Towards the Automatic Replication of Gameplays to Support Game Debugging

Stefano Campanella
Università della Svizzera italiana
Lugano, Switzerland
stefano.campanella@usi.ch

Emanuela Guglielmi
University of Molise
Termoli, Italy
emanuela.guglielmi@unimol.it

Rocco Oliveto
University of Molise
Termoli, Italy
rocco.oliveto@unimol.it

Gabriele Bavota
Università della Svizzera italiana
Lugano, Switzerland
gabriele.bavota@usi.ch

Simone Scalabrino
University of Molise
Termoli, Italy
simone.scalabrino@unimol.it

ABSTRACT

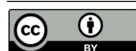
The video game industry has experienced a continuous growth in the last decades. In such a competitive market, it is fundamental to ensure a great gaming experience to the player avoiding, for example, bugs. However, video game testing is an extremely challenging activity, especially considering the extensive number of gaming scenarios that modern video games support (*e.g.,* 3D worlds to explore). Thus, more often than not, numerous bugs are discovered only once the game is released and played by millions of users. For this reason, recent work in the literature suggested to exploit gameplay videos to support developers in identifying possible bugs missed during testing: given the large amount of gameplays posted every day on streaming platforms (> 2M hours), these gameplays are likely to document failures experienced by the player. Empirical evidence show the ability of these techniques to identify parts of the gameplay in which the failure was experienced. However, it could still be difficult for game developers to reproduce the bug. In this paper, we propose the idea of developing a technique able to automate this process, providing the game developer with all actions performed by the player to reach the faulty state shown in the gameplay. We present a simple approach which leverages the on-screen controls overlay available in some gameplay videos. We show that such an approach can replicate 47.2% of gameplays in our preliminary study run on a racing game. We discuss the strong limitations of this first attempt, listing directions for future work we plan to pursue in order to overcome them.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; **Software evolution**.

KEYWORDS

Video Analysis, Gameplay Reproduction, Video Games Testing



This work is licensed under a Creative Commons Attribution 4.0 International License.

FaSE4Games '24, July 16, 2024, Porto de Galinhas, Brazil

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0674-5/24/07
https://doi.org/10.1145/3663532.3664465

ACM Reference Format:
Stefano Campanella, Emanuela Guglielmi, Rocco Oliveto, Gabriele Bavota, and Simone Scalabrino. 2024. Towards the Automatic Replication of Gameplays to Support Game Debugging. In *Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games (FaSE4Games '24), July 16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3663532.3664465

1 INTRODUCTION

Video games have emerged as a dominant force in contemporary society, exerting profound influences on social interactions, cultural paradigms, and economic landscapes. As of 2022, the market revenue of the game industry surpassed 200 billion USDs, with a forecast of 250 billion USDs by 2025 [10, 26, 28]. For being successful in such a market it is not enough to only deliver games ensuring high engagement for the player: Games must exhibit all properties typical of high-quality software, such as good performance and reliability with as few bugs as possible.

Video game testing poses well-known challenges related to constant changes in requirements and design and the need for having an *intelligent* interaction with the game to extensively exercise its functionalities. This is the main reason why test automation techniques are rarely applied by game developers [21, 25], leading game testing to mostly being a manual process. For example, one of the games being subject of the study by Zheng *et al.* [34] was manually tested by 30 players. For these reasons, the research community is actively working on proposing techniques supporting game developers in testing activities (see *e.g.,* [5, 7, 8, 13, 16, 27, 29]). Among those, techniques have recently proposed to identify bugs reported by players in gameplays posted on online platforms such as YouTube[1] and Twitch[2] [11, 18]. Indeed, millions of hours of gameplay are posted on these platforms daily [9] and, as for any other player, the streamers may run into bugs, which are thus documented in these videos. This makes the gameplay videos a relevant source to mine for identifying and reporting failures to the developers. While these approaches [11, 18] are able to identify parts of the gameplay in which a bug/glitch is documented, there is still an open problem to face: How to replicate the sequence of actions that led the game in the failing state. In other words, *how can the game developer reproduce the bug?*

[1] https://www.youtube.com/ [2] https://www.twitch.tv/

In this paper, we propose the idea of developing a technique that, given as input a gameplay video and a set of possible actions that the game supports (*e.g.,* the game's supported keyboard keys), produces as output the sequence of actions which reproduce the portion of gameplay given as input. The main assumption is that gameplay videos show on screen the game actions. This would empower developer to reproduce bugs identified in gameplay videos by state-of-the-art techniques [11, 18].

