Procedural Enemy Generation through Parallel Evolutionary Algorithm

Leonardo T. Pereira  
Instituto de Ciências Matemáticas e de Computação (ICMC)  
Universidade de São Paulo (USP)  
São Carlos, São Paulo, Brazil  
leonardop@usp.br

Breno M. F. Viana  
Instituto de Ciências Matemáticas e de Computação (ICMC)  
Universidade de São Paulo (USP)  
São Carlos, São Paulo, Brazil  
bmfviana@gmail.com

Claudio F. M. Toledo  
Instituto de Ciências Matemáticas e de Computação (ICMC)  
Universidade de São Paulo (USP)  
São Carlos, São Paulo, Brazil  
claudio@icmc.usp.br

Abstract—This research presents a Parallel Evolutionary Algorithm (PEA) that generates enemies with diverse characteristics, such as the enemy’s health, weapons, and movement. Our PEA aims to create enemies matching their difficulty degrees with the difficulty goal given as input parameter. We designed our algorithm in this way to be future used in an online adaptive generation system. We experimented with a set of generated enemies with an Action-Adventure game prototype as a testbed. The results show that players evaluated our approach positively, successfully creating enemies considered easy, medium, or hard to face, as defined by their original fitness target value. Besides, the players found the game fun to play for all difficulty levels played, and the perceived challenge rose as the PEA fitness was higher. In terms of performance results, our PEA converged into the input solution in less than a second for most cases, denoting its future use in online adaptive applications.

Index Terms—enemy generation, procedural content generation, video games, parallel evolutionary algorithm

I. INTRODUCTION

Procedural Content Generation (PCG) are algorithmic methods that can create different types of content for computer applications. PCG is well-known for its use in video games, both new and old, like in the dungeons from the 1980’s Rogue and the weapons from the 2019’s Borderlands 3. Content generation methods can bring a plethora of advantages for the game industry, such as reducing development time and costs up to 40%, and creating products with higher replayability than their non-procedural counterparts [1, ch. 4.1, p. 152-153]. These advantages, together with content generation challenges, have brought many researchers to develop algorithms to create different types of content, such as levels, narratives, and gameplay [2]. Many games reportedly use some PCG: searching the tag “Procedural Generation” for the “Games” type in the Steam game store, one of the largest PC-gaming stores, shows that, as of the writing of this paper, more than 1,700 games have this tag from its more than 30,000 games total1.

One of the significant components in game design is the enemies the player must face. Such a feature arguably makes the gameplay’s core for most genres. Research on intelligent agents for games is relatively abundant. The development of algorithms to control enemies in Real-Time Strategy (RTS) games is very common, with the Starcraft game series being the most common test environment [3]. We also have seen it reach the mainstream media as the AlphaStar Artificial Intelligence (AI) beat Dota 2’s professional sports players [4], and also Starcraft II’s [5]. However, they focus on controlling already existing creatures in a game via AI techniques.

There are relatively few algorithms reported in the literature that creates different enemies’ mechanics and visual properties. The procedural generation of enemies with different statuses, behaviors, weapons, among other features, has not been found in recent literature during our review. Some recent works focus on the placement of enemies through game levels, but only previously-existing ones [6]–[9]. The work most similar to ours is the method introduced by Khalifa et al. [10]. They developed an evolutionary generative process for bullet hell level generation. Their method consists of projectiles with different movement patterns and speeds, as well as bosses.

Our research aims to respond to the following research question: Can we procedurally generate interesting enemies for digital games, that are able to meet a specific difficulty threshold and present some diversity?

We hypothesize that an evolutionary algorithm may be able to do so, given the positive results these algorithms have in previews works of PCG, as presented in Section II. Therefore, we present a system able to answer this question in Section III. The approach we present in this paper targets the generation of enemies by evolving their attributes (e.g., health) and behaviors (e.g., movement type). To do so, we evolve enemies through a Parallel Evolutionary Algorithm (PEA) to provide a set of enemies with different characteristics. We opt for such an approach because we designed it to be, in the future, part of an online adaptive generation system. Therefore, we experimented with our approach for both performance and players’ gameplay feedback. Our results show that the enemies we generated matched the players’ expectations. Besides, our approach is faster enough to be used in online generation.
The main contributions of this paper are stated as follows. We propose a method to help fill the gap of enemy generation, shown in Section II as an area of research with, as far as our review could find, few recent contributions, and most lack the ability to create diverse enemies that fit a specified difficulty. This method introduces a parallel evolutionary algorithm to generate diverse and unique enemies in terms of attributes and behavior. Since we designed our approach to be part of an adaptive system, our fitness function goal is to minimize the distance of difficulty of the enemies with the aimed input difficulty. Therefore, our solution can also be used for human designers to increase their creativity and productivity during the enemy creation. So, we believe the main novelties of our work are the representation of enemies to be used in search-based algorithms to adapt them to different difficulties and the application of a parallel evolutionary algorithm as a solution for enemy procedural generation problems.

