An Approach to Level Design Using Procedural Content Generation and Difficulty Curves

Diaz-Furlong Hector Adrian[1,2]
Centro de Innovacion y Diseño Digital
[1]Benemerita Universidad Autonoma de Puebla
Puebla, Mexico
diazfurlong@gmail.com

Solis-Gonzalez Cosio Ana Luisa[2]
Facultad de Ciencias
[2]Universidad Nacional Autonoma de Mexico
Distrito Federal, Mexico
alsgc@fciencias.unam.mx

Abstract—**Level design is an art which consists of creating the combination of challenge, competition, and interaction that players call fun and involves a careful and deliberate development of the game space. When working with procedural content generation, it is necessary to review how the game designer sets the change in difficulty throughout the different levels. In this paper we present a procedural level generator that can be used for different games and is based on a genetic algorithm. We define a fitness function that does not depend on the game or content type. This function calculates the difference between the difficulty curve defined by the designer and the difficulty curve calculated for the candidate content, so the best content is the one whose difficulty curve best fits the desired curve. To design our generator, we rely on the concept of flow, theories of fun and game design.**

Keywords—procedural content generation; games; genetic algorithms; difficulty; game design

I. Introduction

Procedural content generation (PCG) promises to be a powerful tool for independent game developers, since they don't have a big budget and a large team of designers, artists and programmers, the automatic creation of content can make them more competitive. Even established game companies can benefit from PCG, using it to generate 3D worlds, missions and other types of content. However, the first problem facing a game designer wanting to incorporate PCG techniques is the loss of control over the generated content. One of the main arguments against procedural content generation by the representatives of the gaming industry, at least when discussing online content generation, difficulty scaling [1] and artificial intelligence adaptation [2], is the lack of reliability. Due to the manner in which most commercial games are designed, presenting content that is unplayable to the player is simply unacceptable [3].

Procedural generation tools usually require many tests and corrections to achieve the desired outcomes. As a result, the costs of developing a game that uses procedural generation can get out of control [4]. Furthermore, many of the approaches and PCG algorithms found in the literature are tailored to a particular game or genre, which makes the selection and adaptation of an algorithm for a different game difficult. Even among similar algorithms, the inputs are dependent on the particular game being programmed (e.g., the number and distribution of gaps in Super Mario Bros [5]), thus, an algorithm that has been successfully used to generate content for a game, may not work for another if the input data are not chosen well or the necessary modifications are not made.

Furthermore, the discipline of game design requires careful consideration of spatial relationships. The regions of the game space are commonly differentiated as levels, which are created by a particular facet of the production called level design. The level design is an art which consists of creating the combination of challenge, competition, and interaction that players call fun and involves a careful and deliberate development of the game space, which includes the manual placement of appropriate elements within the game space. A level created in this way can be considered as a static level since the player experience will be consistent every time he or she plays in it [4].

When working with PCG, it is necessary to review how the game designer sets the change in difficulty throughout the levels. Consider the case of a platformer game where the goal is to reach the end of the level by jumping gaps. Increasing the width of the gaps or placing several ones close to one another can increase the difficulty of the level. Because the designer has a direct contact with the arrangement of the gaps and can move them as desired, this becomes a spatial problem. However, if the same game contains procedural generated levels, the designer suddenly loses control over the space. When the level is generated at run time, there is no opportunity to judge a particular part of the level directly and modify it if necessary [4].

Two major problems arise when using PCG techniques to create the levels of a videogame: The levels must be correct, i.e., they can't prevent the player's progress [3], and they must offer a fun experience for the player. Designing a content generator that produces correct and fun levels is a great challenge to overcome for programmers and game designers.

In this paper we present a procedural level generator that can be used for different games and is based on a genetic algorithm. We define a fitness function that does not depend on the game or content type. This function calculates the difference between the difficulty curve defined by the designer and the difficulty curve calculated for the candidate content, so

978-1-4673-5311-3/13/$31.00 ©2013 IEEE

the best content is the one whose difficulty curve best fits the desired curve.

II. Related work

Smelik et al. [6] designed a system to create 3D worlds that incorporates procedural generation techniques. The system allows the designer to draw terrain features on a 2D surface, using different brushes to set the elevation, soil type and placing items such as rivers, roads and cities. Once the designer finishes placing terrain features on a large scale, a procedural generator produces a virtual world trying to preserve the characteristics defined by the user. However, the purpose of this system is to model 3D worlds, which can be used in a game or in some other application, so the generated terrains do not necessarily function as a level and therefore they are not required to be a factor that influences a gameplay experience.

Kerssemakers et al. [7] designed a procedural generator of procedural generators. In their system, an evolutionary algorithm evolves level generators for a platformer game based on agents; these agents draw and erase blocks by walking on the canvas of the level. In each generation of the algorithm, the system interface allows the user to see the kind of levels that each generator produces and select the ones he likes to be the parents of the next generation, or select his favorite and use that generator to create the levels.

