

Using Reinforcement Learning for Load Testing of Video Games

Rosalia Tufano

SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Simone Scalabrino

STAKE Lab
University of Molise
Italy

Luca Pascarella

SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Emad Aghajani

SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Rocco Oliveto

STAKE Lab
University of Molise
Italy

Gabriele Bavota

SEART @ Software Institute
Università della Svizzera italiana
Switzerland

ABSTRACT

Different from what happens for most types of software systems, testing video games has largely remained a manual activity performed by human testers. This is mostly due to the continuous and intelligent user interaction video games require. Recently, reinforcement learning (RL) has been exploited to partially automate functional testing. RL enables training *smart* agents that can even achieve super-human performance in playing games, thus being suitable to explore them looking for bugs. We investigate the possibility of using RL for load testing video games. Indeed, the goal of game testing is not only to identify functional bugs, but also to examine the game’s performance, such as its ability to avoid lags and keep a minimum number of frames per second (FPS) when high-demanding 3D scenes are shown on screen. We define a methodology employing RL to train an agent able to play the game as a human while also trying to identify areas of the game resulting in a drop of FPS. We demonstrate the feasibility of our approach on three games. Two of them are used as proof-of-concept, by injecting artificial performance bugs. The third one is an open-source 3D game that we load test using the trained agent showing its potential to identify areas of the game resulting in lower FPS.

Also, games can hardly benefit from testing automation techniques [37], since even just exploring the total space available in a given game level requires an *intelligent* interaction with the game itself. For example, in a racing game, identifying a bug that manifests when the finish line is crossed requires a player able to successfully drive the car for the whole track (*i.e.*, requires the ability to drive the car). Thus, random exploration is not a viable option here.

Therefore, it comes without surprise that game testing is largely a manual process. Zheng *et al.* [51] report that 30 human testers were employed for testing one of the games used in their study. Also, the challenges in testing games have been stressed by Lin *et al.* [34], who showed that 80% of the 50 popular games they studied have been subject to urgent updates.

To support developers with game testing, researchers have proposed several techniques. These include approaches to test the stability of game servers (*e.g.*, by generating high packet loads) [19, 20, 30], model-based testing [29] using domain modeling for representing the game and UML state machines for behavioral modeling, as well as techniques specifically designed for testing board games [22, 42]. When looking at recent techniques aimed at proposing more general testing frameworks, those exploiting Reinforcement Learning (RL) are on the rise. This is due to the remarkable results achieved by RL-based techniques in playing games with super-human performance reported in the literature [14, 16, 27, 35, 36, 48].

RL is a machine learning technique aimed to train *smart* agents able to interact with a given environment (*e.g.*, a game) and to take decisions to achieve a goal (*e.g.*, win the game). RL is based on the simple idea of trial and error: The agent performs actions in the environment (of which it only has a partial representation) and receives a *reward* that allows it to assess its past actions/behavior with respect to the desired goal.

Recently, researchers started using RL not only to play games but also to test them and, in general, to improve their quality. The common idea behind these approaches is to reduce the human effort in playtesting (*i.e.*, the process of testing a new game to look for bugs before releasing it to the market) using intelligent agents. RL-based agents have been used to help game designers, for example, in balancing crucial parameters of the game (*e.g.*, power-up item effects, characters abilities) [38, 50, 52] and in testing the game difficulty [26, 44]. Also, RL-based agents have been used to look for bugs in games [15, 17, 39, 51].

1 INTRODUCTION

The video game market is expected to exceed \$200 billion in value in 2023 [12]. In such a competitive market, releasing high-quality games and, consequently, ensuring a great user experience, is fundamental. However, the unique characteristics of video games (from hereon, games) make their quality assurance process extremely challenging. Indeed, besides inheriting the complexity of software systems, games development and maintenance require a diverse set of skills covered by many stakeholders, including graphic designers, story writers, developers, AI (Artificial Intelligence) experts, etc.

While agents are usually trained to play a game with the goal of winning, the aforementioned works trained the agent to not only advance in the game but also to explore it to search for bugs. For example, Ariyurek *et al.* [15] combine RL and Monte Carlo Tree Search (MCTS) to find issues in the behavior of a game, given its design constraints and game scenario graph (provided by the game developer). The *ICARUS* framework [39] is able to identify crashes and blockers bugs (e.g., the game get stuck for a certain amount of time) while the agent is playing. Similarly, the approach by Zheng *et al.* [51], also exploiting RL, can identify bugs spotted by the agent during training (e.g., crashes).

While these approaches pioneered the use of RL for game testing, they are mostly aimed at testing functional (e.g., finding crashes) or design-related (e.g., level design) aspects. However, these are not the only types of bug developers look for in playtesting.

In a recent survey, Politowski *et al.* [40] reported that for two out of the five games they considered (i.e., *League of Legends* by Riot and *Sea of Thieves* by Rare) developers partially automated game performance checks (e.g., frame-rate). Similarly, Naughty Dog used specialized profiling tools¹ for finding which parts of a given scene caused a drop in the number of frames per second (FPS) in *The Last of Us*. Truelove *et al.* [46] report that game developers agree that *Implementation response* problems (among which, performance-related ones) may severely impact the game experience. Also, Li *et al.* [33] observed that players frequently complain about performance issues in game reviews.

Significance of research contribution. Despite such a strong evidence about the importance of detecting performance issues in video games, to the best of our knowledge no previous work introduced automated approaches for load testing video games. We present *RELINE* (Reinforcement lEarning for Load testINg gamEs), an approach exploiting RL to train agents able to play a given game while trying to load test it with the goal of minimizing its FPS. The agent is trained using a *reward* function enclosing two objectives: The first aims at teaching the agent how to advance in (and possibly win) the game. The second rewards the agent when it manages to identify areas of the game exhibiting low FPS. The output of *RELINE* is a report showing to developers the identified areas in the game being negative outliers in terms of FPS, accompanied by videos of the gameplays exhibiting the issue. Such “reports” can simplify the identification and reproduction of performance issues, that are often reported in open-source 3D games (see e.g., [1, 4, 6, 8]) and that, in some cases, are challenging to reproduce (see e.g., [3, 7]), even requiring special instructions for their reporting [5].

We experiment *RELINE* with three games. The first two are simple 2D games that we use as a proof-of-concept. In particular, we injected in the games artificial “performance bugs” [23] to check whether the agent is able to spot them. We show that the agent trained using *RELINE* can identify the injected bugs more often than (i) a random agent, and (ii) a RL-based agent only trained to play the game. Then, we use *RELINE* to load test an open-source 3D game [45], showing its ability to identify areas of the game being negative outliers in terms of FPS.

Code and data from our study are publicly available [47].

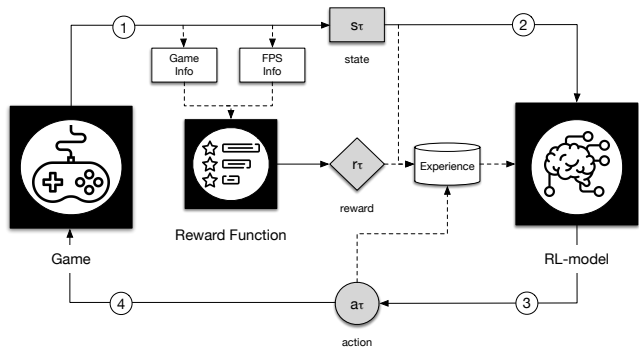


Figure 1: *RELINE* overview

2 RL TO LOAD TEST VIDEO GAMES

In this section we explain, from an abstract perspective, the idea behind *RELINE*. We describe in the study designs how we instantiated *RELINE* to the different games we experiment with (e.g., details about the adopted RL models).

