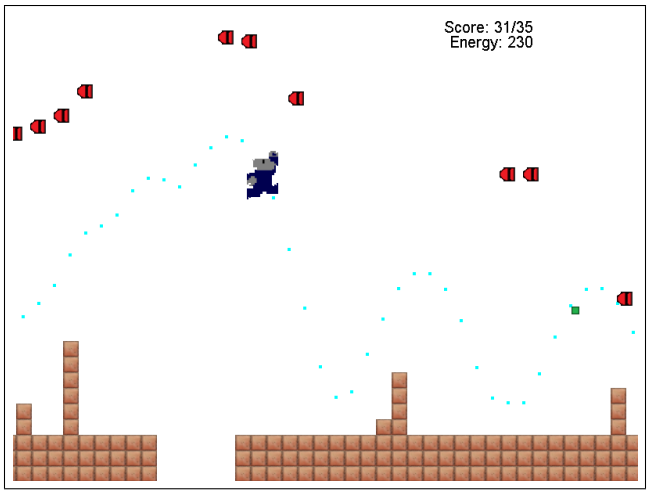

Fig. 4. A screenshot from the game, showcasing the various gameplay elements.

all, the set of gameplay elements we used is defined as $E = \{\emptyset, \gamma, \beta, \varpi, \lambda_1, \lambda_2, \lambda_3, \lambda_4\}$. Figure 4 is a screenshot from the game containing at least one instance of all of the above-mentioned gameplay elements.

6.2 Guideline generation

The training dataset was produced using the first 60 seconds from 10 different songs. Afterwards, the guidelines and equivalent button presses were recorded in-game manually. Since the character velocity and the game's frame rate are known quantities, it is easy to calculate the moment in time that each point in the guideline corresponds to. Likewise, using the sampling rate for the music, the equivalent moments in time for the waveform data are calculated. For each point in the guideline, a half-second window is taken from the waveform data centered at that point. The window is segmented into three sub-windows. Each of these windows is separated into 10 frequency bands, using the FFT [47]. The previous procedures have been implemented using *MATLAB* [28] and *mp3read* [27].

Finally, *LibSVM* [25] is used to train a SVM for these inputs and outputs. In a typical experiment, the linear kernel is used with 10 or 20 different values for the soft margin parameter C. For each of these values a SVM is trained and tested using 10-fold cross validation. Instead of grouping individual input-output pairs into folds, we choose to

create folds based on the songs the samples originated from. So, since in this instance we have the same number of folds and songs, each fold contains all the samples from a single song.

After the SVM is trained, it can be used to produce a button presses sequence for a new song, which can be transformed to an equivalent guideline.

6.3 Gameplay element placement

Level generation is initiated in-game, when a level file is not found. A population of 500 genomes is produced, each gene representing a gameplay element type $z_i \in E$. First, the local fitness of each gene is calculated separately. Each element has a different local fitness function, as illustrated in Figure 5 and shown below, where $g_y(i) \in [0, 1]$ denotes the y-axis part of $\boldsymbol{g}_i$:

$$
\begin{aligned}
&f_{\emptyset}(g_i) = 0 \\
&f_{\gamma}(g_i) = f_{\beta}(g_i) = (10 \cdot g_y(i))^2 \\
&f_{\varpi}(g_i) = \begin{cases} 150 \cdot e^{-\frac{(g_y(i)-1)^2}{0.35^2}} & \text{if } z_{i-1} = \varpi \\ 100 \cdot e^{-\frac{(g_y(i)-1)^2}{0.35^2}} & \text{else} \end{cases} \\
&f_{\lambda}(g_i) = 100 \cdot e^{-\frac{(g_y(i)-0.4)^2}{0.1^2}}
\end{aligned}
$$

The initial probability of appearance p_e for each gameplay element is:

$p_{\emptyset} = 43\%$	$p_{\gamma} = 10\%$	$p_{\beta} = 22\%$	$p_{\varpi} = 15\%$
$p_{\lambda_1} = 3\%$	$p_{\lambda_2} = 2\%$	$p_{\lambda_3} = 2\%$	$p_{\lambda_4} = 3\%$

To conserve this distribution, the global fitness function takes into account how much the frequency appearance of each element in a genome deviates from the above probabilities, as shown previously in (9). The analytical form of the c_d function in (9) that was used in the implementation is

$$c_d(s) = e^{-\frac{s^2}{0.2^2}} \tag{10}$$

It is illustrated in figure Figure 6.

Using the JGAP library [29] and its default configuration, this population is allowed to evolve through 100 epochs. Experimentation has shown that by then the change in level fitness becomes negligible, as evident in Figure 7.

