Reproducing Bugs in Video Games using Genetic Algorithms

Tomás Ahumada
Department of Computer Science (DCC)
University of Chile
Santiago, Chile
tomyahu@gmail.com

Alexandre Bergel
Department of Computer Science (DCC)
University of Chile
Santiago, Chile
http://bergel.eu

***Abstract*—Video games are usually manually tested by a dedicated team. As such, testing is an expensive activity, both financial and emotional as most of the testing is mostly carried out before a release.**

This paper proposes a technique based on using Genetic Algorithm (GA) to reproduce bugs in video games. It consists in searching for a sequence of joystick and keyboard actions that lead to a faulty state of the game. We successfully applied our technique on two different video games, thus suggesting that using GA is a viable technique to reproduce bugs in video games.

***Index Terms*—Genetic Algorithms, Video Games, Testing**

I. Introduction

Modern video games are complex graphical software applications. These pieces of software are known to be difficult to test, because of their graphical and real-time aspect. As such, video games are difficult to automatically test by means of a dedicated testing framework.

Traditional methodologies for testing software usually cover small problems, consisting of uninterrupted sequences of actions. In the case of videogames, some examples of these small problems can be equipping an item, attacking an enemy or jumping on a specific type of surface.

However, many errors and glitches in games can only occur over a large number of frames. Intuitively, these software anomalies are harder to detect. These glitches can include unexpected physics behavior, getting out of the playable area of a level, frame rate drops and more. The videogame industry is facing this problem by having intense testing phases, usually involving human players.

Problem description. There are instances when a tester finds a bug while playing the game, without being able to reproduce it. Being able to accurately reproduce a bug is often an essential step to fixing it. This is the scope of this paper, proposing a technique to reproduce bugs in video games.

Objective. This paper demonstrates that genetic algorithms are effective at finding a sequence of input commands that lead to a faulty or erroneous state of a video game. Such input commands are a valuable asset when communicating bugs to a team of software engineers, who are in charge of implementing and maintaining a video game. There has been a small amount of studies that apply genetic algorithms to videogame testing, one of those studies is ICARUS [3].

Methodology. We employ an implementation of a genetic algorithm in Python in order to generate input commands for games written in the LÖVE framework [2], a popular framework to build video games in Lua [1].

We use our approach to identify bugs in the games *Journey to the Center of Hawkthorn* and *Zabuyaki*. Our findings were shared with the authors of these games.

Contributions. The contributions made in this paper are:

- Applying and evaluating genetic algorithm as a method for searching for input sequences that cause an abnormal behavior or resource consumption in games;
- Exploring and specifying the way input sequences may be encoded with genetic algorithms.

Our results and claims are accompanied by software artifacts supported in this paper, which are:

- A wrapper of the LÖVE framework for running predefined input sequences over games made in LÖVE.
- A piece of software that executes genetic algorithms over this wrapper, in order to generate input sequences that maximize a fitness function.

Outline. The paper is structured as follows: Section II presents the essence of genetic algorithms as a background; Section III describes our techniques to search for input sequences in video games; Section IV presents some case studies; Section V describes the experiments we have conducted to address our case studies; Section VI concludes and outlines future work.

II. Genetic Algorithms Overview

Genetic Algorithms (or GA for short) is a kind of optimization algorithm that is based on Darwinian Evolution and Natural Selection. GA operates by creating a number of random inputs for a function to be optimized. This technique evaluates each possible solution and combines relevant candidate solutions in order to create a new set of possible candidate solutions. This process is repeated until a stop condition is met and the best solution generated is returned as the output of the algorithm.

A. Terminology

A *fitness function* is defined as a function $f : I \rightarrow \mathbb{R}$ where I is the set of possible solutions for the problem to solve. This function is meant to be optimized by the GA, and it is used to evaluate all possible candidate solutions. Thus, this function is designed to quantify how close a solution is to the *optimal solution*, or how *good* the solution is, given the *problem specifications*.