This paper is structured as follows. Section II present the recent literature related to this research. In Section III, we answer the research question we stated in this section by detailing the generative process of our approach, as well as the game prototype developed as a testbed. Section IV presents the work’s results in terms of performance and quality. Finally, Section V presents the conclusions and future works.

II. RELATED WORK

Since the procedural generation of enemies is a pretty recent research area, few works in literature attempted to tackle such a problem. Therefore, in this section, we describe works that somewhat generated enemies during some stage of their generative processes. We also describe some games that somewhat generated NPCs. Table I summarizes our findings and how our work compares to what we found in scientific papers and some commercial games.

Baldwin et al. [6] introduced the Evolutionary Dungeon Designer (EDD), a tool to support game designers in their level creation process. EDD focuses on the generation of 2D dungeon levels, and its evolution is based on game design patterns and performed by a Feasible-Infeasible Two Population Genetic Algorithm (FI2Pop GA). Besides, EDD provides control of frequency, shape, and type over the generated design patterns and the placement of enemies, treasures, doors, and walls. They control enemy placement by setting the number of enemies of a level and the fitness function. However, this version of the tool dealt only with micro-patterns (i.e., grid cells). Later, Baldwin et al. [7] extended the EDD by introducing the evolution of meso-patterns, which are equivalent to rooms: guard chamber, a room with only enemies; ambush, a room with enemies and an entrance; treasure chamber, a room with only a treasure; and guarded treasure, a room with enemies and a treasure. This extension allowed them to perform a more precise enemy placement. Their results were positive considering the diversity of solutions and convergence, but there was no experimental setup with players within a game.

Like the previous works, Sharif et al. [8] shows different strategies based on design patterns for sprite placement in levels. To perform the positioning, they take a sprite from a pool of existing sprites. Some of these sprites are harmful sprites, such as enemies or traps. Nevertheless, their work only performs placement of existing sprites, and they do not create new ones in their generative process.

Liapis [9] developed a two-step automatic evolutionary process for dungeon levels, where both steps evolve the dungeon sketches and levels through FI2Pop GA. The first step generates dungeon sketches to place strategically eight different types of segments (wall, empty, simple, exit, sparse reward, high reward, sparse challenge, and high challenge). These challenges are treated as enemies. In the second step, each segment of a dungeon level evolves independently to create a cavern environment, following connections between the segments and their types. In this stage, the method places enemies in segments and, if the segment has rewards, the enemies are placed strategically around them. The placement strategy is interesting because it forces players to fight the enemies to get the reward.

Khalifa et al. [10] introduced a description language, called Talakat, to define Bullet Hell games’ levels. Bullet Hell games consist primarily of bullets (which we consider enemies) with different damage values, speed, and movement patterns. A Talakat script constitutes a single bullet hell level divided into a spawner section and a boss section. The spawner section defines spawn points to spawn bullets or create new spawners. Each spawner has a set of parameters to determine the bullet it spawns, the speed, angle, and size the bullet receives, the angle and speed at which the spawner rotates, among other features. The boss section defines the boss’s health, position, and behavior, i.e., they generate enemies beyond placing them in levels. The sections consist of numbers and behaviors determining some controllable bullets’ properties. Talakat scripts evolve through Constrained MAP-Elites (CME) – i.e., MAP-Elites with FI2Pop – to generate different levels. They simulate AI agents playing the levels to validate and define the levels’ fitness. The results showed that the approach is capable of generating challenging levels with a wide variety.

Nevertheless, enemies are more than the place they are and their behavior. Enemies have stats like health, damage, weapons, combat styles, among others. We did not found any paper that proposes the generation of enemies with these kinds of parameters. However, the game industry has developed games that perform such enemy generation.