In the literature, Intharah *et al.* [15] introduce DeepLogger in order to reproduce issues through the anlysis of gameplay videos. However, DeepLogger presents some limitations, for this reason we introduced RePlay to overcome these constraints. We started investigating this problem in the simplest scenario in which the executed actions are shown on screen and for which the main task is to export such controls from publicly available gameplay videos. Also, we developed a technique tailored for one specific game, just as a proof of concept of the problem we want to address in the long run. Our approach uses machine learning models to discriminate between frames showing/not showing the game actions overlayed. The ones not showing them are discarded as non-gaming frames (*e.g.,* an advertisement shown on screen) while the others are further analyzed to extract the executed actions. The sequence of executed actions identified in consecutive frames composing the gameplay can be used to replicate that specific gameplay. While we are able to correctly identify ∼80% of the performed actions, ∼20% of errors leads the agent to successfully replicate only ∼50% of the 40 gameplays on which we tested it. Our approach can be applied to different video games, although the complexity of implementing the approach will increase with the number of possible inputs the game supports. Our preliminary work shows that even a simple approach like the one we propose can replicate some gameplay videos (for a specific game). We expect more tailored techniques to effectively address the problem with higher precision and generalizability across games and perhaps make the approach independent from the control overlay in the footage, at the cost of increased demand for computing power.

2 RELATED WORK

In recent years, there has been growing interest in the use of video analysis for supporting Software Engineering tasks [2, 3, 23, 32]. Ponzanelli *et al.* [23] introduce CodeTube, a Web-based recommender system that analyzes the contents of video tutorials and is able to provide, given a query, cohesive and self-contained video fragments, along with related Stack Overflow discussions. White *et al.* [32] show how video analysis can be used to increase the replicability of bugs experienced in Android apps. The authors present an approach for automating the process of reproducing a bug. Previous work presents techniques aimed at identifying issues in video games [14, 19, 20]. Iftikhar *et al.* [14] propose a model-based testing approach for automated black box functional testing of platform games. The authors define a detailed modeling methodology to support automated system-level game testing and guidelines for modeling the platform games for testing using our proposed game test modeling profile. As a more general approach, Paduraru *et al.* [20] introduced a similar tool named RiverGame, used to perform game testing based on artificial intelligence. This tool lets the user automatically test their products from different points of view: the rendered output, the sound played by the game, the animation and movement of the entities, the performance, and various statistical analyses. Mnih *et al.* [19] used gameplay videos to train an AI agent to play Atari games. In this case, the agent was trained by watching gameplay videos and then could play the game by itself.

Some recent studies [1, 29, 33, 35] used Deep Reinforcement Learning to support developers in finding issues in video games (*e.g.,* performance-related). Pfau *et al.* [22] introduced a framework for autonomous playing of games, that also performs testing and bug reporting named ''ICARUS''. Tufano *et al.* [29] introduced RELINE based on Reinforcement Learning, defining a methodology to train an agent to play the game as a human while also trying to identify areas of the game resulting in a drop in FPS.

In some cases, however, such agents might struggle in finding issues that, instead, humans happen to experience while playing the game. For this reason, recent studies [11, 17, 18] focused on the automatic identification of bugs through the analysis of gameplay videos from popular video and streaming platforms (*e.g.,* Twitch and Youtube). Lin *et al.* [18] defined a technique to automatically identify gameplay videos that report bugs through metadata analysis. Their approach, however, is not able to pinpoint the specific parts of the video in which the bug is reported. This makes it unsuitable as a reporting tool for game developers.