An *individual* is defined as a possible solution $x \in I$. Each individual is composed of a set of *gene values*, which represent unitary pieces of information about the individual. An individual can be seen as a vector $x = (x_0, ...x_n) \in I = I_0 \times ...I_n$ where $x_i \in I_i$ is a gene value. A set of individuals is called a *population*.

A *generation* is defined as the creation of a new population in the genetic algorithm. At the beginning of the GA's execution, a set of individuals is created at random and it is defined as Generation 1. Generation $k+1$ is defined as the generation that results when mixing the selected set of individuals from generation k.

Crossover is the operation representing the combination of two individuals to create a new individual. This operation is performed by combining the genes of two individuals $x_1, x_2 \in I$ on the current generation called the *parents*. There are many ways to express a crossover operation but one of the most common is *Single Point Crossover*. In this algorithm, a random number $r \in 1, ..., n$ is selected; let $x, y \in I$ be the parents of the new individual $z \in I$. Let $a[i]$ be the i'th gene of individual a, then z's genes are defined as follows:

$$z[i] = \begin{cases} x[i] & i < r \\ y[i] & i \geq r \end{cases}$$

There is another operation that occurs during the creation of a new generation, called *mutation*. Mutation consists in altering an individual by replacing a gene value with another value. The mutation operation aims to bring diversity in the population.

Another crucial aspect of GA is selecting the individuals for a crossover operation. The *selection* consists in selecting two individuals from a population in order to apply the crossover and the mutation. Selection algorithms usually have a chance of selecting almost any individual of a generation. However the individuals with more chances to be selected are the ones with the best fitness.

III. Searching For Input Sequences

This paper is about evaluating the use of Genetic Algorithms (GA) to obtain information about bugs in video games. The essence of our approach is to use GA to search for a sequence of input values that leads to a faulty state of the game. Typically, each input value represents a simulated joystick action or a keystroke.

A. Modeling the Game Execution

Injecting computed inputs. To apply genetic algorithms to search for a particular input sequence, our fitness function involves injecting some computed inputs in the games. As such, the LÖVE engine is wrapped to redirect computed inputs to the game. These artificial inputs can be injected in any frame during execution.

Avoiding randomness. The game must be completely deterministic in order to effectively apply genetic algorithms. As such, feeding a sequence of generated inputs to two instances of the game must lead to exactly the very same state. When applied to GA, evaluating the fitness function twice for the same input values should result in the exact same fitness. This is an important requirement to have a possible convergence of the genetic algorithm.

Random number generators are used in the LÖVE engine and Lua's mathematical library. Our wrapper sets an arbitrary seed to guarantee the same sequence of pseudo-random values.

Targeting and speeding up the search. The genetic algorithm searches for an input sequence that leads to a game failure. To significantly improve the search, some indications may be manually provided to make the algorithm focus on a restricted portion of the game logic. In particular, the wrapper takes the following inputs when run to target the search:

1) *Frames to skip* – This parameter represents a number of frames at the start of the game execution that are not going to be taken into account for the fitness computation. Additionally, an input sequence can be provided to be run during these frames. This is done in order to skip uninteresting parts of the game like menu navigation, tutorials or levels.
2) *Frames to test* – The search immediately begins after having skipped the frames described above. The search has to focus on the relevant portion of the game, measured in frames. The value *frames to test* is the amount of frames the sequences generated by the GA have to reproduce the target behavior. After that, the game is aborted and the fitness is computed.

Our Wrapper first takes an input sequence and runs it for a number of frames given (the *frames to skip*), where the results will be ignored for fitness computation. When running the genetic algorithm every individual performs the exact same inputs during these frames, so the data obtained in this period is not important as it is the same for every individual in the GA.

