5-2023

Widget detection-based testing for industrial mobile games

Xiongfei WU

Jiaming YE

Ke CHEN

Xiaofei XIE
Singapore Management University, xfxie@smu.edu.sg

Ruochen HUANG

See next page for additional authors

Citation

WU, Xiongfei; YE, Jiaming; CHEN, Ke; XIE, Xiaofei; HUANG, Ruochen; MA, Lei; and ZHAO, Jianjun. Widget detection-based testing for industrial mobile games. (2023). *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice, Melbourne, Australia, May 14-20*. 173-184.

Available at: https://ink.library.smu.edu.sg/sis_research/8229

Author

Xiongfei WU, Jiaming YE, Ke CHEN, Xiaofei XIE, Ruochen HUANG, Lei MA, and Jianjun ZHAO

2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 979-8-3503-0037-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEIP58684.2023.00021

Widget Detection-based Testing for Industrial Mobile Games

Xiongfei Wu[1], Jiaming Ye[1], Ke Chen[2], Xiaofei Xie[3], Yujing Hu[2*]
Ruochen Huang[4], Lei Ma[4,5*] and Jianjun Zhao[1*]
[1]Kyushu University, Japan [2]Fuxi AI Lab of NetEase, China
[3]Singapore Management University, Singapore
[4]University of Alberta, Canada [5]The University of Tokyo, Japan

***Abstract*—The fast advances in mobile hardware and widespread smartphone usage have fueled the growth of global mobile gaming in the past decade. As a result, the need for quality assurance of mobile gaming has become increasingly pressing. While general-purpose testing methods have been developed for mobile applications, they become struggling when being applied to mobile games due to the unique characteristics of mobile games, such as dynamic loading and stunning visual effects. There comes a growing industrial demand for automated testing techniques with high compatibility (compatible with various resolutions, and platforms) and non-intrusive characteristics (without packaging external modules into the source code, e.g., POCO). To fulfill these demands, in this paper, we introduce our experience in adopting the widget detection-based testing technique WDTest, for mobile games at *NetEase Games*. To this end, we have constructed by far the largest graphical user interface (GUI) dataset for mobile games and conducted comprehensive evaluations on the performance of state-of-the-art widget detection techniques in the context of mobile gaming.**

We leverage widget detection techniques to develop WDTest, which performs automated testing using only screenshots as input. Our evaluation shows that WDTest outperforms the widely used tool Monkey in achieving three times more coverage of unique UI in gaming scenarios. Our further experiments demonstrate that WDTest can be applied to general mobile applications without additional fine-tuning. Furthermore, we conducted a thorough survey at NetEase Games to gain a comprehensive understanding of widget detection-based testing techniques and identify challenges in industrial mobile game testing. The results show that testers are overall satisfied with the compatibility testing aspect of widget detection-based testing, but not much with functionality testing. This survey also highlights several unique characteristics of mobile games, providing valuable insights for future research directions.

***Index Terms*—Mobile game testing, GUI testing, GUI detection, software quality assurance.**

I. Introduction

Mobile games have experienced significant gains in popularity due to the advance of mobile devices over the past decade. According to the industry reports [1], the global mobile games market would surpass 112 billion dollars in 2022. In a bid to attract users, modern mobile game industries are often incorporating visually stunning effects and intricate interactions into their products. Meanwhile, quality assurance for industrial mobile games has proven its unrivaled importance. For example, inside *NetEase Games*, one of the leading companies in the global game market, a display issue concerning the character's clothes in a role-playing game (RPG) can lead to a significant decrease in the top-up system of users.

Given the significance of quality assurance, mobile applications often undergo systematic testing before being shipped to users. Although there exists research on testing mobile applications, ranging from simple while useful method Monkey [2], to more sophisticated methods like fuzzing [3], model-based GUI testing [4], and machine learning-based methods [5, 6], these methods struggle to fully meet the requirements for automated mobile game testing in industrial contexts. The demanding nature of task accomplishment and high user interaction in mobile games makes it challenging for these methods to achieve high code/scenario coverage. As a result, current industrial practices for mobile game testing still mainly rely on intensive manual testing and semi-automated testing (e.g., through drafting testing scripts), which are both expensive and limited in scale.

With the advance of testing techniques, visual testing tools that harness the power of computer vision (CV) techniques to identify and recognize UI elements from screenshots have demonstrated their efficacy in both industrial [7] and academic contexts [8, 9]. However, it is still unclear whether widget detection-based testing methods can be applicable to mobile games, and to what extent the potential of these testing methods can be in promoting mobile game testing.

