UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL GUERRA DE PONTES

EVOLUCIONARY PROCEDURAL CONTENT GENERATION FOR A ENDLESS PLATFORM GAME

CAMPINA GRANDE - PB

2020

RAFAEL GUERRA DE PONTES

EVOLUCIONARY PROCEDURAL CONTENT GENERATION FOR A ENDLESS PLATFORM GAME

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Herman Martins Gomes.

CAMPINA GRANDE - PB

2020

P814e Pontes, Rafael Guerra de.
Evolutionary procedural contente generation for na endless platform game. / Rafael Guerra de Pontes. – 2020.

13 f.

Orientador: Prof. Dr. Herman Martins Gomes.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Digital games. 2. Game development. 3. Artificial intelligence. 4. Genetic algorithm. 5. Procedural genetic generation. 6. Automação em coleta de dados. I. Gomes, Herman Martins. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

RAFAEL GUERRA DE PONTES

EVOLUCIONARY PROCEDURAL CONTENT GENERATION FOR A ENDLESS PLATFORM GAME

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Herman Martins Gomes
Orientador – UASC/CEEI/UFCG

Professor Dr. Carlos Eduardo Santos Pires
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 2020.

CAMPINA GRANDE - PB

Evolutionary Procedural Content Generation for an Endless Platform Game

Rafael Guerra de Pontes
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
rafael.pontes@ccc.ufcg.edu.br

Herman Martins Gomes
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
hmg@computacao.ufcg.edu.br

***Abstract*—Making innovative, cohesive and appealing games has become inherently more difficult given the ever increasing competition in the digital games' market. Manually creating game content is expensive and time-consuming. Therefore, alternative approaches for game content creation are relevant for increasing the efficiency of the game development process. This is where procedural techniques step in. Even though they have been used by commercial games since the 1980s, it was only in recent years that this kind of approach has been given the righteous attention in the academic context. In this work, we propose a procedural content generation approach for creating infinite environments for a 2D platform runner game. The approach consists of a Genetic Algorithm that innovatively takes into account environment aesthetics as well as game's physics and rules in its fitness function. Therefore, the created environments should be pleasant and possible to be overcome by the player. An instantiation of the approach was developed using the Godot Game Engine. Time viability for in-game real-time generation and convergence to high/stable fitness values were experimentally evaluated. Our tests indicated parameter ranges that performed best in terms of environment quality and processing time were mutation rates between 0.5% and 1% aligned with a population ranging from 50 to 100 individuals. This approach is expandable to other games that have a tilemap-based environments.**

***Index Terms*—Procedural Content Generation, Genetic Algorithm, Game Development, Artificial Intelligence**

I. Introduction

Digital games have become increasingly popular over the past few decades. In 2014, for example, the gaming industry was generating sales around 47 billion USD annually [1]. Recent research by the Software Entertainment Association (ESA) in 2020 found that 75% of households in the United States have at least one person who played video games. More than 4000 Americans were interviewed [2].

A. Problem Statement

Despite being a promising market, the Digital Game Industry faces a number of challenges. Among them, there is the time needed to create high quality games. Typically, this period lasts between 1.5 to 3 years. This is due to the fact that the construction of each component of AAA[1] games is quite a complex task, as it requires different technical and artistic skills. If we had to divide the components of a game into two parts, these would be: the engine and the content. The first dictates the rules of the game, provides the rendering of the graphics that the player sees throughout the different states of the game. The content corresponds to all the game entities, such as its world, interactive objects, stories, missions, textures, narratives, items, vehicles, characters, sound effects and music [1], [4].

As hardware technology advances, the range of content possibilities that can be rendered increases. Over the years, games have reached greater level of detail, number of simultaneously rendered objects and even the complexity of their behavior. Thus, what used to be simpler to be built by one or a few people, becomes not viable for some projects, due to the high workload involved. To remedy this problem, one can think of hiring more professionals to meet the demand, or even reuse existing content generated for other similar projects owned by the company, negatively impacting originality.

Among the various game genres and content types, this work focuses on the task of creating a playable environment for a 2D endless platform game. This comprises having to consider the constraints imposed by the game rules in the content creation methodology. Furthermore, the game has to present the player with unique playable environments in real-time, as he/she explores it.

B. Solution

In this context, a possible solution for such a task is to use Procedural Content Generation (PCG). Broadly speaking, PCG consists of a technique in which the content is not created manually, but rather algorithmically - or procedurally. In other words, the content is generated partially or totally by some algorithm. In this work, we focus on using an evolutionary algorithm to create the environment for a bi-dimensional (2D) platform game as mentioned before. This approach is an example of a search algorithm which is inspired in the Darwin's Theory of species evolution [5]. Evolutionary Algorithms have been widely used in many artificial systems and are also considered a form of reinforcement learning [6]. Particularly, we used a genetic algorithm which encodes possible solutions for the problem as individuals with a digital representation of its genetic code and apply operations such as mutations,

[1]Read "triple-A" games. They are considered the highest quality games on the market, usually developed by large companies and by a large number of professionals from the most diverse areas [3].

crossovers and natural selection simulations over the course of generations based on a fitness function. Eventually, the best individual is chosen to be incorporated into the game environment as the player explores it in real-time. More details on this technique are given in Section III.

C. Goals

As mentioned, this work focuses on generating the environment of an interactive bi-dimensional platform game. Our proposed approach is be able to generate:

1) the platforms on which the player can walk/run and jump onto[2];
2) the power-ups[3] that help the player achieve the game's goals;
3) a certain density of enemies per area that meets a tolerance range[4]; and
4) an overall cohesive environment that can be explored in real-time as it is procedurally created.

D. Paper Structure

The remainder of this paper is structured as follows. Section II contains some background about the game development industry, procedural content generation and its applications in games, search based approaches and an introduction to the basics of genetic algorithms. Section III presents core aspects in the proposed approach's architecture, such as how the game world is represented, technologies used, how the genetic algorithm's fitness function selects the best candidate solutions and other particularities of the procedural generation. Then, in Section IV some of the main results obtained from the validation tests performed are presented. Section V discusses some related works that propose alternative approaches to similar 2D platform games. Finally, Section VI contains final considerations and conclusions on the contributions of this work.