After running the *frames to skip*, the fitness computation operates on the subsequent frames (the *frames to test*). An example of use is to perform a joystick or keyboard input sequence to skip menus and tutorials of a game. The value for the *frames to skip* variable is usually the length of the related input sequence in frames. For the value of the *frames to test* variable, it depends heavily on the experiment to do. Finally the resulting fitness alongside some execution data will be returned as a string that represents a dictionary in the JSON format.

B. Input Sequence Individual Encodings

We refer to an *input* as a signal given by a joystick or keyboard, which can represent a key press for example. Input *(up, 6),(no_input, 4),(down,8)*

Fig. 1. Example of the genes of an Ephemeral Key Individual.

sequences can be represented in a genetic algorithm in many different ways. This section describes the two encodings we have adopted after researching and iterating on many possible encodings.

Both of these encodings are composed of genes with the same structure. This gene type is called **Ephemeral Key Gene**, a gene that represents a tuple with an input type and the number of consecutive frames the input is active (represented as a whole number).

The first individual encoding is called **Ephemeral Key Individual** and consists of an ordered list of ephemeral key genes. The inputs of the genes of this list are sequentially injected in the game from frame 0 and onward. When a gene of this list stops being active, the next gene is activated and so on until every gene in the list was active at some point.

The individual in figure 1 represents an input sequence that activates the *up* joystick action for six frames (from Frame 1 to Frame 6), then no input is provided during the 4 subsequent frames (frames 7 to 10) and finally the *down* joystick action is performed during the 8 subsequent frames (frames 11 to 18). This simple encoding considers that only one input is provided at a given frame. However, many games expect more than one action to be performed at the same time (e.g., *up* and *right* to jump toward the right hand-side), which motivates for a second input encoding.

The second and more complex individual encoding is called **Multi Ephemeral Key Individual**, and unlike the first one, it *can* represent input sequences where many inputs are active at the same time. This encoding consists of a number of ordered groups of ephemeral key genes. These groups have in common that the sum of the frame duration of all their respective genes is the same.

Each one of these groups represents an input sequence of only one type of input. The input sequence represented by the encoding is the result of playing all the group's input sequences at the same time. Thus, to understand the complete input sequence it is only necessary to understand the input sequence generated by one of the groups.

These groups work in a similar manner as Ephemeral Key Individuals, but each time there is a change of active gene, the input is toggled (is activated if it was inactive at the time, and deactivated otherwise). Consider a group with the *up* input, whose sequence of genes is given by Figure 2.

The *up input* will start inactive, then after each gene finishes, the *up input* will be toggled. In this case the input will remain inactive for the first five frames of execution (frames 1 to 5), then it will be activated for three frames (frames 6 to 8); it will remain inactive for two more frames (frames 9 and 10) before being activated for eight frames at the end (frames 11 to 18).

A complete Multi Ephemeral Key Individual is shown in figure 3. In this figure, the first line represents a group for the

(up, 5),(up, 3),(up, 2),(up, 8)

Fig. 2. Example of a sequence of Ephemeral Key Genes from a group of a Multi Ephemeral Key Individual. In this case the input of the group corresponds to the *up input*.

up input, and the second line represents a group for the *right input*. This individual represents an input sequence that runs the sequences represented by both of these groups at the same time. By analyzing this sequence as with the one in figure 2, it can be observed that in frames 6 to 8 both the *up* and *right* inputs will be active at the same time. This is because the *up* input is active from frames 6 to 8, and the *right input is active* from frames 4 to 15.

(up, 5),(up, 3),(up, 2),(up, 8)
(right, 3),(right, 12),(right, 3)

Fig. 3. Example of a complete Multi Ephemeral Key Individual.

C. Encoding Mutation and Crossover

For performing crossover for the Multi Ephemeral Key Individuals encoding a variation of Single Point Crossover was designed. A random frame mark is selected during execution and the new individual is structured so it behaves in the same way as the first parent before the mark, and then behaves the same as the second parent after it (Figure 4).

