Automatic Identification of Game Stuttering via Gameplay Videos Analysis

EMANUELA GUGLIELMI, DEVISER, University of Molise, Italy
GABRIELE BAVOTA, SEART, Università della Svizzera italiana, Switzerland
ROCCO OLIVETO, DEVISER, University of Molise, Italy
SIMONE SCALABRINO, DEVISER, University of Molise, Italy

Modern video games are extremely complex software systems and, as such, they might suffer from several types of post-release issues. A particularly insidious issue is constituted by drops in the frame rate (*i.e.,* stuttering events), which might have a negative impact on the user experience. Stuttering events are frequently documented in the million of hours of gameplay videos shared by players on platforms such as Twitch or YouTube. From the developers' perspective, these videos represent a free source of documented "testing activities". However, especially for popular games, the quantity and length of these videos make impractical their manual inspection. We introduce HASTE, an approach for the automatic detection of stuttering events in gameplay videos that can be exploited to generate candidate bug reports. HASTE firstly splits a given video into visually coherent slices, with the goal of filtering-out those that not representing actual gameplay (*e.g.,* navigating the game settings). Then, it identifies the subset of pixels in the video frames which actually show the game in action excluding additional elements on screen such as the logo of the YouTube channel, on-screen chats etc. In this way, HASTE can exploit state-of-the-art image similarity metrics to identify candidate stuttering events, namely subsequent frames being almost identical in the pixels depicting the game. We evaluate the different steps behind HASTE on a total of 105 videos showing that it can correctly extract video slices with a 76% precision, and can correctly identify the slices related to gameplay with a recall and precision higher than 77%. Overall, HASTE achieves 71% recall and 89% precision for the identification of stuttering events in gameplay videos.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Maintaining software**; *Software defect analysis.*

Additional Key Words and Phrases: Video game, Performance

ACM Reference Format:
Emanuela Guglielmi, Gabriele Bavota, Rocco Oliveto, and Simone Scalabrino. 2023. Automatic Identification of Game Stuttering via Gameplay Videos Analysis. In *Proceedings of Transactions on Software Engineering and Methodology (TOSEM).* ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The video game industry is a billion-dollar business, with a market value of more than $95B in the US alone, as of 2021[30]. Just like other software systems, video games can be affected by several types of functional and nonfunctional issues.

Manuscript submitted to ACM

Some of such problems are more relevant for video games than for other types of software. One of them is constituted by drops in the frame rate (also referred to as "stuttering"). Such an issue makes the user experience not only less entertaining, but even frustrating when it creates impediments for properly playing the game.

The relevance of stuttering events for game developers is well-known. Politowski *et al.* [25] reported that the developers partially automated game performance testing for two out of the five games considered in their study. Naughty Dog reportedly employed specialized profiling tools [34] while developing and testing *The Last of Us* for detecting stuttering events. Finally, Truelove *et al.* [35] report that game developers agree that *Implementation response* problems may severely impact the game experience.

For these reasons, researchers recently introduced techniques to detect areas of the game affected by stuttering (see *e.g.,* [37]). While these techniques could help in game testing, it is practically impossible to thoroughly test all the possible interactions between the player(s) and the environment. Therefore, stuttering events might still occur in specific conditions (*e.g.,* when many players are in the same area of the game) that are not detected before the release and can only be reported by the end users. This could happen through classic bug-reports possibly accompanied by videos documenting the observed behavior. Video-based bug reports are becoming more and more popular, as also demonstrated by the several research works studying this phenomenon [5, 9, 14, 22]. When it comes to video games, video-based bug reports become fundamental to report issues such as stuttering events. Besides bug reports explicitly opened by end users, stuttering events are implicitly documented in the million of hours of gameplay videos available in platforms such as Twitch [38] or YouTube [47]: Twitch content creators stream, on average, 2.2M hours of videos every day [40], most of which are gameplay videos. Consider, for example, the gameplay video at https://youtu.be/1LHHLaSRW8Y?t=79. At minute 01:22, a stuttering event starts and lasts for a few seconds. Developers could use the information provided in the video to localize the problem and fix it. Previous work explored detecting different types of issues [13, 36], but no previous work specifically focused on stuttering events.

In this paper, we aim at tackling the problem of automatically finding candidate stuttering events in gameplay videos. We preliminarily surveyed 26 professional game developers, aiming at understanding to what extent solving such a problem would be relevant in the first place. Our results suggest that practitioners would be interested in such an approach and that it would be complementary to tools that they already use (*e.g.,* telemetry [49]).

Fig. 1. Gameplay video with extraneous elements

It might be argued that finding stuttering events is conceptually easy: By definition, stuttering occurs when two or more subsequent frames in a gameplay video are identical. However, there are several problems to address. First, there

are game scenes in which identical consecutive frames do not indicate stuttering (*e.g.,* game settings). Thus, a naive approach that checks the equality of consecutive frames might falsely report stuttering events. Second, there might be parts of gameplay videos that do not show the video game itself but other elements, such as the streamer's face.

Fig. 1 depicts a concrete example of a gameplay video showing, besides the game itself, external elements such as the player recording the video. In this case, checking the equality of subsequent frames is also not enough, since while the "game-related pixels" might not change (due to stuttering), the ones depicting the player may change, thus missing the stuttering identification.

To solve those problems, we introduce HASTE (**H**inter for g**A**me **ST**uttering **E**vents). HASTE works in four steps. First, it splits a gameplay video into visually coherent slices, based on major changes in the color schema of the frames shown on screen. Second, HASTE automatically classifies each extracted slice as *gameplay* or *non-gameplay* (*e.g.,* game settings, advertisement) using a machine-learning model. Third, it further analyzes each video slice classified as *gameplay* to identify the parts of the frame actually showing the game (*i.e.,* excluding unrelated elements such as the player or an on-screen chat). Finally, HASTE checks each pair of consecutive frames and it identifies stuttering events in which the relevant pixels (*i.e.,* the ones depicting the game) are (almost) identical.

We validate the steps behind HASTE on a total of 105 videos, of which about 10 hours of contents manually analyzed by the authors. We found that HASTE is accurate both in determining slicing points for the videos (76% precision) and detecting *non-gameplay* events to filter them out (90% precision and 88% recall). In terms of stuttering event identification, HASTE is largely more effective than three simpler baselines with which we compared it, achieving, overall, 71% recall and 89% precision. Finally, we interviewed two senior game developers aiming at understanding the applicability of HASTE in an industrial context. They generally provided positive feedback on the usefulness and applicability of HASTE in a developer's workflow due to the possibility of accessing a large amount of data available online. In addition, they highlight a major challenge related to the use of gameplay videos in the first place: HASTE might find false positives, *i.e.,* stuttering events that are not related to the gameplay video but to other incidental circumstances (*e.g.,* screen recording software, specific hardware configurations). This is a problem shared with several approaches and tools typically used by developers (*e.g.,* static code analysis tools). Still, HASTE could provide some guidance to let developers understand what they should focus on.

2 MOTIVATING STUDY

We ran a survey with game development practitioners to understand the possible relevance of an automated approach that detects stuttering events in gameplay videos.

2.1 Survey Design

The *goal* of this study is to answer the following research question: RQ_0: *To what extent is it useful to identify stuttering events in video games through the use of gameplay videos?* The *context* is represented by objects (*i.e.,* a survey) and subjects (*i.e.,* 26 practitioners from the game development industry).

Participants Selection. We recruited participants through Prolific [27], a platform that helps to select participants for research studies. We looked for candidate participants who (i) were located in the United States and Europe, (ii) had experience in the field of "Computer Science" and "Information Technology (IT)", and (iii) declared to be video game enthusiasts. We initially decided to also have as filter an active job in the video game industry. However, this resulted in only two candidates available on Prolific. Thus, we removed such a filter and relied on the initial questions of the survey to discard participants without game development experience. Based on the filters applied, we invited 248

candidate participants. We payed 5$ to each participant who completed the survey in all its parts (*i.e.,* by also providing relevant answers to the open questions).

Survey Questions. Participants were initially presented with a welcome page, which explained the goal of the study, reported its expected duration (~15 minutes) and other basic information. If they agree to participate, the main survey started, which consisted of four phases. In the first one we collected *demographic information.* As a preliminary question, we asked whether the participant had experience with game development or testing. If the answer to the latter question was no, the survey stopped and the participant was excluded from the study. Otherwise, we asked additional information (*i.e.,* job position and number of years of experience, examples of video games they worked on).

In the second phase, we asked questions aimed at understanding *whether and how developers identify stuttering events.* Specifically, we asked (i) if they actively try to identify stuttering events during beta testing of a video game; (ii) the percentage of bug reports that regards stuttering events, in their experience; and (iii) if they use telemetry to detect stuttering events. Based on the last question, the participants who declared to use telemetry were asked to indicate on a Likert scale from 1 to 5 (the higher the better): (i) what accuracy level do the telemetry-based tool achieve in detecting stuttering events, and how complete the information telemetry provides is to (ii) reproduce the issues and (iii) fix them. The participants had to motivate all their answers in open questions. Finally, we asked if they use other tools to identify stuttering events.

In the third phase, we asked questions aimed at understanding *whether developers use data available online to identify stuttering events in video games.* Specifically, we asked which platforms they usually get data from (YouTube, Twitch, Steam, Discord, or others) and to add details on how they do that (open answer).

In the fourth phase, we asked question to *assess the possible usefulness of an automated approach for identifying issues in gameplay videos.* Specifically, we asked participants to rate, on a Likert scale from 1 to 5 (the higher the better), the extent to which parts of the gameplay video documenting stuttering events would be useful for (i) replicating the problems and (ii) fixing them. We also explicitly asked what other pieces of information would be useful to replicate and fix errors that cause stuttering events. Finally, we asked (iii) to what extent they would be interested in using an automated approach for detecting stuttering events from gameplay videos on a video game they are developing/testing, and (iv) whether they would use this approach in combination with other testing approaches they already use or to replace them. Also in this case, we asked to motivate the answers. We implemented our survey in Google Forms [12]. The complete list of questions and anonymized participants' answers is available in our replication package [2].

Data Collection and Analysis. We kept the survey available to the invited practitioners for eight days because we got immediate feedback from most candidate participants. We collected a total of 53 responses: 3 have been automatically rejected by the platform because the participants exceeded the allowed time limit; 6 have been excluded since the participants declared no experience in game development/testing.

We manually analyzed the remaining 44 to assess the quality of the responses provided by the participants. We discarded the responses for which the open questions were not filled with relevant content, which we used as proxy for the commitment of the participant.

To evaluate the relevant content, we analyzed the individual responses of the participants. In detail, we analyzed the completeness and consistency of open-ended responses. Content was assessed as relevant when the response was complete and directly related to the question. We report some examples of responses that were not considered consistent with the question asked and, thus, excluded from the assessment. As for the question regarding the use of telemetry, a participant provided the definition of telemetry: "telemetry is the term used to denote any source of data [...] There are many popular applications of telemetry in games [...]." However, no motivation was provided. In

addition, another participant provided for two different questions the same generic answer. Specifically, when we asked what other pieces of information is needed to reproduce the stuttering event and to fix a problem, such a participant responded "Disable game DVR. Disable dynamic ticks and HPET. Disable visual effects. Disable Windows features you don't need. Disable full-screen optimizations. [...] Increase virtual memory. Delete unnecessary programs. Update your RAM. Reset your BIOS to optimized defaults."