E. Publication

This work was presented as a full paper in the Computing Track of the Brazilian Symposium on Computer Games and Digital Entertainment (SBGames) 2020[5], which is organized by SBC (Brazilian Computing Society). A brief video presentation is available here.

II. Background

To better understand the context and motivations for procedural content generation approaches, this section presents the relevant background information.

[2]This implies the platforms must be disposed in a way as to make it possible for the player character to move between platforms, given restrictions such as its maximum jump height and horizontal speed.

[3]Items that grant the player some sort of beneficial attributes.

[4]Again, regardless of the difficulty, the game must be beatable.

[5]Proceedings available at this link.

A. Game Development Industry Challenges

As already mentioned, the gaming industry generates billions of dollars every year [1], [7]. However, the workload required to develop appealing games is increasingly demanding. This raises the cost of production both from a financial and time standpoints. Typically, larger games may cost from thousands to millions of dollars to produce and take between 18 and 36 months to be ready for release.

Among the challenges involved in the game development process, the content creation task is particularly important. This may involve the narrative, texture of the characters, their attributes within the game, scenery, voices, music etc. This usually requires hiring several professionals and coordinating them to harmoniously converge to a cohesive game. In this context, alternative approaches may be essential for the feasibility of the game in a reasonable time.

B. Procedural Content Generation

Procedural Content Generation (PCG) is a technique for creating content algorithmically. This content may be anything such as poetry [8], music [9]–[11], narratives [12], game rules, images, videos and so forth.

Togelius et al. [13] suggest that the use of PCG for games does not encompass content that was created by players – online or offline. Also, concerning whether or not PCG implies the use of randomness, some authors abide by the definition that it does, such as Andrew Doull [14]; whereas others like Togelius et al. do not [13], [15]. Therefore, according to the former, it would be possible to create non-stochastic procedural content.

As mentioned, PCG has been used in digital games since the 1980's. At first, it was closely related to the hardware limitations of that time, specially storage devices [4], [7]. One of the early examples of a PCG game was the game Elite [16], made in 1984. It immersed the player into immense galaxies and star systems that were generated procedurally. Its deterministic algorithm was based on the Fibonacci sequence and a chosen numerical seed. This proved to be interesting for players who enjoyed exploring the vast content present in the game world. Furthermore, beyond the hardware limitations, PCG makes infinite game content a feasible task. This increases replayability[6] [4], if done properly.

Even further, it is possible to make environments adapt to each player's gameplay. One may even argue that PCG helps mature the very process of game content creation. Artificial Intelligence (AI) supported PCG in games enables the development of highly creative systems and tools that enhance the process of game design [17]. This is specially important in the competitive game market. Therefore, in order to remain relevant in the commercial scene game companies must seek richer game designs. Attention to detail does not go unnoticed by players, which are more likely to spend their money on games that are genuinely worth it.

[6]Making the player play the game again without feeling tired of it.

More recently, one of the most famous games that use PCG is Minecraft [18]. An interesting fact is that this game was originally created by a single developer: Markus Persson, who founded the company Mojang and released the game under its name. When the player starts the game, a portion of the map is procedurally generated and expands as the player explores beyond its original boundaries. However, because of an astonishing success, Microsoft decided to buy Mojang for 2.5 billion USD and continues its development to this day [19]. Other titles worth mentioning for the use of PCG are: Spelunky [20], Dwarf Fortress [21], No Man's Sky [22], Diablo III [23], Just Cause [24] and Rogue [25].

PCG has gained such traction in the past decade that competitions have been arranged to stimulate exploration of this field. An example of this is the Angry Birds Level Generation Competition, which had its third edition in 2018, in an ACM conference on Computational Intelligence and Games (CIG) [26]. Besides the aforementioned games, there is a framework called Mario AI[7], which presents researchers with a Java interface that exposes several playing agents, level generators, original Mario Bros levels, human readable level and so forth. Among researchers who used it, there is an experiment conducted by Togelius et al. [27], who used the own user gameplay as input for the procedural generation algorithm. Players reported being impressed by the flexibility at which the environment creation was implemented. Another related research was conducted by Ferreira et al. [28], who applied GAs to generate a Mario level. To accomplish it, they separated the generation in four layers: terrain, interactible blocks, enemies and coins. Among the criteria used in the fitness fuction, they accounted for the terrain entropy (how irregular it was) and the even distribution of the other elements across the level.

Because there are many aspects in PCG techniques, taxonomies have been developed to categorize PCG approaches by Hendrix [29]; Kelly and Cabe [30]; Togelius et al. [31]; and Oliveira and Seabra [32]. Among these aspects, one may consider the type of content generated; techniques; generation time; input; scale; variaton; correctness of result; stop condition; determinism; complexity and others.

C. Search-Based Algorithms

This is a category of algorithms whose main objective is to explore candidate solutions for a problem. They are guided by some heuristic which mainly depends on the definition of a fitness function to evaluate how good a given instance of a candidate solution is. After a determined number of iterations or when the desired fitness is found in a candidate, the search may be taken as completed [15]. A relevant question in this kind of approach is how to encode the candidates such that they can be processed by a discrete machine. Encoding real-world problems is not a trivial task, as complexity is often a computationally prohibitive aspect encountered in many scenarios. In order to simplify the encoding, one may use a direct or indirect approach. In the direct approach, every characteristic of the candidates are written explicitly; whereas the indirect encoding condenses more abstract or high level aspects of the candidates in a non-linear fashion. Each approach has advantages and disadvantages regarding computational complexity and level of detail [15], [33], [34]. Given the inherent complexity involved in the search-based approach, optimizations are recommended whenever possible.