The procedural generation of Non-Playable Characters (NPCs) with many properties is present in Spore [11] and No Man’s Sky [12] games. Both present a similar algorithm: a creature is created by randomly assigning different body parts. Although not having any input to create them based on the player’s performance, they have some constraints on not putting together body parts that cannot match another already selected one. However, this constraint is mostly as not to break the procedural animations of the NPCs. The game series Creatures [13], although older, extends this concept, not
only physically evolving NPCs but also making them learn about the environment and the player’s actions via a neural network that receives simulated senses using semi-symbolic approximation techniques as input [18].

Besides the generation of NPCs, *Diablo 3* [14] and *Shadow of Mordor* [15] change some predefined characteristics in their enemies to make the challenge more diverse and unique. The enemies in *Left 4 Dead 2* [16] and *State of Decay 2* [17] are somewhat also adapted to the player. However, this adaptation is not made by changing their characteristics or features, like *Diablo 3* [14] and *Shadow of Mordor* [15]. Instead, they decide where to place them and if new enemies should be spawned (if the player is doing well) or if the game should spawn fewer enemies (if the player is not performing very well) 3 4.

Besides the enemy generation in terms of placement, amount, status, and visuals – as highlighted in Table I –, our approach represents enemies with numerical and nominal values in a similar way to the values in Talakat scripts [10].

### III. Methodology

In this section, we answer the research question we stated in Section I by describing our enemy generation algorithm. First, we describe our enemy representation, then report the generative process that our PEA carries on. Finally, we describe the game prototype of the testbed for our experiments.

#### A. Enemy Representation

We extracted the most common variables from enemies in different games to build our enemy’s genotype, focusing on the Action-Adventure genre since it is our testbed genre. After careful consideration, we came up with the variables: health, damage, attack speed, movement speed, active time, rest time, movement type, weapon type, and projectile speed. Table II details each variable: it contains the variable type, possible range, and a brief description of their impact on gameplay. The range for each variable was set empirically after testing our prototype with different players and selecting the ranges that enabled the enemies to range from easy (but not boring) to hard (but not humanly impossible) configurations.

We defined nominal variables for movements and weapons (Table II) because they represent more complex behaviors and objects, respectively. Table III presents the possible movement behaviors the enemies may have. Table IV presents the possible weapons and their behaviors. Both tables present weight that are used in the difficulty degree calculation. We defined these weights empirically through a test with a small number of players. Although our PEA is relatively simple in terms of genotypical variety, for a commercial game, there can be dozens or even hundreds of possible weapons and behaviors and other enemy-controlling parameters that can be added to the PEA with low effort. In this paper, we considered only the movements and weapons present in our game prototype.

#### B. Enemy Generation Process

The creation of the initial population is carried out by the random filling of each parameter of the enemies (considering the allowed ranges of integers, floats, and the sets of movements and weapons for the nominal values). With the reproduction operators we describe in the following paragraphs, the resulting population then generates an entirely new population that replaces the first one. This process repeats until the algorithm reaches $g$ generations. When the algorithm finishes, we return the $N$ best individuals’ data, where $N$ is the number of enemies to be returned (it is also an input of our algorithm). Therefore, we use the ability of the PEA to evolve whole populations of solutions to have a collection of adequate individuals in a single execution.

The crossover and mutation operators are responsible for the reproduction of enemies. We apply the average crossover [19] operator with a 99% chance of occurring. The operator consists of calculating the average value between two parents for each numerical parameter, e.g., between the attack speed of parents. For the nominal parameters, the crossover operator

### TABLE I

<table>
<thead>
<tr>
<th>Work</th>
<th>Placement</th>
<th>Amount</th>
<th>Status</th>
<th>Visuals</th>
<th>Adaptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baldwin et al. [6]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Baldwin et al. [7]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sharif et al. [8]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Liapis [9]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Khalifa et al. [10]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Spore [11] - - P P -
No Man’s Sky [12] - - P -
Creatures [13] - - ✓ P ✓
Diablo 3 [14] - - P -
Middle Earth: Shadow of Mordor [15] - - P -
Left 4 Dead 2 [16] ✓ ✓ - - ✓
State of Decay 2 [17] ✓ ✓ - - ✓

