GIFdroid: An Automated Light-weight Tool for Replaying Visual Bug Reports

Sidong Feng
Monash University
Melbourne, Australia
sidong.feng@monash.edu

Chunyang Chen
Monash University
Melbourne, Australia
chunyang.chen@monash.edu

ABSTRACT

Bug reports are vital for software maintenance that allow users to inform developers of the problems encountered while using software. However, it is difficult for non-technical users to write clear descriptions about the bug occurrence. Therefore, more and more users begin to record the screen for reporting bugs as it is easy to be created and contains detailed procedures triggering the bug. But it is still tedious and time-consuming for developers to reproduce the bug due to the length and unclear actions within the recording. To overcome these issues, we propose `GIFdroid`, a light-weight approach to automatically replay the execution trace from visual bug reports. `GIFdroid` adopts image processing techniques to extract the keyframes from the recording, map them to states in GUI Transitions Graph, and generate the execution trace of those states to trigger the bug. Our automated experiments and user study demonstrate its accuracy, efficiency, and usefulness of the approach.

Github Link: https://github.com/sidongfeng/gifdroid
Video Link: https://youtu.be/5GIw1Hdr6CE
Appendix Link: https://sites.google.com/view/gifdroid

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

bug replay, visual recording, android testing

ACM Reference Format:
Sidong Feng and Chunyang Chen. 2022. GIFdroid: An Automated Light-weight Tool for Replaying Visual Bug Reports. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516857

1 INTRODUCTION

Software maintenance activities are known to be generally expensive and challenging and one of the most important maintenance tasks is to handle bug reports [9]. A good bug report is detailed with clear information about what happened and what the user expected to happen. It goes on to contain a reproduction step or stack trace to assist developers in reproducing the bug, and supplement information such as screenshots, error logs, and environments. However, clear and concise bug reporting takes time, especially for non-developer or non-tester users who do not have that expertise and are not willing to spend that much effort. Video-based bug reports significantly lower the bar for documenting the bug. First, it is easy to record the screen as there are many tools available, some of which are even embedded in the operating system by default like iOS [5] and Android [7]. Second, video recording can include more detail and context such as configurations, and parameters, hence it bridges the understanding gap between users and developers.

Despite the pros of the video-based bug report, it still requires developers to manually check each frame in the video and repeat it in their environment. According to our empirical study of 13,587 bug recordings from 647 Android apps in our previous study [18], one video is of 148.29 frames on average with a varied resolution makes it difficult for developers to replay them in their setting. Such phenomenon is further exacerbated as 74.2% of recordings are without touch indicators on the screen, resulting in poor action navigation. In addition, only 6.8% of video recordings start from the app launch and most recordings begin 2-7 steps before the bug occurrence, indicating that developers need to guess steps to the entry frame of the video by themselves. Therefore, it is necessary to develop an automated bug replay tool from video-based bug reports to save developers' effort in a bug fix.

There are many related works on bug replay but rarely related to visual bug reports. Some researchers [33] leverage the natural language processing methods with program analysis to generate the test cases from the textual descriptions in bug reports. However, those approaches do not apply to video-based bug reports. There are many works on visual artifacts including recording, tutorials, bug reports [12, 13, 16, 32], but they are specifically for usability and accessibility testing [14, 19, 20, 29, 31]. There are platforms providing both video recording and replaying functionalities [1, 11] which also store the low-level program execution information. They require the framework installation or app instrumentation which is too heavy for end users. Bernal et al [11] proposed a tool named V2S which leverages deep learning techniques to detect touch indicator captured in a video recording, and translate it into a replayable test script. However, it requires high-resolution recording with touch indicators, and complete recording from the app launch to the bug occurrence, which is hard to get in real-world bug reports [18].