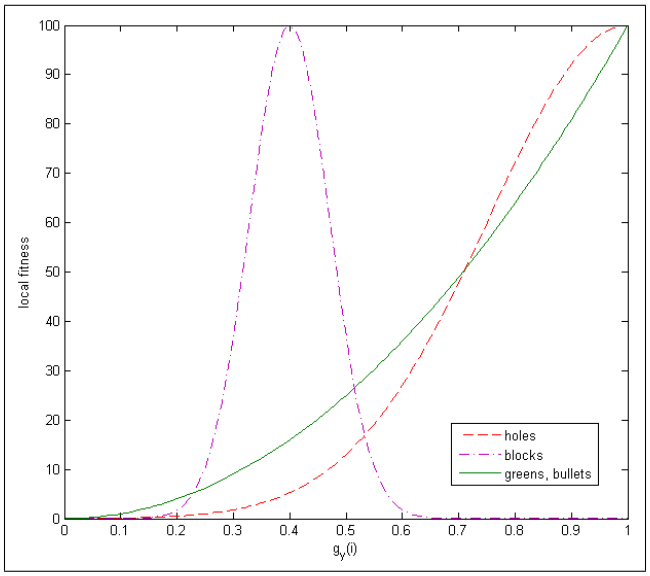

Fig. 5. The local fitness functions of the gameplay elements.

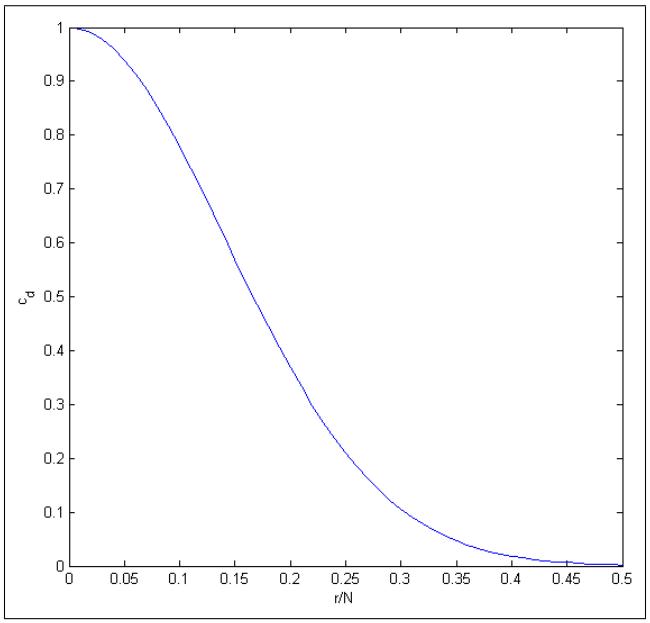

Fig. 6. The c_d function, which controls how much global fitness is decreased depending on the number of distribution errors.

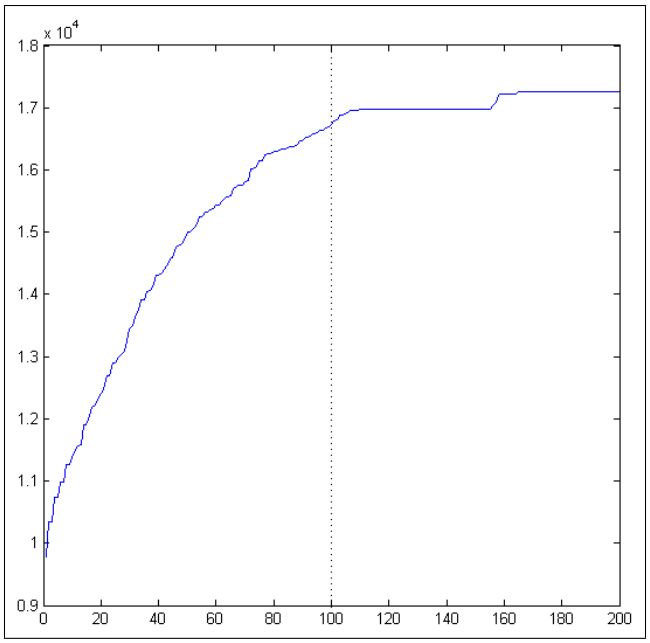

Fig. 7. The convergence curve for global fitness during the course of one execution of the Genetic Algorithm.

The level with the highest fitness from the final epoch is chosen as the best level. As mentioned in Section 5.2, the position of the gameplay elements on the x-axis is equal to that of the corresponding point in the guideline. Their position on the y-axis is chosen deterministically, according to gameplay element type: *Greens* are rewards, so they are placed on the guideline. *Bullets* are placed above the guideline. *Blocks* are allowed to fall down and stack on each other, like in *Tetris* [46].