D. Genetic Algorithms

This kind of search-based approach is inspired in Darwin's Theory of Evolution [5]. Each candidate solution is interpreted as a biological individual with a genetic code that represents its characteristics [35] [6]. This idea was first introduced by Holland [36] and eventually explored in other studies in a variety of applications. In Genetic Algorithms (GAs), the highest scored individuals have a higher chance of surviving over generations (elitism) and higher probability to pass along part of its genetic code by crossing over with another individual [37]. GAs may be implemented with slight differences, but the overall flow of operations is the same, as seen in the Algorithm 1, presented in Section III, which is adopted for implementing our proposal. A population of individuals of size N is initialized with random genes. Then, for each new generation (iteration), a small percentage of the best individuals are kept intact for the next generation (elitism). Then, the rest of the new population is the offspring resulting from crossover operations with the randomly selected pairs of parents from the previous generation. This selection is biased by the fitness of the individuals, such that the higher the fitness, the higher the chances of becoming a parent. This crossover operation is better discussed in Section III-G. The renewed population, then, suffers mutations based on the defined mutation rate. The number of iterations is defined by the user.

One problem in this search-based approach is that there is no guarantee that a valid solution will ever be found. Therefore, it may be necessary to add mechanisms for ensuring minimum restrictions are not broken after all the iterations (generations) [4]. On the other side, GAs are attractive for being able to handle multiobjective, non-continuous and even NP-complete problems. Furthermore, they are relatively simple to understand [37].

III. Solution Architecture

In the following subsections, further details on each step of the approach are presented.

A. Solution Overview

The context of this paper is in the realm of 2D platform games. This type of game usually involves a form of physics simulation in which the characters (or actors) can perform actions such as walk, jump, fall and collide on platforms, walls, and other game entities. The main goal of this work is to procedurally generate a playable game world in real time. To achieve that, a Genetic Algorithm was implemented which

[7] Available at http://marioai.org/.

generates the level dynamically as the main player explores it. Before the game starts, the area around the player spawn point is generated and cached into memory and a chunk of it is decoded and rendered into the visible game world. As the player advances in the horizontal (x) axis towards the current world limits, new chunks are generated using the GA approach and appended to the previous generation global limits. This secondary generation happens in parallel with the game execution, such that the player has a seamless gameplay.

The Algorithm 1 has the overall execution flow of the GA-based game content generation proposed in this work. The representation of the population's individuals is detailed in Section III-D. The initialization of the population is explained in Section III-E. The $iGen$ variable represents the number of each generation. The selection mechanism of best individuals and parents is explained in Section III-G. The mutation is implemented with a pseudo random algorithm that may slightly change each gene with probability proportional to the chosen mutation rate.

P = randomly initialized population of fixed size N;
$iGen$ = 0;
while $iGen < maxGen$ **do**
  E = selectBestIndividuals(P, $elitismRate$);
  C = selectParentsAndCrossover();
  P = union(E, C);
  P = mutateIndividuals(P, $mutationRate$);
  $iGen$ += 1;
end

Algorithm 1: High-level pseudo-code for the Genetic Algorithm used in this work.

B. Technologies Used

In order to implement the proposed approach, the Godot Game Engine[8] was used. It is an open-source software released under the MIT License. It supports 2D and 3D game development. Among the programming languages it supports, we chose to use C#.

C. Test Game

To demonstrate the algorithm, a 2D platform game was developed[9]. In this particular game, the player is able to move towards either horizontal direction and perform actions such as walk, steer mid-air and jump. The goal of the game is to keep playing for as long as possible without letting the player fall down a hole or letting the time run out. In order to extend the time left, the player has to pick up clock items that are found in the game world. Fig. 1 depicts a sample of a rendered chunk of the created game.

[8]Available at https://www.godotengine.org and source code at https://github.com/godotengine/godot.

[9]Game source code and reproduction steps available at https://github.com/rafaelgdp/evopcg.

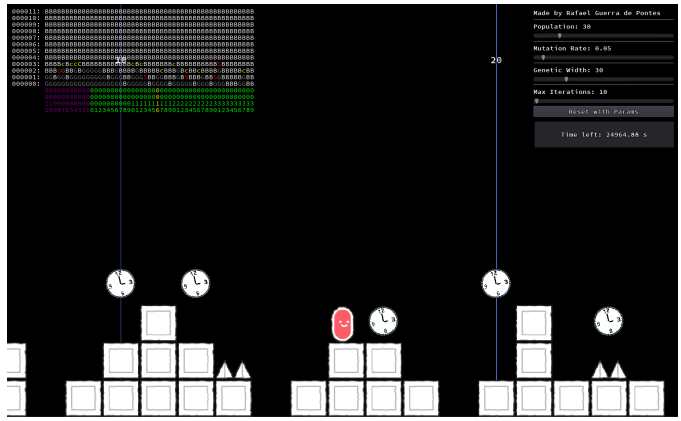


Fig. 1. Screen capture of the Test Game created.

D. World Representation

When rendered into the game, the world consists of a bi-dimensional matrix of collidable/interactible 64×64-pixel blocks that have certain properties within the game rules. Each cell is indexed with a 2D (x, y) coordinate that maps itself to its respective global 64×64 pixel square horizontally and vertically. However, in order to be generated by the GA, the infinite game world is sectioned into finite chunks that represent the space in a given horizontal (x) range.

To achieve this, a special representation was chosen. Instead of using a bi-dimensional matrix of symbols defining the block in a given (x, y) index, a uni-dimensional data structure similar to a doubly linked list was used. Each node in this list is an object that represents the information of all game elements in its x index. More formally, let each node on the doubly-linked list be called GeneColumn (GC). Each GC has properties such as: the ground height, presence/position of spikes, presence/position of a clock, reference to the GC on its immediate left and right. Fig. 2 depicts a simplified UML diagram with some of the properties present in the GC representation. Because each GC has a reference to its immediate neighbors, from any given GC that represents the game world, it is possible to reach or inspect the entire generated game world if necessary, simply by traversing the data structure.

A doubly linked list is an effective data structure to perform insertions in its head and tail dynamically. This is particularly useful in this PCG approach as newly generated chunks are ready to be appended to the game world either to the left or the right. Fig. 3 shows how this data structure maps to the cells in a given chunk of a horizontal (x) range of length 3. As may be seen, each node has all the necessary information for the game renderer to correctly place the cells in the game world. This is also useful for extracting features from the game world that are important for measuring the fitness function in the GA as described in Section III-F.