Sorenson et al. [8] propose a framework for automatic game level creation, using a Feasible-Infeasible Two-Population (FI-2Pop) genetic algorithm. Level designers specify a set of constraints, which determine the basic requirements for a level to be considered playable. The “infeasible population” consists solely of levels which do not yet satisfy all these constraints, and these individuals are evolved towards minimizing the number of constraints violated. When individuals are found to satisfy all the constraints, they are moved to the “feasible population” where they are subjected to a fitness function that rewards levels based on any criteria specified by the level designers. Essentially, this population is for generating levels that are not only playable, but also fun.

Togelius et al. [9] used a multiobjective evolutionary algorithm to generate terrains for strategy games, defining various objectives for the algorithm motivated by player experience considerations. Their experiments show that there exist a number of partial conflicts between these objectives, which makes the algorithm useful as a design support tool, allowing the designers to visualize the necessary tradeoffs as a Pareto front and choose maps making an informed aesthetic judgment.

Shaker et al. [10] introduced an approach for automatic level design and adaptation using grammatical evolution. They defined a design grammar to represent the underlying structure of the levels in Super Mario Bros and performed evolution to select the best individuals, based on their potential to optimize a particular affective state for a specific playing style. To this end, they used models that map game content to reported player experience as fitness functions.

III. Tools for the Level Designer

The importance of providing the level designer with editing tools to influence the procedural generation process arises from the assumption that his experience (both as a designer and player) gives him an intuitive sense of what makes a level fun and what not. So making his intuition impact the generation process may result in better designed levels. However, although the design of characteristics of the level and its elements via editing tools gives the designer more control, it puts him in the same role that he has when not using procedural generation, i.e., manually placing the elements of the level and their distribution.

In this paper, we propose to give the designer an editing tool, not of the particular characteristics of the level, but of the level of difficulty it will have. This way, the designer may indirectly affect the level design and in turn, the experience he wants to offer to the player. In order to design this tool, we rely on the concept of Flow of Mihaly Csikszentmihalyi [11], theories of fun [12] and game design [13].

Using difficulty as a parameter for the generation algorithm is not entirely new. There are researchers who seek to adapt the difficulty of the game according to the player's skill level [1], [2], [14], [15], [16], [17] and still others have proposed fitness functions for genetic algorithms, based in the difficulty level as part of a general methodology for creating levels [8]. Also, there are games like Cloudberry Kingdom whose levels were generated by an algorithm whose governing parameter is the difficulty level [18]. However, if the difficulty of a level is set only as a numeric value, the designer of the game will have no control over the dramatic arc of the level.

In most games we find that each level has its own dramatic arc, that is, the difficulty is not constant throughout the level. In platformer games we can find more enemies in certain parts of the level or even a boss at the end. In games such as tetris it may come a time when the speed of the falling pieces increases and the conflict grows if the player does not remove the stack of pieces that has accumulated. Therefore, the designer needs to establish not only a numeric value for the difficulty but a difficulty curve.

A difficulty curve is a graphical representation of how the difficulty changes during the game. There are two main types: time-based and distance-based. The first one places the difficulty spikes according to time played, the latter places difficulty spikes depending on where the challenges are. Both have their advantages and some work better depending on the game being designed. For games like Asteroids and Geometry Wars, time-based curves are the best option. Platformer games, generally, have a path from beginning to end, so distance-based curves function very well [19].

To avoid losing the generality of our methodology, we will work with difficulty curves that are not based on distance or time, but on a variable x defined in the [0, 1] interval. Thus, we can use this curve regardless of the game being designed, we just need to map the time/distance interval (or the parameter that determines the difficulty of the game in particular) to this interval. Similarly, the difficulty value will be defined in the [0,1] interval and can be mapped to the space that best suits our game. Then, the difficulty curve is a function $d : [0,1] \to [0,1]$.

We developed a program that allows the user to define points of the difficulty curve and then it interpolates them using splines. The curve is stored as three arrays: the first two correspond to the coordinates of the points defined by the user and the third stores the second derivatives of the cubic polynomials. With these three arrays we can calculate any point on the curve.

IV. Our Game: The Chronicles of Balam

In this work we use PCG techniques to generate levels of a new videogame called The Chronicles of Balam: The Lost Panda, which was designed and developed by the authors. The main objective of our game is to carry a baby panda to his kingdom. To achieve this, the player must travel through different levels, each of which is constructed as a spiral track with tubular form. The player must slide within the interior of the tube that forms the track, avoiding obstacles, pitfalls and collecting items. The goal in each level is to reach the end of the runway. Along the way, the player can find fruits that restore a nutrition shield that protects him from diseases; he can also collect empty soda cans which can be recycled for a special power and there are mosquitoes that can harm the player and infect him with malaria, which can be cured using an Artemisia leaf. Fig. 1 shows a screenshot of the game.