For performing mutation, a random gene is selected and the length of its input is varied in a small amount of frames (a maximum of 10). After performing mutation, it is important to make sure that the sequence from the group selected lasts the same amount of frames as before; if the sequence lasts less than before, the last gene is extended. And if it lasts more than before, the sequence is cropped so it retains its original length.

IV. Case Studies

We applied genetic algorithms to reproduce bugs in two games: *Journey to the Center of Hawkthorn* [1], a platformer RPG (will be referred as Hawkthorn in the remaining of the paper); and *Zabuyaki* [2], a side-scrolling beat'em up. Hawkthorn was selected because it is an open source game that has public issues on GitHub. The game was developed in 4 years, from 2012 to mid 2016, it has 59 contributors on GitHub, and has more than 5,000 commits.

[1]https://projecthawkthorne.com/
[2]https://www.zabuyaki.com/

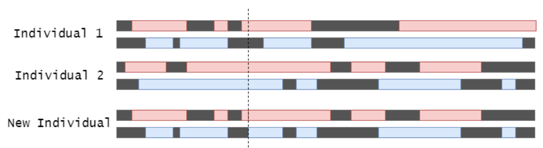

Fig. 4. A visualization of crossover between two Multi Ephemeral Key Individuals.

The first issue tackled was a glitch that allowed the player to go inside the roofs of some levels, called Issue #2456 [3]. What was known of the bug was that it can be performed by running to a place the player can crawl under, crouch briefly and attacking when coming out of crouch. It was discovered by github user LoubiTek in June 2015 and the way of reproducing it was found by github user niamu that same month.

Even though there is a way of reproducing the bug, more accurate details are unknown; like the frame window for reproducing it for example.

Zabuyaki is also an open source game in GitHub. Development in this game started mid 2016 and still continues at the time of writing this paper. The development team consists of 4 people and the repository has more than 6,000 commits. The game currently has two complete playable characters, Rick and Chai.

After contacting the developers, two issues were brought to the table, named by the developers as the D545 and D694 Tasks. The first task was about a two-part move of a playable character, Chai's special dash. The second task consisted of performing combos with both finished characters.

The D545 Task's problem is that sometimes when performing Chai's two-part moves, the first part will trigger and the second part will not. Another problem related to this task is that sometimes when performing the moves, the second part will trigger without the first part landing. These issues were last encountered in August 2018, and the developers were unsure if they still occur.

The D694 Task's problem is that when performing specific combos with Chai and Rick, the last move of the combo sometimes does not hit or do any damage. In a video sent by the developers portraying the issue it could be observed that the hitboxes of the move seemed to hit the target but no damage was done. The combos in particular are the following:

- Rick's Dash Attack Combo: combo1 → combo2 → dashAttack
- Chai's Dash Attack Combo: combo1 → combo2 → dashAttack
- Rick's Offensive Special Combo: combo1 → forwardCombo2 → forwardCombo3 → offensive Special

In every experiment in this section the genetic algorithm tested 30 generations with a population of 100 individuals, unless specified otherwise. Additionally, in every experiment we used Multi Ephemeral Key Individuals, as all the issues tackled required simultaneous inputs at some point.

V. Experiments

A. Hawkthorn: Issue #2456

There is a small amount of levels in the game where it is necessary to crawl to navigate. From those, there is an even smaller amount that are designed as tight spaces which is where this issue is said to occur. Three of these instances were found in the game: one in the *forest* level, other in the *black-caverns-2* level and one final in the *vents-2* level.

[3]https://github.com/hawkthorne/hawkthorne-journey/issues/2456

$$fitness = \sum_{f=1}^{n} k^f \, d_f$$

Fig. 5. Fitness function for reproducing Issue #2456 in Journey to the Center of Hawkthorn. n represents the frames to test, k is a constant value equal to 0.9954 and d_f is the distance between the player and the position that it has to get to on frame f. The k value between 0 and 1 defined as 0.9954 for the experiments.