We ended up with 26 valid responses. We summarize the quantitative data through boxplots and barplots, and report examples of interesting open answers we collected.

2.2 Results

Most of the participants (73%) reported that they have (or had in the past) experience as developers/testers in the game development industry, while the remaining 27% still had game development experience, but not in industry (*e.g.,* open-source projects). Out of the ones with industrial experiences, 26% are/were testers, 42% are/were developers, and the remaining 32% have/had other roles (*e.g.,* AI specialist). The average experience in game development of the participants is 2 years, with 19% of them having more than 3 years of experience. Fig. 2 summarizes the data collected for the subsequent parts of the survey we discuss below.

Identification of Stuttering Events. Most of the participants (76%) reported that they are interested in identifying stuttering events during the beta-testing of a video game. The average reported percentage of bug reports related to stuttering is ~15%, with a median ~10% (top-left part of Fig. 2). As expected, most of the participants use telemetry for detecting stuttering events (62%). While they mostly agree that telemetry is useful for detecting stuttering events (median/mode of 4), not all of them think that it can be useful for reproducing and fixing the issue (median for reproduction: 3.5; median for fixing: 4). Some participants reported that telemetry does not provide sufficient information to pinpoint the cause of the issue. For example, one practitioner reported that, sometimes, stuttering events are hardware-related and, thus, difficult to reproduce without the exact setup of the end user.

Usage of Online Platforms. The majority of participants (77%) use YouTube to collect bug-related information about games, with several of them also focusing on more communication-oriented platforms such as Discord (58%) and forums (50%). Some of the participants claim to use YouTube to collect information about game issues since "... following a video that specifically addresses the problem is often the best approach. On the other hand another participant states that "Sometimes I search the forums for common problems in games and then try to find them in the game I am testing." In addition, some participants who use more than one platform say that "If something is public enough, then it should go back to the developers and testers [...] using social media is a great way to get information about what is causing any number of problems."

We observed that the large majority of participants use YouTube, while a few of them use Twitch (~5%). While no participant explicitly reported this, it might be the case that they generally prefer the former over the latter because it contains higher-quality videos (often edited before being posted), while Twitch mostly features live/past-live videos.

Usefulness of an Approach for Identifying Stuttering Events in Gameplay Videos. The bottom part of Fig. 2 summarizes participants' feelings about the usefulness of an approach for automatically detecting stuttering events in gameplay videos. Most of them believe such an approach would be more useful to reproduce and fix stuttering-related issues than telemetry (even if marginally, given that the median, in this case, is 4 for both the questions). The word cloud in the lower-right part of Fig. 2 reports information developers might need to replicate/fix stuttering-related issues (besides gameplay videos). Most of them report *hardware* information (*CPU*- and *GPU*-related information). Finally, 86% of participants would like to use such an approach in combination with existing tools (*e.g.,* telemetry),

while the remainder said that it could replace them. None of the participants reported a lack of interest in testing such an approach (with a median of 4 and no answer below 3).

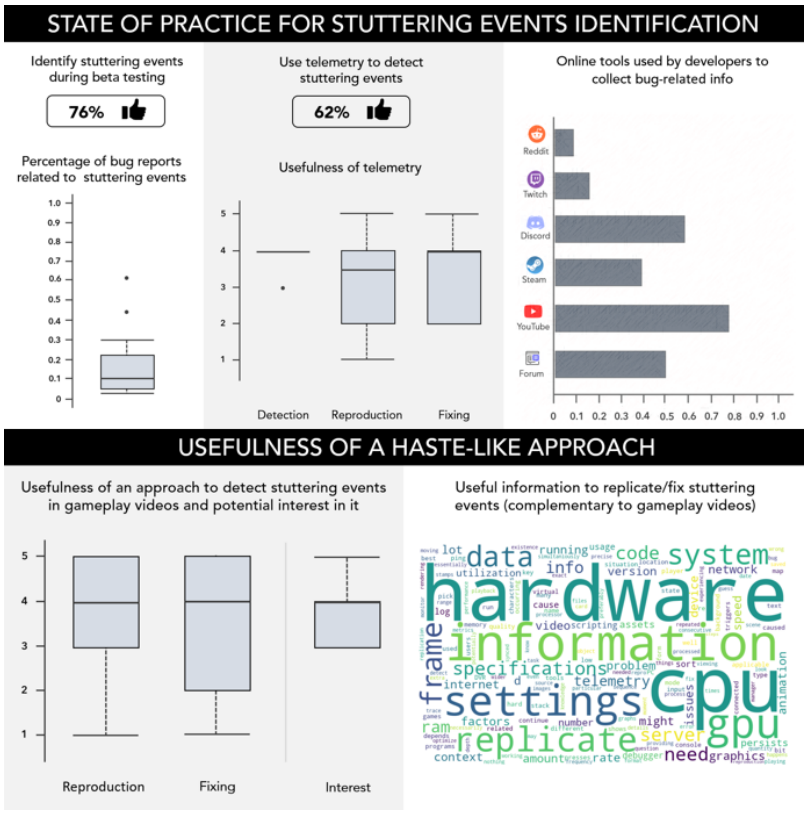

Fig. 2. Survey results.

3 HASTE: HINTER FOR GAME STUTTERING EVENTS

Fig. 3 overviews the steps behind HASTE. The process starts with a list of gameplay videos which developers want to analyze. These are supposed to be gameplay videos related to video games they maintain. The list of videos is provided to the *Gameplay Videos Crawler*, which is in charge of downloading them. Our current implementation supports the download of gameplay videos from YouTube [47] and Twitch [38]. More often than not, the downloaded videos include, besides the gameplay itself, also additional content which is not interesting to identify stuttering. For example, the player (*i.e.,* the person recording the video) might interrupt the gameplay to talk to the viewers. For this reason, HASTE implements a mechanism to identify, within a given video, the fragments representing actual gameplay. This is accomplished through two components: (i) the *Video Slicer* splits the video into different slices, based on drastic changes in the video frames (*e.g.,* the gameplay is interrupted to show an advertisement, thus causing most of the pixels on screen to change); (ii) a *Video Slice Classifier*, which is a machine learning model trained to discriminate between video slices showing and not showing gameplay. Video slices not classified as gameplay are discarded, while the "gameplay slices" are further analyzed. Indeed, it is not possible to just compare subsequent frames to check whether they are

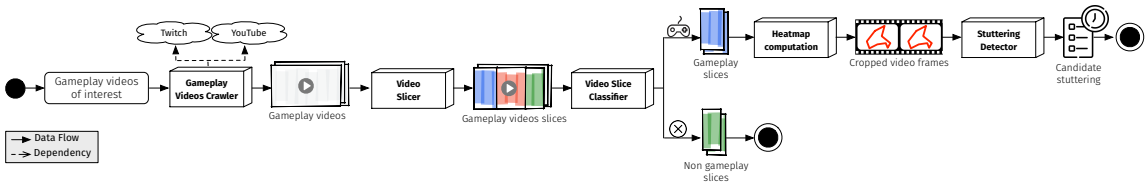

Fig. 3. HASTE overview.

identical or not to identify stuttering events. This is due to the presence on screen of additional elements such as the camera recording the player (see *e.g.,* Fig. 1). For example, it is possible that despite a game stuttering and, thus, a perfect match between the pixels representing the game in two subsequent frames, the latter are different due to the player moving or to other elements appearing on the screen (*e.g.,* subscription notifications). For this reason, HASTE features a *Heatmap Computation* step aimed at identifying, in a given gameplay slice, the pixels of its frames which change more frequently.

These are supposed to be the pixels in which the game is displayed and allow to crop the frames to only focus on this part when computing the similarity between subsequent frames. Cropped subsequent frames being almost identical are identified as candidate stuttering. In the following we detail each of the HASTE's steps.

3.1 Video Slicer

Given a gameplay video as input, the *Video Slicer* extracts and stores all its frames. This is done by using the OpenCV library [23]. To make the following steps of HASTE faster, all frames are resized to a resolution of 320×180. Each pair of subsequent frames is then compared using their color histogram representation. Color histograms allow to summarize an image as the number of its pixels having a pixel falling in specific color ranges. Using color histograms it is possible to compare images by checking whether their "distribution of colors" is similar. Histograms are a typical method used for extracting features from images and one of the most basic methodologies for computing image similarity [32]. Instead of using the classic RGB (Red, Green, and Blue) color space, we use HSL (Hue, Saturation, and Level). Such a color space allows to better detect the similarity between frames even when some effects are employed (*e.g.,* fade-in, fade-out, transitions). For example, if a fade-out effect is used, the hue and the saturation of all the pixels will likely remain the same, while the level (*i.e.,* the brightness) will uniformly reduce for all of them. A drawback of this representation is that the similarity computation ignores the shapes and texture of the compared images. However, given the goal of the *Video Slicer* (*i.e.,* splitting a video when drastic changes in the video frames are observed), a similarity computation based on color distributions is sufficient. For example, if a gameplay is interrupted by an advertisement, we expect substantial changes in the color distribution of the frames.

We compute the similarity between each pair of subsequent frames in the video using the `compareHist` function from OpenCV [23], which takes as input the two histograms to compare (*i.e.,* the two images) and the method to use for the similarity computation. For the latter, we use `HISTCMP_CORREL`, computing a correlation between the two histograms and thus returning a value between -1 and 1, with 1 indicating a high similarity between images. Following well-known guidelines used to interpret correlation coefficients [8], we split the video if two subsequent frames have a correlation lower than 0.3 (*i.e.,* no positive correlation). Such a process results in a set of video slices extracted from the input video. We discard slices shorter than 150 frames (5 seconds) since, as it will be clear later, HASTE exploits

information extracted from video slices to automatically classify them as *gameplay* or *non-gameplay* slices. A slice shorter than 5 seconds does not contain enough information to allow for a reliable classification.

It is worth highlighting that the *Video Slicer* is not meant for extracting slices representing parts of the game that humans would find *semantically coherent* but rather at detecting *visually coherent* slices. For example, let us imagine a cutscene in which two visually different characters discuss in a shot-reverse shot in which they are alternately shown on screen. We are not interested in having a single semantically coherent slice for the whole dialogue: Having a slice for each (visually coherent) camera cut is good as well, as long as gameplay and non-gameplay parts (*e.g.,* advertisements) are not in the same slice since the next step assumes this for detecting and discarding non-gameplay slices.

3.2 Video Slice Classifier

Once extracted the slices from a video under analysis, they are provided as input to a machine learner in charge of classifying them as *gameplay* or *non-gameplay* slices. HASTE exploits the Weka [42] implementation of the Random Forest algorithm [6] for this task. The Random Forest builds a collection of decision trees with the aim of solving classification-type problems, where the goal is to predict values of a categorical variable from one or more continuous and/or categorical predictor variables. In our work, the categorical dependent variable is the type of video slice (*i.e., gameplay* or *non-gameplay*), and we use features extracted from the video slice as predictor variables. Two families of features are exploited: heatmap-based and similarity-based. We detail in the following these features, while in our study design (Section 4) we explain how we trained and evaluated the classifier.