Guglielmi *et al.* [11] introduced GELID, a novel approach for automatically extracting relevant information from gameplay video segments in which streamers reported issues through subtitle and image analysis. Given as input a set of gameplay videos from the same video game, such an approach clusters together segments of the different gameplay videos reporting similar issues (*e.g.,* a specific bug occurred to many players).

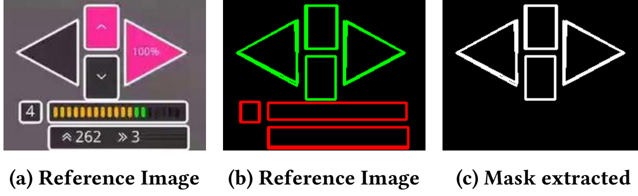
To the best of our knowledge, the only approach in the literature to reproduce issues highlighted in gameplay videos is DeepLogger [15]. Given a gameplay video, such an approach relies on a CNN to extract user inputs. DeepLogger takes into account discrete inputs (*i.e.,* keys pressed or not pressed) and leaves the extraction of continuous inputs (*e.g.,* joysticks or mouse) as an open problem. Thus, this network can only predict user input logs for a game where training data is available. On the other hand, RePlay considers both discrete and continuous inputs. This comes at the cost of relying on input overlays, which are available only on a subset of gameplay videos.

3 REPLAY

We present RePlay, an approach to extract the sequence of actions performed by a player to replicate a given gameplay. The main steps performed by RePlay are: (i) overlay detection, to identify the game controls shown on screen, if any; (ii) the extraction of the actions performed by the player; and (iii) the game reproduction. Some parts of our implementation, meant to be a proof-of-concept, are tailored for a specific video game, namely *Trackmania*, a popular racing game series developed by Nadeo[3] and published by Ubisoft[4]. We chose Trackmania due to the limited number of controls to be identified and consequent player actions to be predicted. Indeed, the possible commands are limited to acceleration, braking, and

[3] https://www.nadeo.com/ [4] https://www.ubisoft.com/en-us/

turning left/right. Also, each game starts with the beginning of the track: It is easier to identify the initial status of the game to replicate, as opposed, *e.g.,* to open-world games.



(a) Reference Image **(b) Reference Image** **(c) Mask extracted**

Figure 1: Process used to rank and extract mask.

3.1 Overlay Detection

As one can see from Fig. 1a, overlays have synthesized iconography, which greatly simplifies extraction and parsing. This, given a gameplay video as input we first need to detect the game control overlay (*i.e.,* the part of the screen showing the actions performed by the player). More specifically, we need to (i) filter the frames in which the overlay is present (control overlays might be temporarily obstructed in certain frames and, thus, not reliable for input extraction), and (ii) identify the overlay elements that represent input devices (*e.g.,* buttons). Overlays may vary not only in position but also in type across different videos, depending on the input mechanisms used during gameplay. These overlays can take many different forms, including joystick movement indicators and on-screen prompts for keyboard and mouse inputs. Furthermore, the overlay may vary depending on the game's specific input requirements, such as displaying key bindings alongside corresponding controls or highlighting specific actions. Thus, to achieve our two goals, we first identify the position of the overlay in the specific video, then we filter out frames that do not have the overlay, and we use the shapes identified to distinguish the parts of the overlay.

Identifying the overlay and building the mask. Our first step is to run PyShapes [24] on each frame, with the goal of identifying regular shapes (*e.g.,* rectangles and triangles) shown on screen. Then, we automatically identify a frame in which the overlay featuring the game controls is clearly visible and identifiable. To do this, we manually define a set of rules regarding the number, the types, and the relationships between the (detected) shapes that are supposed to compose the overlay of the game at hand. The mask of the overlay is extracted from one of the frames for which all such rules are met. Note that the rules are highly game- and overlay-dependent. For Trackmania, the mask must include a triangle pointing to the right, a triangle pointing to the left (for analog stick inclination) and two rectangles in the center (*i.e.,* acceleration and reverse). If we find exactly two triangles and two rectangles with the previously-mentioned relationships, we assume we found a good candidate for extracting the overlay mask. The mask is defined on the basis of the edges of the detected shapes. An example of this is shown in Figure 1b, where the green shapes will be considered as relevant, while the red ones will not. The process involves locating a frame with clearly visible controls and exporting a corresponding mask (Fig. 1c).