The instance in the *forest* level consists of a short space where the player has to crawl to pass in a small cave. The instance in the *black-caverns-2* level consists of a moving platform that moves through a small space, the idea is to crouch in the moving platform to pass. Finally the instance in the level *vents-2* is very similar to the instance in *forest*, but the roof of the crawling space is connected to the ceiling of the level.

In order to reproduce this glitch, we designed a fitness function that rewards an individual for going as close to a defined in-game position as possible (Figure 5). For every site, a specific spot inside the wall over the crouch-able space was selected as the fitness function target position. The fitness function used is designed like this so individuals in the first generations try to get as close as possible at the beginning of the execution, and in later generations the last part of execution is optimized.

For these experiments some inputs were restricted as they represented noise to find the target behavior. The inputs enabled were the ones related to moving, jumping, crouching and attacking. The amount of frames to test an individual was set to 500 as it is known that it is possible to produce this issue in a fraction of this amount of frames.

For the second site, an input sequence was developed so individuals wait in the moving platform for a bit before starting to act. This is done so the genetic algorithm does not have to also learn to keep itself in the moving platform for some time. This sequence took 125 frames, so the frames to test were set to 375 frames in this particular case.

In the first site the glitch could not be reproduced, the genetic algorithm instead produced individuals that got as close as possible by crawling near the position. In the second site, the glitch was produced in Generation 1, the next generations the genetic algorithm tended to produce individuals that stayed as close as possible to the exact point inside the wall. For the final site, the genetic algorithm performed the glitch in Generation 4, and the rest of the generations showed a similar evolution that the second site experiment.

By analyzing the data from the experiments it was discovered that in site 2, the resulting individuals crouched, attacked, **and after attacking, came out of crouch**. In site 3 the produced individuals had to do some preparation to set up the glitch. After that, they performed the same actions as the individuals in site 2 to get to the target position.

After manually reducing the resulting output sequence from site 2, an input sequence of three inputs that produces the glitch was created. This sequence consists in pressing the

$$fitness = \frac{d}{8} + d_s + 10\, d_{sl2}(1 + 9\,(d_{sl2} - d_{sl}))$$

Fig. 6. No-Hit Fitness function. d represents the number of dash attacks performed. d_s the number of special dash attacks performed. d_{sl} the number of special dash attacks that landed. And d_{sl2} the number of times the second part of the special attack was triggered.

$$fitness = \frac{d}{8} + d_s + 10\, d_{sl}(1 + h_0)$$

Fig. 7. No-Damage Fitness function. d represents the number of dash attacks performed. d_s the number of special dash attacks performed. d_{sl} the number of special dash attacks that landed. And h_0 the number of frames the attack did 0 damage.

crouch button, then when the player character gets below the wall, pressing the attack button and before one third of a second passes, releasing the crouch button. The glitch causes the player character to go up until its not colliding into a collision box, which gets them inside the wall.

The resulting individuals were able to perform the glitch in two of the three sites. More information about the glitch was found, apparently when the player character is in crouch state and attacks, if the crouch button is released the player stands up without checking if it can stand up. More so, information on the frame window for releasing the crouch button was discovered.

The results do not imply that the glitch can not be performed on the first site though. It is believed that the glitch can be performed the same way as in site 3. The effects of the glitch will be different in site 1, because in site 3 the wall is connected to the roof, which is not true in site 1. By trying to perform the glitch in site 1 it is expected that the player character gets propelled upwards until it no longer collides with a wall.

B. Zabuyaki Task D545 Point 1 Chai's Special Dash

As mentioned before, this issue refers to when Chai's Special Dash's first part does not land, but the second part is triggered anyways.

Two fitness functions were designed in order to reproduce the error. As the target behavior is rather complex, the functions were designed so doing other similar moves also rewards an individual in a small manner. The first fitness function will be called *No-Hit Fitness Function* and can be observed in Figure 6. The second fitness function will be called *No-Damage Fitness Function* and it is defined in Figure 7.