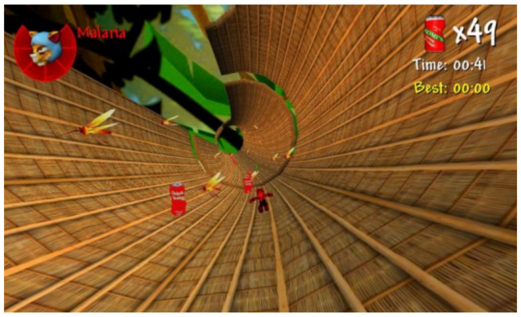

Fig. 1. The objective of our game is to carry a baby panda to the sanctuary. To do this the player must slide through tubular tracks, collecting items, avoiding obstacles and jumping holes in the ground. The Chronicles of Balam: The Lost Panda is a game designed and developed by the authors.

V. Content Representation

In general, procedural generation algorithms generate content from a more compact representation, either from a data structure, grammar, or artificial genotype. Defining the content's representation is not an easy task and usually depends on the type of content and the generation algorithm [3].

In this work, we use the Vasconcelos genetic algorithm proposed by Kuri-Morales [20] to generate the levels of our game. Each level is represented as a binary coded artificial genotype, which allows us to define a general methodology and relegate special considerations (inherent to the particular game being designed) to the genotype to phenotype mapping.

In this section, we present the first approach we took to define the content's representation, namely, choosing the content's representation from the construction of the game prototype; allowing the designer to focus on the gameplay mechanics that involve the generated content. However, as will be discussed in section VII, this representation proved to be ineffective due to the way we built our prototype, so we had to define a second representation, which will be discussed after introducing the generation process. Nevertheless, we deemed appropriate to present the first approach we followed to better convey the lessons we learned and the importance of choosing the right representation.

The creation of a prototype is an essential part of game design. A prototype is a functional model of the game that allows you to test its feasibility and improve it without investing a lot of time and money. Prototypes are sketches whose purpose is to allow the designer to focus on a small set of the gameplay mechanics and see how they work. When developing a prototype, all the designer needs to worry about are the fundamental mechanics of the game, if they can keep the interest of the players then the design is solid. There are many types of prototypes, including physical prototypes, visual, video, software, etc. A project may require several prototypes, each addressing a particular part of the game [13].

The first thing we did to develop a physical prototype of our game, was to divide the level according to the actions the player can carry out. Each level consists of a coiled hollow tube within which the character automatically slides forward; therefore, the first level division consisted in making slices of the tube as shown in Fig. 2 subsection (a). The player can move right and left around the circumference of the tube, to represent this gameplay mechanic, we rolled out each slice of the tube like a strip of paper and divide it into 16 tiles (subsections (b) and (c)), so that each tile represents a valid position of the character.

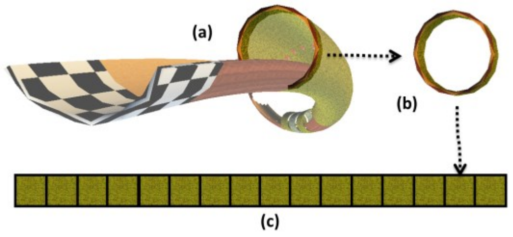

Fig. 2. Level division according to the player's actions. a) The track is divided into slices corresponding to forward motion of the player. b) Each slice is rolled out like a paper strip whose beginning and end are connected. c) The strip is divided into 16 tiles corresponding to the movement that the player can do sideways.

Using this discretization of the level, we build a physical prototype with paper cards, placed in 100 rows of 16 cards each. The cards have different pictures representing the various obstacles and resources that the player can find, namely, apples, peaches, watermelons, mosquitoes, leafs, holes, and floor tiles. Therefore, we have 8 different cards which can be represented by 3 bits as shown in Fig. 3. Each level segment consists of 16 cards and is represented by $16 \times 3 = 48$ bits. If there are 100 segments, the level is represented by $48 \times 100 =$ 4, 800 bits. Then, the output of our generation algorithm is a string of 4,800 bits representing a level of the game.

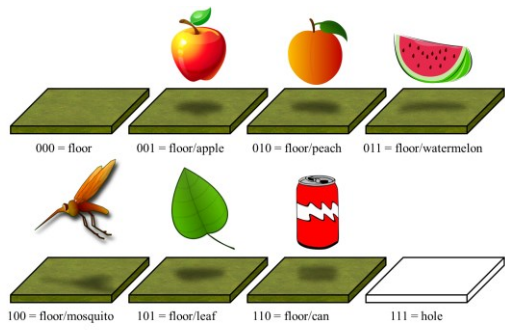

Fig. 3. Binary encoding of the elements which can be found in a level of the game.

VI. The Level Generation Process

Fig. 4 shows a schematic of the content generation process. The designer defines the desired difficulty curve and the content representation (which depends on the game and particular type of content). A genetic algorithm generates a population of candidate content at random. Each candidate goes through a difficulty curve calculator; the resulting curve is compared to the curve defined by the designer using the following equation,

$$f(c) = 1 - \sum_{i=0}^{n-1}\left(d\left(\frac{i}{n-1}\right) - d_m(c,i)\right)^2 \qquad (1)$$

Equation (1) gives the fitness value for the candidate content c, where n is the number of sampled points of the difficulty curves (either in space or time, in our case the segments of the track), $d(i/(n-1))$ is the difficulty of segment i given by the difficulty curve created by the designer and $d_m(c,i)$ is the difficulty of segment i of content c given by a player experience model, m. Then, the fitness value is equals to 1 minus the mean square error between the calculated difficulty curve and the curve defined by the designer. That is, the genetic algorithm searches for the content whose difficulty curve best fits the curve created by the designer.