Finally, the level is refined to remove any problematic placements. Any *blocks* that are near the guideline are removed. leading to a substantial improvement in the quality of the game level.

7 Experimental results and discussion

We presented a methodology for generating music-based levels for a novel one-button platform game. The novelty of the proposed approach lies in the use of evolutionary and machine learning techniques for content generation. Content generation has been transformed to learning the human behavior of rhythmical button pressing using Sup-

port Vector Machines and to the optimization of a gameplay element placement fitness function that is achieved through the Genetic Algorithm.

It is worth mentioning at this point that the proposed methodology can be easily adapted to different input data or different gameplay mechanics or even genres. For example, music could be replaced with an audio narration or written text of a story. Feature extraction would include voice recognition first if the data is audio, followed by a syntactic and semantic analysis of the story to recognize high-level ideas, like pacing and style. Then the audio would be played or a text-to-speech system would be used to accompany gameplay.

On the gameplay mechanics and genre front, possibilities are essentially endless. Adding more buttons to the control scheme only requires increasing the dimension of the button presses sequence. Changing the game mechanics would only require a different way to calculate the guideline from button presses.

Additionally, certain high-level aspects of the level generation process are open to configuration. For example, the game difficulty could be controlled by automatically adapting the local fitness functions of the gameplay elements, or their distribution, or even the deterministic algorithms are are used to calculate their position on the y-axis.

Our ongoing research includes the subjective evaluation of game levels. Test subjects are asked to download a version of the game along with a level, e.g., http://users.auth.gr/~ntzimoul/rmj/game+level.zip. After playing the level, they are instructed to answer some questions regarding their experience. A sample survey can be seen at http://users.auth.gr/~ntzimoul/rmj/survey. Initial results seem to indicate that levels generated with the Genetic Algorithm are less frustrating compared to levels generated with a simple algorithm that decides where to place elements using hard thresholds.

In conclusion, the results are very promising and provide a new perspective on procedural content generation using computational intelligence techniques.

REFERENCES

[1] Michael Toy, Glenn Wichman and Ken Arnold, *Rogue*, 1980.

[2] Glenn Wichman, *A Brief History of "Rogue"*, 1997, http://www.wichman.org/roguehistory.html.

[3] Tarn Adams, Zach Adams, *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*, August 2006, http://www.bay12games.com/dwarves/features.html.

[4] Marshall G. Jones, *Learning to Play; Playing to Learn: Lessons learned from computer games*, Annual Conference of the Association for Educational Communications and Technology, Albuquerque, New Mexico, February 1997.

[5] Jason Mitchell, Moby Francke and Dhabih Eng, *Illustrative Rendering in Team Fortress 2*, International Symposium on Non-Photorealistic Animation and Rendering, 2007.

[6] Nintendo, *Super Mario Bros.*, September 1985.

[7] Black Isle Studios, *Fallout 2*, September 1998.

[8] Amanda Greenslade, *Gamespeak: A glossary of Gaming Terms*, The Specusphere, June 2006, http://www.specusphere.com/joomla/index.php?option=com_content&task=view&id=232&Itemid=32#platformgame.

[9] Semi Secret Software, *Canabalt*, 2009, http://adamatomic.com/canabalt/.

[10] Ninjadoodle, *One Button Bob*, 2010, http://www.ninjadoodle.com/one-button-bob/.

[11] Berbank Green, *One Button Games*, Gamasutra, June 2005, http://www.gamasutra.com/view/feature/2316/one_button_games.php.

[12] Konami, *Dance Dance Revolution*, November 1998, http://www.konami.com/games/ddr/.

[13] Harmonix, *Guitar Hero*, November 2005, http://guitarhero.com.

[14] Dylan Fitterer, *Audiosurf*, February 2008, http://www.audio-surf.com.

[15] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B. Sandler, *A Tutorial on Onset Detection in Music Signals*, IEEE Transactions on Speech and Audio Processing, volume 13, issue 5, pages 1035-1047, 2005.

[16] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin, *A Practical Guide to Support Vector Classification*, Technical report, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, July 2003.

[17] Introversion Software, *Procedural Content Generation*, Game Career Guide, June 2007, http://www.gamecareerguide.com/features/336/procedural_content_.php.

[18] David Braben and Ian Bell, *Elite*, 1984.

[19] Matt Barton and Bill Loguidice, *The History of Elite: Space, the Endless Frontier*, Gamasutra, April 2009, http://www.gamasutra.com/view/feature/3983/the_history_of_elite_space_the_.php.

[20] Gillian Smith, Mike Treanor, Jim Whitehead and Michael Mateas, *Rhythm-Based Level Generation for 2D Platformers*, International Conference On The Foundations Of Digital Games, 2009.