Heatmap-based features. Given the set of frames $S = \{F_1, F_2, \ldots, F_n\}$ in a given slice, we start by creating a 320×180 HM matrix in which each entry represents a pixel of the frames in S (*e.g.,* the entry in position $[1, 1]$ represents the pixel in the top-left corner of each frame). At the beginning, all HM entries are initialized to 0. Then, we perform a pixel-by-pixel comparison between each subsequent pair of frames in S (*e.g.,* F_1 *vs* F_2) and, if a pixel in position $[i, j]$ has a different color in the two frames, we increase by one the value of $HM[i, j]$. At the end of this process, HM will represent a heatmap showing how frequently each pixel in the S's frames changed. Once created HM, we compute its minimum, maximum, median, average, first and third quartile, and use those six statistics as features for the Random Forest. Our assumption is that the frequency with which the pixels change could help in discriminating *gameplay* slices from *non-gameplay* slices. For example, we assume that *non-gameplay* slices in which the gamer is talking in full screen foreground tend to have less pixel-level changes as compared to *gameplay* slices featuring the video game.

Similarity-based features. We start from the same set of frames S in the slice and compute the correlation between the color histograms of each subsequent pair of frames using `HISTCMP_CORREL`.

This process results in a distribution of correlations having $n - 1$ values, where n is the cardinality of S. Indeed, if $S = \{F_1, F_2, F_3\}$, we compute the following correlations (Cr): $Cr(F_1, F_2)$ and $Cr(F_2, F_3)$. For such a distribution we compute the same six descriptive statistics previously described and provide them as additional features to the Random Forest. The rationale here is that the color histogram similarity between frames could be different between *gameplay* and *non-gameplay* slices. For example, an advertisement on screen is likely to exploit a more variegate and dynamic set of colors as compared to a gameplay, which is usually characterized by a quite stable color scheme for a given game scene.

3.3 Cropping Frames Using the Heatmap

Video slices classified by the Random Forest as *gameplay* are further analyzed to identify in them the pixels representing the actual game. Indeed, as previously explained, before looking for stuttering events we must be able to focus the

Fig. 4. Example of heatmap applied to several frames of the same slice.

attention on the pixels relevant for the game. For a given *gameplay* video slice, we exploit the previously built heatmap *HM* and compute its 90% percentile. This means identifying the top-10% of pixels which more frequently change in the slice. It is indeed reasonable to think that the portion (*i.e.,* subset of pixels) showing the actual game on screen is the one changing more frequently, since it represents the core action of the game. Instead, parts of the screen showing ads, chat messages, or the streamer, are likely to change less frequently.

All pixels falling below the 90% percentile are removed from all frames in *S* (*i.e.,* the frames of a given video slice). In Fig. 4 we show an example of how such a process allows to focus the attention only on the part of the frames featuring the game in action. In this example we use a transparency to still make it visible the part that would be cut from the frames (*i.e.,* the one in black). Since all frames belong to the same video slice, they are all cropped using the same heatmap. The output of this step are cropped frames belonging to each *gameplay* slice (*i.e.,* frames only including the top-10% of changing pixels within the slice).

3.4 Stuttering Detector

In the last step of HASTE, the cropped frames are provided as input to the *Stuttering Detector* which is in charge of comparing them looking for candidate stuttering events. The stuttering identification is based on a simple idea: If subsequent cropped frames are identical, we can safely assume that a stuttering event is happening, since the cropped frames, as explained, are supposed to focus on the game action. Given the set of cropped frames in a video slice, we compute the similarity between each pair of subsequent frames by using the Structural Similarity Index (SSIM) [41]. SSIM has been proposed by Wang *et al.* as a way to overcome the shortcomings of classic image similarity techniques (such as the previously described color histogram similarity) and it is based on the idea of comparing the images' textures. The authors showed that the SSIM is the metric better capturing image similarity as perceived by humans [41], which is what we need to identify stuttering events (*i.e.,* cases in which humans perceive a high similarity between subsequent frames, giving the impression of lagging). We used the SSIM implementation available in the *scikit* image library [31], which returns a value between 0.0 and 1.0, with the latter identifying identical images.

We opted for a quite conservative detection of stuttering events since our goal is not to flood developers with recommendations about possible stuttering events, but rather to provide precise recommendations (even at the cost of losing some valid data points). In particular, the *Stuttering Detector* reports a candidate stuttering if the SSIM is at least $1 - \epsilon$ for two consecutive pairs of cropped frames. ϵ is a small number that developers can choose to increase or decrease the number of candidate stuttering events returned by HASTE. If ϵ is 0, even a small imprecision in the previous phases (*e.g.,* in cropping) would result in a missed detection of a stuttering event. Therefore, we set $\epsilon = 0.0001$, so that, given the resolution of our frames ($320 \times 180 = 57{,}600$ pixels), HASTE is able to tolerate a 5-pixel noise. We say that a stuttering event is there if *two* consecutive pairs of cropped frames meet the similarity threshold to further deal

with possible imprecisions of the frames-cropping stage: When we find two subsequent frames meeting our threshold, we ask for a "confirmation" to the next pair of frames as well.

The number of consecutive pairs that should meet the threshold can be further increased in case developers want to reduce the chances to obtain false-positive recommendations. Also, cases in which more than 30 subsequent pairs of frames meet the similarity threshold are not treated as stuttering, since they are more likely to be due to events such as the player pausing the game. The final output is a set of candidate stuttering events, reporting, for each analyzed video, the frames at which the issues start (if any).

4 EMPIRICAL STUDY DESIGN

The *goal* of our study is to evaluate the effectiveness of HASTE in automatically identifying stuttering events in gameplay videos. In particular, we want to assess HASTE in terms of its: (i) accuracy in automatically splitting gameplay videos into visually coherent slices; (ii) ability to correctly classify actual gameplay video slices; and (iii) ability to automatically identify stuttering events in gameplay videos. We aim at answering the following RQs:

RQ_1: *What is the accuracy of HASTE in automatically splitting gameplay videos into visually coherent slices?* RQ_1 assesses the accuracy of HASTE in automatically splitting gameplay videos into visually coherent slices (*e.g.,* fragments showing the same game scene, an advertisement). RQ_1 evaluates the *Video Slicer* (Section 3.1).

RQ_2: *To what extent is HASTE able to correctly classify video slices relevant to the gameplay?* RQ_2 evaluates the *Video Slice Classifier* (Section 3.2) focusing on its ability to classify the video fragments extracted by the *Video Slicer* as *gameplay* or *non-gameplay*.

RQ_3:*To what extent is HASTE able to automatically identify stuttering events in gameplay videos?* Our last research question evaluates HASTE as a whole, verifying whether the stuttering events it identifies are true positives.

4.1 Context Selection

We built three video datasets, each one aimed at answering one of the formulated research questions. All datasets have been extracted from either Twitch [38] or *YouTube* [47]. For the former, the *Gameplay Videos Crawler* relies on the Twitch command-line interface (CLI) [39] while for the latter it exploits the Pytube library [28]. Both streaming platforms allow to download videos by specifying characteristics of interest such as their resolution and FPS.

4.1.1 RQ_1: Dataset for the evaluation of the Video Slicer. We collected 20 gameplay videos from Twitch starting from the list of most popular videos, which is based on the popularity of the streamer and on the number of visualizations. We targeted English videos having a duration between 10 and 40 minutes and being downloadable at 30 FPS at least. Concerning the length thresholds, these have been defined to focus on videos which are long-enough to justify the usage of the *Video Slicer* (*e.g.,* slicing a 30-second video would probably be useless) while considering a maximum length representative of most videos uploaded on these platforms. Table 1 reports the exact duration of each video. Overall, this dataset features over seven hours of gameplay videos. All videos have been dowloaded in *.mp4* format in 1280 × 720 pixels resolution at 30 FPS.

4.1.2 RQ_2: Dataset for the evaluation of the Video Slice Classifier. To evaluate the *Video Slice Classifier* we need to build a dataset to train and evaluate the Random Forest model. This means creating a dataset of video slices labeled as *gameplay* or *non-gameplay*. Manually building such a dataset would be quite expensive. Thus, we designed the following automated process.

To collect instances of *non-gameplay* slices, we downloaded the top-5 videos (in terms of visualizations) from each of the top-10 YouTube non-game-related video categories (*i.e.,* basketball, cooking, football, Formula 1, motors sport, movies, music, news, sport, trends). Then, we run on them our *Video Slicer* and collected a total of 2,945 slices that we can safely label as *non-gameplay*. Similarly, as representative of *gameplay* slices, we downloaded 5 gameplay videos from each of the six game-related categories in YouTube (*i.e.,* action, adventure, beat, riders, shooter, sport-game). We made sure that the downloaded videos only contained gameplay scenes, without any sort of interruption due to advertisements, the gamer speaking, etc. Again, we run the *Video Slicer* on the 30 gameplay videos, obtaining 1,546 slices labeled as *gameplay*. This is the dataset on which the Random Forest will be trained and tested.

4.1.3 RQ_3: Dataset for the evaluation of HASTE as a whole. The third and last dataset aims at assessing the ability of HASTE in identifying stuttering events in gameplay videos. We selected 10 videos from the list of popular gameplay videos on YouTube. These videos have not been used to build the previously described datasets. Given the goal of RQ_3, we selected videos containing in their meta data (*i.e.,* title, description, comments) the word "stuttering". These videos should provide data points useful to assess the ability of HASTE in identifying stuttering events. Table 3 reports the duration of the 10 selected videos, which is ~2 hours, in total.

4.2 Data Collection and Analysis

To address **RQ_1** we ran the *Video Slicer* on the set of 20 Twitch gameplay videos, collecting a total of 1,909 extracted slices. Out of these, we manually analyzed a sample of 320 video slices, ensuring a margin of error of ±5% with a confidence level of 95%.

The estimation has been performed applying a sample size calculation formula for an unknown population [29]. The goal of the manual validation was to assure that the splitting performed by HASTE actually resulted in the creation of visually coherent slices, representing *e.g.,* a specific game scene, an introductory countdown, etc. Each of the 320 manually analyzed slices has been independently inspected by two of the authors who classified them as "visually coherent" or not. Conflicts, arisen for 13 slices (4%), have been solved through an open discussion. We report the precision of the *Video Slicer* as the percentage of reported slices which have been classified as visually coherent. Such an evaluation lacks an assessment of the recall ensured by the *Video Slicer*. In other words, we are not assessing whether points in the video which should have resulted in new slices have been missed by HASTE. To partially address this limitation, when manually inspecting the 320 slices we also verified whether each of them contained additional "valid splitting points" that have been missed by HASTE. We also report this data when answering RQ_1.