A. Calculating the Difficulty Curve

Our fitness function requires that we can calculate the difficulty curve associated with the candidate content. Therefore, it is important to have a player experience model. Togelius et al [21] describe different types of player experience models, which we believe can be used to calculate the difficulty curve of a level. For example, an artificial agent that is trained to play racing games can be used to plot a difficulty curve of the track (e.g., measuring the speed of the car, the number of crashes, the number of times it had to decelerate, etc.). These player experience models are outside of the scope of this paper, but we must emphasize that the player experience model can be as complex and sophisticated as desired.

In this work, we decided to address procedural content generation from the viewpoint of the game designer. Therefore, the player experience model we used is based in game design theories [13]. From the formal and dramatic elements that form the game, the designer can determine what makes it difficult, i.e., determine which elements create the game conflict.

In our game, the difficulty level is affected by the shape of the track and the number of obstacles. Regarding the shape of the track, we know that the number of floor tiles in each segment may vary, from 0 up to 16. Therefore, we determined that a difficulty factor is the number of floor tiles that the segment has. We define the first difficulty factor of segment i of content c as:

$$d_1(c,i) = 1 - \frac{numberOfFloorTiles(c,i)}{16} \qquad (2)$$

If there are zero floor tiles in segment i, then its difficulty is 1 (very hard). If there are 16 floor tiles (the maximum value) then the difficulty is 0 (very easy).

Along the track there can be obstacles that make the segments more difficult. We define the second difficulty factor of segment i for content c as:

$$d_2(c,i) = \begin{cases} 1, & \text{if } numberOfFloorTiles(c,i) = 0 \\ \dfrac{numberOfObstacles(c,i)}{numberOfFloorTiles(c,i)}, & \text{otherwise} \end{cases} \qquad (3)$$

So that, if the number of floor tiles equals zero then the difficulty is 1 (very difficult), if the number of floor tiles is greater than zero, we divide the number of obstacles in the segment between the number of floor tiles. If the number of obstacles is zero the difficulty is 0 (very easy), if the number of obstacles equals the number of floor tiles then the difficulty is 1 (very hard).

Then, we define the difficulty of segment i as a weighted sum of the two difficulty factors defined in (2) and (3):

$$d_m(c,i) = w_1 d_1(c,i) + w_2 d_2(c,i) \qquad (4)$$

The subscript m only indicates that the difficulty is defined by a player experience model, distinguishing it from the difficulty curve created by the designer, $d(x)$. The weights w_1 and w_2 are defined in the program code, in our tests we assigned the same weight to both factors, that is, $w_1 = 0.5$ and $w_2 = 0.5$.

B. Building the Level

To build the geometry of the level from the genotype that represents it, we used a function of a spiral in three dimensions, defining the coordinates of the center of each segment of the track as:

$$x = \alpha \sin(\delta \times i) \qquad (5)$$

$$y = \alpha \cos(\delta \times i) \qquad (6)$$

$$z = \beta \times \delta \times i \qquad (7)$$

Where i is the segment number, n the number of segments of the level, $\delta = 2\pi / n$, $\alpha = n / 2$ and $\beta = -150$. The value of β was assigned through experimentation, so that the size of the track was adequate to the size of the player's character.

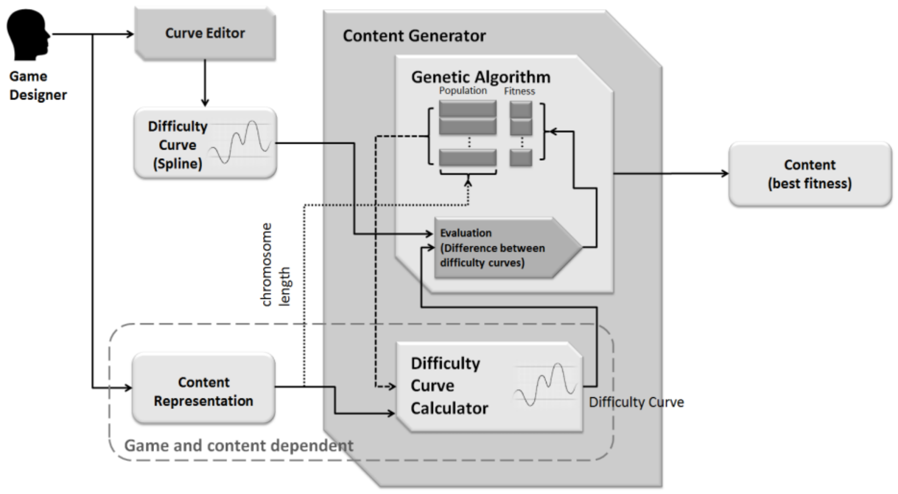

Fig. 4. Schematic of the content generation process. The game designer defines the content's representation and designs the difficulty curve that he wants the content to have. The representation defines the length of the chromosome for each individual in the population. These are processed by the difficulty curve calculator providing the difficulty curve for each individual. This curve is compared to the curve created by the designer and calculates the fitness value of the chromosome. Finally, the chromosome with the best fitness is output from the generator.