This work ✓ ✓ ✓ P -

TABLE II
LIST OF PARAMETERS OF THE ENEMY’S GENOTYPE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Range</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health</td>
<td>Integer</td>
<td>1-5</td>
<td>Total health.</td>
</tr>
<tr>
<td>Damage</td>
<td>Integer</td>
<td>1-4</td>
<td>Damage done.</td>
</tr>
<tr>
<td>Attack Speed</td>
<td>Float</td>
<td>0.75-4.0</td>
<td>Projectile’s shot frequency (1/Attack Speed).</td>
</tr>
<tr>
<td>Movement Speed</td>
<td>Float</td>
<td>0.8-3.2</td>
<td>Multiplies movement direction’s vector.</td>
</tr>
<tr>
<td>Active Time</td>
<td>Float</td>
<td>1.5-10.0</td>
<td>Time (in sec.) that the enemy moves before resting.</td>
</tr>
<tr>
<td>Rest Time</td>
<td>Float</td>
<td>0.3-1.5</td>
<td>Time (in sec.) the enemy rests before moving.</td>
</tr>
<tr>
<td>Projectile Speed</td>
<td>Float</td>
<td>1.0-4.0</td>
<td>Multiplied by the projectile’s trajectory vector.</td>
</tr>
<tr>
<td>Weapon Type</td>
<td>Nominal</td>
<td>-</td>
<td>Enemy’s weapon. Each may have different properties.</td>
</tr>
</tbody>
</table>

TABLE III
LIST OF MOVEMENT TYPES THAT ENEMIES CAN HAVE. ALL THE MOVEMENTS ARE 2-DIMENSIONAL MOVEMENTS. DV ABBREVIATES DIRECTION VECTOR AND RDV ABBREVIATES RANDOM DIRECTION VECTOR.

<table>
<thead>
<tr>
<th>Movement</th>
<th>Weight</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>Stay still.</td>
</tr>
<tr>
<td>Random</td>
<td>1.04</td>
<td>Selects a RDV to move towards in the actual active cycle.</td>
</tr>
<tr>
<td>Random1D</td>
<td>1</td>
<td>Selects an axis from a RDV to move towards in the current cycle.</td>
</tr>
<tr>
<td>Flee</td>
<td>1.1</td>
<td>DV opposes the player’s direction.</td>
</tr>
<tr>
<td>Flee1D</td>
<td>1.08</td>
<td>Selects an axis from the player’s location opposing vector in the current cycle.</td>
</tr>
<tr>
<td>Follow</td>
<td>1.15</td>
<td>DV points towards the player’s direction.</td>
</tr>
<tr>
<td>Follow1D</td>
<td>1.12</td>
<td>Selects an axis of the vector towards the player’s location in the current cycle.</td>
</tr>
</tbody>
</table>

TABLE IV
LIST OF WEAPONS AVAILABLE FOR ENEMIES. ‘*’ HIGHLIGHTS THE WEIGHTS OF THE WEAPONS THAT SHOT PROJECTILES, THESE WEIGHTS ARE MULTIPLIED BY THE PROJECTILE SPEED AND ATTACK SPEED.

<table>
<thead>
<tr>
<th>Weapon</th>
<th>Weight</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1</td>
<td>Damage on contact.</td>
</tr>
<tr>
<td>Sword</td>
<td>1.5</td>
<td>Holds a sword in front of itself. Increases reach.</td>
</tr>
<tr>
<td>Shield</td>
<td>1.6</td>
<td>Protects the enemy from frontal attacks.</td>
</tr>
<tr>
<td>Bullet</td>
<td>0.3*</td>
<td>Shoots a bullet towards the player. Damages on contact.</td>
</tr>
<tr>
<td>Bomb</td>
<td>0.3*</td>
<td>Shoots a bomb towards the player. Explodes in 2 seconds.</td>
</tr>
</tbody>
</table>

selects at random, with equal probability, the parents’ corresponding value. For instance, if one parent has the Random movement and the other the Flee movement, the child has a 50% chance to inherit each. When the crossover does not happen, the first parent is sent to the intermediate population.

Our approach presents a multi-gene mutation [20], which means that our mutation operator can change all the enemy’s genes (i.e., the enemy’s parameters). Each gene has, individually, a 10% chance of mutating. For each gene, we calculate the chance of mutating the parameter. If the calculated chance is to mutate the parameter, a new value is randomly chosen, respecting the allowed ranges. The mutation is very important for the numerical values, as it helps avoid a local convergence to a middle-ground value. Regarding the nominal parameters, the value is randomly selected from the list of nominal values.

The selection of parents is made through a 2-individual tournament, in which the winner is the fittest enemy. Besides, all the random operations follow a uniform distribution.