To answer **RQ_2**, we trained and tested the Random Forest classifier on the dataset of 1,546 *gameplay* and 2,945 *non-gameplay* video slices previously described. We used the WEKA's default configuration for the Random Forest classifier, *i.e.,* we set the number of trees to 100, the number of randomly chosen attributes to 0 and the maximum depth of the trees to "unlimited". We used a 10-fold cross validation to assess the performance of the trained model. Since our dataset is slightly unbalanced (66% of the slices are *non-gameplay*), we also experimented with re-balancing our training set in each of the 10-fold iterations using SMOTE [7], an oversampling method which creates synthetic samples from the minor class. Since we did not observe major improvements (results in our replication package [2]), we discuss in the paper the results without re-balancing. In particular, we report the confusion matrix output of the 10-fold validation (and, thus, the true positives, true negatives, false positives, and false negatives) and the corresponding recall and precision values for both the *gameplay* and the *non-gameplay* categories. We compare our approach with a very simple baseline that reports all the video segments as gameplay videos.

To address **RQ$_3$** we started by manually building an oracle reporting stuttering events in the 10 YouTube videos composing the dataset for this RQ. The first author looked at the overall 2 hours of videos with the goal if identifying stuttering events. The latter were documented by writing down the second in the video and the corresponding frame at which the event started. The 44 candidate stuttering events identified have been further validated by a second author, who confirmed 41 of them (7% of conflicts). An open discussion aimed at solving conflicts led to the final creation of our oracle, composed by 42 stuttering events spread across the 10 videos as reported in Table 3. HASTE has then been run on the 10 videos, collecting the 62 candidate stuttering events reported by it. Based on these, we compute its recall and precision when considering the built oracle as the ground truth.

We also manually analyzed the reported stuttering events that were outside of the oracle (*i.e.,* that we did not identified by watching the video). Indeed, the perception of a stuttering event is quite subjective and subtle in some cases (micro-stuttering). Thus, even instances not present in our oracle might be true positives. Also this analysis has been performed independently by two authors, with conflicts arisen on 1 (1.61%) of the inspected instances. Such an analysis allows to compute an overall precision for HASTE (*i.e.,* how many of the stuttering events it reports are true positives).

To better interpret the performance of HASTE, we compare it with two main baselines. The first, named *SSIM baseline*, is basically the last step of HASTE (*i.e.,* the *Stuttering Detector*) without all previous components in the pipeline. This means that a stuttering event is identified if at least two pairs of subsequent frames have a SSIM $\geq 1 - \epsilon$ (*i.e.,* 0.9999, in our case). Differently from the complete approach, the SSIM is computed (i) between all pairs of subsequent frames in the video since the *non-gameplay* slices are not discarded by the Random Forest, and (ii) on the entire frame, including external elements shown on screen but unrelated to the gameplay (*e.g.,* the player), since cropping is not applied. The *SSIM baseline* allows to assess the boost in performance provided to HASTE by all steps preceding the similarity computation. The second baseline, *Pixel-sim baseline*, is the most simple and natural approach one could devise to detect stuttering events. It computes a pixel-by-pixel similarity between subsequent frames in a video, and it says that there is a stuttering event when an exact match between them is found. In addition to those two main baselines, we also compare HASTE with a version of HASTE (HASTE ^{NoFiltering}) that does not rely on the Video Slice Classifier component, *i.e.,* which does not filter out non-gameplay. This would further allow to highlight the usefulness of such a step.

We run the baselines on the same dataset of videos used for HASTE computing, also in this case, the precision and recall with respect to the manually built oracle. The *Pixel-sim baseline* and *HASTE* ^{NoFiltering} reported a total of 95 and 213 candidate stuttering events, respectively. We manually classified all instances not matching the ones in the oracle using the same procedure previously described for HASTE and computed the baseline precision. Since the *SSIM baseline* reported a much higher number of stuttering events (388), we performed the same analysis, but on a statistically significant sample (95%±5% confidence) of those not matching our oracle (186 manually analyzed instances).

5 RESULTS DISCUSSION

We discuss the achieved results by research question.

5.1 RQ$_1$: Evaluation of Video Slicer.

Table 1 reports the accuracy of the *Video Slicer* component in identifying valid splitting points in the provided videos. For each video, we report: (i) its ID, which can be mapped to our replication package [2]; (ii) its length; (iii) the number of slices (*i.e.,* frames in which the video should be split) identified by HASTE; (iv) the size of the sample we manually

Table 1. RQ_1: Accuracy of the Video Slicer

Video	Length	#Identified	#Analyzed	TP	Precision	#Missed
1	30:46	563	94	58	0.62	8
2	13:41	63	11	7	0.64	2
3	20:28	78	13	10	0.77	1
4	16:15	53	9	9	1.00	1
5	29:17	143	24	19	0.79	4
6	15:42	61	10	7	0.70	3
7	23:53	97	16	16	1.00	2
8	19:33	180	30	20	0.66	2
9	20:39	35	6	6	1.00	0
10	35:17	28	5	4	0.80	2
11	19:45	52	9	9	1.00	1
12	29:11	108	18	13	0.72	3
13	17:56	5	1	1	1.00	0
14	24:13	83	14	12	0.86	0
15	13:10	7	1	1	1.00	3
16	18:42	25	4	4	1.00	1
17	22:36	38	6	6	1.00	0
18	23:48	22	4	3	0.75	1
19	28:49	254	43	37	0.86	4
20	17:27	14	2	2	1.00	0
Overall	**7:21:08**	**1,909**	**320**	**244**	**0.76**	**38**

analyzed; (v) the true positive (TP) instances we identified in the analyzed sample (*i.e.,* points in which the splitting was valid); (vi) the corresponding precision, computed as the number of TPs divided by the size of the analyzed sample; and (vii) the number of missed slices (*i.e.,* additional splitting points we identified that were missed by HASTE). Worth commenting is the number of slices identified by HASTE (1,909), an average of 95 per video. Such a number may look surprisingly high. However, it is worth remembering that the goal of the *Video Slicer* is not to extract slices that are meant to be visualized by humans, but to make sure that each slice embeds a set of frames having a very similar graphical layout.

Indeed, this is crucial for the subsequent steps of HASTE. For example, computing the *HM* heatmap on a set of frames visualizing completely different content would not make sense, since it would mean focusing on the top-10% of changing pixels which, however, may represent different objects (*e.g.,* in a frame, the game action, in another frame, an advertisement). Thus, the slicing must cluster together similar subsequent frames into one slice, creating a new slice when the pixels on screen substantially change. Based on our analysis, 76% of the identified splitting points are correct, representing major changes of the content depicted on screen. One may argue what false positives are in this context. In other words, how is it possible that an approach based on image similarity may identify wrong split points (*i.e.,* subsequent frames that basically represent the same scene but that are perceived as different by HASTE and thus split). This happens, for example, when a special effect is shown on screen for a few seconds (*e.g.,* some smoke appears in the game for a few frames), leading the color histogram similarity to low values between the last frame without the special effect and the first frame showing it. While HASTE slices the video at this point, this is a wrong decision, since both the frames before and after the special effect represent the exact same scene and, thus, could in theory share the same heatmap.

Fig. 5. Example of gradient effect resulting in the loss of a valid splitting point

Concerning the "#Missed Slices" column, in the 320 slices we manually inspected, we found 38 valid splitting points missed by HASTE. These are mostly due to the application of a gradient effect when the video moves from one scene (*e.g.,* the player introducing the following video content) to another (*e.g.,* the actual gameplay). The gradient effect applied over a set of frames $\{F_1, F_2, \ldots, F_n\}$ makes HASTE failing since there is no drastic change from a frame F_i to a frame F_{i+1}, with the image on screen slowly changing. However, such a process results in an overall drastic change between F_1 and F_n, which is missed by HASTE. Fig. 5 shows a concrete example of the gradient effect resulting in the loss of a valid splitting point.

The issue of missed slices, while minimal in our current dataset, poses a potential challenge for datasets with a prevalent use of gradient transitions between scenes. his effect results in a gradual change that HASTE currently fails to detect due to its reliance on identifying significant frame-to-frame changes. A possible basic solution is to extend the comparison between frames, in addition to the immediately following frames, to a larger time window, which could help identify the cumulative effect of gradual changes. By analyzing the difference between frames in a specific interval, HASTE could detect significant transitions occurring after a series of smaller changes. Alternatively, a more expensive method might include using machine learning techniques on a dataset that includes a significant number of gradient transitions. A model could allow the recognition of gradual change patterns typical of gradient effects, enabling the identification of cutoff points that would otherwise go undetected. In general, however, this remains as an open issue that future work should address.

5.2 RQ_2: Evaluation of Video Slice Classifier.

Table 2 reports the confusion matrix resulting from the classification performed by the *Video Slice Classifier*.

The rows report the *gameplay* (GP) and the *non-gameplay* (NGP) slices in the oracle, which have been classified by HASTE as reported in the columns. For example, out of the 1,537 GP slices in the oracle (1,174 + 363), 1,174 have been correctly identified by HASTE, while 363 have been misclassified as NGP. This results in a recall of 0.77 for the GP slices. The precision for GP slices is 0.80, since HASTE wrongly classifies 296 NGP slices as GP. Thus, eight out of ten slices identified as GP by HASTE are actual gameplay slices.

The precision and recall values are even better for the identification of NGP slices. While the latter are not at the core of HASTE, the excellent results achieved demonstrate that the features used for training the *Video Slice Classifier* are able to filter out slices that are irrelevant for the identification of stuttering events.

The Random Forest does also pair each prediction with a "confidence level", a value between 0.50 and 1.00 that indicates how confident the model is in the provided classification. Such a confidence level is computed as the number of classification trees in the forest that "voted" for a specific output (*e.g., gameplay*). For example, an output prediction $\langle gameplay, 0.90 \rangle$ indicates that the model is 90% confident about the *gameplay* classification. We studied how the confidence of the classifications impacts their quality. In particular, we verified whether by setting a threshold on the confidence of the classification it is possible to effectively exclude false positives (*i.e.,* NGP classified as GP). Such a

Table 2. RQ_2: Automatic classification of video slices as *gameplay* (GP) and *non-gameplay* (NGP) by Video Slice Classifier and the baseline.

Video Slice Classifier	GP	NGP	Precision	Recall
GP	1,174	363	0.80	0.77
NGP	296	2,657	0.90	0.88
Baseline	**GP**	**NGP**	**Precision**	**Recall**
GP	1,573	0	0.35	1.00
NGP	2,953	0	0.00	0.00

scenario could be interesting for developers who want to receive less "stuttering reports" from HASTE which, however, are more likely to belong to actual gameplay slices. We found that by only considering classifications having a confidence of at least 0.70 (other slices are just discarded as if they are NGP), the precision in the identification of GP slices raises to 0.89 and it further increases to 0.98 by only considering classifications having a confidence of at least 0.90. This has a price to pay in terms of recall which drops to 0.56 (for the 0.70 confidence) and 0.27 (0.90). Still, our analysis shows that using a high threshold on the classification confidence can be a viable solution for developers interested in receiving less, but more likely to be correct, recommendations.

The baseline naturally achieves a perfect recall of 1, which is higher than the one achieved by Video Slice Classifier (0.77) on gameplay segments. However, it achieves a very low precision on such a class (0.35). The practical implication of such low precision is an unacceptably high rate of false positives, which could undermine the usefulness of the classifier in real-world applications.