RELINE requires three main components: the *game* to load test, a *RL model*, representing the agent that must learn how to play the game while load testing it, and a *reward function*, used to reward the agent so that it can evaluate the worth of its actions for reaching the desired goal (i.e., playing while load testing). The *RL model* is trained through the 4-step loop depicted in Fig. 1 (see the circled numbers). The continuous lines represent steps performed at each iteration of the loop, while the dashed ones are only performed after a first iteration has been run (i.e., after the agent performed at least one action in the game). When the first episode (i.e., a run of the game) of the training starts (step 1), at each time step τ the game provides its state s_τ . Such a state can be, for example, a set of frames or a numerical vector representing what is happening in the game (e.g., the agent’s position). The *RL model* takes as input s_τ (step 2) and provides as output the action a_τ to perform in the game (step 3). When the agent has no experience in playing the game at the start of the training, the weights of the neural network in the RL model are randomly initialized, producing random actions. The action a_τ is executed in the game (step 4), which, in turn, generates the subsequent state $s_{\tau+1}$.

After the first iteration (i.e., after having received at least one a_τ), the game also produces, at each iteration, the data needed to compute the reward function. In *RELINE* we collect (i) the information needed to assess how well the agent is playing the game (e.g., time since the episode started and the episode score), and (ii) the FPS at time τ . It is required that the game developer instruments the game and provide APIs through which *RELINE* can acquire such pieces of information. We assume that this requires a minor effort.

The *reward function* aims at training an agent that is able to (i) play the game, thanks to the information indicating how well the agent is playing, and (ii) identify low-FPS areas, thanks to the information about the FPS. The output of the *reward function* is a number representing the reward obtained by the agent at time τ . In *RELINE*, the reward function for a given action is composed of two sub-functions: A *game reward function*, depending on how good the action is in the game (rg_τ), and a *performance reward function*, depending on how the action impacts the performance (rp_τ).

¹<https://youtu.be/yH5MgEbB0ps?t=3494>

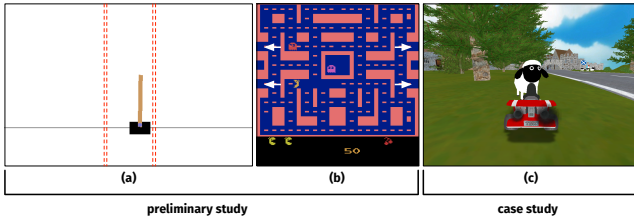


Figure 2: Screenshots of games used in the preliminary study—Section 3 (a) CartPole and (b) MsPacman, and in the case study—Section 4 (c) SuperTuxKart.

We combine such functions in $r_\tau = rg_\tau + rp_\tau$. The game reward function clearly depends on the game under test: A function designed for a racing game likely makes no sense for a role-playing game. In general, defining the reward function for learning to play should be performed by considering (i) what the goal of the game is (e.g., drive on a track), and (ii) which information the game provides about the “successful behavior of the player” (e.g., is there a score?). Even if less intuitive, the performance reward function is game-dependent as well: Assuming a tiny FPS drop (e.g., -1%), the reward can be small for a role-playing game, in which it likely does not affect the whole experience, while it should be high for an action game, in which it could even cause the (unfair) player’s defeat. Unlike the game reward function, we expect however minor changes to be required to adapt the performance reward function to a different video game (i.e., tuning of the thresholds to use).

The state s_τ , the action a_τ , and the reward r_τ are then stored in an experience buffer. When enough experience has been accumulated, it is used to update the network weights. How experience is stored and used to update the network depends on the used RL model.

The episode ends when a final state is reached. Again, the definition of the final state depends on the game, and it could be based on a timeout (e.g., each episode lasts at most 90 seconds) or on a specific condition that must be met (e.g., the agent crosses the finish line). Once the episode ends, the game is reinitialized and the loop restarts. The training is performed for a number of episodes sufficient to observe a convergence in the total reward achieved by an agent during an episode (e.g., if the trained agent obtains a reward of 100 for ten consecutive episodes the training is stopped).

3 PRELIMINARY STUDY: INJECTING ARTIFICIAL PERFORMANCE ISSUES

This preliminary study aims at assessing the ability of *RELINE* in identifying artificial “performance bugs” [23] we simulate in two 2D games. It is important to highlight that the *goal* of this study is only to demonstrate the applicability of *RELINE* for load testing games as a proof-of-concept. A case study on a 3D open-source game is presented in Section 4.

3.1 Study Design

We select two 2D games, CartPole [2] and MsPacman [10]. The former — Fig. 2-(a) — is a dynamic system in which an unbalanced pole is attached to a moving cart, and the player must move the cart to balance the pole and keep it in a vertical position.

The player loses if the pole is more than 12 degrees from vertical or the cart moves too far from the center. The latter — Fig. 2-(b) — is the classic Pac-Man game in which the goal is to eat all dots without touching the ghosts. Both games employ simple 2D graphics which bound the player’s possible moves in only one (e.g., left and right, for CartPole) or two (e.g., left, right, up, and down, for MsPacman) dimensions. This is one of the reasons we selected these games for assessing whether a RL-based agent that learned how to play them can also be trained to look for artificial “performance bugs” we injected. Also, both games are integrated in the popular GYM Python toolkit [9] developed by OpenAI [18].

GYM can be used for developing and comparing RL-based agents in playing games. It acts as a middle layer between the *environment* (the game) and the *agent* (a virtual player). In particular, GYM collects and executes *actions* (e.g., go left, go right) generated by the agent and returns to it the new state of the *environment* (i.e., screenshots) with additional information such as the score in the episode. GYM comes with a set of integrated arcade games including the two we used in this preliminary study.

3.1.1 Bug Injection. We injected two artificial “performance bugs” in CartPole and four in MsPacman. The idea behind them is simple: When the agent visits specific areas for the first time during a game, the bugs reveal themselves (simulation of heavy resource loading). A natural way of achieving this goal would have been to introduce the bugs in the source code of the game and to implement the logic to spot FPS drops in the agent accordingly. This, however, would have slowed down the training of the agent. Therefore, we chose to use a more practically sound approach, inspired by the simulation of Heavy-Weight Operation (HWO) operator for performance mutation testing [23]: We directly assume that the agents observe the bugs when they visit the designated areas and act accordingly.

In CartPole, the agent can only move on the x axis (i.e., left or right). When the game starts, the agent is in position $x = 0$ (i.e., center of the axis) and it can change its position towards positive (by moving right) or negative (left) x values. The two bugs we injected manifest when $x \in [-0.50, -0.45]$ and $x \in [0.45, 0.50]$ — dashed lines in Fig. 2-(a). We use intervals rather than specific values (e.g., -0.45) because the position of the agent is a float: if it moves to position -0.450001, we want to reward it during the training for having found the injected bug. Concerning MsPacman, we assume that a performance bug manifests when the agent enters the four *gates* indicated by the white arrows in Fig. 2-(b).

As detailed in Section 3.1.4, we assess the extent to which *RELINE* is able to identify the bugs we injected while playing the games. To have a baseline, we compare its results with those of a RL-based agent only trained to play each of the two games (from hereon, *rl-baseline*), and with a *random agent*. Since *RELINE* will be trained with the goal of identifying the bugs (details follow), we expect it to adapt its behavior to not only successfully play the game, but to also exercise more often the “buggy” areas of the games.

3.1.2 Learning to Play: RL Models and Game Reward Functions. We trained the *rl-baseline* agent (i.e., the one only trained to learn how to play) for CartPole using the *cross-entropy method* [41] as RL model. We choose this method because, despite its simplicity, it has been shown to be effective in applications of RL to small environments such as CartPole [31].

The core of the cross-entropy method is a feedforward neural network (FNN) that takes as input the state of the game and provides as output the action to perform. The state of the game for CartPole is a vector of dimension 4 containing information about the x coordinate of the pole’s center of mass, the pole’s speed, its angle with respect to the platform, and its angular speed. There are two possible actions: go right, go left. Once initialized with random weights, the agent (*i.e.*, the FNN) starts playing while retaining the experience acquired in each episode: The experience is represented by the state, the action, and the reward obtained during each step of the episode.