Since our approach is designed for future application in an online adaptive generation system, we opted for a simple fitness function for better performance instead of a simulation-based one. Therefore, our fitness function minimizes the distance of difficulty of the enemies with the aimed input difficulty. The difficulty function sums all the parameters, weighting the nominal ones according to the empirical found values. This function is sums of the equations 1, 2 and 3 – that are weapon factor, projectile factor, and movement factor, respectively – and others parameters presented in Equation 4.

\[ f = health + active + 1/rest + f_w + f_p + f_m, \]  

where, damage is the damage the enemy does to the player, and weapon is the weight of the weapon it currently holds.

\[ f_p = \text{projectile} \times (\text{attack\_speed} + \text{projectile\_speed}), \]  

where, projectile is 1 if the enemy can throw any kind of projectile and 0 otherwise, projectile\_speed is the projectile’s speed shot by the enemy’s weapon, and attack\_speed is the rate at which the enemy throws the said projectile.

\[ f_m = \text{movement\_speed} \times \text{movement}, \]  

where, movement\_speed is the movement speed, and movement is the weight of the enemy’s movement behavior.
where, health is the enemy’s health, active is the time the 
enemy moves without stop before resting, and rest is the time 
the enemy rests between cycles of active movement. Besides, 
the rest time has value inversely proportional to the difficulty, 
as the lower it is, the more difficult the enemy tends to be. We 
highlight that although we use a simple fitness function, we 
decided to use an EA. Thus, at the end of a single evolution, 
we would have a whole population of good solutions with 
some degree of variety. Also, the representation of enemies, 
which is one of our main novelties, may be expanded to 
more complex enemies that may require harder-to-solve fitness 
functions without many changes in our methodology.

Regarding parallelism, the evolutionary process is carried 
out by the Unity3D’s Data-Oriented Technology Stack 
(DOTS)\(^5\). The DOTS present a data-oriented design that uses 
the Entity Component System (ECS), which optimizes object 
referencing and boosts processing for large amounts of data, 
and the Burst Compiler, which maximizes registers’ use and 
other features. Both ECS and Burst Compiler are optimized 
for parallel job operations in Unity3D\(^5\). Therefore, this stack 
is suitable for evolving in parallel a large population of 
individuals in a reduced time frame. For each generation of 
our population, we handle the evolutionary operators for each 
new individual as different parallel jobs.

Furthermore, DOTS also allowed us to have huge popula-
tions for our PEA. Thus, we have greater diversity in our 
solution candidates by having such a massive population, 
which, except when computational time is hindered, usually 
leads to faster convergence [21]. Therefore, besides the speed 
brought by the parallelism, DOTS enabled us to converge to 
good solutions quickly, taking us a step closer to an online 
enemy creation algorithm.

So far, we have described only the generation of enemies’ 
statuses and visuals, which are defined by the enemies’ 
weapons and difficulties. Next section, we present the game 
prototype and detail the process of enemy placement.

C. Game Prototype

We adapted the game prototype developed by Pereira et al. 
[22] according to the needs of our research. The prototype is 
an Action-Adventure game that mimics the main mechanics from 
The Binding of Isaac’s combat [23] and the dungeon 
exploration of The Legend of Zelda [24]. Although our focus 
is to experiment with the combat mechanics, we also added 
dungeon exploration to enhance the players’ immersion and 
retain their attention through different playthroughs. This feature 
is important as we want to collect data both from successful 
attempts and ones that the player died, as the number of failed 
attempts may reflect the difficulty. Furthermore, the game 
contains rewards and other kinds of items, which the player 
may collect throughout the levels.


\(^6\) Get Started with the Unity\(^6\) Entity Component System (ECS), C# Job 
System, and Burst Compiler: https://software.intel.com/en-us/articles/get-
started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-
and-burst-compiler.

Fig. 1. Game prototype screenshot.

Fig. 2. Enemies’ spawn points represented as yellow circles.

The protagonist of the game is a yellow robot. The player 
must control it to explore the levels’ rooms, collect items (e.g., 
keys and treasures), open doors and find the levels’ goals 
(a green triangle). Besides overcoming locked doors puzzle 
challenges, the player must defeat enemies (gray robots) by 
shooting green projectiles. Enemies fill all rooms, but the 
starting and final rooms, and the player must defeat them to 
proceed to other rooms. After the player wins or loses a level, 
a score screen is shown to give feedback about: the victory 
or failure, the highest combo reached, the amount of treasure 
collected, and the number of visited rooms. The players can 
only progress in the game if they win the levels. Fig. 1 presents 
screenshot from our game prototype. As we observe in the 
figure, all doors are locked, the health points are full (the hearts 
at the top-right), and, in the mini-map, the neighboring rooms 
of each room and highlights the visited rooms (at the bottom-
right). Below we list the main game features that are or will 
be present in our prototype.