When the genetic code is decoded and rendered into the game world as a bi-dimensional block matrix, each block visually corresponds to an entity that is within the area of

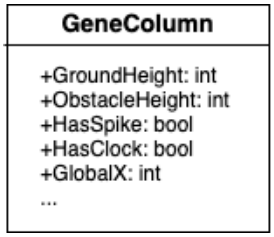


Fig. 2. Simplified UML representation of a GeneColumn.

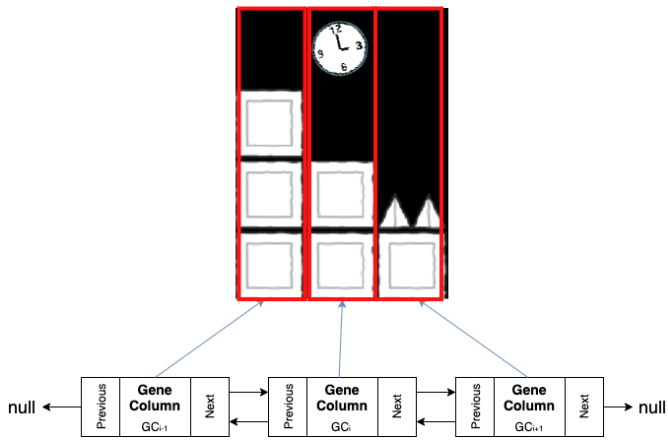


Fig. 3. Mapping from the doubly-linked list-like data structure that carries the genectic code from the generated individuals to its representation in the game world.

a 64×64 pixels square. The environment is, then, formed by contiguous blocks that are laid side by side. The possible types of environment blocks are described in Table I. The table also shows characters used by a simplified representation of the game world matrix for in-game visualization/debugging purposes.

E. Population Initialization

Whenever a new chunk of terrain needs to be generated, a new population of N individuals (environment chunks) is created. Each individual consists of two connected sub-chunks. One is an immutable reference chunk and another is a mutable chunk where the actual generation takes place. The reference chunk is simply there to help the fitness function of the GA to determine the best individual that connects to the reference chunk. Otherwise, the generated chunk could be drastically different from its neighboring area and still score a high fitness. The reference chunk is static, which means the mutations and crossovers in the GA do not affect it. However, the reference chunk is used alongside the mutable chunk by the GA to determine the individual fitness.

TABLE I
CHARACTER-ENCODED GAME BLOCKS USED FOR GENETIC CODE REPRESENTATION

Character	Meaning	Description
B	Blank	Absence of a block. The player may pass through this area.
G	Ground	A block on which the player and other physics entities may land on and collide with.
S	Spike	A collidable block that harms the player character.
C	Clock	A pickable item that grants extra time to the player.
c	Placed Clock	A clock that was already placed (rendered) in the environment or picked by the player.

F. Fitness Function

One of the most important design aspects of GAs is the fitness function. It is used to rank individuals of a population which are closest to a desired solution. We measured it with an integer value. The higher the value, the better the individual potentially is to be used as a solution. Upon initialization, each individual receives a given numeric fitness. For each positive feature encountered within its genetic code, a positive integer is added to its fitness. Likewise, for each negative feature, a certain amount is subtracted. If a feature is extremely negative (such as an impossible level), a significant amount would be decreased. The following features were considered:

- Safe jumps: for each safe block the player can land on, several jump simulations are performed to the neighbor safe blocks. For each possible safe jump, a small amount is added to the fitness. If the simulations identify an area in which there are no possible safe jumps, a great penalty is applied to the fitness. Fig. 4 depicts some parabolas that simulate possible player jump trajectories from a given starting block position. Each parabola starts at the safe block being inspected and ends precisely on 64-pixel separated spots that coincide with contiguous block widths.
- Ground level: for each subsequent column in the matrix, the ground height is compared to the the value immediately before. If the difference is greater than the maximum player jump height, a huge amount is decreased, because this would mean an impossible level. Likewise, if the ground is very irregular, but still playable, a smaller amount is decreased.
- Holes: if there are holes that separate two platform chunks longer than the player jump could overpass, the fitness is also be significantly decreased, as this means the game is also be impossible.
- Enemy density: if there are contiguous spikes (blocks that decrease the time left) such that they would be impossible to be surpassed by the player jump, the fitness would also decrease significantly. Likewise, even if surpassable, there should not exist too many nor too few spikes for a given horizontal (x) range (density). Therefore, the fitness function penalizes if there are less or more spikes per area than a tolerance range.

More formally, the fitness function is defined by (1).

$$fitness(i) = if(i) + sf(i) + hf(i) + cf(i) + jf(i) \quad (1)$$

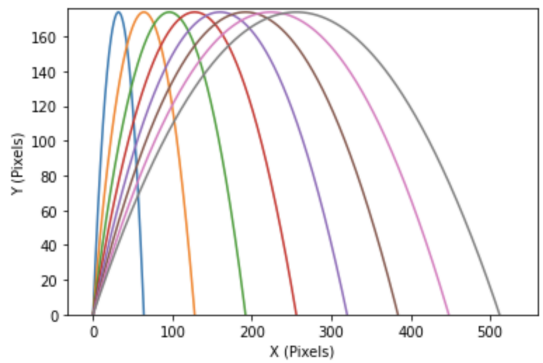


Fig. 4. Player jump simulations from a starting block at $x = 0$ to the right. Each landing x to the right is separated by 64 pixels to represent block distances.