We introduce `GIFdroid`, a light-weight image-processing approach to automatically replay the general video (GIF) based bug reports for Android apps. First, we extract keyframes (i.e., fully

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516857

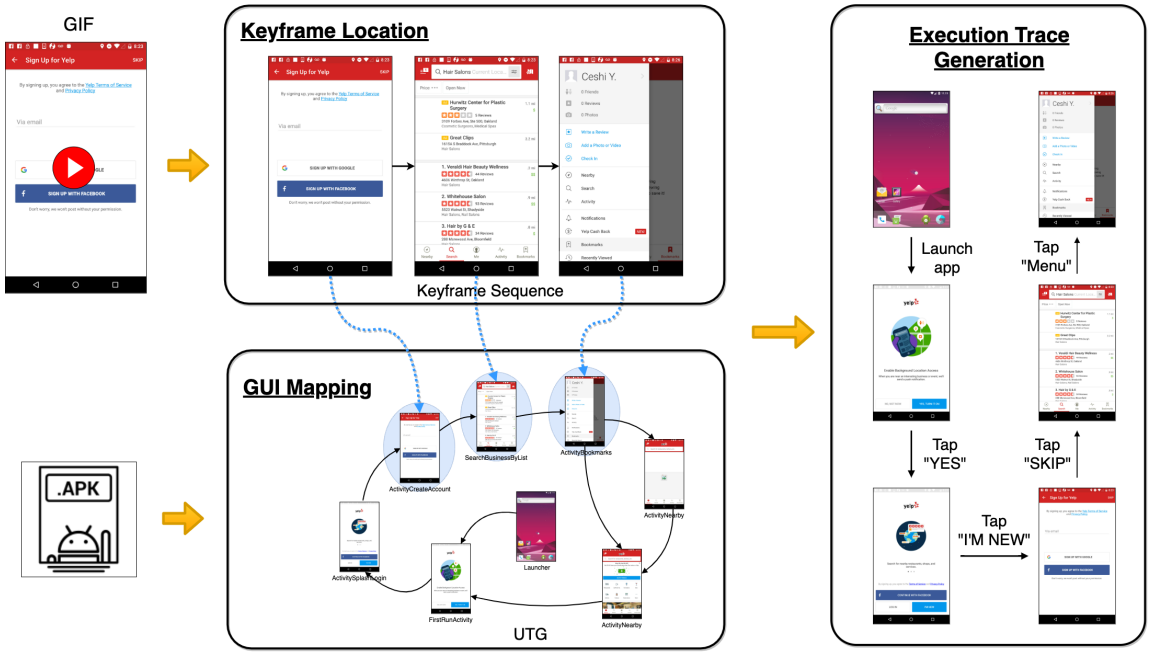

Figure 1: The overview of GIFdroid.

rendered GUIs) of a recording by comparing the similarity of consecutive frames. Second, a sequence of located keyframes is then mapped to the GUI states in the existing UTG (UI Transition Graph) of the app by calculating image similarity based on pixel and structural features. Third, given the mapped sequence, we propose a novel algorithm to not only address the defective mapped sequence, but also auto-complete the missing trace between app launch to the entry frame of the video, resulting in an optimal execution trace to automatically repeat the bug trigger.

To evaluate the effectiveness of our tool, we evaluate our approach for 61 video recordings from 31 apps. Our approach significantly outperforms other baselines and successfully reproduce 82% video recordings. Apart from the accuracy of our tool, we also evaluate the usefulness of our tool by conducting a user study on replaying bugs from 10 real-world video recordings in GitHub. Through the study, we provide the initial evidence of the usefulness of `GIFdroid` for bootstrapping bug replay.

This paper makes the following contributions:

- We present the first light-weight image-processing based approach, `GIFdroid`, to reproduce bugs for Android apps directly from the general GIF recordings.
- A comprehensive evaluation including automated experiments and a user study to demonstrate the accuracy, efficiency and usefulness of our approach.

2 OUR FULLY AUTOMATED APPROACH

Given an input bug recording, we propose an automated approach to localize a sequence of keyframes in the GIF and subsequently map them to the existing UTG (UI Transition Graph) to extract the execution trace. The overview of our approach is shown in Figure 1, which is divided into three main phases: (i) the *Keyframe Location* phase, which identifies a sequence of keyframes of an input visual recording, (ii) the *GUI Mapping* phase that maps each located keyframe to the GUIs in UTG, yielding an index sequence, and (iii) the *Execution Trace Generation* phase that utilizes the index sequence to detect an optimal replayable execution trace. A detailed overflow of our approach which contains examples of elaborated diagrams, pseudocode, approaches is shown in our previous work [18].