The goal is to keep the pole in balance as long as possible or until the maximum length of an episode (that we set to 1,000 steps) is reached. The *game reward function* is defined so that the agent receives a +1 reward for each step it manages to keep the pole balanced. The total score achieved is also saved at the end of each episode. After $n = 16$ consecutive episodes the agent stops playing, selects the $m = 11$ (70%) episodes having the highest score, and uses the experience in those episodes to update the weights of the FNN (n and m have been set according to [31]).

Instead, we trained the *rl-baseline* agent for MsPacman using a Deep Q Network (DQN) [35]. In our context, a DQN is a Convolutional Neural Network (CNN) that takes as input a set of contiguous screenshots of the game (in our case 4, as done in previous works [35, 36]) representing the state of the game and returns, for each possible action defined in the game (five in this case: go up, go right, go down, go left, do nothing), a value indicating the expected reward for the action given the current state (Q value). The multiple screenshots are needed to provide more information to the model about what is happening in the game (*e.g.*, in which direction the agent is moving). The goal of the DQN is the same as the FNN: selecting the best action to perform to maximize the reward given the current state. Differently from the previous model, the DQN is updated not on entire episodes but by randomly batching “experience instances” among 10k steps saved during the most recent episodes. An “experience instance” is saved after each step τ , and is represented by the quadruple $(s_{\tau-1}, a_{\tau}, s_{\tau}, r_{\tau})$, where $s_{\tau-1}$ is the input state, a_{τ} is the action selected by the agent, s_{τ} is the resulting state obtained by running a_{τ} in $s_{\tau-1}$ and r_{τ} is the received reward.

The CNN is initialized with random weights, and the agent starts playing while retaining the experience of each step. When enough experience instances have been collected (10k in our implementation [31]), the CNN starts updating at each step selecting a random batch of experience instances. The reward function for MsPacman provides a +1 reward every time the agent eats one of the dots and a 0 reward otherwise.

3.1.3 Instantiating RELINE: Performance Reward Functions. To train RELINE to play while looking for the injected bugs, we use a simple *performance reward function*: In both the games, we give a reward of +50 every time the agent, during an episode, spots one of the injected artificial bugs. As previously mentioned, the bugs reveal themselves only the first time the agent visits each buggy position; this means that the performance-based reward is given at most twice for CartPole and four times for MsPacman.

3.1.4 Data Collection and Analysis. We compare RELINE against the two previously mentioned baselines: *rl-baseline* and the *random agent*. Both RELINE and *rl-baseline* have been trained for 3,200 episodes on CartPole and 1,000 on MsPacman. The different numbers are due to differences in the games and in the RL model we exploited. In both cases, we used a number of episodes sufficient for *rl-baseline* to learn how to play (*i.e.*, we observed a convergence in the score achieved by the agent in the episodes).

Once trained, the agents have been run on both games for additional 1,000 episodes, storing the performance bugs they managed to identify in each episode. Since different trainings could result in models playing the game following different strategies, we repeated this process ten times. This means that we trained 10 different models for both RELINE and *rl-baseline* and, then, we used each of the 10 models to play additional 1,000 episodes collecting the spotted performance bugs. Similarly, we executed *random agent* 10 times for 1,000 episodes each. In this case, no training was needed.

We report descriptive statistics (mean, median, and standard deviation) of the number of performance bugs identified in the 1,000 played episodes by the three approaches. A high number of episodes in which an approach can spot the injected bugs indicate its ability to look for performance bugs while playing the game.

3.2 Preliminary Study Results

Table 1 shows for each of the two games (CartPole and MsPacman) the number k of artificial bugs we injected and, for each of the three techniques (*i.e.*, RELINE, *rl-baseline*, and the *random agent*), descriptive statistics of the number of episodes (out of 1,000) they managed to identify at least n of the injected bugs, with n going from 1 to k at steps of 1.

For both games, it is easy to see that the *random agent* is rarely able to identify the bugs. Indeed, this agent plays without any strategy as it is able to identify bugs only by chance in a few episodes out of the 1,000 it plays. This is also due to the fact that the *random agent* quickly loses the played episodes due to its inability to play the game. This confirms that these approaches are not suitable for testing video games.

Concerning CartPole, both RELINE and *rl-baseline* are able to spot at least one of the two bugs in several of the 1,000 episodes. The median is 984 for RELINE and 706 for *rl-baseline*. The success of *rl-baseline* is soon explained by the characteristics of CartPole: Considering where we injected the bugs — see Fig. 2(a) — by playing the game it is likely to discover at least one bug (*e.g.*, if the player tends to move towards left, the bug on the left will be found). What it is instead unlikely to happen by chance is to find both bugs within the same episode. We found that it is quite challenging, even for a human player, to move the cart first towards one side (*e.g.*, left) and, then, towards the other side (right) without losing due to the pole moving more than 12 degrees from vertical. As it can be seen in Table 1, RELINE succeeds in this, on average, for 102 episodes out of 1,000 (median 47), as compared to the 5 (median 1) of *rl-baseline*. This indicates that RELINE is pushed by the reward function to explore the game looking for the injected bugs, even if this makes playing the game more challenging. Similar results have been achieved on MsPacman.

Table 1: Number of episodes (out of 1,000) in which *RELINE*, *rl-baseline*, and the *random agent* identify the injected bugs.

| Game | #Injected Bugs | #Bugs found | <i>RELINE</i> | | | <i>rl-baseline</i> | | | <i>random agent</i> | | |
|----------|----------------|-------------|---------------|--------|-------|--------------------|--------|-------|---------------------|--------|-------|
| | | | mean | median | stdev | mean | median | stdev | mean | median | stdev |
| CartPole | 2 | 1 | 965 | 984 | 47 | 715 | 706 | 107 | 12 | 11 | 4 |
| | | 2 | 102 | 47 | 177 | 5 | 1 | 7 | 0 | 0 | 0 |
| MsPacman | 4 | 1 | 971 | 989 | 59 | 700 | 680 | 228 | 24 | 23 | 5 |
| | | 2 | 966 | 985 | 63 | 356 | 343 | 169 | 17 | 16 | 3 |
| | | 3 | 914 | 941 | 87 | 114 | 80 | 90 | 1 | 1 | 1 |
| | | 4 | 879 | 907 | 106 | 25 | 23 | 17 | 1 | 1 | 1 |

In this case, the DQN is effective in allowing *RELINE* to play while exercising the points in the game in which we injected the bugs. Indeed, on average, *RELINE* was able to spot all four injected bugs in 879 out of the 1,000 played episodes (median=907), while *rl-baseline* could achieve such a result only in 25 episodes.

Summary of the Preliminary Study. *RELINE* allows obtain agents able not only to effectively play a game but also to spot performance issues. Compared to *rl-baseline*, the main advantage of *RELINE* is that it identifies bugs more frequently while playing.

4 CASE STUDY: LOAD TESTING AN OPEN SOURCE GAME

We run a case study to experiment the capability of *RELINE* in load testing an open-source 3D game. Differently from our preliminary study (Section 3), we do not inject artificial bugs. Instead, we aim at finding parts of the game resulting in FPS drops.

4.1 Study Design

For this study, we use a 3D kart racing game named *SuperTuxKart* [45] – see Fig. 2-(c). This game has been selected due to the following reasons. First, we wanted a 3D game in which, as compared to a 2D game, FPS drops are more likely because of the more complex rendering procedures. Second, *SuperTuxKart* is popular open-source project that counts, at the time of writing, over 3k stars on GitHub. Third, it is available an open-source wrapper that simplifies the implementation of agents for *SuperTuxKart* [11].

The existence of a wrapper like the one we used is crucial since it allows, for example, to advance in the game frame by frame (thus simplifying the generation of the inputs to the RL model), to execute actions (e.g., throttle or brake), and to acquire game internals (e.g., kart centering, distance to the finish line). Also, using this wrapper, it is possible to compute the time needed by the game to render each frame and, consequently, calculate the FPS. Finally, the wrapper allows to have simplified graphics (e.g., removing particle effects, like rain, that could make the training more challenging).

4.1.1 Learning to Play: RL Models and Game Reward Functions. The training of the *rl-baseline* agent has been performed using the DQN model previously applied in *MsPacman*.