Both functions were designed so individuals learn in a step-wise fashion how to land a special dash on an enemy. That is the reason behind rewarding them for performing dash attacks and special dash attacks without landing them. However the best way to obtain a high fitness value is to perform the move itself and the issue.

The *No-Damage Fitness Function* was designed based on the theory that the issue as observed in 2018 was that the special dash landed but then it did not produce any damage as the enemy may have moved out of the way. Once a special dash has landed, the genetic algorithm will attempt to minimize the amount of times the player character damages an enemy with the first part of the move.

$$fitness = \frac{d}{8} + d_s + 10\, d_{sl}(1 + 9\,(d_{sl1} - d_{s2}))$$

Fig. 8. No-Second Fitness function. d represents the number of dash attacks performed. d_s the number of special dash attacks performed. d_{sl} the number of special dash attacks that landed. And d_{sl2} the number of times the second part of the special attack was triggered.

For these experiments the amount of frames to test was set to 800 and the amount of generations to test was set to 20. Additionally, some parameters for individual initialization were set so individuals with many short inputs are more common than individuals with a few long ones.

In both experiments the genetic algorithm was able to perform the special dash attack on enemies without being able to fully reproduce the issue (a part of the bug was not reproduced). The No-Hits Fitness Function's experiments, produced individuals that landed up to three special dash attacks. For the No-Damage Fitness Function's experiments, the individuals managed to damage the enemy only once with the first part of the move.

C. Zabuyaki Task D545 Point 2 Chai's Special Dash

The objective of the experiments in this section is to perform and land the first part of Chai's Special Dash (on an enemy) without triggering the second part. Similar to Point 1 the function designed for this task rewards and individual in a small manner for performing similar moves as the target move while awarding a high fitness for performing the issue. This function will be called *No-Second Fitness Function* and its definition can be observed in Figure 8.

This fitness function is almost the same as the No-Hit Fitness Function from Point 1. The difference is in the last term of the function where it greatly rewards an individual for performing a special dash and even more for performing the issue.

These experiments also used 800 frames and 20 generations. The individuals initialization variables were set in the same manner as the previous experiment.

In this experiment, **the GA was able to reproduce the objective behavior** by generation 8. The first part of the move is an aerial kick, and the player character descends while performing it. In the execution of the genetic algorithm the player character touched the ground while doing this part and the rest of the move was not performed after it.

After sending a recording of this behavior to the developers, they confirmed that it corresponds to the behavior they were looking for. The use of the input sequence file created by the genetic algorithm led to the confirmation of a theory the developers had. This theory was "if the player touches the ground while performing the first part of the move, the second part will be cancelled".

D. Zabuyaki Task D694 Rick's Dash Attack Combo

As mentioned earlier the D694 Issue refers to combos performed by the finished characters where the last part of the

$$fitness = k\,(0.1\,c_1 + c_2 + \frac{c_{da}}{1+d_{da}})$$

Fig. 9. Rick's Dash Attack Fitness function. If the individual respects the constraints defined by the fitness function, k has the value 1, otherwise it becomes 0. c_1 takes the value of 1 if the first move of the combo was performed during execution and 0 otherwise. c_2 takes the value of 1 if the first and second moves of the combo was performed *as a combo* during execution and 0 otherwise. c_{da} takes the value of 1 if the objective combo is performed during execution and 0 otherwise. d_{da} is the total damaged done by the dash attack at the end of a combo.

combo does not hit. The first combo to do experimentation on is Rick's Dash Attack Combo.

For this task, we asked the developers for debug tools they had created to visualize hitboxes during playtime. This is because, as mentioned before, in a video sent by the developers the hitboxes seemed to hit when this issue happened. So it was necessary to observe the moves' hitboxes in order to verify whether the issues occurs. A new fitness function also had to be developed in order to reproduce this issue.