2.1 Keyframe Location

Note that GUI rendering takes time, hence many frames in the visual recording are showing the partial rendering process. The goal of this phase is to locate keyframes i.e., states in which GUI are fully rendered in a given visual recording.

Inspired by signal processing, we leverage the image processing techniques to first build a perceptual similarity score for consecutive frame comparison based on Y-Diff. Y-Diff is the difference in Y (luminance) values of two images in the YUV color space. We adopt the luminance component because people perceive a sequence of graphics changes as a motion and luminance is a major input for the human perception of motion. To calculate the Y-Diff for consecutive frame, we apply the perceptual comparison metric, SSIM (Structural Similarity Index) [28]. To further make decisions on whether the frame is a keyframe, we look into the similarity scores of consecutive frames in the visual recording. We find that the keyframe tends to be *steady* state where the consecutive frames are similar for a relatively long duration. We empirically set threshold to decide whether two frames are similar, and duration to localize the keyframe of recording.

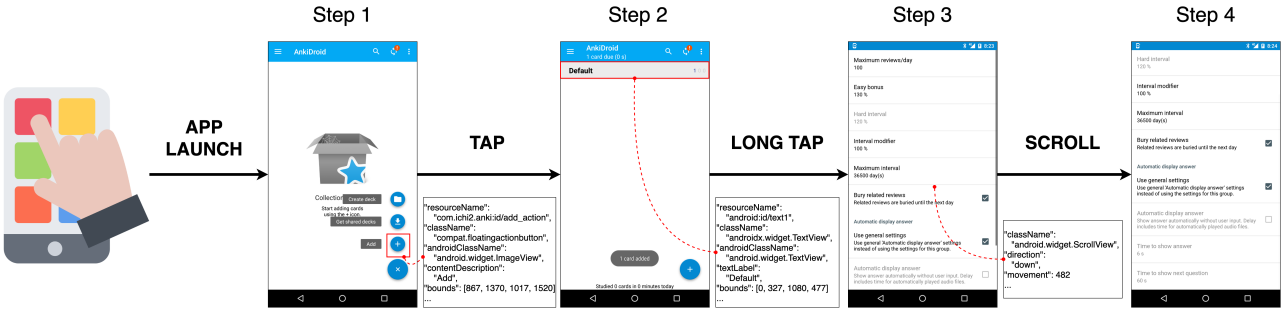

Figure 2: Example of execution trace generated by `GIFdroid`.

2.2 GUI Mapping

It is easy for developers to get the UI transitions graph (UTG) of their own app [15, 30], which is widely used to illustrate the transitions across different GUIs triggered by typical elements such as pop-ups, text boxes, text view objects, spinners, list items, progress bars, checkboxes. In this paper, we adopt the Firebase [2] to collect UTG, a widely-used automated GUI exploration tool developed by Google, while other tools can also be used.

Once we have the UTG, we infer actions from the recording by directly mapping the keyframes extracted from the recording to states/GUIs within the UTG. To achieve this, we first extract both pixel and structural features of the keyframe and each GUI screenshot by using SSIM [28] and ORB (Oriented FAST and Rotated BRIEF) [23]. While SSIM detects the features within pixels and structures, it still has several fundamental limitations that exist in visual recordings, e.g., image distortion [26]. To address this, we further supplement a robust local invariant feature extraction method, ORB. Based on the features extracted by SSIM and ORB, we compute a similarity value S_{ssim} and S_{orb}, respectively. We then determine the similarity S_{comb} between the keyframe and states in UTG by combining two feature similarities score: $S_{comb} = w \times S_{orb} + (1 - w) \times S_{ssim}$ where w is a weight for S_{ssim} and S_{orb}, taking a value between 0 to 1. Smaller w value weights S_{ssim} more heavily, and larger value weights S_{orb} more heavily. We empirically choose 0.5 as the w value for the best performance.