We use the previously mentioned *PySuperTuxKart* [11] to make the agent interact with the game. For the sake of speeding up the training, the screenshots extracted from the game have been resized to 200x150 pixels and converted in grayscale before they are provided as input to the model. Moreover, as previously done for *MsPacman*, multiple (four) screenshots are fed to the model at each step. Thus, the representation of the state of the game provided to the model is a 4x200x150 tensor. The details of the model and its implementation are available in our replication package [47].

A critical part of the learning process is the definition of the *game reward function*. Being *SuperTuxKart* a racing game, an option could have been to penalize the agent for each additional step required to finish the game. Consequently, to maximize the final score, the agent would have been pushed to reduce the number of steps and, therefore, to drive as fast as possible towards the finish line. However, considering the non-trivial size of the game space, such a reward function would have required a long training time. Thus, we took advantage of the information that can be extracted from the game to help the agent in the learning process.

SuperTuxKart provides two coordinates indicating where the agent is in the game: `path_done` and `centering`.

The former indicates the path traversed by the agent from the starting line of the track, while the latter represents the distance of the agent from the center of the track. In particular, `centering` equals 0 if the agent is at the center of the track, and it moves away from zero as the agent moves to either side: going towards right results in positive values of the `centering` value, going left in negative values. We indicate these coordinates with x (`centering`) and y (`path_done`), and we define δ_y as the path traversed by the agent in a specific step: Given y_i the value for `path_done` at step i , we compute δ_y as $y_i - y_{i-1}$. Basically, δ_y measures how fast the agent is advancing towards the finish line.

Given x and δ_y for a given step i , we compute the reward function as follows:

$$rg_i = \begin{cases} -1 & \text{if } |x| > \theta \\ \max(\min(\delta_y, M), 0) & \text{otherwise} \end{cases}$$

First, if the agent goes too far from the center of the track ($|x| > \theta$), we penalize it with a negative reward. Otherwise, if the agent is close to the center ($|x| \leq \theta$), we can have two scenarios: (i) if it is not moving towards the finish line ($\delta_y \leq 0$), we do not give any reward (the minimum reward is 0); (ii) if it is moving in the right direction ($\delta_y > 0$), we give a reward proportional to the speed at which it is advancing (δ_y), up to a maximum of M .

In our experimental setup, we set $\theta = 20$ because it roughly represents the double of $|x|$ when the agent approaches the sides of the road in the level, and $M = 10$ as it is the same maximum reward also given by the *performance reward function*, as we explain below. Finally, we reward the agent when it crosses the finish line with an additional +1,000 bonus.

4.1.2 Instantiating RELINE: Performance Reward Function. To define the *performance reward function* of RELINE for SuperTuxKart, the first step to perform is to define a way to reliably capture the FPS of the game during the training. In this way, we can reward the agent when it manages to identify low-FPS points. As previously said, we use PySuperTuxKart to interact with the game and such a framework keeps the game frozen while the other instructions of RELINE (e.g., the identification of the action to execute) are run. Since the framework runs the game in the same process in which we run RELINE and since we do not use threads, we can safely use a simple method for computing the time needed to render the four frames: We get the system time before (T_{before}) and after (T_{after}) we trigger the rendering of the frames and we compute the time needed at step i as $rT_i = T_{after} - T_{before}$. Such a value is negatively correlated with the FPS (higher rendering time means lower FPS).

The *performance reward function* we use is the following:

$$rp_i = \begin{cases} 10 & \text{if } |x| \leq \theta \wedge rT_i > t \\ 0 & \text{otherwise} \end{cases}$$

We give a performance-based reward of 10 when the agent takes more than t milliseconds to render the frames at a given point (causing an FPS drop). We explain the tuning of t later. We do not give such a reward when $|x| > \theta$ (the kart is far from the center) since we want the agent to spot issues in positions that are likely to be explored by real players (i.e., reasonably close to the track).

Finally, in RELINE we do not give a fixed +1,000 bonus reward when the agent crosses the finish line but we assign a bonus computed as $10 \times \sum_{i=1}^{steps} rp_i$, i.e., proportional to the total performance-based reward accumulated by the agent in the episode. This is done to push the agent to visit more low-FPS points during an episode.

4.1.3 Data Collection and Analysis. As done in our preliminary study, we compare RELINE with *rl-baseline* (i.e., the agent only trained to play the game) and with a *random agent*.

Training *rl-baseline* and RELINE. While we used different reward functions for the two RL agents, we applied the same training process for both of them. We trained each model for 2,300 episodes, with one episode having a maximum duration of 90 seconds or ending when the agent crosses the finish line of the racing track (the agent is required to perform a single lap). We set the 90 seconds limit since we observed that, by manually playing the game, ~70 seconds are sufficient to complete a lap. The 2,300 episodes threshold has been defined by computing the average reward obtained by the two agents every 100 episodes and by observing when a plateau was reached by both agents. We found 2,300 episodes to be a good compromise for both agents (graphs plotting the reward function are available in the replication package [47]).

The trained *rl-baseline* agent has been used to define the threshold t needed for the RELINE’s reward function (i.e., for identifying when the agent found a low-FPS point and should be rewarded).

In particular, once trained, we run *rl-baseline* for 300 episodes, storing the time needed by the game to render the subsequent four frames after every action recommended by the model.² This resulted in a total of 48,825 data points s_{FPS} , representing the standard FPS of the game in a scenario in which the player is only focused on completing the race as fast as possible.

Starting from the 48,825 s_{FPS} data points collected in the 300 episodes played by the trained *rl-baseline* agent, we apply the five- σ rule [24] to compute a threshold able to identify outliers. The five- σ rule states that in a normal distribution (such as s_{FPS}) 99.99% of observed data points lie within five standard deviations from the mean. Thus, anything above this value can be considered as an outlier in terms of milliseconds needed to render the frames. For this reason, we compute $t_b = \text{mean}(s_{FPS}) + 5 \times \text{sd}(s_{FPS})$ as a candidate base threshold to identify low-FPS points. However, t_b cannot be directly used as the t value of our reward function. Indeed, we observed that the time needed for rendering frames during the RELINE’s training is slightly higher as compared to the time needed when the trained *rl-baseline* agent is used to play the game. This is due to the fact that the load on the server (and in particular on the GPU) is higher during training. To overcome this issue, we perform the following steps.

At the beginning of the training, we run 100 *warmup episodes* in which we collect the time needed to render the four frames after each action performed by the agent. Then, we compute the first (Q_1^{tr}) and the third (Q_3^{tr}) quartile of the obtained distribution and compare them to the Q_1 and Q_3 of the distribution obtained in the 300 episodes used to define t_b (i.e., those played by the trained *rl-baseline* agent). During the *warmup episodes*, the agent selects the action to perform almost randomly (it still has to learn): Therefore, it would not be able to explore a substantial area of the game (i.e., of the racing track), thus not providing a distribution of timings comparable with the ones obtained when the trained *rl-baseline* agent that played the 300 episodes. For this reason, during the 100 *warmup episodes* of the training, the action to perform is not chosen by the agent currently under training, but by the trained *rl-baseline* agent (i.e., the same used in the 300 episodes). This does not impact in any way the load on the server that remains the one we have during the training of RELINE since the only change we have is to ask for the action to perform to the *rl-baseline* agent rather than to the one under training. However, the whole training procedure (e.g., capturing the frames and updating the network) stays the same.

We compute the additional “cost” brought by the training in rendering the frames during the game using the formula $\delta = \max(Q_1^{tr} - Q_1, Q_3^{tr} - Q_3)$. We use the first and third quartiles since they represent the boundaries of the central part of the distribution, i.e., they should be quite representative of the values in it. We took as δ the maximum of the two differences to be more conservative in assigning rewards when the agent identifies low-FPS points. The final value t we use in our reward function when training RELINE to load test SuperTuxKart is defined as: $t = t_b + \delta = 18.36$.³

²Since we wanted to measure the frames rendering time in a standard scenario in which the agent was driving the kart, we stopped an episode if the agent got stuck against some obstacle.