To build each segment of the track, we have to define the vertices that make up the walls of the segment. Each tile of the segment will have associated a pair of vertices as shown in Fig. 5. The vertex v_{i1} exists if any of the tiles t_{i-1} or t_i of segment s_j or s_{j-1} is a floor tile. Similarly, the vertex v_{i2} exists if any of the tiles t_{i-1} or t_i of segment s_j or s_{j+1} is a floor tile.

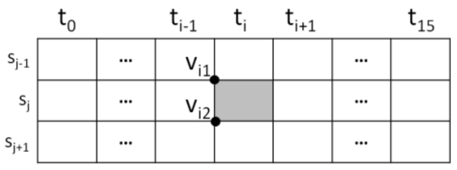

Fig. 5. Pair of vertices associated to tile i of segment j. The vertex v_{i1} exists if any of the tiles t_{i-1} or t_i of segment s_j or s_{j-1} is a floor tile. Similarly, the vertex v_{i2} exists if any of the tiles t_{i-1} or t_i of segment s_j or s_{j+1} is a floor tile.

The coordinates of existing vertices are calculated first as the coordinates of points on a circle in the XY plane:

$$v_x = r\cos(\Delta\theta \times t) \tag{8}$$

$$v_y = r\sin(\Delta\theta \times t) \tag{9}$$

$$v_z = 0 \tag{10}$$

Where $r = 10$ and corresponds to the radius of the tube that shapes the track, t is the number of the current tile. We apply two transformations to this vector. First, we rotate the vertices so that the segment follows the spiral. To do this, we calculate a direction vector as:

$$\vec{d}_i = \vec{c}_{i+1} - \vec{c}_i \tag{11}$$

Where $\vec{c}_{i+1}$ and $\vec{c}_i$ are the points on the spiral defining the beginning and end of segment i (see Fig. 6), and their coordinates are calculated using (5), (6) and (7). This direction vector is normalized:

$$\vec{u}_i = \frac{1}{\left\|\vec{d}_i\right\|}\vec{d}_i \tag{12}$$

Then, we calculate the axis of rotation as:

$$\vec{e} = (0,0,1) \times \vec{u}_i \tag{13}$$

Here, × denotes the cross product. The angle of rotation is:

$$\phi = \sin^{-1}\left((0,0,1)\cdot\vec{u}_i\right) \tag{14}$$

Where • denotes the dot product. The coordinates of the vertices in the XY are rotated by ϕ radians around the axis $\vec{e}$. Then, they are translated to the beginning of the segment, c_i, or the end, c_{i+1}, depending on the vertex in question.

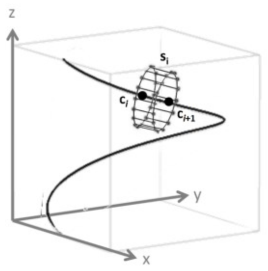

Fig. 6. Spiral curve that shapes the track. Each segment, s_i, is constructed from two points on the curve, c_{i1} and c_{i+1}, from which the vertices that form the walls of the segment are calculated.

Once the vertices are stored, we assign indices of vertices to form the triangles that make up the faces of each tile. Also, each vertex has associated a pair of texture coordinates that define the texture of the face, they are mapped onto a single image that is divided in several sections with different textures, so that the track may have different textures for each segment or group of segments.

The elements like the fruits, mosquitoes, etc., are models created by hand and they are loaded only once by the program. The position of each element is calculated in a very similar fashion as the position of the vertices of the segment. When drawn, the same model is used to draw the element at that position.

VII. Content Representation Revision

Fig. 7 shows a level created by our generator. Subsection (a) shows the level seen from afar and subsection (b) shows a portion of the level seen from the inside of the track. This level is representative, since all levels generated by the generator share two features that can be seen in Fig. 7. The first is that there are no empty segments in the level, i.e., no segments that the player has to jump. The second is that the inside of the track has a lot of elements (fruits, mosquitoes, etc.), which makes it look messy and obviously not very well designed.

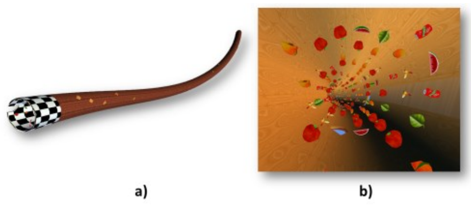

Fig. 7. A level generated by the genetic algorithm. a) Seen from a distance. The whole level is continuous and has no empty segments that the player must jump. b) View from inside the track. The interior has many elements that make it look messy and not well designed.