When the player enters a room, the game spawns enemies 
from the pool of enemies generated by the PEA. We selected 
the enemies randomly to spawn in the rooms to provide a 
variety of enemies. The enemies are added in rooms until 
the sum of all their fitness is greater than three times for the 
difficulty level the player chose. This control enables some 
rooms to have 3 to 4 enemies, as their fitness values might be 
slightly below the target fitness. Then, each enemy would be 
assigned to a different spawn point in the room.

Regarding the enemy placement, we decided to divide the 
room into a grid with 25 cells (five rows and five columns) 
to define the spawn points. We then removed the cells close
to doors, giving players enough space to move and avoid hits when entering a new room. The resulting possible spawn points are presented as the yellow circles in Fig. 2.

IV. RESULTS

This section reports the computational results of PEA’s performance and diversity achieved by our method, and the players’ feedback and gameplay performance.

A. PEA’s Performance

Our first set of experiments test the convergence time and quality of solutions given by the PEA. We executed the PEA for nine different sets of input parameters, 200 times each, and measured the generation time. All tests were run in a relatively common PC setup: AMD FX-8320 Eight-Core 3.50GHz Processor, 16 GB DDR3 RAM, 223GB SSD memory, NVIDIA GeForce GTX 1060 3GB graphics card. Table V shows the results of this experiment, considering three different fitness objectives: Easy; Medium; with fitness 17.5; and Hard, with fitness 22.5.

The table contains the average execution time, average and standard deviation for the whole population’s fitness, and the average and standard deviation of the fitness considering only the 20 best individuals (the N value chosen for our game prototype, as mentioned previously) across the 200 executions for each input. We can observe that the algorithm is fast, even with large populations. Taking less than 0.2s to evolve a population of 10,000 individuals over 10 generations, and 1.7s for 100 generations. Therefore, we may be able to later apply our approach for online enemy generation.

The algorithm can evolve the 20 best individuals very close to the difficulty goal input. The worst-case scenario average is 0.28 units away from the target, with a standard deviation of 0.17. This difference gets a little higher for the whole population: for the hard difficulty, the average was more than 2.5 units below the desired fitness and had a standard deviation of 2.277. Therefore, we can not guarantee that the whole population harbors good solutions for this difficulty. Furthermore, this result is somewhat expected since we evolve 10,000 individuals. However, we can ensure that at least the 20 best individuals, which already guarantees each PEA’s execution, can conceive a good variety of feasible individuals. This result is very useful to entertain players and increase the uniqueness of each playthrough.

Trying to settle for good results in the lowest time possible, we decided to create our population of enemies using 10,000 individuals and evolve the population for 30 generations for each difficulty setting for the experiments with players. Table V highlights in bold such test cases.

B. PEA’s Diversity

As we select a group of the best solutions from a single population, concerns about the diversity of said solutions can arise. So, we evaluated the diversity by plotting a parallel coordinates visualization of each numeric parameter from the ten best solutions of a single execution for each difficulty setting. We normalized each value to the interval 0 to 1, using the minimum and maximum values presented for said parameter between the ten evaluated instances. We selected the 10 best instead of the 20 best used in the experiments with players to make the visualization easier to read.

The results presented in Fig. 3 show that for most parameters, there was a significant diversity, especially when increasing the difficulty. However, the health and damage parameters did not present much variety, especially on the easy setting because they have a larger impact on the fitness; i.e., when an enemy with two health points is one whole fitness unit (and, therefore, one difficulty unit) above another with one health point; but the other parameters are floating-point values and can have more subtle differences.

We believe that this diversity inside a single-population EA arose from the fact that we were able to use a very large population because of our parallel approach. Therefore, we had much more solutions spread in local optima across the search-space, even after a considerable amount of generations.

C. Experiment with Players

Our experiment with players allowed them to select between three difficulty settings: easy, with difficulty degree of 14; medium, with difficulty degree of 17.5; and hard, with difficulty degree of 22.5. We calculated these values with our difficulty function and determined empirically in previous tests with a smaller number of testers. For each difficulty setting, we executed the PEA with the presented parameters. Then, we saved the 20 best non-equal individuals (a total of 60 different enemies), aiming to provide enemy variety for the levels. In these executions, the largest observed difference from the target fitness for any given difficulty was 0.0845.