³We identify as low-FPS points the ones in which the FPS is lower than 218. Such a number is still very high, more than enough for any human player, in practice. Note that we run the game using high-performance hardware and, most importantly, with the lowest graphic settings. The equivalent in normal conditions would be much lower.

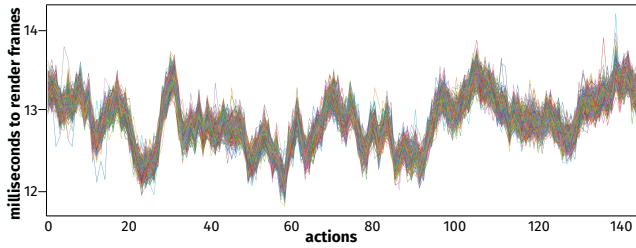


Figure 3: Rendering times for 300 episodes (same actions).

Thus, if *RELINE* is able, during the training, to identify a point in the game requiring more than t milliseconds to render four frames, then it receives a reward as explained in Section 4.1.2.

The training of *rl-baseline* took ~ 3 hours, while *RELINE* requires substantially more time due to the fact that, after each step performed by the agent, we collect and store information about the time needed to render the frames (this is done million of times). This pushed the training for *RELINE* up to ~ 30 hours.

Reliability of Time Measurements. It is important to clarify that the FPS of the game can be impacted by the hardware specifications and the current load of the machine running it. In other words, running the same game on two different machines or on the same machine in two different moments can result in variations of the FPS. For this reason, all the experiments have been performed on the same server, equipped with 2 x 64 Core AMD 2.25GHz CPUs, 512GB DDR4 3200MHz RAM, and an nVidia Tesla V100S 32GB GPU. Also, the process running the training of the agents or the collection of the 48,825 s_{FPS} with the trained *rl-baseline* agent was the only process running on the machine besides those handled by the operating system (Ubuntu 20.04). On top of that, the process was always run using the `chrt -rr 1` option, that in Linux maximizes the priority of the process, reducing the likelihood of interruptions.

Despite these precautions, it is still possible that variations are observed in the FPS not due to issues in the game, but to external factors (e.g., changes in the load of the machine). To verify the reliability of the collected FPS data, we run a constant agent performing always the same actions in the game for 300 episodes. The set of actions has been extracted from one of the episodes played by the *rl-baseline* agent, that was able to successfully conclude the race. Then, we plotted the time needed by the game to render the four frames following each action made by the agent. Since we are playing 300 times exactly the same episode, we expect to observe the same trend in terms of FPS for each game. If this is the case, it means that the way we are measuring the FPS is reliable enough to reward the agent when low-FPS points are identified.

Fig. 3 shows the achieved results: The y -axis represents the milliseconds needed to render four frames in response to an agent’s action (x -axis) performed in a specific part of the game. While, as expected, small variations are possible, the overall trend is quite stable: Points of the game requiring longer time to render frames are consistently showing across the 300 episodes, resulting in a clear trend. We also computed the Spearman’s correlation [43] pairwise across the 300 distributions, adjusting the obtained p -values using the Holm’s correction [28].

We found all correlations to be statistically significant (adjusted p -values < 0.05) with a minimum $\rho=0.77$ (strong correlation) and a median $\rho=0.91$ (very strong correlation). This confirms the common FPS trends across the 300 episodes.

Running the Three Techniques to Spot Low-FPS Areas. After the 2,300 training episodes, we assume that both the RL-based agents learned how to play the game, and that *RELINE* also learned how to spot low-FPS points. Then, as also done in our preliminary study, we train both agents for additional 1,000 episodes, storing the time needed to render the frames in every single point they explored during each episode (where a point is represented by its coordinates, i.e., $centering=x$ and $path_done=y$). We do the same also with the *random agent*.

Data Analysis. The output of each of the three agents is a list of points with the milliseconds each of them required to render the subsequent frames. Since each agent played 1,000 episodes, it is possible that the same point is covered several times by an agent, with slightly different FPS observed (as previously explained, small variations in FPS are possible and expected across different episodes). We classify as low-FPS points those that required more than t milliseconds to render the four subsequent frames more than 50% of times they have been covered by an agent.

This means that, if across the 1,000 episodes a point p is exercised 100 times by an agent, at least 51 times the threshold t must be exceeded to consider p as a low-FPS point. In practice, a developer using *RELINE* for identifying low-FPS points could use a higher threshold to increase the reliability of the findings. However, for the sake of this empirical study, we decided to be conservative.

Then, we compare the characteristics of the low-FPS points identified by the three approaches. Specifically, we analyze: (i) how many different low-FPS points each approach identified; (ii) the number of times each low-FPS point has been exercised by each agent in the 1,000 episodes; (iii) the *confidence* of the identified points (i.e., the percentage of times an exercised point resulted in low FPS). Given the low-FPS points identified by each agent, we draw violin plots showing the distribution of timings needed to render the frames when the agent exercised them (the higher the timings, the lower the FPS). We compare these distributions using Mann-Whitney test [21] with p -values adjustment using the Holm’s correction [28]. We also estimate the magnitude of the differences by using the Cliff’s Delta (d), a non-parametric effect size measure [25] for ordinal data. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [25].

4.2 Study Results

Fig. 4 summarizes the main findings of our case study. Fig. 4-(a) shows the distribution of time needed to render the game frames (i.e., our proxy for FPS) for four groups of points. The first violin plot on the left (i.e., Regular FPS) shows the timing for points that have never resulted in a drop of FPS in any of the 3,000 episodes played by the three agents (1,000 each). These serve as baseline to better interpret the low-FPS points exercised by the agents. The other three violin plots show the distributions of timing for the low-FPS points identified by *RELINE* (blue), *rl-baseline* (green), and the *random agent* (red).

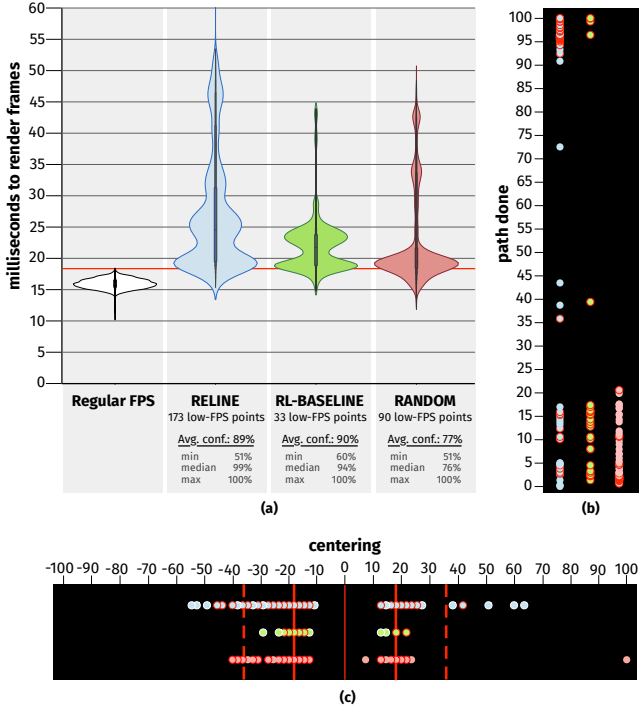


Figure 4: Results of the study: (a) reports the distributions of timings for the low-FPS points with summary statistics, while (b) and (c) depict the *path done* and *centering* coordinates at which the such points were observed, respectively.

Below each violin plot we report the number of low-FPS points identified by each agent and descriptive statistics (average, median, min, max) of the confidence for the low-FPS points. A 100% confidence means that all times that a low-FPS point has been exercised in the 1,000 episodes played by the agent it required more than $t = 18.36$ milliseconds to render the subsequent frames. The t threshold is represented by the red horizontal line. On average, *RELINE* exercised each low-FPS point 89 times in the 1,000 episodes, against the 210 of *rl-baseline* and the 829 of the *random agent* (the same point can be exercised multiple times in an episode).