As the issue involves a complex behavior, it was not clear at the beginning of the experiments how to design a fitness function for this problem. The game does not keep track of combos internally, so detecting combos had to be done by the fitness function created. For this, a heuristic was developed in order to define what is a combo and when is a combo occurring.

A combo is defined as a sequence of moves executed sequentially where every move lands on a single target. Additionally, a sequence of moves will not be considered a combo if between each move's triggering times there are more than m_f frames (where the default value for m_f is 10).

The fitness function is defined based on *milestones*, rewarding individuals more as they get more steps of the combo. The initial number of generations set was 50, but was then reduced to 20 because most experiments tended to converge in the first generations. Also, the number of frames tested was set to 400, and the individuals initialization parameters were set so initial individuals have somewhat longer inputs than in the previous experiments.

We did many experiments in order to iterate on the fitness function's methods of parameter computation, constraints, m_f value, etc. During these experiments, the constraints "the individual must not kill an enemy" and "the individual can not turn around during the execution" were added. After 20 experiments, the issue was not reproduced.

In the last experiments the best individuals were able to reproduce the combo. In some experiments with an m_f value of 20 the individuals were able to manipulate the AI to move out of the way of the combo. As there is not a simple constraint that can be added to the fitness function in order to avoid this behavior, no more experiments were done for this instance of the issue.

E. Zabuyaki Task D694 Chai's Dash Attack Combo

This instance of the issue is about the same combo as before but executed as the character Chai.

As Rick's Dash Attack Fitness Function (Figure 9) produced individuals that reproduced the combo, the fitness function was maintained by changing the way it computed the parameters for the move. This included modifying the c_1, c_2, c_{da} and d_{da} to detect Chai's moves instead of Rick's. No further modifications were made to the fitness function for these experiments. For these experiments, three values for m_f were used: 10, 15 and 20. The number of generations was set to 50 and the frames to test were increased to 500.

For the first two values of m_f, the combo was performed without reproducing the issue, for the last value the issue was reproduced. In this last experiment, the genetic algorithm performed a different combo from the objective combo in order to reproduce the issue. The combo performed was:

- combo1 $\rightarrow$ combo2 $\rightarrow$ combo3Forward $\rightarrow$ dashAttack

The window between the two last moves of the original combo allowed the genetic algorithm to perform an extra move between them, producing the issue. The combo is performed on various enemies at once, all the three first moves of the new combo hit and the last one did not hit, despite the hitboxes overlap.

The developers were contacted about the reproduction of the issue and confirmed again that the behavior produced was the one they were looking for. Through analysis on the input sequence produced, it was determined that the bug was directly linked to the timing of the moves and it did not show signs of being connected to the distance from the target. Because the issue was reproduced, no more further experimentation was made with other moves.

VI. Conclusions and Future Work

The technique used in this paper for generating input sequences using genetic algorithms can be used to generate specific target behaviors in games. Based on the results obtained in the experiments, the difficulty of using this technique is to design a fitness function appropriate for generating behaviors. On this topic, fitness functions based on milestones like the ones used in Zabuyaki provide a structure on how to design them for behaviors that can be divided in steps.

A future research opportunity can be to experiment with the encodings developed and their crossover and mutation functions. The encodings used in the paper are internally structured as time-lapses. It could be interesting to see the development of effective encodings that do not have this structure.

References

[1] P Christensson. Lua definition. https://techterms.com/definition/lua. Accessed: 2020-03-25.

[2] Löve (game engine). Löve (game engine) - wikipedia. https://en.wikipedia.org/wiki/L%C3%B6ve_(game_engine). Accessed: 2020-03-25.

[3] Johannes Pfau, Jan Smeddinck, and Rainer Malaka. Intelligent completion of adventure riddles via unsupervised solving. In *Automated Game Testing with ICARUS*, pages 153–164, 10 2017.