Filtering out frames without the overlay. We first extract from a given frame the pixels that are in the overlay mask. Then, based on the assumption that we do not know the color of the edges of the overlay, we convert such pixels in grayscale. We obtain an linear array of pixels in the order they appear in the image scanning first the rows (left-right) and then the columns (top-bottom). We extract measures to aggregate such an array of pixels, each of which is represented by a single value ranging between 0 and 255. Such measures are: (i) maximum (*i.e.,* the brightest color), minimum (*i.e.,* the darkest color), mean, variance, contrast (*i.e.,* $\sum_{i=1}^{n}(x_i - \bar{x})^2$), and homogeneity (*i.e.,* $\sum_{i=1}^{n} \frac{x_i}{1+(\bar{x}-\bar{x})^2}$) where x stands for the pixel value of each pixel in the array. We train a Random Forest model [4] using such measures as features to classify whether or not the frame contains the overlay.

3.2 Control Extraction

Once detected the controls overlay, we extract the inputs from each frame, creating a sequence that can be played back.

For each frame, we extract a value representing the activation of each control supported by the game. We support two types of input devices: *buttons*, which can be either pressed or not (*i.e.,* they can be represented through boolean values), and mono-dimensional *sticks*, that can have in-between positions (*i.e.,* they can be represented as decimal numbers). We assume that each shape in the overlay corresponds to a specific game input device, which can be either a button or a joystick, and that developers manually map each overlay shape to the input device at hand. For example, the control overlay of Trackmania is composed of (i) top and bottom arrows that map the acceleration and reverse commands (buttons), and (ii) left and right arrows that represent the inclination angle of the analog stick. The top and bottom arrows are mapped to the respective buttons on the pad, while the left and right arrows are mapped to the same device (main analog stick of the pad): the right and left arrow values are mapped to positive and negative x-axis values of the analog stick, respectively. We use two different procedures to extract the input value for overlay shapes representing buttons and sticks.

Buttons. For buttons, the shape in the overlay is completely colored with an overlay-dependent color when the button is pressed, while it might be transparent (depending on the overlay at hand) when the button is not pressed. Since the color used by the overlay to represent the "button pressed" event strictly depends on the overlay at hand and on its settings, we take such a color (*active color*) as an input for this step. We first extract all the pixels contained in the overlay shape. Then, we check if all the pixels are within a certain distance from the active color: If they are, it means the button is pressed, otherwise it is not. We do this because the active color might not fully cover the game capture below, but it might be semi-transparent. To compute the distance between pixel colors we compute the difference between each color channel in the RGB space (red, green, and blue) and sum them. We say that a pixel is equal to the active color if the distance between them is lower than a threshold t. For our experiments, we use the threshold $t = 50$, which was determined through manual experimentation to optimally suit the purpose of detecting similar colors.

Sticks. For sticks, the shape in the overlay is gradually colored (again, with an overlay-dependent color) to represent the percentage of stick inclination. To predict the percentage in the arrows, we use a regression machine-learning model. The model can estimate the percentage value of inclination of a stick for a given overlay element. To do this, since we assume a shape represents mono-directional sticks (we do not support bi-directional sticks), we first summarize the whole shape content by extracting some pixels from it.

Specifically, for the triangular shapes used in the overlay of Trackmania, we extract the pixel lines beside the upper- and lower-edges of the triangle. In addition, an orthogonal line is derived from the center of the left or right base to the opposite vertex. This extraction process aims to provide the model with information about how colored is the triangle. We use the Bresenham's line algorithm [31] to get the pixel lines from the overlay shape.

For each line of pixels we compute two measures. The first one represents the size of the longest sequence of consecutive pixels that are within a certain distance from the active color, while the second one represents the percentage of pixels that are within a certain distance from the active color (*i.e.,* we also count pixels that are not consecutive, for example when some other elements go above the overlay). We compute the distance between pixel colors in the same way we do for buttons and filter them based on the same threshold t. We use such measures, extracted for each line, as features for a linear regression model that predict the stick inclination in a range between 0 and 100, based on the overlay behaviour. A training set is needed for this step, which can be easily built for overlays that also report the numeric value of the percentage of inclination.