RELINE identified 173 low-FPS points, as compared to the 33 of *rl-baseline* and the 90 of the *random agent*. The confidence is similar for *RELINE* (median=99%) and *rl-baseline* (median=94%), while it is lower for the *random agent* (median=76%). Thus, the low-FPS points identified by the two RL-based agents are, overall, quite reliable. Concerning the number of low-FPS points identified, *RELINE* identifies more points as compared to *rl-baseline* (173 vs 33). This is expected since it has the explicit goal of load testing the game. However, what could be surprising at first sight is the high number of low-FPS points identified by the *random agent* (90). Fig. 4-(b) and Fig. 4-(c) help in interpreting this finding.

Fig. 4-(b) plots the *path_done* (y coordinate) for each low-FPS point identified by each agent, using the same color schema of the violin plots (e.g., blue corresponds to *RELINE*).

If multiple points fall in the same coordinate (i.e., same *path_done* but different *centering*), they are shown with a red border. The scale of the *path_done* has been normalized between 0 and 100, where 0 corresponds to the starting line of the track and 100 to its finish line. Similarly, Fig. 4-(c) plots the *centering* (x coordinate) for the low-FPS points. The line at 0 represents the center of the track, while the continuous lines in position ~ -18 and ~ 18 depict the limits of the track. Finally, the dashed lines represent the area of the game we asked *RELINE* to explore: based on our reward function, we penalize the agent for going outside the $[-20, +20]$ range that, normalized, corresponds to $\sim [-36, +36]$. Also *rl-baseline* is penalized outside of this area.

As expected, the *random agent* is not able to advance in the game: The low-FPS points it identifies are all placed near the starting line — red dots in Fig. 4-(b). This indicates that a random agent can be used to exercise a specific part of a game, but it is not able to explore the game as a player would do. This is also confirmed by the red dots in Fig. 4-(c), with the *random agent* exploring areas of the game far from the track and that a human player is unlikely to explore. Also, it is worth noting that in SuperTuxKart each episode lasts (based on our setting) 90 seconds if the agent does not cross the finish line. However, as shown in our preliminary study, in other games such as MsPacman a *random agent* could quickly lose an episode without having the chance to explore the game at all.

The low-FPS points identified by *RELINE* (blue dots) and by *rl-baseline* (green) are instead closer to the track and, for what concerns *RELINE*, they are within or very close the area of the game we ask it to explore — see dashed lines in Fig. 4-(c). Thus, by customizing the reward function, it is possible to define the area of the game relevant for the load testing.

Looking at Fig. 4-(b), we can see that *RELINE* is also able to identify low-FPS in different areas of the game with, however, a concentration close to the beginning and the end of the game. It is difficult to explain the reason for such a result, but we hypothesize two possible explanations.

First, it is possible that the “central” part of the game simply features less low-FPS areas. This would also be confirmed by the fact that *rl-baseline* only found one low-FPS point in that part of the game. Also, the training and the reward function could have driven *RELINE* to explore more the starting and the ending of the game. The starting part is certainly the most explored since, at the beginning of the training, the agent is basically a random agent. Thus, it mostly collects experience about low-FPS points found in the beginning of the game since, similarly to the *random agent*, it is not able to advance in the game. It is important to remember that the data in Fig. 4 only refers to the 1,000 games played by *RELINE* after the 2,300 training games, so we are not including the random exploration done at the beginning of the training in Fig. 4. However, once the agent learns several low-FPS points in the starting of the game, it can exercise them again and again to get a higher reward.

Concerning the end of the game, we set a maximum duration of 90 seconds for each game, but we know that a well-trained agent can complete the lap in ~ 70 seconds. It is possible that the agent used the remaining time to better explore the last part of the game before crossing the finish line, thus finding a higher number of low-FPS points in that area. Additional trainings, possibly with a different *reward function*, are needed to better explain our finding.

Concerning the violin plots in Fig. 4-(a), we can see that *RELINE* and *rl-baseline* exhibit a similar distribution, with *RELINE* being able to identify some stronger low-FPS points (*i.e.*, longer time to render frames). All distributions have, as expected, the median above the t threshold, with *RELINE*'s one being higher (24.54 vs 21.69 for *rl-baseline* and 19.39 for *random agent*). The highest value of the distributions is 65.92 (60.7 FPS) for *RELINE*, against 44.81 (89.3 FPS) for *rl-baseline* and 50.73 (78.8 FPS) for *random agent*. Remember that all these values represent milliseconds to load frames after an action performed by the agents.

Table 2: Results of Mann-Whitney test (adjusted p -value) and Cliff's Delta (d) when comparing the distributions of rendering times – boldface indicates higher times.

| Test | p -value | OR |
|--------------------------------------|------------|---------------|
| <i>RELINE vs rl-baseline</i> | <0.001 | 0.34 (Medium) |
| <i>RELINE vs random agent</i> | <0.001 | 0.36 (Medium) |
| <i>rl-baseline vs random agent</i> | <0.001 | 0.16 (Small) |

Table 2 shows the results of the statistical comparisons among the three distributions. In each test, the approach reported in boldface is the one identifying stronger low-FPS points (*i.e.*, more extreme points requiring longer rendering time for their frames). The adjusted p -values report a significant difference (p -value < 0.001) in favor of *RELINE* against both *rl-baseline* and the *random agent* (in both cases, with a medium effect size). Thus, the low-FPS points identified by *RELINE* tend to require longer times to render frames. Fig. 2-(c) shows an example of low-FPS point identified by *RELINE*: Crashing against the sheep results in a drop of FPS.

Finally, it is worth commenting about the overlap of low-FPS points identified by the three agents. Indeed, *RELINE* and *rl-baseline* found 14 low-FPS points in common (*i.e.*, same x and y coordinates), while the overlap is of 11 points for *RELINE* and *random agent*, and 10 for *rl-baseline* and *random agent*. The most interesting finding of this analysis is that *rl-baseline* was able to identify only 19 points missed by *RELINE*, while the latter found 159 points missed by *rl-baseline*. This supports the role played by the *reward function* in pushing *RELINE* to look for low-FPS points.

Summary of the Case Study. *RELINE* is the best approach for finding low-FPS points in SuperTuxKart. A *random agent* is not able to spot issues that require playing skills, and *rl-baseline* only finds a small portion of the low-FPS points.

5 THREATS TO VALIDITY

Threats to Construct Validity. The main threats to the construct validity of our study are related to the process we adopted in our case study (Section 4) to identify low-FPS points. Based on our experiments, and in particular on the findings reported in Fig. 3, our methodology should be reliable enough to identify variations in FPS. Still, some level of noise can be expected, and for this reason all our analyses have been run at least 300 times, while 1,000 episodes were played by each of the experimented approaches.

Concerning our preliminary study (Section 3), it is clear that the bugs we injected are not representative of real performance bugs in the subject games. However, they are inspired from a performance mutation operator defined in the literature [23]. Our preliminary study only serves as a proof-of-concept to verify whether, by modifying the reward function, a RL-based agent would adapt its behavior to look for bugs while playing the game.

Threats to Internal Validity. In our case study, to ease the training we did not use the “real” game, but its wrapped version, *i.e.*, PySuperTuxKart [11]. While the core game is the same, the version we adopted does not contain the latest updates and it includes additional Python code that may affect the rendering time. We assume that such a time is constant for all the frames since it simply triggers the frame rendering operation in the core game. Besides, we forced the game to run with lowest graphics settings to speed up rendering: For example, we excluded dynamic lighting, anti-aliasing, and shadows. Therefore, the low-FPS points found in PySuperTuxKart may be irrelevant in the original game or with other graphic settings. Also, we applied the five- σ rule to define a threshold for defining what a low-FPS point is. The threshold we set might be not indicative of relevant performance issues.

Still, the goal of our study was to show that once set specific requirements (*e.g.*, the threshold t , the area to explore, etc.), the agent is able to adapt trying to maximize its reward. Thus, we do not expect changes in the threshold to invalidate our findings.