Where,

$$if(i) = 200 \times w(i) \tag{2}$$
$$sf(i) = -omsd(i) \times sp \tag{3}$$
$$hf(i) = -ohc(i) \times hp \tag{4}$$
$$cf(i) = k \times it(i) - abs(it(i) - ct(i)) \tag{5}$$
$$jf(i) = -chp(i) \times ch(i) \tag{6}$$

Most of these parameters are functions that have a given individual i as input and output i's respective:

- $if(i)$: initial fitness of an individual;
- $w(i)$: genetic width of an individual. It's the number of GCs in an individual chunk;
- $omsd$: over maximum contiguous spike distance. It's a sum of all lengths of contiguous spike chunks that are over the maximum allowed in a given individual. This value depends on player jump height and maximum horizontal speed;
- sp: spike penalty (constant equal to 50);
- $ohc(i)$: number of oversized hole columns of an individual;
- hp: oversized hole column penalty (constant equal to 100);
- $ct(i)$: sum of clock extra times in the given area covered by an individual chunk;
- k: a constant equal to 5;
- $it(i)$: ideal extra time an individual chunk should have, defined in (11);
- $ch(i)$: contiguous hazard blocks of an individual. It's a sum of all contiguous unjumpable hazardous blocks such as spikes or holes both from the left to right and vice versa;
- chp: contiguous hazard penalty (constant equal to 50).

Concerning the jump simulations, multiple parabolas are used as a simplified model for the infinitely possible ways a player can actually jump from one safe block to another. Fig. 4 depicts the simulated trajectories of jumps performed from a given safe block in a $X = 0$ towards the possible right blocks. The equations of the parabolas are of the following form:

$$y = jc \times x \times (x - d_l) \tag{7}$$

Where,

$$jc = \frac{d - d^2/2}{H} \tag{8}$$
$$jt = -jf/g \tag{9}$$
$$H = jf \times jt + \frac{g \times jt^2}{2} \tag{10}$$

Or, in words,

- jc: is a numerical constant directly related with the player jump force.
- jf: is the player jump force (its initial y velocity when taking off from the ground).
- jt: is the time in seconds the jump takes to reach its maximum height.
- g: is the gravity acceleration in $pixels/second^2$.
- H: is the maximum height the player jump can reach.
- d_l is the distance in pixels to the l-th simulation landing block.

For evaluation the ideal time-traversal considerations for clock placement, we have the following equation:

$$it(i) = m \times cw(i) \times tpw(i)/ps \tag{11}$$

Where,

- $it(i)$: is the ideal extra time the clocks in a given individual chunk should grant the player in order for it to traverse it horizontally in a straight line;
- $cw(i)$: chunk width. It's the number of horizontal blocks a given individual chunk has;
- $tpw(i)$: tile pixel width of an individual;
- ps: player maximum horizontal speed. It's measured in pixels per second;
- m: multiplier. Since it's humanly improbable to collect all the clocks optimally in order to avoid a timeout, this number increases the ideal time to a more player-friendly time.

G. Elitism and Crossover

When evolving over the generations, the GA copies a small percentage of the population to the next. Even though these individuals are copied as is, they can still be changed by mutation, according to the mutation rate. The rest of the next generation population is created by recombining the genetic codes of individuals from the previous generation in pairs (here, called parents). Individuals from the previous generation are chosen as parents with probability proportional to its fitness value in comparison to the others. This means that higher valued individuals have a higher chance of being chosen as parents, but even the lowest fitness valued individual can still be chosen. When a pair of parents is chosen, the next step is to choose the crossover point. Since all individuals of a given

population represent contiguous blocks of the same horizontal (x) range in the game world, this crossover operation happens in a single point, such that the left parent passes along the genes for the left part of the chunk, and the right parent to the right portion. The crossover point is randomly chosen in a central region of the horizontal range. If the chosen point has an Obstacle Height that is too different from the other (i.e. the player cannot jump from the lower to the higher), this means that this crossover point is incompatible. The algorithm, then, attempts to choose other contiguous points to the left and right alternatively in the crossover region. If no crossover point is found, the algorithm gives up trying to find a compatible crossover point and the crossover point is the last value in the trials. Fig. 5 depicts an example of this operation. Notice that the colors represent the genes from the left and right parent. Since the maps tend to be around the same average height, not finding a compatible crossover point is rare. Even if this happens, the resulting individual would potentially present a low fitness value and be eventually discarded over the generations.

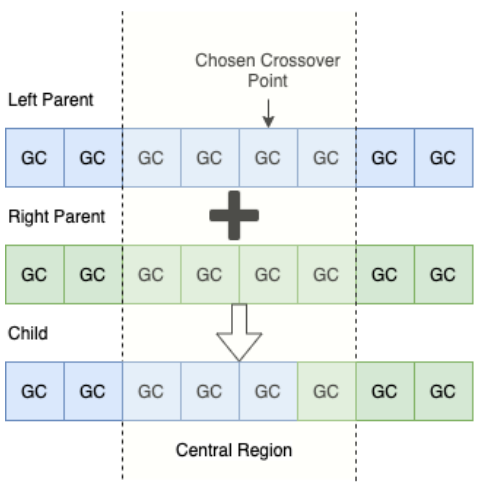


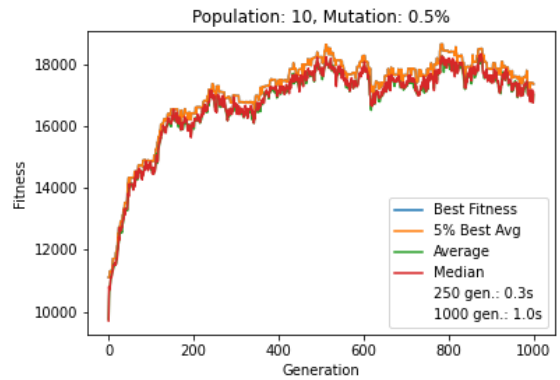
Fig. 5. Crossover operation. A preferably compatible crossover point is randomly chosen in the central region.