3.3 Gameplay Reproduction

We implement an agent through which we execute the sequence of commands extracted from the gameplay video. We implemented the agent using the emulator for the gamepad available in *vgamepad* [30]. We take as input the output of the previous step to create a simplified list of commands for the gamepad APIs. We set the frame-rate of the game at 30 FPS. Then, to replicate the commands, we make sure that we replicate the input devices status inferred in the previous step each $\frac{1}{30}$ of a seconds. If a given frame i is classified as without overlay during the first step, there will be no input associated to it. In such cases, we assume the input status did not change from the previous frame and, thus, we keep the input devices status extracted for the frame $i - 1$.

4 EMPIRICAL STUDY DESIGN

The *goal* of our study is to understand if it is possible to detect command acquisition and replicate the game through RePlay. We aim to answer the following research questions (RQs):

- RQ_1: *To what extent is RePlay effective in identifying frames that include the controls overlay?*
  RQ_1 assesses the performance and reliability of the approach in identifying frames that contain the control overlay.
- RQ_2: *To what extent does RePlay allow to infer the input commands?*
  RQ_2 evaluate the accuracy of the approach to infer input commands in the gameplay analysis.
- RQ_3: *To what extent does the agent allow to reproduce the games?*
  With this last RQ, the goal is to evaluate the accuracy of the agent to reproduce the gameplay, based on the command gathered with RePlay

4.1 Context Selection

To evaluate the first two steps of RePlay (RQ_{1-2}), we built three annotated datasets. First, we randomly sampled 3 gameplay videos of Trackmania from Twitch (~10 hours), downloaded them, and extracted the frames after fixing the frame-rate at 30 FPS. We sampled 496,770 frames and annotated them by indicating whether the overlay was visible or not. This allowed us to define the dataset $D_{overlay}$, which contains pairs $\langle frame, overlay \rangle$. We further sampled 165 frames and manually annotate them with the information regarding the status of the two buttons (pressed or not pressed) based on the information provided in the overlay (*i.e.,* their color).

As a result, we defined the dataset $D_{buttons}$, which contains triples $\langle frame, button, status \rangle$, where each frame is repeated twice (one for each *button*) and *status* is a categorical variable (*pressed* or *not pressed*). Finally, some overlays show inside the triangles the percentage number indicating the stick inclination. We rely on them to build a third dataset of randomly sampled 246 frames. We manually labeled them with the percentage indicated in the triangles. Thus, we obtained the dataset D_{stick}, which contains triples $\langle frame, stick, percentage \rangle$, where each frame is repeated twice (one for each *stick*) and *percentage* is a numerical (integer) variable ranging from 0 to 100. To evaluate the last step of RePlay (RQ_3) we built a last dataset. We randomly sampled 9 videos reporting entire game sessions of Trackmania from Twitch, all of them different from the ones used for building the previous datasets. Besides, we recorded 31 gameplay videos using the same maps and overlay shown in the Twitch clips. We use the whole RePlay on such videos to extract commands and define the agents to replicate them. We ran each game with the agent as a player and recorded the videos. As a result, we obtained D_{games}, which contains pairs $\langle video_{original}, video_{replicated} \rangle$, where $video_{original}$ is the video of the game played by a human player and $video_{replicated}$ is the one played by the agent defined with RePlay. Figure 2 shows an example of a frame paired with the portion of the image exported.

Figure 2: Example of frame and controls portion exported.

4.2 Data Collection and Data Analysis

To address $\mathbf{RQ_1}$, we use $D_{overlay}$. For each instance $\langle frame, overlay \rangle$, we first extracted the features previously defined for the first step of RePlay from $frame$. Then, we trained and tested our model on the

dataset containing such features and the *overlay* value as a label. We used the implementation and default configuration of Random Forest available in Weka [12]. We ran a 10-fold cross validation to assess the performance of the trained model. We compute and report accuracy, precision, and recall.