We uploaded our game prototype to a public server and asked on social networks for anyone interested in playing it. The game presented instructions about the experiment and gameplay. We ensured to follow protocols such as stating that the users were to remain calm, as they were not being evaluated, only the game itself was. We also stated we would be collecting gameplay data and asking them a questionnaire.

### Table V

<table>
<thead>
<tr>
<th>Input</th>
<th>Time</th>
<th>AVG Best</th>
<th>AVG</th>
<th>STD Best</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-10^2-10</td>
<td>0.168s</td>
<td>17.4</td>
<td>16.59</td>
<td>0.170</td>
<td>1.870</td>
</tr>
<tr>
<td>M-10^3-10</td>
<td>0.174s</td>
<td>17.60</td>
<td>16.57</td>
<td>0.017</td>
<td>1.872</td>
</tr>
<tr>
<td>M-10^4-10</td>
<td>0.175s</td>
<td>17.50</td>
<td>16.54</td>
<td>0.002</td>
<td>1.883</td>
</tr>
<tr>
<td>M-10^5-30</td>
<td>0.508s</td>
<td>17.50</td>
<td>16.79</td>
<td>0.011</td>
<td>1.914</td>
</tr>
<tr>
<td>M-10^6-30</td>
<td>0.512s</td>
<td>17.50</td>
<td>16.81</td>
<td>0.001</td>
<td>1.922</td>
</tr>
<tr>
<td>M-10^7-30</td>
<td>0.660s</td>
<td>17.50</td>
<td>16.81</td>
<td>0.000</td>
<td>1.922</td>
</tr>
<tr>
<td>M-10^8-10^2</td>
<td>1.706s</td>
<td>17.50</td>
<td>16.81</td>
<td>0.001</td>
<td>1.929</td>
</tr>
<tr>
<td>E-10^4-30</td>
<td>0.578s</td>
<td>14.00</td>
<td>14.01</td>
<td>0.001</td>
<td>1.619</td>
</tr>
<tr>
<td>H-10^6-30</td>
<td>0.512s</td>
<td>22.50</td>
<td>19.88</td>
<td>0.011</td>
<td>2.277</td>
</tr>
</tbody>
</table>
TABLE VI
EXPERIENCE QUESTIONNAIRE ON A 5-POINT LIKERT SCALE. Q1 THROUGH Q3 ARE ANSWERED ONLY ONCE PER PLAYER.

Question
Q1 - How would you best describe your overall experience with games (knowledge, playing time, skills, etc.)?
Q2 - How would you best describe your experience (knowledge, playtime, skill, etc.) with Action-Adventure games (e.g.: The Legend of Zelda, The Binding of Isaac, Darksiders, Uncharted, etc.)?
Q3 - What difficulty level do you usually choose to play your games in?
Q4 - How much do you agree with the statement: “This level was fun to play”?
Q5 - How much do you agree with the statement: “The combat against the enemies was hard”?
Q6 - How much do you agree with the statement: “The combat against the enemies’ difficulty was adequate to the selected difficulty level”?

at the end to answer if they felt like it, but we would not collect any personal data. All our respondents remained anonymous, as it was an opinion survey. Table VI presents the questionnaire we applied in our experiment. We also allowed them to quit at any time during the experiment section.

After reading this introduction, the player selects the difficulty level they would like to play (easy, medium, or hard), and the game starts. If they died, they could restart or quit the game, which would take them to our questionnaire screen. They could retry any number of times. After answering our questionnaire, the user could choose to leave the game or go back to the difficulty select screen and start over.

In terms of data collection, no personal data was collected. We identified the users only by their session starting time and a random id – it was not possible to check if any user came back in another session. We also collected implicit data such as if the player succeeded or failed to complete the level.

D. Players’ Feedback

A total of 16 players answered our questionnaires, but not all answered for all difficulties (Fig. 6). As we observe in Table VI, we divide the questions into two groups. The first is demographic data, which were answered only once per player, with the first three questions about the players’ experience with games and preferred difficulty. The second group is their actual answers about our game prototype and enemies created by the PEA. The first three questions are independent of the chosen difficulty setting and are shown in Fig. 4, on the left column (Fig. 4a, 4c, 4e). We observe that most players consider themselves very experienced with video-games, and reasonably experienced with Action-Adventure games. They also prefer the medium difficulty setting. The hard difficulty comes in second. Only a small sample was used to play in the easy, very easy, or very hard settings. Therefore, we expected most of them to play in the medium difficulty, which is confirmed in the next section.