Based on the combined similarity between the keyframe and each GUI screenshot, we select the highest score to be the index of the keyframe. Consequently, a sequence of keyframes is converted to a sequence of the index in the UTG.

2.3 Execution Trace Generation

After mapping keyframes to the GUIs in the UTG, we need to go one step further to connect these GUIs/states into a trace to replay the bug. However, this process is challenging due to two reasons. First, the extracted keyframe (Section 2.1) and mapped GUIs (Section 2.2) may not be 100% accurate, resulting in a mismatch of the groundtruth trace. Second, different from the uploaded GIF which may start the recording anytime, the recovered trace in our case must begin from the launch of the app.

Therefore, the trace generation algorithm needs to consider both the wrong extraction/mapping in our previous steps, and the missing trace between the app launch and first keyframe in the visual bug report. To overcome these issues, our approach first generates all candidate sequences in UTG between the app launch to the last keyframe from GIF by adopting Depth-First Search traversal (DFS), that takes a path on UTG and starts walking on it and check if it reaches the destination. To avoid cyclic path, we record all visited nodes, so that one node cannot be visited twice. By regarding the extracted keyframes as a sequence, our approach then further extracts the Longest Common Subsequence (LCS) between it and all candidate sequences. Once the LCSs are detected, we select the candidate sequence that has the longest LCS as the execution trace due to it replays most keyframes (or index nodes) in the visual recording. Besides, our goal is to help developers reproduce the bug with the least amount of time/steps. Therefore, we choose the optimal execution trace with the shortest sequence.

3 TOOL IMPLEMENTATION AND USAGE

`GIFdroid` was implemented entirely in Python with modularity to encourage reuse and extension by future developers and researchers. To obtain the efficiency of `GIFdroid`, we utilized the robust implementation for each approach, i.e., Open-CV [3] for ORB and Scikit-Image [6] for SSIM. Time complexity is an important metric to measure the efficiency of algorithm. If an algorithm has to scale, it computes the result within a finite and practical time bound even for large UTG of n vertices and m edges. We implemented dynamic programming to reduce the time complexity of execution trace generation to $O(n^3)$.

Since `GIFdroid` is fully automated, our model can automatically deal with the bug recording immediately once uploaded. Note that our tool can be finished offline, especially for long recordings which require much processing time. Figure 2 shows an example of the output (i.e., execution trace) generated by `GIFdroid`. It is similar to "how-to" instruction to guide developers to repeat the actions, such as action type, where to tap. It can be further proceeded to generate sendevent commands for fully automating the bug replay.

4 EVALUATION

The goal of our study is to evaluate the performance of our tool `GIFdroid` in terms of accuracy, efficiency, and usability. We first conducted a large-scale experiment for automated evaluating of our approach consists of three main phases including keyframe location, GUI mapping, and trace generation. Then, we conducted a user study to evaluate the usefulness of the generated execution

trace for replaying visual bug recording into real-world development environments. We elaborate more detailed information in our previous work [18].

4.1 Automated Evaluation

We artificially constructed a dataset as the groundtruth for evaluating each step within our approach, instead of using the real-world bug recordings due to two reasons. First, many real-world bug reports have been fixed and the app is also patched, but it is hard to find the corresponding previous version of the app for reproduction. Second, the replay of some bug reports (e.g., financial, social apps) requires much information like authentication/database/hardware to generate the UTG which are beyond the scope of this study. Therefore, we manually generated 61 visual recordings from 31 open-source Android apps [11]. They are top-rated on Google Play covering 14 app categories (e.g., development, productivity, etc.). To make the artificial recordings as similar as the real-world one, we generated the visual recordings by different creation tools (32 from video conversion, 22 from mobile apps, 7 from emulator screen recording), varied resolutions (27 1920 × 1080, 23 1280 × 800, 11 900×600), diverse length (30-305 frames), and differed playing speed (7-30 frames per second). We asked two experienced developers to manually label keyframes from recordings, GUI mapping between recording and UTG, and real trace in the UTG as the groundtruth for each phase. To mitigate the potential threat of data labeling, each human annotator finished the labelling individually and they discussed the difference until an agreement was reached. Finally, we obtained 539 reproduction steps as our evaluation dataset.