Therefore, in this paper, we introduce our experience of developing and deploying our widget detection-based testing framework, WDTest, in NetEase Games. NetEase Games is the online game division of NetEase Inc, and is responsible for the development and operation of some of the most widely played mobile and PC games in markets. Currently, NetEase Games operates more than one hundred games and continues to be ranked in the top five global mobile game publishers. In 2021, the annual gross profit of NetEase Games reached 7.371 billion dollars, with monthly active users (MAU) base that expanded to 80.3 million.

As shown in Figure 1, WDTest follows the previous practice [7] and primarily consists of two components: the widget detection component that utilizes existing object detection techniques to extract widget information from the underlying screenshot, and the action planning component, responsible

[*]Corresponding Authors

for selecting a specific widget among the detected widgets and choosing an appropriate action to take.

To establish a solid basis to study industrial mobile game testing, we first construct a large collection of GUI dataset of mobile games, consisting of 33 mobile games from 12 different mobile game publishers and covering 11 different types of mobile games. Since the existing studies only evaluate the accuracy of widget detection on a limited dataset (consisting of nine mobile games) [10], or the results are obtained on a dataset of general applications [6, 7]. Our experimental results indicate that models trained on our new game GUI dataset have significant performance improvement and can be generalized to a wider scope of applications without fine-tuning. As for the action planning component, we adopt a randomized strategy. The results of the evaluation conducted on three external mobile games, namely Apex Legends Mobile, D10, and GI4, indicate that WDTEST is capable of covering three times the number of unique UIs compared to the widely used testing tool Monkey [2] using a full randomized action planning strategy.

Unlike general-purpose mobile apps, mobile games possess unique characteristics, as their UIs are often more complicated [11] and have unique behaviors such as dynamic loading. While the evaluation results have demonstrated the effectiveness of widget detection-based testing, its practicality in industrial settings remains not very clear. To obtain a relatively more complete picture of widget detection-based testing and identify challenges in industrial mobile game testing, we conduct a survey with a carefully designed questionnaire in NetEase Games Testing Center. Eventually, 42 mobile game testers answered our questionnaire. The results show that WDTEST can alleviate the burden on manual testers for testing the usability and stability of mobile games. However, 57% of the testers are not still satisfied with WDTEST for functional testing, since the current action planning strategy is randomized clicking and lacks semantic comprehension of the underlying buttons being clicked. Furthermore, we examine the differences between general mobile applications and mobile games, highlighting three unique characteristics of the latter.

In summary, this paper makes the following key contributions:

- We initiate an early step towards widget detection-based testing for mobile games and identify challenges in adopting GUI widget detection-based testing methods in practice.
- We construct a much larger and more precise GUI dataset for mobile games, based on which we perform an empirical study to better understand the difference between general-purpose mobile apps and mobile games. Our dataset is made publicly available at [12] to enable further in-depth research in this direction.
- We introduce our experience and lessons learned from developing and deploying the first widget detection-based testing framework WDTEST in *NetEase Games*.
- We conduct a survey in *NetEase Games* to investigate how testing practitioners view the GUI widget detection-based testing methods. Based on the results, we discuss and pinpoint some practical directions towards building more effective automated testing techniques for mobile games.

The rest of this paper is organized as follows. Section II provides background knowledge about object detection models and their evaluation metrics. Section III presents the performance of widget detection models in the context of mobile games. Section IV evaluates WDTEST in industrial settings. Section V describes the opinions of practitioners towards widget detection-based testing and the differences between mobile apps and mobile games. Section VI discusses our findings and several implications. Section VII reviews internal and external threats. After discussing related work in Section VIII, Section IX concludes this paper.

II. BACKGROUND AND GAME GUI COLLECTION

In this section, we first briefly introduce several representative techniques for GUI widget detection. Then, we discuss metrics for widget detection and image complexity, followed by the process in which we construct GUI dataset of mobile games.

A. Techniques for GUI Widget Detection

1) Generic object detection techniques: Object detection is to identify where objects are located in a given image (object localization) and which category each object belongs to (object classification) [13]. Old-fashioned (traditional) object detection techniques such as Scale-Invariant Feature Transform (SIFT) [14], rely on handcrafted features and shallow trainable classifiers to identify objects. Their performance easily stagnates in handling complex and varied object shapes. Deep learning-based techniques adopt neural networks (e.g., convolutional neural networks) to learn features from images and classify the located objects. With the fast progress over the past few years, deep learning-based techniques have achieved state-of-the-art performance on a wide range of object detection benchmarks [15].