Threats to conclusion validity. In our data analysis we used appropriate statistical procedures, also adopting p -value adjustment when multiple tests were used within the same analysis.

Threats to External Validity Besides the proof-of-concept study we presented in Section 3, our empirical evaluation of *RELINE* includes a single game. This does not allow us to generalize our findings. The reasons for such a choice lie in the high effort we experienced as researchers in (i) building the pipeline to interact with the game, (ii) finding and experimenting with a reliable way to capture the FPS, (iii) defining a meaningful reward function that allowed the agent to successfully play the game in the first place and, then, to also spot low-FPS points. These steps were a long trial-and-error process with the most time consuming part being the trainings needed to test the different reward functions we experimented before converging towards the ones presented in this paper. Indeed, testing a new version of a reward function required at least one week of work with the hardware at our disposal (including implementation, training, and data analysis).

This was also due to the impossibility of using multiple machines or to run multiple processes in parallel on the same server. Indeed, as explained, using the exact same environment to run all our experiments was a study requirement. It is worth noting that, because of similar issues, other state-of-the-art approaches targeting different game properties were experimented with only one game as well (see *e.g.*, [17, 38, 49, 52]). We believe that instantiating *RELINE* on a new game would be much easier by collaborating with the game developers. While this would only slightly simplify the definition of a meaningful reward function, the original developers of the game could easily provide through APIs all information needed by *RELINE* (including, *e.g.*, the FPS), cutting away weeks of work.

6 RELATED WORK

Three recent studies [33, 40, 46] suggest that finding performance issues in video games is a relevant problem, according to both game developers [40, 46] and players [33]. Nevertheless, to the best of our knowledge, no previous work introduced automated approaches for load testing video games. Therefore, in this section, we discuss some important works on the quality assurance of video games in general. We first introduce the approaches defined in the literature for training agents able to automatically play and win a game. Then, we show how such approaches are used for play-testing for (i) finding functional issues and (ii) assessing game/level design (e.g., finding unbalanced levels or mechanics).

6.1 Training Agents to Play

Reinforcement Learning (RL) is widely used to train agents able to automatically play video games. Mnih *et al.* [35, 36] presented the first approach based on high-dimensional sensory input (i.e., raw pixels from the game screen) able to automatically learn how to play a game. The authors used a Convolutional Neural Network (CNN) trained with a variant of Q-learning to train their agent. The proposed approach is able to surpass human expert testers in playing some games from the Atari 2600 benchmark.

Vinyals *et al.* [48] introduced SC2LE, a RL environment based on the game *StarCraft II* that simplifies the development of specialized agents for a multi-agent environment.

Hessel *et al.* [27] analyzed six extensions of the DQN algorithm for RL and they reported the combinations that allow to achieve the best results in terms of training time on the Atari 2600 benchmark.

Baker *et al.* [16] explored the use of RL in a multi-agent environment (i.e., the *hide and seek* game). They report that agents create self-supervised autotutorials [32], i.e., curricula naturally emerging from competition and cooperation. As a result, the authors found evidence of strategy learning not guided by direct incentives.

Berner *et al.* [14] reported that state-of-the-art RL techniques were successfully used in OpenAI Five to train an agent able to play Dota 2 and to defeat the world champion in 2019 (Team OG). Finally, Mesentier *et al.* [22] reported that AI agents could be easily trained to explore the states of a board game (*Ticket to Ride*) performing automated play-testing.

6.2 Testing of Video Games

Functional testing of video games aims at finding unexpected behaviors in a game. Defining the test oracle, i.e., determining if a specific game behavior is defective, is not trivial. Several categories of test oracles were identified to determine if a bug was found: *crash* (the game stops working) [39, 51], *stuck* (the agent can not win the game) [39, 51], *game balance* (game too easy or too hard) [51], *logical* (an invalid state is reached) [51], and *user experience bugs* (related to graphic and sound, e.g., glitches) [39, 51]. While heuristics can be used to find possible crash-, stuck-, and game-balance-related bugs [51], logical and user-experience bugs may require the developers to manually define an oracle.

Iftikhar *et al.* [29] proposed a model-based testing approach for automatically perform black-box testing of platform games. More recent approaches mostly rely on RL.

Pfau *et al.* [39] introduced ICARUS, a framework for autonomous play-testing aimed at finding bugs. ICARUS supports the fully automated detection of *crash* and *stuck* bugs, while it also provides semi-supervised support for *user-experience* bugs.

Zheng *et al.* [51] used Deep Reinforcement Learning (DLR) in their approach, Wuji. Wuji balances the aim of winning the game and exploring the space to find *crash*, *stuck*, *game balance*, and *logical* bugs in three video games (one simple, *Block Maze* and two commercial, *L10* and *NSH*).

Bergdahl *et al.* [17] defined a DLR-based method which provides support for continuous actions (e.g., mouse or game-pads) and they experimented it with a first-person shooter game.

Wu *et al.* [49] used RL to automatically perform regression testing, i.e., to compare the game behaviors in different versions of a game. They experimented with such an approach on a Massive Multiplayer Online Role-Playing Game (MMORPG).

Ariyurek *et al.* [15] experimented RL and Monte Carlo Tree Search (MCTS) to define both synthetic agents, trained in a completely automated manner, and human-like agents, trained on trajectories used by human testers.

Finally, Ahumada and Bergel [13] proposed an approach based on genetic algorithms to reproduce bugs in video games by reconstructing the correct sequence of actions that lead to the desired faulty state of the game.

6.3 Game- and Level-Design Assessment

One of the main goals of a video game is to provide a pleasant gameplay to the player. Assessing the game balance and other aspects related to game- and level-design is, therefore, of primary importance.

For this reason, previous work defined several approaches for automatically finding game- and level-design issues in video games. Zook *et al.* [52] proposed an approach based on Active Learning (AL) to help designers performing low-level parameter tuning. They experimented such an approach on a *shoot 'em up* game.

Gudmundsson *et al.* [26] introduced an approach based on Deep Learning to learn human-like play-testing from player data. They used a CNN to automatically predict the most natural next action a player would take aiming to estimate difficulty of levels in *Candy Crush Saga* and *Candy Crush Soda Saga*.

Zhao *et al.* [50] report four case studies in which they experiment the use of human-like agent trained with RL to predict player interactions with the game and to highlight possible game-design issues. On a similar note, Pfau *et al.* [38] used deep player behavioral models to represent a specific player population for *Aion*, a MMORPG. They used such models to estimate the game balance and they showed that they can be used to tune it.

Finally, Stahlke *et al.* [44] defined PathOS, a tool aimed at helping developers to simulate players' interaction with a specific game level, to understand the impact of small design changes.

7 CONCLUSIONS AND FUTURE WORK

We presented *RELIN*, an approach that uses RL to load test video games. *RELIN* can be instantiated on different games using different RL models and reward functions.

Our proof-of-concept study performed on two subject systems shows the feasibility of our approach: Given a reward function able to reward the agent when artificial performance bugs are identified, the agent adapts its behavior to play the game while looking for those bugs.

We performed a case study on a real 3D racing game, SuperTuxKart, showing the ability of *RELINE* to identify areas resulting in FPS drops. As compared to a classic RL agent only trained to play the game, *RELINE* is able to identify a substantially higher number of low-FPS points (173 vs 33).

Despite the encouraging results, there are many aspects that deserve a deeper investigation and from which our future research agenda stems. First, we plan additional tests on SuperTuxKart to better understand how the agent reacts to changes in the reward function (e.g., is it possible to find more low-FPS points in the central part of the game?). Also, with longer training times it should be possible to train an agent able to play more challenging versions of this game featuring additional 3D effects (e.g., rainy conditions), possibly allowing to find new low-FPS points. We also plan to instantiate *RELINE* on other game genres (e.g., role-playing games), possibly by cooperating with their developers.

In our replication package [47], we release the code implementing the models used in our study and the raw data of our experiments.

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 851720). Any opinions, findings, and conclusions expressed herein are the authors’ and do not necessarily reflect those of the sponsors.