In section V we defined the content representation, using 3 bits to represent the elements that can be found in each tile, with a total of 8 different elements per tile. Each segment has 16 tiles; therefore, the total number of unique segments that can be generated with this representation is 8^{16} = 281, 474, 976, 710, 656. Furthermore, an empty segment is one that has 16 empty tiles, the genetic algorithm initial population is created randomly, but the probability of randomly creating an empty segment is $\frac{1}{8^{16}} = 3.55 \times 10^{-15} \approx 0$.

On the other hand, in order for the track to not be full of items and look messy, we want there to be segments which have only floor tiles or only two or three elements at best, but the probability of creating a segment with 16 floor tiles is the same as creating an empty segment, so there is a problem with our representation. The problem is that our representation includes combinations that result in levels that are not interesting, and others that cannot even be played, for example, a level with all empty segments cannot be played. Therefore, we need to find a representation that includes only interesting combinations.

In his theory of fun, Raph Koster [12] proposes that games are fun because they present us with new patterns, which we must recognize and master and, in doing so, our brain rewards us by releasing endorphins into our system. We therefore decided to define a representation of content that represents only the patterns that we believe may result in an interesting decision of the player.

Fig. 8 shows four of the 64 patterns that we identified as interesting. Each circle represents a segment level (a slice of the track) seen from the front, along with the elements that are present in that segment. For example, the first pattern is a segment with no elements; while the last is full of mosquitoes and one soda can, so that the player must take the risk to pick up the soda can which is surrounded by mosquitoes. While having only 64 patterns limit the levels that can be created by our generator, identifying patterns of interest gives the designer more control over the generated content. Another way to identify these patterns could be to define a grammar or an algorithm that generates patterns of interest.

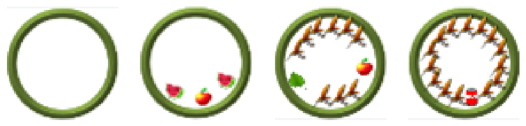

Fig. 8. Four of the 64 patterns of elements we defined as interesting.

Once we defined the patterns of interest, we can represent each segment as the index of the corresponding pattern. In our case, we need only 6 bits to represent this index, since we have 64 patterns. We also included in our representation some construction rules for the track. First, the level must have a start and a goal that cannot be composed of empty segments and also, as a matter of design, must consist of 3 segments with the same number of floor tiles. Then, we include 3 bits at the beginning of our genotype that define the number of floor tiles that have the first and last 3 segments of the track, 000 meaning three floor tiles, 001 five, 010 seven, 011 nine, 100 eleven, 101 thirteen, 110 fifteen and 111 sixteen (see Fig. 9).

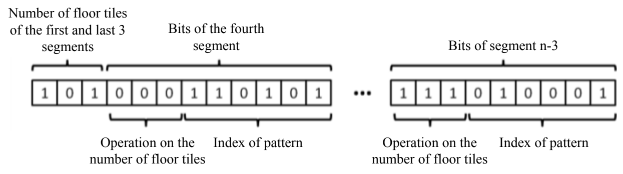
Number of floor tiles of the first and last 3 segments
Bits of the fourth segment
Bits of segment n-3
1 0 1 0 0 0 1 1 0 1 0 1
...
1 1 1 0 1 0 0 0 1
Operation on the number of floor tiles
Index of pattern
Operation on the number of floor tiles
Index of pattern

Fig. 9. Example of a genotype using the second representation, where *n* is the number of segments of the level.

Each segment has assigned 3 bits that represent an operation over the number of floor tiles from the previous segment: 000 means the number of floor tiles remains the same, 001 adds a floor tile to the right end and takes one from the left, 010 adds a floor tile to the left end and takes one from the right, 011 adds a floor tile to both ends, 100 takes one floor tile from both ends, 101 all tiles are floor, 110 the number of floor tiles is cut in half, 111 all tiles are holes (an empty segment). The genotype to phenotype mapping makes sure that these operations can be applied (it doesn't allow more than two subsequent empty segments).

VIII. Results

To test the performance of the generators we designed three different difficulty curves using the program described in section III. Fig. 10 shows the three curves. Curve A maintains a nearly constant difficulty level, staying around the normal difficulty level. Curve B has random ups and downs and curve C was designed with the player's experience in mind, in this curve the difficulty level starts easy and the difficulty increases gradually towards the middle of the level, with lower difficulty levels interspersed, the difficulty increases before reaching the goal.

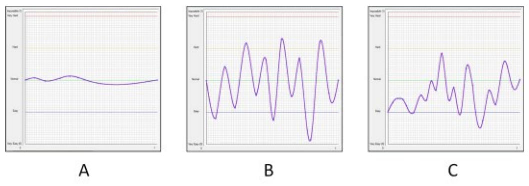

Fig. 10. The three difficulty curves used as input for the content generator.

We ran the genetic algorithm a hundred times for each curve and for both representations of the content: the representation described in section V (GG1) and the representation described in section VII (GG2). The population size was set to 50, the crossing probability to 0.9, the mutation probability to 0.005 and the number of generations to 500. In total, six hundred levels were generated.