IV. Experimental Evaluation and Results

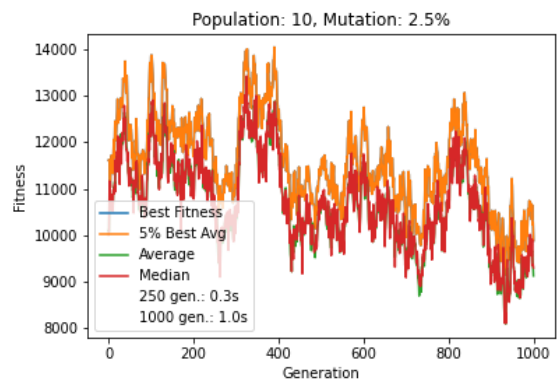
To evaluate the proposed approach, batches of simulations were run with different population sizes and mutation rates for map chunks with 100 GCs in length. These tests were run on a Macbook Pro 2018, with a 2,2 GHz 6-Core Intel Core i7 processor, 16GB 2400 MHz DDR4 RAM. Each parametric batch consisted of the following configurations:

- Population sizes: 10, 25, 50, 75, 100, 150, 250, 500, 1000.
- Mutation rates: 0.05%, 1%, 2.5%, 5%, 10%.
- Elitism: 5%, 10%, 20%.

Taking into account the test game created, the 100 GCs generated imply that the chunk created consists of a horizontal length of $100 \times 64 = 6400$ pixels. If the player has a maximum horizontal speed of 500 pixels/second, it means this area could be walked in at least 12.8 seconds (if there were no obstacles



(a)



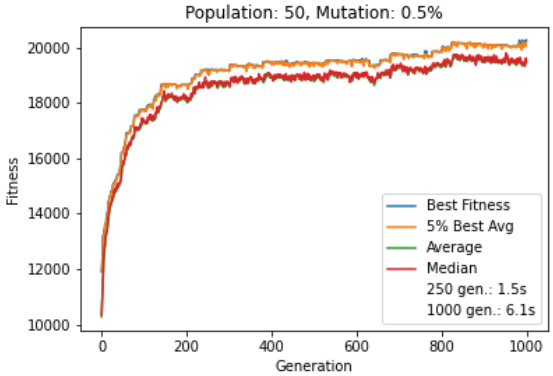
(b)

Fig. 6. Simulations with population size 10 and mutation rates of 0.5%, 1%, 2.5% and 5%. The legend also shows the time lapse in seconds for 250 and 1000 generations.

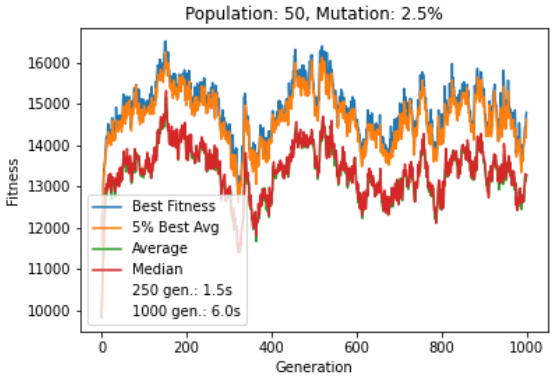
nor the need for mid-air steering maneuvers). Even though it is more probable that this is not the actual time a real player would take, it is a good baseline for the chunk generation algorithm processing time.

Figs. 6, 7, 8 show the fitness over the generations for some of the experiments. The blue and orange lines represent the fitness of the best individual and the mean of 5% best individuals in each generation, respectively. The green and red lines are the mean and median of the fitness in each generation. An indication for the processing time for the first 250 and 1000 generations are presented in the label. They revealed that, for this type of generation, the higher the mutation rate, the less likely it is for the algorithm to find better solutions over the iterations. Mutation rates of 2.5% or higher were very unstable across all simulated population sizes. Even worse, there were scenarios in which the fitness of the best individuals over the generations were getting lower values, as depicted by the sample images. This means that mutation rates of 2.5% or higher are not recommended for the generation of desirably playable maps with this particular approach.

However, low mutation rates of 0.5% and 1% rendered much better individuals over the generations even with small population sizes as low as 10. The experiments also showed

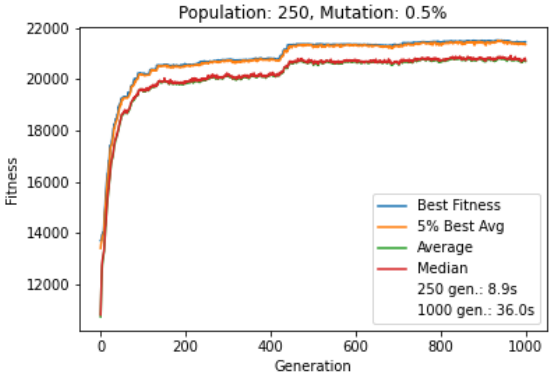


(a)

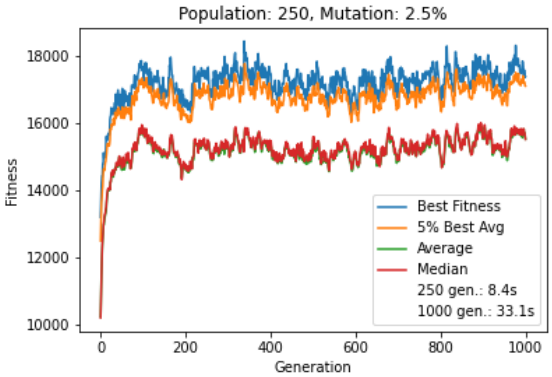


(b)

Fig. 7. Simulations with population size 50 and mutation rates of 0.5%, 1%, 2.5% and 5%. The legend also shows the time lapse in seconds for 250 and 1000 generations.



(a)



(b)

Fig. 8. Simulations with population size 250 and mutation rates of 0.5%, 1%, 2.5% and 5%. The legend also shows the time lapse in seconds for 250 and 1000 generations.

that higher population sizes yield better individuals in less iterations. Nonetheless, the higher the population size, the higher is the processing time per generation. Therefore, it is necessary to choose the right balance between quality and generation processing time. For this type of generation and the aforementioned hardware used, the population size of 50 and mutation rate 0.5% took around 1.5 second to process 250 generations and reached a satisfactory fitness. This is less than 12% of the optimal player area walk time. Therefore, nice playable maps can be generated in a safe time window such that the player can move at its top speed and never reach an area before it can actually be algorithmically generated.