5.3 RQ_3: HASTE for Identifying Stuttering Events.

Table 3 compares HASTE and the two baselines in terms of recall and precision achieved with respect to the oracle, *i.e.,* the set of 42 stuttering events we manually identified in the inspected ∼2 hours of videos. For each approach and video, we provide the number of stuttering events identified (#Found), the number of true positives (*i.e.,* correctly identified events) and the corresponding recall and precision.

HASTE is able to correctly identify 30 stuttering events (0.71 recall), and nearly half of the reported stuttering events are true positives accordingly to our oracle (0.48 precision). Therefore, even by considering all candidate stuttering events not matching our oracle as "wrong", HASTE works reasonably well, with one out of two reported stuttering being correct. HASTE was not able to identify 12 stuttering events documented in our oracle. This happened mostly due to imprecisions for the Random Forest classifier that wrongly labels some slices as *non-gameplay*, thus excluding them from the stuttering analysis. For example, for the video with ID=3 we miss three out of the three stuttering events, since they fall within slices classified as *non-gameplay*. An example of such a scenario is available in the video hosted at https://youtu.be/ok9TV-lZyJk?t=56, with the slice starting at second 57 wrongly classified as *non-gameplay* by HASTE, with the consequent miss of the stuttering documented in our oracle (second 63).

In relative terms, it can be noticed that HASTE performs significantly better than both baselines. The *Pixel-sim baseline*, which simply checks whether consecutive frames are equal at pixel level, achieves the worst results in terms of recall (0.29) accompanied by a low precision as well (13%). This shows that a trivial approach is largely insufficient for the problem at hand. Several of the wrongly identified stuttering events belong to parts of the videos unrelated to gameplay, such as those showing on screen the game settings which, by their nature, are quite static and tend to

Table 3. RQ_3: Recall and precision in the identification of stuttering events we manually identified (oracle).

			HASTE				SSIM baseline				Pixel-sim baseline			
Video	**Length**	**#Oracle**	**#Found**	**TP**	**Recall**	**Prec.**	**#Found**	**TP**	**Recall**	**Prec.**	**#Found**	**TP**	**Recall**	**Prec.**
1	15:15	1	3	1	1.00	0.33	29	1	1.00	0.03	1	1	1.00	1.00
2	8:49	4	8	1	0.25	0.13	29	3	0.75	0.10	7	2	0.50	0.29
3	7:27	1	4	1	1.00	0.25	19	1	1.00	0.05	14	1	1.00	0.07
4	17:21	10	11	7	0.70	0.64	194	5	0.50	0.03	57	3	0.30	0.05
5	11:59	1	2	0	0.00	0.00	1	0	0.00	0.00	1	0	0.00	0.00
6	12:04	1	3	1	1.00	0.33	10	0	0.00	0.00	5	0	0.00	0.00
7	6:14	22	29	18	0.81	0.62	74	17	0.77	0.23	10	5	0.23	0.50
8	21:16	0	0	0	-	-	25	0	-	0.00	0	0	-	-
9	5:10	2	1	1	0.50	1.00	0	0	0.00	-	0	0	0.00	-
10	14:37	0	1	0	-	0.00	7	0	-	0.00	0	0	-	-
	2:00:12	42	62	30	0.71	0.48	388	27	0.64	0.07	95	12	0.29	0.13

feature equal pairs of subsequent frames. Also the *SSIM baseline* tends to recommend false positive stuttering events in these cases, since it does not benefit from the exclusion of *non-gameplay* slices performed by the Random Forest. The *SSIM baseline* has the worst precision (0.07), due to the additional tolerance it has compared to the other baseline. Indeed, while the *Pixel-sim baseline* identifies a stuttering only if two subsequent frames are identical, the *SSIM baseline* inherits the design decision of our approach, with the 0.9999 threshold.

Concerning the recall value (0.64), it may look surprising that *SSIM baseline* achieves a lower recall than HASTE. Indeed, the additional steps behind our approach (in particular the *Video Slicer* and the *Video Slice Classifier*) are mostly aimed at removing false positives, *i.e.,* stuttering events detected when there is no gameplay on screen. The lower recall is completely due to the mask employed in HASTE for cropping the 10% most frequently changing pixels in the slice. The top part of Fig. 6 shows how HASTE "sees" two subsequent frames before computing their SSIM, which is instead computed on the entire frames (shown in the bottom of Fig. 6) by the *SSIM baseline*. The cropped frames allow HASTE to exclude the face of the player from the similarity computation, thus correctly identifying the stuttering, since the core part of the game scene is identical between the two images. Instead, the baseline considers the changes in "irrelevant" parts of the screen which make it missing the stuttering since the similarity falls below the set threshold.

It is worth noting that on the video with ID=2 both baselines achieved a higher recall as compared to HASTE. This is due to the lack in this video of external elements on screen (*e.g.,* the player), which makes all strategies adopted in HASTE to exclude false positives useless, and just leading to the lost of valid stuttering events.

We further analyzed the stuttering events identified by the experimented approaches but not matching our oracle (*i.e.,* correct stuttering we missed while creating the oracle). This could happen especially in the case of micro-stuttering events which are hard to spot for humans, but that might still be relevant for developers [43]. We show the results of such an analysis in Table 4, in which we report, for each approach, (i) the number of events analyzed, (ii) the number of true positives found that do not belong to the oracle ($TP_{\neg O}$), and (iii) the computed precision. Remember that for the *SSIM baseline* we analyzed a statistically significant sample (186 instances) of the 388 reported events, while we analyzed all the events for HASTE and the *Pixel-sim baseline*. HASTE confirms its superiority, by achieving a 0.78 precision compared to the 0.19 of the best-performing baseline. Overall, when considering both the true positives falling and not falling within our oracle, 89% of the stuttering events detected by HASTE are actual stuttering or micro-stuttering events. Only 7 out of 62 identified events are false positives.

Table 5 reports the results of the comparison between HASTE and HASTE ^{NoFiltering}. HASTE ^{NoFiltering} is able to correctly identify a larger number of valid stuttering events as compared to HASTE (0.73 recall), but it is less precise

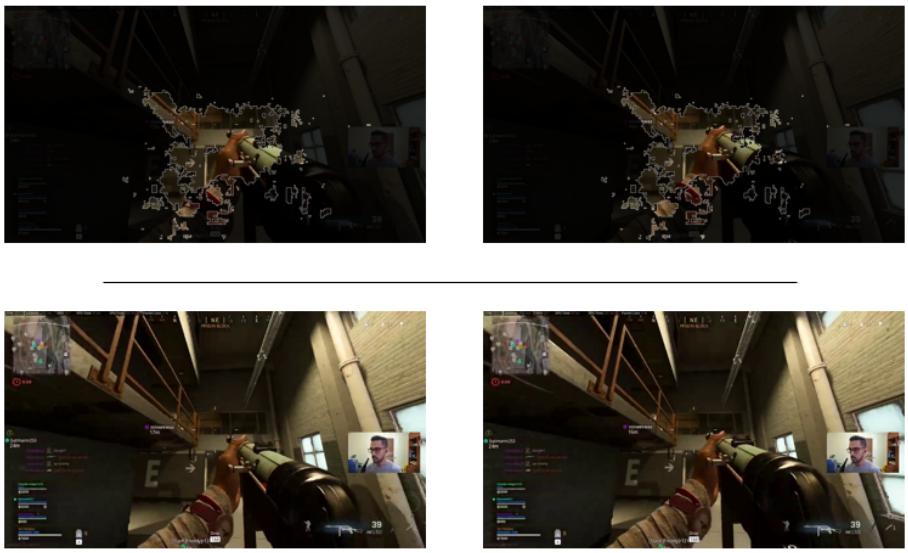

Fig. 6. Comparison between HASTE and *SSIM baseline*: On top how HASTE "sees" two subsequent frames before computing their SSIM; on the bottom the complete frame how seen by the baseline for the similarity computation

Table 4. RQ_3: Precision in identifying micro-stuttering events not in the oracle. † means we analyzed a sample of events.

	HASTE			SSIM baseline			Pixel-sim baseline		
Video	**#Analyzed**	**TP$_{\neg O}$**	**Precision**	**#Analyzed†**	**TP$_{\neg O}$**	**Precision**	**#Analyzed†**	**TP$_{\neg O}$**	**Precision**
1	2	1	0.50	14	0	0.00	0	0	-
2	7	6	0.85	13	4	0.31	5	3	0.60
3	3	2	0.67	9	1	0.11	13	4	0.31
4	4	3	0.75	98	10	0.10	54	4	0.09
5	2	1	0.50	1	0	0.00	1	0	0.00
6	2	1	0.50	5	2	0.40	5	1	0.25
7	11	10	0.91	30	19	0.63	5	1	0.25
8	0	0	-	13	0	0.00	0	0	-
9	0	0	-	0	0	0.00	0	0	-
10	1	1	1.00	3	0	0.00	0	0	-
	32	25	0.78	186	36	0.19 ± 0.05	83	13	0.16

(0.15 precision). The higher recall can be explained with wrong classifications of segments as non-gameplay, which did not allow HASTE to detect stuttering events in them. The lower precision, instead, shows that the actual non-gameplay videos that HASTE ^{NoFiltering} contain consecutive identical frames that the last step of HASTE detects as stuttering events. In summary, we can conclude that the Video Slice Classifier is fundamental to significantly reduce the number of false positives.

We further analyzed the stuttering events identified by the experimented approaches but not matching our oracle (*i.e.*,, correct stuttering we missed while creating the oracle). We show the results of such an analysis in Table 6, in which we report, for each approach, (i) the number of events analyzed, (ii) the number of true positives that do not belong to the oracle (TP$_{\neg O}$), and (iii) the computed precision. Such an analysis confirms the superiority of HASTE, which achieves 0.78 precision, compared to 0.15 obtained by HASTE ^{NoFiltering}.

Table 5. Comparison between HASTE and HASTE ^{NoFiltering} in terms of recall and precision in the identification of stuttering events we manually identified (oracle).