2) Selection of detection techniques: Inspired by previous studies [6, 7, 10], to be more comprehensive, we choose to adopt both deep learning-based object detection techniques as well as old-fashioned techniques. After investigation, seven object detection techniques are chosen to evaluate their performance on the game dataset. The selected techniques include five deep learning-based methods, one method that developed upon old-fashioned techniques, and one method that combines both old-fashioned and deep learning-based techniques. We give a high-level summary of these methods as follows.

Faster R-CNN: Faster R-CNN is a two-stage anchor-based object detection algorithm [16]. Faster R-CNN employs a novel idea of the region proposal network (RPN), which is a fully-convolutional network sliding on the last feature map of the feature extraction network and predicts whether there is an object or not along with the corresponding bounding box of those objects. Then, the proposals generated by RPNs are fed into two fully connected layers to perform object class classification and the bounding box regression.

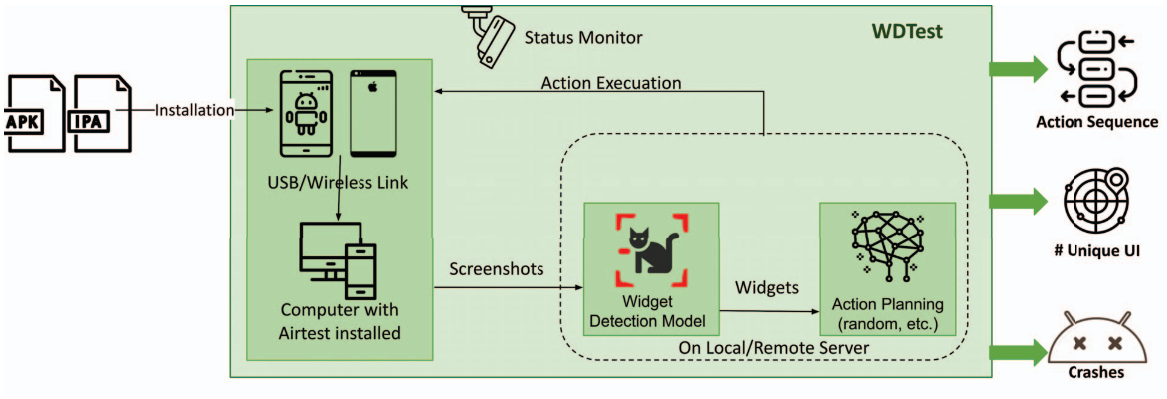


Fig. 1. Overview of the WDTEST framework.

YOLO Family: Compared with the Faster R-CNN model, YOLO is a one-stage anchor-box-based object detection technique. It treats object detection tasks as a regression problem and extracts features from input images as a unified architecture. Different from the manually defined anchor box of Faster R-CNN, the one-stage model uses the k-means method to cluster the groundtruth bounding boxes, and takes the box scale and aspect ratio of the k-centroids as the anchor boxes. For each grid of the feature map, it regresses the box coordinates and classifies the object in the bounding box. We adopt YOLOv3 [17] and YOLOv5 [18] in our experiment.

CenterNet Family: CenterNet is a one-stage anchor-free object detection technique [19]. Faster R-CNN and YOLO rely on a set of predefined anchor boxes, whose performance may be affected by the ratio aspect of anchor boxes. Instead of generating bounding boxes using anchor boxes, CenterNet directly predicts the position of the top-left and bottom-right corners as well as the center of the object, after which it assembles them to output the bounding box. CenterNet2 can be treated as a two-stage version of CenterNet, with its first stage estimating object probabilities and the second stage conditionally classifying objects [20].

Xianyu: Xianyu is a tool developed by Alibaba to reverse-engineer GUIs [6]. Xianyu first binarizes the image and slices the image in both horizontal and vertical directions. Then, it utilizes edge detection to detect the edges and obtains contours in the binarized image, followed by the flood fill algorithm to merge qualified neighbor points. In this paper, we use the element detection part of Xianyu.

UIED: UIED is a combined GUI detection approach, which blends multiple techniques to seek a balance with their combination [21] in practice. Specifically, UIED adopts optical character recognition (OCR) to detect text elements (i.e., GUI elements are filled with text information) and leverages deep learning-based models to detect non-text GUI elements. Based on this, UIED collects the results from both OCR and the GUI detection model and aggregates the predictions of bounding boxes to improve overall detection precision.