In Fig. 4’s right column (Fig. 4b, 4d, 4f) we observe that players had the most fun in the easy and hard difficulties. We believe this occurred because players who do not enjoy very challenging games are happy with the easy level, while those who prefer challenges were entertained with the hard setting. It is important to note that the number of answers for each difficulty is different, as players could play at any difficulty, any number of times. For the medium setting, the
answers were mostly neutral, slightly positive. In this case, we suppose that the players did not evoke a strong feeling of joy nor challenge for most players, or some of the casual players tried these levels and faced more difficulty than expected (as we can confirm with the great loss rate in the next section). Even so, we can conclude that the overall answer was that the levels were indeed fun to play, disregarding difficulty.

Fig. 5 shows the players perception of difficulty for each setting on the left column (Fig. 5a, 5c, 5e). The easy difficulty was considered not challenging for most players (as expected), while the medium one had mixed reviews, tipping a little more for the opinion that it was difficult. Moreover, the hard difficulty was unanimously considered difficult. Those answers further confirm the ability of our PEA to generate a variety of enemies that also respect the input difficulty setting.

Lastly, even though the medium difficulty tipped a little to the harder side in the previous answers, Fig. 5 shows that most users agreed the difficulty was adequate to what they expected on the right column (Fig. 5b, 5d, 5f). The same was true for the remaining difficulties. Again, the medium one was a little more controversial, as it was probably a little harder than expected.

E. Players’ Performance

We collected the number of failed and succeeded attempts for each difficulty degree. Fig. 7 shows the results regarding the players’ performance. We observe that the medium level had the most attempts, reiterating the preference of users to play in such difficulty, as shown in Fig. 4. When comparing success rates, the easy difficulty was obviously the one with the greatest one, followed by the medium and the hard one. These results highlight that our algorithm can successfully create enemies with different difficulty settings, and these settings were progressively harder, as intended.

These data also show that our target fitness for each difficulty setting could be a bit lower to allow more players to
complete the game. It is usually intended for easier difficulties to allow victory in the majority of attempts. While our current medium fitness could be substituted for the easy one, allowing a little above half of the players to win. Then, the current medium difficulty could be used as the hard setting, as the majority of players failed, and the current hard difficulty as very hard, as only one player did win.
V. CONCLUSION

We presented a PEA able to evolve various enemies for an Action-Adventure game that closely matched the input difficulty degree. The convergence was fast enough that it could even fit loading times for many games, allowing us to create various enemies with similar difficulties in less than a second for most input configurations. We tested the quality of our solutions against real players through a game prototype where they selected their desired difficulty and faced many rooms filled with random sets of enemies created by our PEA using that difficulty as input.

Both explicit data – provided by the users answering a questionnaire – and implicit data – collected during their gameplay – showed that, overall, the difficulty of the generated enemies matched what was expected by the players and provided a fun and challenging gameplay. The major drawback observed was that the fitness for each difficulty could be better adjusted to other settings, as the perceived difficulty was a little above than we expected.

These very positive results tell us that it may be possible to create these enemies online. Besides, coupled with a Machine Learning algorithm or a similar approach, it may be feasible even adapting the enemies for players, making the game more difficult as the user gets a better game. Although our current setting may be a little too simple for demanding a PEA to search for the best enemies, as it is a preliminary experiment, it is done in a way that can be easily expanded. Dozens of new behaviors and weapons can be added with ease. These are numbers common in commercial games and would increase the complexity enough that the PEA would prove to be a viable alternative to other search-based algorithms.

Since the chosen fitness is the goal difficulty, our approach could also be used by human designers. We emphasize that although our algorithm demands an input fitness’ weights for the nominal enemies’ attributes, it still may be a much better alternative than creating enemies from scratch. Firstly, our algorithm can save much time for designers since the PEA creates new, feasible, and balanced enemies at every execution for the same parameters. Secondly, many of these weights have some leeway for wild-guessing, as the algorithm will still converge to the desired fitness, this being the most important value to fine-tune.

Our next steps are to add even more variety to the generated enemies to create more interesting gameplay, especially through the use of Quality-Diversity algorithms, which focus on guaranteeing the diversity while creating good solutions. We also want to test the limits of our PEA, and make the algorithm generate enemies in an online fashion as part of the game itself. After that, we plan to develop a tool to assist designers in the enemy creation process, releasing it as an asset for the Unity Engine.

REFERENCES