			HASTE ^{NoFiltering}				HASTE			
Video	**Length**	**#Oracle**	**#Found**	**TP**	**Recall**	**Prec.**	**#Found**	**TP**	**Recall**	**Prec.**
1	15:15	1	7	1	1.00	0.14	3	1	1.00	0.33
2	8:49	4	19	**2**	0.50	0.11	8	1	0.25	0.13
3	7:27	1	7	1	1.00	0.14	4	1	1.00	0.25
4	17:21	10	135	7	0.70	0.05	11	7	0.70	0.64
5	11:59	1	2	0	-	-	2	0	0.00	0.00
6	12:04	1	5	1	1.00	0.2	3	1	1.00	0.33
7	6:14	22	32	18	0.81	0.56	29	18	0.81	0.62
8	21:16	0	4	0	-	-	0	0	-	-
9	5:10	2	1	1	0.50	1.00	1	1	0.50	1.00
10	14:37	0	5	0	-	-	1	0	-	0.00
	2:00:12	42	213	31	0.73	0.15	62	30	0.71	0.48

Table 6. Comparison between HASTE and HASTE ^{NoFiltering} in terms of precision in identifying micro-stuttering events not in the oracle.s

	HASTE ^{NoFiltering}			HASTE		
Video	**#Analyzed**	**TP$_{\neg O}$**	**Precision**	**#Analyzed**	**TP$_{\neg O}$**	**Precision**
1	6	1	0.17	2	1	0.50
2	17	8	0.47	7	6	0.85
3	6	2	0.33	3	2	0.67
4	128	4	0.03	4	3	0.75
5	2	1	0.50	2	1	0.50
6	4	1	0.25	2	1	0.50
7	14	10	0.71	11	10	0.91
8	4	0	-	0	0	-
9	0	0	-	0	0	-
10	5	1	0.20	1	1	1.00
	182	28	0.15	32	25	0.78

5.4 Generalizability of HASTE

To assess the generalizability of HASTE beyond the dataset we used for evaluating it, we run it on a generic gameplay video in which we manually validated the presence of stuttering events even if was not declared in the video metadata. In detail, we started by manually building an oracle reporting stuttering events on a gameplay video. To select the gameplay video, we used Reddit to look for a video game and a specific area notoriously affected by stuttering. Specifically, we selected a video regarding a driving session from the video game Grand Theft Auto V. Such a video was a normal gameplay video for which no stuttering information was reported in the metadata (*e.g.*, title or description). The first author looked at the 18 minutes of the video with the goal of manually identifying stuttering events. As described in the evaluation of RQ$_3$ stuttering events were documented by noting the second in the video and the corresponding frame at which the event began. We identified 7 stuttering events. We then ran HASTE on such a video. It reported 8

candidate stuttering events. Based on these, we compute its recall and precision when considering the built oracle as the ground truth. We obtanied 0.86 recall and 0.75 precision. We manually analyzed the reported stuttering events that were outside of the oracle.

5.5 Discussion

In this section, we provide practitioners with suggestion on how they could use HASTE in practice.

Using Gameplay Videos from Twitch or YouTube. An interesting point to discuss regards the distinctions between acquiring gameplay videos from the two dominant platforms used for hosting gameplay videos, *i.e.,* YouTube and Twitch. Twitch has become a leader in the live streaming sector, primarily because it was the pioneer in executing the streaming model effectively. This early and successful focus on live streaming helped Twitch carve out a dominant position in this particular niche. On the other hand, YouTube has always been at the forefront of video hosting services, offering a platform where users can upload, share, and watch generic video-on-demand (VOD) content. In our study, we used YouTube videos as the main resource to evaluate our third research question (RQ_3) related to the identification of stuttering events. This choice is based on YouTube's extensive collection of curated shorter videos (between 5 and 20 minutes) showing stuttering events within games. These videos provide a focused lens to analyze and validate stuttering events identified by HASTE. On the other hand, we use videos available on Twitch to evaluate our first research question (RQ_1) related to the evaluation of video slices, where we have the ability to analyze longer videos to identify more potential scene changes within the video that may not occur in shorter videos. The extended duration of streams on Twitch provides a larger dataset for initial analysis and refinement of our approach, ensuring that HASTE can efficiently handle extended gameplay videos. On Twitch, during live streaming, streamers often pause gameplay to offer commentary or advertising content. This variation of content within a single stream increases the complexity of our analysis, allowing us to evaluate different elements within long-form video content. For this reason, we considered both platforms in our evaluation. It is worth noting that HASTE is not limited to one of them and it can be used on generic videos (from both platforms, or even others). The identification of stuttering events does not depend on the video source on which the analysis is conducted, neither from a technological point-of-view (*e.g.,* reliance on specific APIs) nor from a methodological perspective (since the only information needed is the source video).

Replicating Problems and Debugging. HASTE is mainly intended to provide developers with video bug reports from online platforms. This means that it has two key limitations. First, HASTE does not provide developers with indications on how to replicate the problem observed in the videos. Replicating a bug could be particularly difficult in video games since, in some cases, the input sequence that manifested the failure need to be provided with perfect timing for a successful replication. Besides, even when timing is perfect, the inherent non-deterministic nature of some video games might make it difficult to replicate some problems. There have been some attempts to tackle this problem in the literature [13], but the research in this area is still in its infancy and future work should specifically aim at solving this problem. A second limitation of HASTE is that it does not provide developers with feedback on how to debug and fix those issues. Achieving this goal, however, is beyond the scope of HASTE, which is meant to simply highlight problems, similarly to the several tools daily used by software developers such as linters or test cases.

Graphics Settings: Impact on Stuttering events in Video Games. The presence of stuttering events might be more prevalent when the player chooses settings that provide a better graphic experience since the hardware is more stressed in such a scenario. Nevertheless, stuttering events are *never* something expected from the players, above all during gameplay. Since the hardware resources are sometimes insufficient to provide the best graphic experience with the highest frame rate available, some video games allow the player to choose between high graphic quality with limited

Table 7. Interview participants details.

Full Name	Position	Company	Game Development Experience
Lorenzo Valente	Lead Developer	Tiny Bull Studios, Italy	7+ years
Jonathan Simeone	Full Stack Developer	Datasound, Italy	7+ years

FPS (*e.g.,* 30) or lower graphic quality with higher FPS (*e.g.,* 60 or 120). Still, such a trade-off should never result in the manifestation of stuttering events (indeed, it is provided to avoid them). Stuttering events are clearly more problematic when detected with the game's default graphics settings, which might impact the highest number of players.

Effectively Using HASTE. Through the analysis of gameplay videos HASTE lacks access to hardware information, making it challenging to definitively attribute the identified issues to software, hardware, or configuration-related limitations (*e.g.,* high quality settings on older hardware). Therefore, as any other tool, HASTE might provide false positives. To address this problems, we suggest practitioners to prioritize stuttering events based on the frequency they are reported by HASTE. If the same stuttering event is reported in several videos, it is likely that it is not just an hardware-related or configuration-related problem, but rather a game-related one. On the other hand, isolated stuttering events might indicate false positives (*e.g.,* related to the hardware or the configuration used). Besides, practitioners could use HASTE in combination with other testing tools, as also suggested by the developers we interviewed.

6 INDUSTRIAL APPLICABILITY OF HASTE

We evaluated the level of interest of game developers in HASTE by conducting semi-structured interviews. In this section, we report the design and the results of such a study. Note that, differently from the survey we preliminarily conducted to verify the relevance of the problem and of our methodology, in this case we aim at receiving feedback on the specific approach we defined (HASTE).

6.1 Interviews Design

The *goal* of this additional analysis is to assess the practical applicability of HASTE in an industrial context. Specifically, we want to assess whether game developers would consider exploiting HASTE in their testing activities to identify stuttering events in video games. We conducted semi-structured interviews and involved two *game developer* (see Table 7) to understand whether they perceive HASTE as a valuable asset that aligns with their needs and objectives in identifying and addressing stuttering events within their game development process. We selected the two participants using convenience sampling (both of them are former students at the University of Molise).

Before each interview, one of the authors explained the objective of the study and described how HASTE works. We showed that the process starts by conducting a search on YouTube for a particular game of interest. In the second step, HASTE can be executed on the given gameplay video to identify stuttering events in it. As a result, HASTE gives as output a set of times in the video in which potential stuttering events occur.

Both interviews lasted about 30 minutes and were conducted by one of the authors, who recorded and transcribed them for the following analyses. The interviews were based on a reflective strategy. In this way we encourage participants to share their experiences, thoughts and insights in a more introspective way. For example, by asking open-ended and exploratory questions, we encourage participants to reflect on their experiences and provide detailed and nuanced answers.

Table 8. Interview questions.

Question	Type of response
Stuttering Evaluation (repeated for three stuttering events)	
Does the identified stuttering event actually result from a problem within the game?	Yes/No
Please motivate your answers	Open response
Does the provided information about the stuttering event offer enough details to replicate the issue?	Yes/No
Please motivate your answers	Open response
Overall Evaluation	
Would you use this tool before the release of the game?	Yes/No
Please motivate your answers	Open response
Would you use this tool after the release of the game?	Yes/No
Please motivate your answers	Open response
Would you use HASTE in combination with other tools?	Yes/No
How useful is HASTE overall for identifying potential stuttering events?	5-point Likert scale
Please motivate your answers	Open response
Do you have suggestions on how HASTE could be improved?	Open response

We preliminarily executed HASTE on one of the gameplay videos we used for our main evaluation of HASTE randomly sampled from the ones we used and identified 3 stuttering events. In detail we use the gameplay video related to the video game *Fortnite*. We asked the participants open the video on YouTube and we told them the time at which HASTE detected each stuttering event (one at a time). We gave them the freedom to navigate the video directly on YouTube to possibly get context regarding the event. After they analyzed each of them, we asked for feedback aimed at understanding whether (i) the identified stuttering event is really an issue with the game and (ii) the information about the stuttering event is sufficient to reproduce the problem (see the top part of Table 8).

After the evaluation of the three events, we asked questions aimed at getting feedback on the whole HASTE (see the bottom part of Table 8). Specifically, we asked whether they would use this tool to identify stuttering events: (i) in the testing phase before the game release, (ii) in the testing phase after the game release, and (iii) in combination with other tools. Based on the last questions, participants were asked to indicate on a Likert scale from 1 to 5 (the higher the better): the usefulness of HASTE in identifying potential stuttering events. Finally, they were asked to provide possible suggestions on how HASTE could be improved. For all questions, participants were asked to motivate their answers.

6.2 Results

Identifying Stuttering Events. Both Lorenzo and Jonathan confirmed that the parts of the video we made them watch (reported by HASTE) contained stuttering events. As for the first question, Lorenzo stated that while the one observed is a stuttering event, it is difficult to determine from the video whether it is a problem due to the game or to the hardware. On the other hand, Jonathan claimed that the observed events are due to a possible rendering problem of the game. Specifically, he points out that, in one of them, stuttering occurs when the player turns the view and a larger game scene with more details is displayed. Similarly, in another case, the player is entering inside a narrower passage: Jonathan states it is very likely that the stuttering event is due to the game since often such parts are used to unload game elements of the previous scene and load the ones that need to be rendered later, thus causing performance issues. In detail he states, "*Regardless of the video card or the hardware, even if it has high power, if the scene is heavy, there is still a drop in fps although not quite as noticeable as it is on a less powerful video card. However, it remains an obvious optimization problem*". In addition, he points out that "*In some games where the graphics are particularly impressive, there are also scenes that, despite all possible optimization, remain heavy as they are rich in detail. Optimizing those scenes would mean having less details and thus risk losing player engagement*".

Reproduction of Stuttering Events. Both Lorenzo and Jonathan reported that HASTE does not provide sufficient information about the identified stuttering events to reproduce the conditions that caused of the issue. Again, Jonathan states "*Based on the observed stuttering events, probably due to a rendering problem, we have some information inherent to what caused the stuttering and thus might allow us to reproduce the conditions that caused the problem. [...] However, there are other aspects not known through the gameplay video that could affect the event*". For example, both point out that some of the reported problems may be hardware-related. The devices on which the video game runs, or more specifically, the video card used, may have a not negligible impact on the presence of possible stuttering events.