[21] Kieron Gillen and Dylan Fitterer, *New Wave: Dylan Fitterer on Audiosurf*, Rock Paper Shotgun, March 2008, http://www.rockpapershotgun.com/2008/03/21/new-wave-dylan-fitterer-on-audiosurf/.

[22] James Gosling and Sun Microsystems, *Java*, 1995.

[23] Boris van Schooten, *JGame - a Java game engine for 2D games*, 2006, http://www.13thmonkey.org/~boris/jgame/.

[24] Matthew McGowan, Matthias Pfisterer, Michael Scheerer, Daniel Szabo, Micah Spears and Paul Santon, *JLayer - MP3 library*, 1999, http://www.javazoom.net/javalayer/javalayer.html.

[25] Chih-Chung Chang and Chih-Jen Lin, *LIBSVM : a library for support vector machines*, 2001, http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[26] Corinna Cortes and Vladimir Vapnik, *Support-Vector Networks*, Machine Learning, volume 20, issue 3, pages 273-297, 1995.

[27] Dan Ellis, *mp3read and mp3write for Matlab*, http://labrosa.ee.columbia.edu/matlab/mp3read.html.

[28] MathWorks, *MATLAB R2009b*, 2009, http://www.mathworks.com/products/matlab/.

[29] Klaus Meffert, Javier Meseguer, Enrique D. Marti, Audrius Meskauskas, Jerry Vos and Neil Rotstan, *JGAP - Java Genetic Algorithms Package*, 2007, http://jgap.sourceforge.net.

[30] Michael Nitsche, Calvin Ashmore, Will Hankinson, Robert Fitzpatrick, John Kelly and Kurt Margenau, *Designing Procedural Game Spaces: A Case Study*, FuturePlay, October 2006.

[31] Bethesda Softworks, *The Elder Scrolls II: Daggerfall*, August 1996.

[32] Stefan Greuter, Jeremy Parker, Nigel Stewart and Geoff Leach, *Real-time procedural generation of 'pseudo infinite' cities*, Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia, pages 87-95, February 2003.

[33] Allegorithmic, *Substance Air*, 2003, http://www.allegorithmic.com/?PAGE=PRODUCTS.air.

[34] Interactive Data Visualization, *SpeedTree*, 2002, http://www.speedtree.com/.

[35] Gearbox Software, *Borderlands*, October 2009, http://www.borderlandsthegame.com.

[36] Will Wright and Maxis, *Spore*, September 2008, http://www.spore.com.

[37] Michael Booth, *The AI Systems of Left 4 Dead*, Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford, 2009.

[38] andrewdoull, droid and Richard Tew, *Algorithms for Procedural Content Generation*, Procedural Content Generation Wiki, 2008, http://pcg.wikidot.com/category-pcg-algorithms.

[39] Christophe de Dinechin and Infogrames, *Alpha Waves*, 1990.

[40] I.Buciu, C.Kotropoulos and I.Pitas, *Demonstrating the stability of support vector machines for classification*, Signal Processing Elsevier, vol. 86, no. 9, pages 2364-2380, 2006.

[41] J.J. Moré and S.J. Wright, *Optimization Software Guide*, SIAM Press, pages 35-38, 1993.

[42] M. Aizerman, E. Braverman and L. Rozonoer, *Theoretical foundations of the potential function method in pattern recognition learning*, Automation and Remote Control, volume 25, pages 821837, 1964.

[43] Christopher J. C. Burges, *A Tutorial on Support Vector Machines for Pattern Recognition*, Data Mining and Knowledge Discovery, volume 2, pages 121167, 1998.

[44] Darrell Whitley, *A genetic algorithm tutorial*, Statistics and Computing, Volume 4, pages 65-85, June 1994.

[45] Harris Drucker, Chris J.C. Burges, Linda Kaufman, Alex Smola and Vladimir Vapnik, *Support Vector Regression Machines*, Advances in Neural Information Processing Systems, volume 9, pages 155-161, MIT Press, 1997.

[46] Alexey Pajitov, Tetris, 1984, http://www.tetris.com.

[47] Alan V. Oppenheim, R.W. Schafer and J.R. Buck, *Discrete-time signal processing*, 3rd Edition, N.J.: Prentice Hall.

[48] Trevor Hastie, Robert Tibshirani and Jerome Friedman, *The Elements of Statistical Learning*, 2nd Edition, Springer, 2009.

[49] Noor Shaker, Georgios N. Yannakakis and Julian Togelius, *Towards Automatic Personalized Content Generation for Platform Games*, to be presented at AIIDE, 2010.