To answer **RQ**$_2$, we ran two separate evaluations for buttons and analog sticks. As for the former, we use $D_{buttons}$: For each instance $\langle frame, button, status \rangle$, we ran our button pression detection approach on the $frame$ for the shape of the $button$ at hand. We compare the predicted button pression status with the $status$ label. We report accuracy, precision, and recall for such a step. As for the latter, we use D_{sticks}: Again, for each instance $\langle frame, stick, percentage \rangle$, we extracted from the $frame$ the features defined for the second step of RePlay (stick inclination) for the shape of the specific $stick$ at hand. We trained and test a linear regression model on the dataset containing the extracted features and the $percentage$ value as labels, again using the implementation and default configuration available in Weka. We ran a 10-fold cross validation to assess the performance of such a model. We report the obtained Mean Absolute Error (MAE) and Relative Absolute Error (RAE).

To address **RQ**$_3$, we manually compared, for each instance of D_{games}, the $video_{original}$ and $video_{replicated}$ by extracting 1-second clips every 3 seconds of the videos. We started from the beginning of the videos and proceeded until we found a difference in the clips, by synchronising the two videos based on the start of the control sequences. We consider the clips different if any of the control sections are not matching exactly. We measure, for each video, time percentage of correctly replicated game by dividing the time at which the first difference has been found by the video duration.

4.3 Replication Package

We publicly release the implementation of RePlay and the script we used to run the experiment and the dataset of results for each RQs in our replication package [6].

5 EMPIRICAL STUDY RESULTS

The results of RQ$_1$ show that the model has an accuracy of 98% and precision and recall values of 0.99 in overlay detection. Based on the evaluation results, we can conclude that the model to identify frames showing the overlay, in our context, is highly effective. Regarding the button pression detection (RQ$_2$), RePlay achieves 99.0% of precision, 97% of recall 96% of AUC. As for the analog stick (again, RQ$_2$) inclination prediction, instead, RePlay achieves Mean Absolute Error of 3.16 and a Relative Absolute Error of 12.88%. While the button detection is very promising, we observed that the models makes a small error in predicting the stick inclination which could hamper the perfect replication of a game.

Finally, as for RQ$_3$, we found that our agents can reproduce, on average 47.23% of the games we analyzed. While this result might seem slightly underwhelming, it is important to consider the difficulty of the tracks. In particular, the agent finds in tight turns and very rapid changes of direction especially difficult. This also happens when the player performs maneuvers very close to the boundaries of the map, such as walls. On the other hand, if the turn is simple, the agent is generally able to complete the entire turn exactly as in the original video. This observed behavior of the agents may have a strong impact on the accuracy of specific games.

We conducted an additional analysis to evaluate to what extent the agents' actions are synchronized with the players actions. To this end, we continued analyzing the 1-second clips also after the first error and only focused on the overlays. This allowed us to check what would have happened if the agent did not make a mistake. In this case, we found that, on average, 81.21% of the commands are performed correctly.

The results obtained during the analysis aimed to answer RQ$_3$ remains reliable and enables us to evaluate the quality of the proposed approach, as well as the implementation of the agent involved in the gameplay replication process.

6 THREATS OF VALIDITY

In this section we summarize the threats of validity of our work.

Construct Validity. In our evaluation, we assumed that the gameplay videos, that we want to reproduce, report an overlay that shows user input commands through coloured each key input.

However, gameplay videos do not always feature an overlay showing input playback, and when they do, they do not necessarily follow the standards used in our study as a reference.

Internal Validity. The selection of the threshold for extracting commands through colour similarity introduces a potential bias that may influence the overall accuracy of overlay commands extraction. Additionally, the accuracy of the models involved in the processing applied and the potential error that resides within them can have an impact on the overall outcome.

External Validity. While certain components of our approach (*i.e.,* frame identification) are agnostic to the specific game used, others may exhibit dependence (*i.e.,* controls overlay).