Another point to consider is that the rate at which the fitness increases over first 100 generations is very high. However, after this point, the fitness tends to increase at a much lower rate. As the graphs show, after 200 generations, there is not much gain in the subsequent generations. Therefore, one may also choose to add stop criteria such as a lower number of generations around 250 or when the rate of best fitness increase decreases significantly. This iteration threshold is important specially if a larger population size is used due to the computationally prohibitive time demand that more individuals entangle, as our tests indicated.

Concerning the Elistim parameter, there was no significant change in percentages of 10% and 20%, so the graphs only show the results for 5% of elitism. This level leads to miscegenation in the subsequent populations, preserving only the very best individuals in each.

Finally, it is also worth noting that a GA does not guarantee that the generated map is playable. Therefore, it is also necessary to add a validation algorithm to check the playability of the generated map before actually using it in the game world. However, from the experiments, even the maps generated by low population sizes of 30 and 5% mutation rate over 10 iterations were all playable. Therefore, it is reasonable to say that maps generated with population sizes of 75, and 250 generations are probably playable and computable in a safe time lapse.

V. Related Work

Other proposals to create PCG based on Genetic Algorithms for platform games have been made. A recent one was conducted by Classon and Andersson [38], which focused on incorporating features that impacted on game difficulty in the fitness function. Therefore, the generation algorithm had the desired level of difficulty as an input. The tests they performed

concluded that the algorithm was successful in providing outputs compatible with the difficulty defined in the input parameter. The game they created had similar features to our test game such as spikes, a character that can walk and jump, but it also had other features such as springs that enhanced the jump height, moving platforms, trees, coins and enemy chicks. Among the variables they considered in their fitness function, were: the average width of all gaps, the number of gaps and the spatial diversity of gaps placed in a level. The authors also incorporated a validation in their algorithm to inhibit placement of spikes in undesired spots, such as under a pit. The present work also does a similar check. Besides focusing on the desired difficulty as the main goal of the GA, the levels generated were all finite in length. Their approach was different from ours in that they relied on human players to test the game. They measured time taken to complete the level, number of deaths and a questionnaire was used to gather subject perceived difficulty. The results were then compared to the values in the fitness function and showed they were correlated.

Even though GAs have been used for PCG, other approaches are also possible in PCG for games. In their study, Mendes et al. [39] developed a deterministic seed-based approach for generating levels for a 2D bullet hell game. They used an integer variable as input that is used to define the initial values for all the content creation methods. Their algorithm iterates over validity checks on the minimum requirements for the rooms that were initially created with the provided seed. These checks make sure that the pattern of rooms and entity placement meet a certain acceptable pattern. The rooms of the game are interconnected by contiguous pathways or by portals. One of the checks is in the placement of portals between rooms in the maze. Because the portal placement in their game is quite complex, they used a logic gate circuit to make this validity check, reducing programming complexity.

Another study that used a different PCG approach than a GA for a similar goal was conducted by Summerville and Mateas [40], which examined the use of Long Short-Term Memory Recurrent Neural Networks (LSTMs) for generating levels for of the *Super Mario Bros* game. The downside is that the LSTM Neural Network relied on a dataset of human-made levels for training. Their dataset consisted of real maps from the original games and eight networks were trained with different induced sequences. Among the metrics they considered were the percentage that could be completed by a simulated agent, the percentage of the level taken up by empty space, the linearity of the level (how much it resembled a straight line), the number of jumps required to complete a level and so forth. They concluded that including the player path in the metrics significantly improved the quality of the generated levels, according to their quality metrics. They believe this consideration may even help human game level designers create better levels without the need of exhaustive play testing.

As may be seen, PCG approaches for games vary greatly, depending on the game at hand, hardware, type of content, available time for processing and other factors. Our approach is unique and expandable for similar block-based game environments. Our doubly-linked-list game world representation with GCs as nodes is specially useful for appending dynamically generated chunks. This enables the creation of infinite worlds. Furthermore, incorporating or removing features in the generated world is simply a matter of editing the fitness function. The GA will, then, search for solutions that match best with the game generation goals. Therefore, we hope to have contributed to the area of online PCG for 2D platform games, specially infinite ones.

VI. Conclusion

The experiments presented in this paper showed that the proposed evolutionary approach is suitable for generating 2D platform game environments in a sufficient amount of time such that it may be used to generate new chunks in real time (online) as the player explores the map. However, to achieve good maps, it is crucial that a proper fitness function be modeled such that the individuals generated carry genes with the desired features. Furthermore, the parameter that impacts the most on the convergence of the algorithm to a increasingly better individual over the generations is the mutation rate. When it is set to values between 0.5% and 1%, the population has a much higher probability of evolving to better individuals. On the other hand, mutation rates of 5% or more are chaotic and may even render worse individuals over the generations.

Even though higher population sizes tend to converge less chaotically than smaller ones, the processing time penalty is prohibitive for real-time level generations. Therefore, it is the right balance of low mutation rate and medium-sized populations that yield the best evolutionary procedural content generation for this specific real-time application. An aspect that may be explored in a future work is to incorporate the calculation of the likelihood of evolution leap, which could be used to measure the performance for each simulation parameter [41]. With this approach, the algorithm may dynamically stop iterating when the fitness evolution rate has decreased significantly.

We hope to have contributed to game developers and researchers in the field of procedural game content creation as this approach may be easily extended to other block based game environments with new features. All that is required is that they be incorporated into the fitness function. To achieve this, one has to implement methods that extract the presence or absence of desired/undesired features in the individuals and add/subtract integer values from the fitness value. It is also important to apply great penalties in the fitness for extremely negative features (specially ones that render impossible levels). If the new fitness function is properly implemented, the algorithm will search for maps that match the generation goal. After this, it is necessary to edit the method that translates the GCs that represent the map to the 2D matrix of characters that represent the blocks of the chunk to be rendered in the game. If new types of blocks are necessary, it is important to assign a unique character for each new type of block or game entity.