REFERENCES

- [1] [n.d.]. 3D.City - Performance Issue 42. <https://github.com/lo-th/3d.city/issues/42>.
- [2] [n.d.]. CartPole. <https://gym.openai.com/envs/CartPole-v0/>.
- [3] [n.d.]. Dwarfcorp - Performance Issue 583. <https://github.com/Blecki/dwarfcorp/issues/583>.
- [4] [n.d.]. Dwarfcorp - Performance Issue 64. <https://github.com/Blecki/dwarfcorp/issues/64>.
- [5] [n.d.]. Dwarfcorp - Performance Issue 711. <https://github.com/Blecki/dwarfcorp/issues/711>.
- [6] [n.d.]. Dwarfcorp - Performance Issue 904. <https://github.com/Blecki/dwarfcorp/issues/904>.
- [7] [n.d.]. Dwarfcorp - Performance Issue 966. <https://github.com/Blecki/dwarfcorp/issues/966>.
- [8] [n.d.]. Geostrike - Performance Issue 214. <https://github.com/Webiks/GeoStrike/issues/214>.
- [9] [n.d.]. Gym. <https://gym.openai.com/>.
- [10] [n.d.]. MsPacman. <https://gym.openai.com/envs/MsPacman-v0/>.
- [11] [n.d.]. PySuperTuxKart. <https://github.com/supertuxkart/stk-code>.
- [12] [n.d.]. VIDEO GAMES : INDUSTRY TRENDS, MONETISATION STRATEGIES & MARKET SIZE 2020-2025 <https://www.juniperresearch.com/researchstore/content-digital-media/video-games-market-report>.
- [13] Tomás Ahumada and Alexandre Bergel. 2020. Reproducing Bugs in Video Games using Genetic Algorithms. In *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*. IEEE, 1–6.
- [14] Open AI. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).
- [15] S. Ariyurek, A. Betin-Can, and E. Surer. 2021. Automated Video Game Testing Using Synthetic and Humanlike Agents. *IEEE Transactions on Games* 13, 1 (2021), 50–67. <https://doi.org/10.1109/TG.2019.2947597>
- [16] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. 2019. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528* (2019).
- [17] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén. 2020. Augmenting Automated Game Testing with Deep Reinforcement Learning. In *2020 IEEE Conference on Games (CoG)*. 600–603. <https://doi.org/10.1109/CoG47356.2020.9231552>
- [18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv:arXiv:1606.01540*
- [19] Bum Hyun Lim, Jin Ryong Kim, and Kwang Hyun Shim. 2006. A load testing architecture for networked virtual environment. In *2006 8th International Conference Advanced Communication Technology*, Vol. 1. 5 pp.–848. <https://doi.org/10.1109/ICACT.2006.206095>
- [20] C. Cho, D. Lee, K. Sohn, C. Park, and J. Kang. 2010. Scenario-Based Approach for Blackbox Load Testing of Online Game Servers. In *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 259–265. <https://doi.org/10.1109/CyberC.2010.54>
- [21] W. J. Conover. 1998. *Practical Nonparametric Statistics* (3rd edition ed.). Wiley.
- [22] Fernando De Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. 2017. AI as evaluator: Search driven playtesting of modern board games. In *WS-17-01 (AAAI Workshop - Technical Report)*. AI Access Foundation, 959–966. 31st AAAI Conference on Artificial Intelligence, AAAI 2017.
- [23] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. 2021. Performance mutation testing. *Software Testing, Verification and Reliability* 31, 5 (2021). <https://doi.org/10.1002/stvr.1728>
- [24] E.W. Grafarend. 2006. *Linear and Nonlinear Models: Fixed Effects, Random Effects, and Mixed Models*. Walter de Gruyter. <https://books.google.ch/books?id=uHW2wAEACAAJ>
- [25] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Erlbaum Associates.
- [26] Stefan Freyr Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartłomiej Kozakowski, Richard Meurling, and Lele Cao. 2018. Humanlike playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 1–8.
- [27] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
- [28] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.
- [29] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood. 2015. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 426–435. <https://doi.org/10.1109/MODELS.2015.7338274>
- [30] YungWoo Jung, Bum-Hyun Lim, Kwang-Hyun Sim, HunJoo Lee, IlKyu Park, JaeYong Chung, and Jihong Lee. 2005. VENUS: The Online Game Simulator Using Massively Virtual Clients. In *Systems Modeling and Simulation: Theory and Applications*. 589–596.
- [31] Maxim Lapan. 2018. *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Packt Publishing.
- [32] Joel Z Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. 2019. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *arXiv preprint arXiv:1903.00742* (2019).
- [33] Xiaozhou Li, Zheyang Zhang, and Kostas Stefanidis. 2021. A data-driven approach for video game playability analysis based on players’ reviews. *Information* 12, 3 (2021), 129.
- [34] Dayi Lin, C. Bezemer, and A. Hassan. 2016. Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering* 22 (2016), 2095–2126.
- [35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
- [36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
- [37] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How is video game development different from software development in open source?. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 392–402.
- [38] Johannes Pfau, Antonios Liapis, Georg Volkmar, Georgios N Yannakakis, and Rainer Malaka. 2020. Dungeons & replicants: automated game balancing via deep player behavior modeling. In *2020 IEEE Conference on Games (CoG)*. IEEE, 431–438.
- [39] Johannes Pfau, Jan David Smedindin, and Rainer Malaka. 2017. Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '17 Extended Abstracts)*. 153?164.

- [40] Cristiano Politowski, Fabio Petrillo, and Yann-G ael Gu eh eneuc. 2021. A Survey of Video Game Testing. *arXiv preprint arXiv:2103.06431* (2021).
- [41] Reuven Y. Rubinstein and Dirk P. Kroese. 2004. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Springer-Verlag.
- [42] Adam M. Smith, Mark J. Nelson, and Michael Mateas. 2009. Computational Support for Play Testing Game Sketches. In *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'09)*. AAAI Press, 167?172.
- [43] C. Spearman. 1904. The Proof and Measurement of Association Between Two Things. *American Journal of Psychology* 15 (1904), 88–103.
- [44] Samantha N. Stahlke, Atiya Nova, and Pejman Mirza-Babaei. 2020. Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design. *Proceedings of the Annual Symposium on Computer-Human Interaction in Play* (2020).
- [45] supertuxkart. [n.d.]. <https://github.com/supertuxkart>.
- [46] Andrew Truelove, Eduardo Santana de Almeida, and Iftekhar Ahmed. 2021. We'll Fix It in Post: What Do Bug Fixes in Video Game Update Notes Tell Us?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 736–747.
- [47] Rosalia Tufano. 2021. <https://github.com/RosaliaTufano/rlgameauthors>.
- [48] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, P. Georgiev, A. S. Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich K uttler, J. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. V. Hasselt, D. Silver, T. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, D. Lawrence, Anders Ekeremo, J. Repp, and Rodney Tsing. 2017. StarCraft II: A New Challenge for Reinforcement Learning. *ArXiv abs/1708.04782* (2017).
- [49] Yuechen Wu, Yingfeng Chen, Xiaofei Xie, Bing Yu, Changjie Fan, and Lei Ma. 2020. Regression Testing of Massively Multiplayer Online Role-Playing Games. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 692–696.
- [50] Yunqi Zhao, Igor Borovikov, Ahmad Beirami, Jason Rupert, Caedmon Somers, Jesse Harder, Fernando de Mesentier Silva, John Kolen, Jervis Pinto, Reza Pourabolghasem, Harold Chaput, James Pestrak, Mohsen Sardari, Long Lin, Navid Aghdaie, and Kazi A. Zaman. 2019. Winning Isn't Everything: Training Human-Like Agents for Playtesting and Game AI. *CoRR abs/1903.10545* (2019). <http://arxiv.org/abs/1903.10545>
- [51] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. 2019. Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 772–784.
- [52] Alexander Zook, Eric Fruchter, and Mark O. Riedl. 2014. Automatic playtesting for game parameter tuning via active learning. *ArXiv abs/1908.01417* (2014).