Keyframe location: We employed three evaluation metrics (i.e., precision, recall, F1-score) and set up four state-of-the-art keyframe extraction methods as the baselines, including ILS-SUMM [24], Hecate [25], PySceneDetect [4], and Comixify [22]. The performance of `GIFdroid` is much better than that of other baselines, i.e., 32%, 106%, 14% boost in recall, precision, and F1-score compared with the best baseline. The issues with these baselines are that they are designed for general videos which contain more natural scenes like human, plants, animals etc. However, different from those videos, our visual bug recordings belong to artificial artifacts with different rendering processes. Therefore, considering the characteristics of visual bug recordings, our approach can work well in extracting keyframes.

GUI mapping: We adopted Precision@k and compared it with 10 mature image processing baselines, including pixel level (e.g, euclidean distance [17], color histogram [27], fingerprint [8]), and structural level (e.g., SSIM [28], SIFT [21], SURF [10], ORB [23]). In contrast with baselines, our method outperforms in all metrics, 85.4%, 90.0%, 91.3% for Precision@1, Precision@2, Precision@3 respectively. Our method that combines SSIM and ORB leads to a substantial improvement (i.e., 9.7% higher) over any single feature, indicating that they complement each other.

Execution trace generation: We calculated the sequence similarity as the metric and set up one video replay generation method V2S [11] and an ablation study of GIFdroid without LCS as our baselines. Our method achieves 89.59% sequence similarity which is much higher than that of baselines. In addition, adding LCS can mitigate the errors introduced in the first two steps in our approach, resulting in a boost of performance from 82.63% to 89.59%. Although applying LCS takes a bit more runtime (i.e., 13.25 seconds on average), it does not influence its real-world usage as it can be automatically run offline. In detail, `GIFdroid` fully reproduces 82% (50/61) of the visual recordings, signals a strong replay-ability.

We manually check the instances where our method failed to reproduce scenarios. Instances where slightly biased are largely due to inaccuracies in keyframe location (i.e., missing keyframes) and GUI mapping (i.e., incorrect GUI mapping). Instances that failed to reproduce are due to the inaccuracies on the last index, as our method depends on the last index to end the search.

4.2 Usefulness Evaluation

We recruited 8 participants including 6 graduate students (4 Master, 2 Ph.D) and 2 software developers to participate in the experiment. All students have at least one-year experience in developing Android apps and have worked on at least one Android apps project as interns in the company. Two software developers are more professional and work in a large company (Alibaba) about Android development. We first gave them an introduction to our study and also a real example to try. Each participant was then asked to reproduce the same set of 10 randomly selected visual bug recordings from GitHub which were of diverse difficulty ranging from 6 to 11 steps until triggering bugs. The study involved two groups of four participants: the experimental group P_1, P_2, P_3, P_4 who got help with the generated execution trace by our tool, and the control group P_5, P_6, P_7, P_8 who started from scratch. We recorded the time used to reproduce the visual bug recordings in Android. Participants had up to 10 minutes for each bug replay.

Although most participants from both experimental and control groups can successfully finish the bug replay on time, the experiment group reproduced the visual bug recording much faster than that of the control group (with an average of 171.4 seconds versus 65.0 seconds). In fact, the average time of the control group is underestimated, because three bugs fail to be reproduced within 10 minutes, which means that participants may need more time. In contrast, all participants in the experiment group finished all the tasks within 2 minutes.

We summarised two reasons from their feedback why it takes the control group more time to finish the reproduction than the experiment group. First, some visual recording is quite complicated which requires participants in the control group to watch the visual recordings several times for following procedures. The GUI transitions within the recording may also be too fast to follow, so developers have to replay it. Second, it is hard to determine the trigger from one GUI to the next one. That trial and error makes the bug replay process tedious and time-consuming. It is especially severe for junior developers who are not familiar with the app code.