Finally, besides being expandable in terms of game features and block types, this work may also be extended in the future for 3D block-based environments, similar to Minecraft worlds. To achieve that, a possible first step would be to expand the proposed one-dimensional doubly-linked list structure for a bi-dimensional quadruply-linked matrix. In this scenario, suppose the vertical axis is now z. Therefore, each node (Gene Column or GC) would have information on the placement of blocks in its vertical z axis. Each GC would also have references to its four neighbors (two in the x and two in the y direction). Hence, each individual in the GA would represent a 3D chunk. Instead of expanding only in the x direction, it would dynamically grow in both the x and y directions.

REFERENCES

[1] S. Dahlskog and J. Togelius, "Patterns and procedural content generation," in *Proceedings of the Workshop on Design Patterns in Games (DPG 2012), co-located with the Foundations of Digital Games 2012 conference*, 2012.

[2] Entertainment Software Association, "Essential Facts about the Video Game Industry," https://www.theesa.com/esa-research/2020-essential-facts-about-the-video-game-industry/, 2020.

[3] C. Hadzinsky, "A look into the industry of video games past, present, and yet to come," 2014.

[4] L. N. Ferreira, "Uma abordagem evolutiva para geração procedural de níveis em jogos de quebra-cabeças baseados em física," Ph.D. dissertation, Universidade de São Paulo, 2016.

[5] C. Darwin, "The origin of species on the basis of natural selection," 1859.

[6] S. Russel, P. Norvig *et al.*, *Artificial intelligence: a modern approach*. Pearson Education Limited, 2013.

[7] R. Khaled, M. J. Nelson, and P. Barr, "Design metaphors for procedural content generation in games," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 1509–1518.

[8] K. James, *Cut-up Consciousness and Talking Trash*. Leiden, The Netherlands: Brill — Sense, 2009.

[9] H. Zulić *et al.*, "How ai can change/improve/influence music composition, performance and education: Three case studies," *INSAM Journal of Contemporary Music, Art and Technology*, vol. 1, no. 2, pp. 100–114, 2019.

[10] K. Collins, "An introduction to procedural music in video games," *Contemporary Music Review*, vol. 28, no. 1, pp. 5–15, 2009.

[11] R. L. De Mantaras and J. L. Arcos, "Ai and music: From composition to expressive performance," *AI magazine*, vol. 23, no. 3, pp. 43–43, 2002.

[12] C. Fernández-Vara and A. Thomson, "Procedural generation of narrative puzzles in adventure games: The puzzle-dice system," in *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, ser. PCG'12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–6.

[13] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is procedural content generation?: Mario on the borderline," in *Proceedings of the 2nd international workshop on procedural content generation in games*. ACM, 2011, p. 3.

[14] A. Doull, "The death of the level designer," *URL http://pcg. wikidot. com/the-death-of-the-level-designer*, 2008.

[15] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 141–150.

[16] I. Bell and D. Braben, "Elite game," http://www.iancgbell.clara.net/elite/, 1984.

[17] G. M. Smith, "Expressive design tools: Procedural content generation for game designers," Ph.D. dissertation, UC Santa Cruz, 2012.

[18] Mojang, "Minecraft game," https://minecraft.net, 2011.

[19] K. Stuart and A. Hern, "Minecraft sold: Microsoft buys mojang for $2.5bn," 2014. [Online]. Available: https://www.theguardian.com/technology/2014/sep/15/microsoft-buys-minecraft-creator-mojang-for-25bn

[20] M. LLC, "Spelunky," 2009-2013. [Online]. Available: https://spelunkyworld.com

[21] B. . Games, "Dwarf fortress," 2002-2020. [Online]. Available: http://www.bay12games.com/dwarves/features.html

[22] H. Games, "No man's sky," 2013-2020. [Online]. Available: https://www.nomanssky.com

[23] B. E. Inc, "Diablo iii." [Online]. Available: https://us.diablo3.com

[24] S. E. Ltd, "Just cause." [Online]. Available: https://justcause.square-enix-games.com/

[25] M. C. Toy and K. C. Arnold, "A guide to the dungeons of doom," *Computer Systems Research Group. University of California, Berkeley. Nd Web*, vol. 9, 2012.

[26] P. Kaufmann and P. A. Castillo, *Applications of Evolutionary Computation: 22nd International Conference, EvoApplications 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24–26, 2019, Proceedings*. Springer, 2019, vol. 11454.

[27] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[28] L. Ferreira, L. Pereira, and C. Toledo, "A multi-population genetic algorithm for procedural generation of levels for platform games," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 45–46.

[29] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, pp. 1–22, 2013.

[30] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation," *ITB Journal*, vol. 14, no. 3, pp. 342–351, 2006.

[31] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[32] N. Oliveira and R. D. Seabra, "Towards a comprehensive classification for procedural content generation techniques," *XV Simpósio Brasileiro de Jogos e Entretenimento Digital-SBGames*, 2016.

[33] W. L. Raffe, F. Zambetta, and X. Li, "Evolving patch-based terrains for use in video games," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 363–370.

[34] N. Othman, J. Decraene, W. Cai, N. Hu, M. Y. H. Low, and A. Gouaillard, "Simulation-based optimization of starcraft tactical ai through evolutionary computation," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2012, pp. 394–401.

[35] K. A. De Jong, *Evolutionary computation: a unified approach*. MIT press, 2006.

[36] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[37] K. F. Man, K. S. Tang, and S. Kwong, "Genetic algorithms: concepts and applications [in engineering design]," *IEEE Transactions on Industrial Electronics*, vol. 43, no. 5, pp. 519–534, 1996.

[38] J. Classon and V. Andersson, "Procedural generation of levels with controllable difficulty for a platform game using a genetic algorithm," 2016.

[39] D. D. M. Mendes, D. M. C. Silva, H. N. Rabelo, I. B. Amaral, V. S. Granja, and M. S. Nery, "Processo de design de fases por lógica booleana para geração procedural de conteúdos em jogos digitais," *XVIII Simpósio Brasileiro de Jogos e Entretenimento Digital-SBGames*, 2019.

[40] A. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms," *arXiv preprint arXiv:1603.00930*, 2016.

[41] K. Sugihara, "A case study on tuning of genetic algorithms by using performance evaluation based on experimental design," in *Proceedings of the 1997 joint conference on information sciences*, 1997.