Practical Application of HASTE. In relation to the possibility of using this tool *before the release* of the game, Lorenzo stated that, during beta-testing, he would rather rely on testers: "*I would rely on their experience without going through gameplay videos, gather information through game logs or direct screen recordings*". On the other hand, Jonathan evinces in HASTE a support in beta-testing: "*Maybe the tester can miss some stuttering event because they are thinking about so many other things, and maybe they do not notice the micro lag. So it could be a useful tool to increase the support testing"*. In relation to the use of HASTE *after the release* of the game, both Lorenzo and Jonathan were positive. In particular, they recognized the potential of the enormous amount of information now available through the continuously evolving streaming platforms. Lorenzo states: "*This tool could be used at scale, on a much larger pool of streamers. By analyzing their gameplays, a game developer can get a lot of information. However, the actual causes of the problem remain to be considered*". Again, Jonathan shows strong confidence: "*Absolutely yes, the game continues to be tested by end buyers and I continue to monitor it. This would allow me to find any problems that passed unnoticed in beta testing*". Both participants would use HASTE in combination with other tools. In details, Lorenzo states: "*Yes, I would use it in combination with the tool provided in the engine we use in development. Specifically, I would use the engine tool first and HASTE in the second phase*". Jonathan points out: "*I would use it with any other tool that automatically allows me to identify issues missed in testing. The strength comes based on the fact that gameplay videos are always on the rise because the streamer's profession is now emerging as a real profession. So this tool can be very useful for constant monitoring*".

Usefulness of HASTE tool. In terms of usefulness of HASTE in the identification of stuttering events, Lorenzo gives a rating of 3 out of 5. He states "*I am uncertain still about the same considerations expressed above about possible problems due to hardware that could affect stuttering events. [...] However, it might be useful to get this information from streamers*". In relation to the possibility of improvement HASTE, he suggests taking information from the logs generated by games during gameplay: "*If HASTE highlights video-side stuttering events and at the same time another tool notices an abnormal sequence of error logs within the log file then there would be more certainty about what caused the stuttering event*". In relation to the same aspect, Jonathan gives a rating of 5 out of 5: "*The proposed events are actually stuttering events. Regardless of what caused them, the tool identified them. Still, more in-depth analysis of these events may be needed*". He also suggests that to improve HASTE it would be interesting to allow developers to give feedback on identified stuttering events in order to recognize false positives. Therefore, a continuous learning model could be useful to improve HASTE through feedback from those who use it.

Summing up. The two participants mostly provided positive feedback on HASTE, but emphasized its limitations. One of them was worried about false positives, while the other one was more enthusiastic about the approach. While we acknowledge that HASTE can provide (even several) false positives, we argue that this is not necessarily an impediment for all developers. The same problem affects most static code analysis tools and still some developers want to used them because they have other advantages (*e.g.,* quick feedback). As the interviews point out, also HASTE has a clear advantage: It is capable of exploiting the very large amount of information available online and reduce the effort of developers interested in analyzing gameplay videos to find and fix performance issues. In addition, the interviews highlight a key

point: The usefulness of HASTE strongly depends on its integration into the workflow of game developers. Despite our very limited sample of developers involved, the interviewees reported that they would use HASTE at different stages. Finally, the participants suggested to (i) implement a continuous learning system based on developer feedback to improve the results and (ii) integrate it with information available from game logs, which however are harder to acquire than gameplay videos and could be more easily available during beta-testing. Finally, the participants highlighted that it may be difficult in some cases to understand whether an observed lag is due to a software (game) issue or to an hardware one. While we agree, game developers may consider an identified stuttering event as relevant only if found in several videos possibly from different streamers, thus increasing the chance that the observed issue is independent from the hardware and is, thus, software-related.

7 THREATS TO VALIDITY

Construct validity. Those are mainly related to imprecisions made when building the oracles used in our evaluation and, more in general, when manually inspecting the output of HASTE and of the baselines. The manual analyses always involved at least two authors. As for the dataset used to assess the performance of the *Video Slice Classifier*, it has been automatically built starting, however, from manually-classified videos assigned in YouTube to specific categories. Thus, we are confident about the quality of the assigned *gameplay*/*non-gameplay* slices.

It is possible that the stuttering events identified in gameplay videos are not due to issues with the game itself but to external factors, such as a heavy computation running on the player's machine or suboptimal hardware configuration. However, gameplay videos are usually recorded by professional players equipped with proper hardware and interested in recording an enjoyable playing session.

Internal validity. We did not tune some of the HASTE's parameters. For instance, we discarded video slices shorter than 5 seconds, since we assumed that any sort of statistical feature we could compute on their frames (*e.g.,* the median of the *HM* heatmap distribution) would be unreliable. Similarly, we identify candidate stuttering only if at least two subsequent pairs of frames exhibit a SSIM higher than the set threshold (0.9999). The latter is a possible parameter to tune as well. Our decision of avoiding fine-tuning these parameters was driven by the high cost of manually validating different variants of HASTE. However, the lack of parameters fine-tuning does not invalidate our findings, but makes the reported performance a sort of lower-bound for what could be achieved by systematically adjusting the HASTE parameters. To answer RQ_1, we mostly relied on the labeling performed by the authors, who might have been biased in the evaluation. To alleviate this threat, we involved two external validators to evaluate the 320 manually-analyzed slices. Such validators were asked to see the labels we provided and classify them either as "correct" or "incorrect." Both the evaluators reported that all the instances had been correctly validated. This suggest that there was no significant bias, thanks to the methodology we used to label the segments. We involved the same validators to re-assess the labels we assigned to the segments to answer RQ_3, which focused on the identification of stuttering events. Specifically, we asked them to double-check the label we assigned to (i) the 42 stuttering events identified by the authors, and (ii) the ones that HASTE or any baseline classified as stuttering event (343 events), totaling 385 evaluations. We used the same methodology adopted for the check of RQ_1. Again, the evaluators reported no disagreement with our evaluations.

We set the threshold for splitting the video in segments in the Video Slicer component of HASTE to 0.3. We chose such a value because we wanted to find clear cut points. We analyzed the distribution of similarities between pairs of consecutive frames in the videos we analyzed to answer RQ_1. Fig. 7 reports the results of such an analysis. It is clear that similarity values below 0.8 are always outliers for all the videos. This shows that varying the threshold in this range would result in the inclusion/exclusion of relatively few data points (outliers).

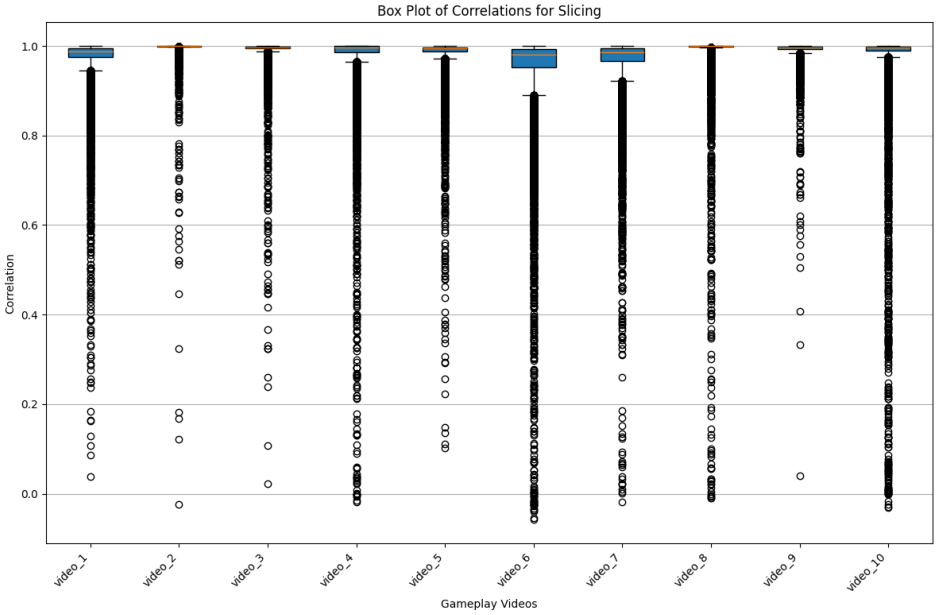

Fig. 7. Distribution of color histogram similarity values between pairs of consecutive frames.

We chose to set the threshold for the slicing at 5 seconds. It might be argued that different thresholds values might allow to obtain better results. We conducted an analysis on the video segments discarded (lower than 5-seconds). In detail, we examined the video segments of each gameplay analyzed in our study. We identified 75 segments shorter than 5 seconds, of which 58 reported a duration of 0 seconds (less than 30 FPS detected). The cumulative duration of the remaining 17 segments was 43 seconds from a total of about two hours of gameplay videos analyzed. Specifically, two segments had a duration of 4 seconds (*e.g.*,, depicting a part of gameplay where the player accesses a game map), while three segments had a duration of 3 seconds, and the remainder mostly lasted only one second. In addition, in 3 out of the 10 videos analyzed, no segments were deleted based on the 5-second threshold. We observed that these short interruptions mostly occur when lighting significantly changes in the game. For example, these short segments often correspond to bomb explosions, which light up the scene, switches to night mode via infrared, or brief accesses and configurations of game maps. We chose not to combine these short video segments because our goal is to identify scene changes to distinguish between gameplay and non-gameplay segments. Merging segments could increase the risk of classification errors. Therefore, the 5-second threshold was determined as a pragmatic balance between excluding too many short, potentially uninformative segments and maintaining the integrity of significant scene changes within our analysis framework. On the other hand, the impact of the using a slightly larger threshold would have been irrelevant. We analyzed how many segments are in the range between 5 and 10 seconds, and we observed that there are only 13 segments between 5 and 10 seconds.

We chose 3 as a minimum number of consecutive identical frames to trigger the detection of a stuttering event. This, however, could be a sub-optimal choice. To evaluate how many consecutive identical frames are needed to identify

Table 9. Identification of stuttering events with different numbers of identical consecutive frames.

	2 frames	3 frames	4 frames
Evaluator 1	0	3	3
Evaluator 2	0	1	1
Evaluator 3	0	2	2

stuttering events, we conducted an evaluation with humans to understand to what extent they could detect stuttering in different scenarios. We focused on an approximately one-minute gameplay video. We introduced stuttering events by duplicating frames at three different points in the video, creating three potential stuttering scenarios to assess the visibility and perceptibility of stuttering for users. These scenarios included (i) two consecutive identical frames, (ii) three consecutive identical frames (the validation approach used in HASTE), and (iii) four consecutive identical frames. We asked three external evaluators to watch the videos and mark the seconds in which they perceived stuttering events. The results of this validation process revealed that none of the evaluators identified stuttering with only two duplicate frames. However, in scenarios with three and four consecutive identical frames, the results varied. We report the details of the results in Table 9.

These results suggest that duplication of two consecutive frames is probably not sufficient to be perceived as stuttering by viewers, while duplication of three or more frames leads to consistent detection of stuttering, thus supporting our heuristic choice in HASTE.

External validity. We initially aimed to include in our survey developers from around the world to get a more comprehensive perspective. In the end, we decided not to do this because, in an initial version of the survey, we received several responses from bots. To reduce the manual effort in discarding such responses, we decided to limit the survey and get responses only from the United States and European countries. As a result, however, we might have missed several participants, especially from countries with significant game development industries such as Canada and Japan. To analyze the impact of their exclusion, we simulated a new run of the survey we conducted on Prolific with all the filters we used in our survey, except for the nationality, that we set to Canada and Japan. Prolific warned that fewer than 25 eligible participants would be available, and it did not report the exact number for privacy-related reasons. Thus, we believe that our selection criteria did not significantly alter the results we would have achieved by including such nationalities as well.