Fig. 11 shows the difficulty curves of the levels with the best fitness values generated with the GG1 representation. As shown in the figure, these levels have difficulty values that do not go over 0.2 and the difficulty curve does not fit the desired curve. Recall that the calculation of the difficulty curve is based on the number of empty segments and the number of obstacles in each segment (see section VI.A). However, we have seen that the probability of creating holes and obstacles using GG1 representation is very low, so it is expected that we find more resources and floor tiles than obstacles and holes, this explains why the levels have a low difficulty and why their difficulty curves do not fit the desired curve.

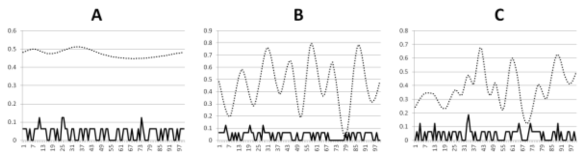

Fig. 11. Difficulty curves of the levels with the best fitness values generated using the GG1 representation. The dotted line is the desired difficulty curve and the solid line is the curve of the generated level.

Fig. 12 shows the difficulty curves of the levels with the best fitness values generated with the GG2 representation. It can be observed that the difficulty curve fits more closely the desired curve than the levels generated using the GG1 representation. It is noteworthy that the algorithm and the fitness function are the same in both cases, which shows the impact of changing the representation of content.

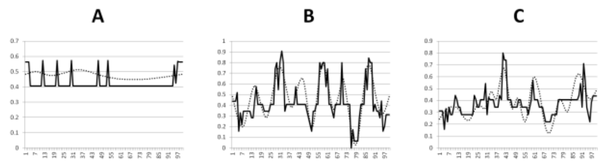

Fig. 12. Difficulty curves of the levels with the best fitness values generated using the GG2 representation. The dotted line is the desired difficulty curve and the solid line is the curve of the generated level.

To evaluate the quality of content, we conducted playtesting sessions with 22 players between 15 and 27 years old. For our playtesting sessions we selected 6 levels of the 600 generated, one level for each difficulty curve and content representation. At the start of the session, the player began with a tutorial level that explained the controls and game mechanics. Then, he played each of the six selected levels, answering a series of questions when completing each level. We asked the player to rate from 1 to 10 the difficulty level, the fun factor and level design. Fig. 13 shows the average values of the results from the playtesting sessions.

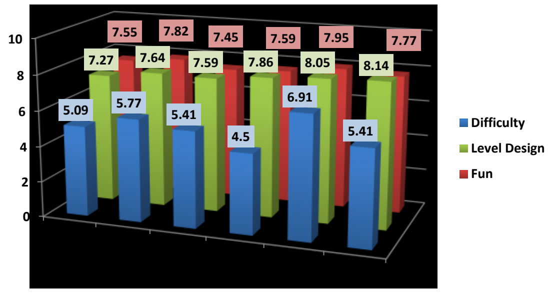

Fig. 13. Playtesting sessions results. 22 players rated from 1 to 10 the difficulty, fun and design of each level. The names of the levels indicate the difficulty curve and representation used, for example, level GG1_A was generated using curve A and the GG1 representation.

Table I shows the number of obstacles and empty segments of the levels played by the players. Level GG2_B is the one with more empty segments and according to the players is the most difficult. All the other levels, except GG2_C, have the same number of empty segments, but level GG1_B is the one with more obstacles and is the second most difficult. This leads us to believe that our definition of the difficulty factors (number of obstacles and empty segments) to calculate the difficulty curve is consistent with the difficulty perceived by the players. Furthermore, for both representations, the easiest levels were the ones generated with curve A, which has the lowest difficulty (the other two have higher difficulty spikes), and the hardest levels were the ones generated with curve B, which has the highest difficulty spikes.

TABLE I. Level Characteristics

Level	Obstacles	Empty segments
GG1_A	227	0
GG1_B	247	0
GG1_C	225	0
GG2_A	13	0
GG2_B	42	9
GG2_C	30	1

Regarding the level design, according to the players the worst designed levels were the ones created using curve A, whose difficulty level remains almost constant, while the ones with more variation were better rated. The best designed level was the one created with curve C and the GG2 representation. This is exactly what we expected because curve C was designed with the player's experience in mind and the GG2 representation uses patterns of interest to build the levels.

IX. CONCLUSIONS

In this paper we present a procedural level generator that can be used for different games and is based on a genetic algorithm. We define a fitness function that does not depend on the game or content type. This function calculates the difference between the difficulty curve defined by the designer and the difficulty curve calculated for the candidate content, so the best content is the one whose difficulty curve best fits the desired difficulty curve. We believe difficulty curves can be applied to a large number of games because they have the flexibility to represent distances or playtime. Even for games that do not have a linear progression, we believe that we can adapt a difficulty curve that does not represent distance but targets within the overall mission. The study of the structure of games (its formal and dramatic elements), leads us to believe that it is possible to tailor difficulty curves to the game we want to design.

Through the playtesting sessions we observed the following: The changes in the difficulty curves were perceived by players, they indicated that the levels generated by the curve with the highest peaks were the hardest, while the ones generated with the curve that maintained an intermediate level of difficulty were the easiest. The difficulty level made an impact on the experience of the players, indicating that the more difficult levels were the most fun. In particular, the graphs obtained with the GG2 representation, show a relationship between the fun factor and the difficulty level. With these results and based on the concept of flow of Mihaly Csikszentmihalyi, we believe that the control over the difficulty of the generated content allows the designer to influence the experience he wants to offer to the player.