5 CONCLUSION

The visual bug recording is trending in bug reports due to its easy creation and rich information. To help developers automatically reproduce those bugs, we propose `GIFdroid`, an image-processing approach to covert the recording to executable trace to trigger the bug in the Android app. Our automated evaluation shows that our

GIFdroid can accurately reproduce 82% (50/61) visual recordings from 31 Android apps. The user study on replaying 10 real-world visual bug recordings confirms the usefulness of our GIFdroid in boosting developers' productivity.

In the future, we will take the human factor into the automated approach consideration, exploring how human collaborate with the machine for replaying the bugs in the visual bug report.

REFERENCES

[1] 2021. Android Developer: Playback capture. https://developer.android.com/guide/topics/media/playback-capture.

[2] 2021. Firebase - Build and Run Successful Apps. https://firebase.google.com/.

[3] 2021. OpenCV. https://opencv.org/.

[4] 2021. Python and OpenCV-based scene cut/transition detection program & library. https://github.com/Breakthrough/PySceneDetect.

[5] 2021. Record the screen on your iPhone, iPad, or iPod touch. https://support.apple.com/en-us/HT207935.

[6] 2021. Scikit-Image. https://scikit-image.org/.

[7] 2021. Take a screenshot or record your screen on your Android device. https://support.google.com/android/answer/9075928?hl=en.

[8] Mohammad A Alsmirat, Fatimah Al-Alem, Mahmoud Al-Ayyoub, Yaser Jararweh, and Brij Gupta. 2019. Impact of digital fingerprint image quality on the fingerprint recognition accuracy. *Multimedia Tools and Applications* 78, 3 (2019), 3649–3688.

[9] John Anvik, Lyndon Hiew, and Gail C Murphy. 2005. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 35–39.

[10] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *European conference on computer vision*. Springer, 404–417.

[11] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.

[12] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.

[13] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.

[14] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.

[15] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.

[16] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. *arXiv preprint arXiv:2101.09194* (2021).

[17] Per-Erik Danielsson. 1980. Euclidean distance mapping. *Computer Graphics and image processing* 14, 3 (1980), 227–248.

[18] Sidong Feng and Chunyang Chen. 2021. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. *arXiv preprint arXiv:2112.04128* (2021).

[19] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, Tingting Zhou, and Yankun Zhen. 2021. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*. 59–69.

[20] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 398–409.

[21] David G Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.

[22] Maciej Pesko, Adam Svystun, Paweł Andruszkiewicz, Przemysław Rokita, and Tomasz Trzciński. 2019. Comixify: Transform Video Into Comics. *Fundamenta Informaticae* 168, 2-4 (2019), 311–333.

[23] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.

[24] Yair Shemer, Daniel Rotman, and Nahum Shimkin. 2019. ILS-SUMM: Iterated Local Search for Unsupervised Video Summarization. *arXiv preprint arXiv:1912.03650* (2019).

[25] Yale Song, Miriam Redi, Jordi Vallmitjana, and Alejandro Jaimes. 2016. To click or not to click: Automatic selection of beautiful thumbnails from videos. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 659–668.

[26] Shiqi Wang, Abdul Rehman, Zhou Wang, Siwei Ma, and Wen Gao. 2011. SSIM-motivated rate-distortion optimization for video coding. *IEEE Transactions on Circuits and Systems for Video Technology* 22, 4 (2011), 516–529.

[27] Xiang-Yang Wang, Jun-Feng Wu, and Hong-Ying Yang. 2010. Robust image retrieval based on color histogram of local feature regions. *Multimedia Tools and Applications* 49, 2 (2010), 323–345.

[28] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[29] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't Do That! Hunting Down Visual Design Smells in Complex UIs against Design Guidelines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 761–772.

[30] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[31] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1286–1297.

[32] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 748–760.

[33] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.