In our survey we could only involve 26 participants. It is possible that larger or different samples of practitioners could have resulted in different conclusions. Nevertheless, when analyzing the open answer questions, we noticed recurring themes and a lack of new themes (which might indicate that we reached saturation). For this reason, we believe that, despite the limitations, the sample is sufficient for our purposes. Being the evaluation mostly based on manual analysis (with the exception of RQ_2), we limited our study to a total of 30 videos (20 for RQ_1 and 10 for RQ_3), excluding the 75 used for RQ_2. Still, this involved multiple authors manually analyzing over 10 hours of videos. Two primary concerns revolve around the relatively small sample sizes in both the survey and structured interviews. In the initial survey, only 26 participants were involved. However, it is imperative to underscore that this subset was carefully selected from a broader population, but we focus only on participants connected with the world of video game development. This strategic selection ensured that responses were not only more informative but also aligned with the specific prerequisites outlined for our study. Similarly, the structured interviews were constrained to a mere two participants. This limitation was attributed to the challenges in interfacing with video game developers actively

engaged in their professional roles. Despite the small number, it's crucial to acknowledge that these interviews involved two senior developers. This allows us to give robust foundation to the insights gleaned, considering the experience and expertise possessed by the interviewees.

8 RELATED WORK

We discuss techniques proposed in the literature for quality assurance in video games through mining of gameplay videos and playtesting. In literature there are several works relying on video analysis in software engineering in general [4, 20, 26, 44].

In recent years, video screen recordings are becoming increasingly common for users to communicate problems to developers because they effectively convey what the user sees [5, 9]. Similarly, the analysis of gameplay videos can play a key role in game testing. Through the analysis of gameplay videos, developers can identify and document software bugs that may not be easily detected by traditional testing methods. Video-based bug reports are becoming increasingly popular for mobile applications [10, 17, 46]. Feng *et al.* [11] introduce CAPdroid, an automatic approach that use image processing and convolutional deep learning models to segment bug recordings, infer user action attributes, and generate subtitle descriptions. Krieter *et al.* [16] present a method for analyzing mobile application usage in detail by generating log files based on mobile screen output.

Video games can face a multitude of challenges. TrueLove *et al.* [35] introduce a taxonomy to identify the reported issues within these games. These challenges encompass aspects like game balance, including issues tied to the game's artificial intelligence (AI).

A strategy to support developers in finding quality issues in video games consists in analyzing gameplay videos released by players. Since the phenomenon of publishing gameplay videos is relatively recent, only a few studies have emphasized their value for identifying problems in video games.

Lewis *et al.* [18] were the first to realize the possible usefulness of analyzing gameplay video. They introduced a taxonomy of video game bugs based on a collection of gameplay videos, when the phenomenon was not yet so spread.

Mnih *et al.* [21] used gameplay videos as input for a convolutional neural network to learn how to play Atari games.

More recently, Lin *et al.* [19] manually labeled 96 gameplay videos to train and test a Machine Learning-based approach for automatically detecting gameplay videos that report functional bugs. Taesiri *et al.* [33] present a search method that retrieves relevant video from large archives of gameplay videos related only on game physics problem. To the best of our knowledge, HASTE is the first approach that automatically detects stuttering events from gameplay videos.

Previous work defined approaches to support developers in testing video games, aiming at identifying functional and nonfunctional unexpected behaviors before the release. Iftikhar *et al.* [15] proposed a model-based testing approach for performing black-box testing of platform games. A crucial challenge of such approaches is that some issues might only occur after executing a specific set of moves, which requires a certain level of intelligence. Therefore, Deep Reinforcement Learning (RL) has been explored to provide competitive and intelligent "human-like" support. Pfau *et al.* [24] introduced ICARUS, a framework for autonomous video game playing, testing, and bug reporting from which it is possible to extract information about the problems identified (*e.g.*, *crash* and *stuck* events).

Zheng *et al.* [48] present Wuji, an approach for automatically finding *crash*, *stuck*, *logical*, and *balance* problems by using evolutionary algorithms, RL and multi-objective optimization.

Wu *et al.* [45] defined an approach based on RL to perform regression testing, while Ariyurek *et al.* [3] defined synthetic and human-like agents, based on a combination of RL and Monte Carlo Tree Search (MCTS).

Ahumada and Bergel [1] introduced an approach based on genetic algorithms to allow developers to reproduce functional bugs by reconstructing the sequence of actions that lead to a specific faulty state of the game.

Guglielmi *et al.* [13] introduced GELID an automated approach to identify gameplay videos segments in which streamers reported issues. Given a set of gameplay identify issues through subtitle and image analysis. In order to extract relevant information from gameplay videos (i) identify video segments in which streamers experienced anomalies; (ii) categorize them based on their type (*e.g.,* logic or presentation); cluster them based on (iii) the context in which appear (*e.g.,* level or game area) and (iv) on the specific issue type (*e.g.,,* game crashes). In addition, TrueLove *et al.* [36] based on the work just mentioned intoduce an automated approach based on machine learning to identify whether a segment of a gameplay video contains occurrences of bugs. On the other hand, thier approach is designed to process video segments regardless of the contents of the transcript text.

To the best of our knowledge, the only approach that aims at achieving a goal similar to HASTE is RELINE, defined by Tufano *et al.* [37]. RELINE is the first technique to automatically detect game areas in which the frame-rate drops (*i.e.,* areas that might trigger stuttering events). To do this, the authors trained a RL-based agent able to play a given game with the aim of (i) achieving the best results in the game, like a player would do, and (ii) minimizing the frame-rate. While RELINE is meant to be executed by developers before the release, HASTE supports them in beta-testing and after the release, when gameplay videos from human players are available. Therefore, HASTE and RELINE play different roles in testing video games and they are not interchangeable and the two techniques can be used together.

9 CONCLUSION AND FUTURE WORK

Stuttering is a relevant problem in video game development since it can significantly impact the gaming experience and, thus, lead to poor perceived quality of the product. The identification of stuttering events is, however, quite challenging. Indeed, the "search space" in which they could manifest is huge in modern video games.

We presented HASTE, an approach that allows video game developers to automatically detect stuttering events documented in gameplay videos from Twitch and YouTube so that they can try to reproduce and fix them. We validated the three main steps of HASTE on a total of 105 videos. As for the two preliminary steps, our results show that HASTE (i) is able to accurately split videos in visually coherent slices, and (ii) is able to distinguish slices containing gameplay from the ones with other contents (*e.g.,* ads). When looking at HASTE as a whole, we found that it is able to achieve significantly better results than the baselines, with an overall 71% recall and 89% precision.

The results of two interviews we conducted with expert video game developers to assess the applicability of HASTE in an industrial context highlight its strengths in identifying potential stuttering events, leveraging the vast amount of data available online. On the other hand, such interviews also highlight the limitations of HASTE: First, it might not provide enough information to reproduce the issue, and second, it might report false positives (*i.e.,* stuttering events not due to the game but to other incidental problems). Both participants provide valuable insights on addressing current limitations.

Future work will include both a broader evaluation of HASTE and experiments aimed at fine-tuning some of its parameters. All code and data used in our study is publicly available in our replication package [2].

REFERENCES

[1] Tomás Ahumada and Alexandre Bergel. Reproducing bugs in video games using genetic algorithms. In *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*. IEEE, 1–6.

[2] Anonymus. Replication Package of "Automatic Identification of Game Stuttering via Gameplay Videos Analysis". https://figshare.com/s/600d3be6169203ce6cac.

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

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

[5] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.

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

[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[8] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge.

[9] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 957–969.

[10] Sidong Feng and Chunyang Chen. Gifdroid: Automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.

[11] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. Read It, Don't Watch It: Captioning Bug Recordings Automatically. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2349–2361.

[12] Google Forms. https://www.google.com/forms/about/. [Online].

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

[14] Gang Hu, Linjie Zhu, and Junfeng Yang. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.

[15] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. 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.

[16] Philipp Krieter and Andreas Breiter. Analyzing mobile application usage: generating log files from mobile screen recordings. In *Proceedings of the 20th international conference on human-computer interaction with mobile devices and services*. 1–10.

[17] Hiroki Kuramoto, Masanari Kondo, Yutaro Kashiwa, Yuta Ishimoto, Kaze Shindo, Yasutaka Kamei, and Naoyasu Ubayashi. Do visual issue reports help developers fix bugs? a preliminary study of using videos and images to report issues on github. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 511–515.

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

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

[20] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. Code, camera, action: How software developers document and share program knowledge using YouTube. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 104–114.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[22] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE international conference on software testing, verification and validation (icst)*. IEEE, 33–44.

[23] OpenCV. https://opencv.org. [Online].

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

[25] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 90–99.

[26] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 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.

[27] Prolific. https://www.prolific.co/. [Online].

[28] PyTube. https://github.com/pytube/pytube. [Online].

[29] B. Rosner. *Fundamentals of Biostatistics* (7th edition ed.). Brooks/Cole, Boston, MA.

[30] Satista. https://www.statista.com/topics/868/video-games. [Online].

[31] Scikit-image. https://scikit-image.org. [Online].

[32] John Richard Smith. *Integrated spatial and feature image systems: Retrieval, analysis and compression*. Columbia University.

[33] Mohammad Reza Taesiri, Finlay Macklon, and Cor-Paul Bezemer. CLIP meets GamePhysics: Towards bug identification in gameplay videos using zero-shot transfer learning. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 270–281.

[34] The Last of Us. https://youtu.be/yH5MgEbBOps?t=3494. [Online].

[35] Andrew Truelove, Eduardo Santana de Almeida, and Iftekhar Ahmed. 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.

[36] Andrew Truelove, Shiyue Rong, Eduardo Santana de Almeida, and Iftekhar Ahmed. Finding the Needle in a Haystack: Detecting Bug Occurrences in Gameplay Videos. *arXiv preprint arXiv:2311.10926* (2023).

[37] Rosalia Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, and Gabriele Bavota. 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

[38] Twitch. https://www.twitch.tv/. [Online; June 2011].

[39] Twitch-dl. https://github.com/ihabunek/twitch-dl/. ([n. d.]). [Online].

[40] Twitch Stream Time. https://twitchtracker.com/statistics/stream-time. [Online].

[41] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

[42] WEKA. http://www.cs.waikato.ac.nz/ml/weka/. [Online].

[43] What is microstuttering and how do I fix it. https://www.pcgamer.com/what-is-microstutter-and-how-do-i-fix-it/. [Online].

[44] Huijuan Wu, Yuepu Guo, and Carolyn B Seaman. 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.

[45] Yuechen Wu, Yingfeng Chen, Xiaofei Xie, Bing Yu, Changjie Fan, and Lei Ma. Regression Testing of Massively Multiplayer Online Role-Playing Games. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 692–696.

[46] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[47] YouTube. https://www.youtube.com. [Online; June 2011].

[48] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning.

[49] Georg Zoeller. Game development telemetry in production. *Game analytics: Maximizing the value of player data* (2013), 111–135.