Unlike other studies that use difficulty as a parameter for the generation algorithm, we do not define the difficulty as a scalar but as a curve. This allows the designer to affect the dramatic arc of the level. As expected, the results show that the variation in the difficulty curve affects the level design. For both representations of content, the worst designed levels were created with the curve showing little variation.

X. FUTURE WORK

Finally, we believe that with additional work, we can improve the manner in which the difficulty curve is calculated; in particular, the process could incorporate an objective player experience model, based on facial expressions recognition and machine learning. We also plan to use our generator to create levels of games like Super Mario Bros, to be able to compare our method with others that generate levels of the same game.

REFERENCES

[1] P. Spronck, I. Sprinkhuizen-Kuyper and E. Postma, "Dificulty scaling of game AI", Proc. Fifth Int'l Conf. Intelligent Games and Simulation, pp. 33-37, 2004.

[2] S. Bakkes, P. Spronck and J. van den Herik, "Rapid and Reliable Adaptation of Video Game AI", IEEE Transactions on Computational Intelligence and AI in Games, vol. 1, no. 2, 2009

[3] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne, "Search-based procedural generation: a taxonomy and survey" IEEE Transactions on Computational Intelligence and AI in Games, September 2011. *(references)*

[4] H. M. Decker-Davis, "Tunning dificulty when making procedurallly generated levels", [Online]. Available: http://www.gamecareerguide.com/features/1120/tunind_di¢culty_when_making_.php?, 2012.

[5] C. Pedersen, J. Togelius and G. N. Yannakakis, "Modeling player experience for content creation", IEEE Trans. Computational Intelligence and AI in Games, vol. 2, no 1, pp. 54-67, 2010

[6] R. Smelik, T . Tutenel, K. Jan de Kraker and R. Bidarra, "Integrating procedural generation and manual editing of virtual worlds", PCGames, June 2010.

[7] M . Kerssemakers, J. Tuxen, J. Togelius and G. N. Yannakakis, "A procedural procedural level generator generator», in IEEE Conference on Computational Intelligence and Games, 2012.

[8] N. Sorenson and P. Pasquier, "Towards a generic framework for automated video game level creation", Proc. European Conf. Applications of Evolutionary Computation, pp. 130-139, 2010.

[9] J. Togelius, M. Preuss and G. N. Yannakakis, "Towards multi-objective procedural map generation", in Proc. Workshop Procedural Content Generation, Foundations of Digital Games, 2010.

[10] N. Shaker, G. N. Yannakakis, J. Togelius, M. Nicolau and M. O'Neill, "Evolving Personalized Content for Super Mario Bros Using Grammatical Evolution", AAAI Conference on Artificial Intelligence and Interactive Digital Entretainment, 2012.

[11] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*, Harper Collins, 1990.

[12] R. Koster, *A Theory of Fun for Game Design*, Paraglyph Press, 2005.

[13] T. Fullerton, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, Morgan Kaufmann, 2008.

[14] H. Lida, N. Takeshita and J. Yoshimura, "A metric for entertainment of boardgames: its implication for evolution of chess variants", in Proc. Int'l Wireless Comm. Expo, pp. 65-72, 2003.

[15] J. K. Olesen, G. N. Yannakakis and J. Hallam, "Real-time challenge balance in an RTS game using rtNEAT", Proc. IEEE Symp. Computational Intelligence and Games, pp. 87-94, 2008.

[16] G. Lankveld, P. Spronck and M. Rauterberg, "Dificulty scaling through incongruity", Proc. Fourth Int'l Artificial Intelligence and Interactive Digital Entertainment Conf., pp. 228-229, 2008.

[17] G. Andrade, G. Ramalho, H. Santana and V. Corruble, "Extending reinforcement learning to provide dynamic game balancing", Proc. Workshop Reasoning, Representation, and Learning in Computer Games, 19th Int'l Joint Conf. Artificial Intelligence, pp. 7-12, 2005.

[18] J. Fisher, "How to make insane, procedural platformer levels", Gamasutra: The Art & Business of Making games [Online] Available: http://www.gamasutra.com/view/feature/170049/how_to_make_insane_procedural_.php, 2012.

[19] R. Vazquez, "How tough is your game? Creating dificulty graphs", Gamasutra: The Art & Business of Making games [Online] Available: http://www.gamasutra.com/view/feature/134917/how_tough_is_your_game_creating_.php, 2011.

[20] A. F. Kuri-Morales, "Solution of simultaneous non-linear equations using genetic algorithms", WSEAS Transactions on Systems, pp. 44-51, WSEAS Press, Issue 1, Vol 2, 2003.

[21] G. N. Yannakakis and J. Togelius, "Experience-Driven Procedural Content Generation", IEEE Transactions On Affective Computing, vol. 2, no 3, 2011