7 CONCLUSION AND FUTURE WORK

Video games are one of the most solid pillars of the development industry, so improving the stages of the software development process can be important to support companies in creating better gaming experiences. We presented RePlay, a simple approach able to (i) distinguish frames with command overlay from those without, (ii) extract commands and (iii) replicate the games. Despite the very promising results, RePlay also has clear limitations. First, it requires that the gameplay videos report an overlay that shows user input commands. However, this is not always true. Second, some steps of RePlay are closely dependent on the game under consideration, restricting its generalizability unless these aspects are tailored for the targeted game. Also, we assume that game overlays are defined as specific geometric shapes. To adapt this approach to new games, it is required (i) the extraction of commands and (ii) the definition of an appropriate mask. To mitigate both such limitations, future research can explore the use of reinforcement learning, similarly to previous work [1, 19, 29]. This strategy would allow to make our approach less video game-dependent and even overlay-independent.

REFERENCES

[1] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2019. Automated video game testing using synthetic and humanlike agents. *IEEE Transactions on Games* 13, 1 (2019), 50–67.

[2] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, Xin Xia, and Bo Zhou. 2017. Extracting and analyzing time-series HCI data from screen-captured task videos. *Empirical Software Engineering* 22, 1 (2017), 134–174.

[3] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. 2015. scvRipper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 673–676.

[4] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[5] 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

[6] Stefano Campanella, Emanuela Guglielmi, Rocco Oliveto, Gabriele Bavota, and Simone Scalabrino. 2023. Replication Package of "Towards the Automatic Replication of Gameplays to Support Game Debugging". https://doi.org/10.6084/m9.figshare.24581826.

[7] 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

[8] 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.

[9] "Satista Gaming". 2023. Number of hours streamed on leading gaming live stream platform. https://www.statista.com/statistics/1030809/hours-streamed-streamlabs-platform/.

[10] "Satista Gaming". 2023. Satista Gaming. https://www.statista.com/topics/1680/gaming.

[11] Emanuela Guglielmi, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2023. Using gameplay videos for detecting issues in video games. *Empirical Software Engineering* 28, 6 (2023), 136.

[12] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[13] 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

[14] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah 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)*. IEEE, 426–435.

[15] Thanapong Intharah and Gabriel J Brostow. 2018. Deeplogger: Extracting user input logs from 2d gameplay videos. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*. 221–230.

[16] 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.

[17] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. 2010. What went wrong: a taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games*. 108–115.

[18] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. 2019. Identifying gameplay videos that exhibit bugs in computer games. *Empirical Software Engineering* 24, 6 (2019), 4006–4033.

[19] 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).

[20] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2022. RiverGame-a game testing tool using artificial intelligence. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 422–432.

[21] 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.

[22] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. 2017. Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. Association for Computing Machinery, New York, NY, USA.

[23] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Codetube: extracting relevant fragments from software development video tutorials. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 645–648.

[24] "PyShapes". 2023. PyShapes. https://github.com/sudoRicheek/PyShapes.

[25] Ronnie ES Santos, Cleyton VC Magalhães, Luiz Fernando Capretz, Jorge S Correia-Neto, Fabio QB da Silva, and Abdelrahman Saher. 2018. Computer games are serious business and so is their quality: particularities of software testing in game development from the perspective of practitioners. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[26] "Satista". 2023. Satista. https://www.statista.com/statistics/292056/video-game-market-value-worldwide/.

[27] 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.

[28] "Truelist". 2023. Truelist. https://truelist.co/blog/gaming-statistics/.

[29] Rosalia Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, and Gabriele Bavota. 2022. Using Reinforcement Learning for Load Testing of Video Games. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2303–2314. https://doi.org/10.1145/3510003.3510625

[30] "Vgamepad". 2023. Vgamepad. https://github.com/yannbouteiller/vgamepad.

[31] William E Wright. 1990. Parallelization of Bresenham's line and circle algorithms. *IEEE Computer Graphics and Applications* 10, 5 (1990), 60–67.

[32] Huijuan Wu, Yuepu Guo, and Carolyn B Seaman. 2009. Analyzing video data: A study of programming behavior under two software engineering paradigms. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 456–459.

[33] 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.

[34] 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.

[35] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie 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)*. IEEE, 772–784.

Received 2024-03-28; accepted 2024-04-26