B. Metrics for GUI Widget Detection and Image Complexity

Widget detection metrics. Intersection over Union (IoU) is an important and common evaluation metric to approximate the performance of object detection techniques [22]. An IoU threshold indicates the requirement of the precision of prediction. The selection of the IoU threshold can largely affect the performance of deep learning models. In consideration of the effect on downstream applications (e.g., widget detection guided GUI testing), a moderately strict criterion to evaluate the performance of models should be adopted. In previous works, for real-world object detection tasks, the IoU threshold is often assigned with a loose value (i.e., 0.3 or 0.6) [9, 23], which means a prediction is considered correct when half of the prediction bounding box area is overlapped with the ground-truth bounding box area. However, in GUI widget detection, a more strict threshold is required based on the following two reasons: 1) The widgets are placed compactly in a GUI. The loose threshold may lead to incorrect estimations of model performance. 2) The detection results serve for downstream applications (e.g. widget detection guided GUI testing).

Image complexity. The complexity of an image can indicate various aspects of the image content and is an important factor for testing various image processing methods. Among various measures of image complexity, spatial information (SI) has demonstrated its strong correlation with compression-based image complexity and is widely adopted as the basis for estimating image complexity [24]. Given s_h and s_v to denote grey-scale images filtered with horizontal and vertical Sobel kernels, respectively, the $SI_r = \sqrt{s_h^2 + s_v^2}$ is the magnitude of spatial information at every pixel. According to [24], SI_{mean} can be an ideal predictor for estimating image complexity. SI_{mean} can be calculated as $SI_{mean} = \frac{1}{P}\sum SI_r$, where P represents the number of pixels in the image.

C. Action Execution Components

Inside WDTEST framework, to execute a selected action (see Figure 1), we utilize an open-sourced project named

Airtest[1]. Airtest is a cross-platform UI automation framework for games and applications, which is based on image identification and supports Windows, Android, and iOS, etc. In WDTEST, we mainly use the input simulation function of Airtest to capture screenshots and perform click (touch) actions. Detailed usage information could be found on the official website of Airtest. We also provide some examples on our website [12] to ease the usage by practitioners.

D. Collection of Game GUIs

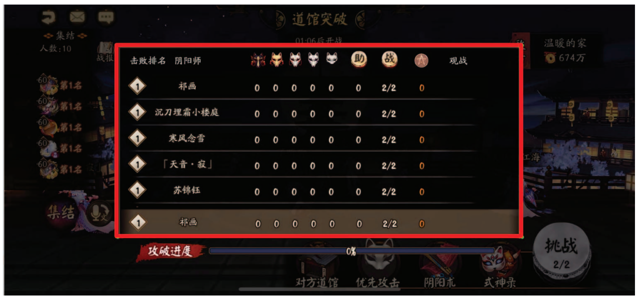
To the best of our knowledge, up to the present, there is only one available dataset of game GUI benchmark [10]. However, this dataset only contains 2,993 GUIs with 38,776 widgets which may be insufficient for training modern object detection models. Furthermore, the diversity of game types in this dataset is also somehow limited. To achieve a solid evaluation of the performance of the chosen object detection models, we spent lots of effort in constructing a new mobile game GUI dataset, which currently consists of 33 mobile games and covers 11 types of popular mobile game types. The process of constructing a widget detection image dataset mainly follows three key steps.

First, we need to define the target widget detection categories. To do this, we adopt the categories used in existing works [10, 25]. Then, we adapt them into widget detection categories for mobile games. As a result, 12 categories are summarized.

The second step is collecting a set of candidate images to represent the proposed categories. We select 33 mobile games published by 12 different mobile game companies which cover 11 game types to enrich the diversity of our dataset. While most of the widget types can be easily recognized from their name, except for *Background Return*. *Background Return* is a widget on which if there is no explicit button displayed on the screen, the user can only return to the previous UI by clicking the dimmed background. As shown in Figure 2, users can only return to the previous UI by clicking the area outside the red box. Detailed explanations and examples for every widget type could be found on our supplementary website [12]. Then, we successfully recruit 12 volunteers in *NetEase Games Testing Center* to play these mobile games with their screen being recorded. The volunteers are required to explore as many scenes of the mobile game as possible, and each mobile game is played for at least 10 minutes. The screen recordings are converted into images, which are then de-duplicated using a third-party tool. After removing the duplicated images, a total of 14,442 GUIs are collected.

The third step is annotating the collected screenshots in step 2. Although Ye et.al. [10] have proposed an automated approach to extract GUI widgets in mobile games. The automated approach proposed by Ye et.al. struggles in effectively detecting nested GUI widgets. Thus, we recruit eight developers along with the first four authors of this paper to manually annotate the images. From the 33 selected games, we obtain

[1] Airtest: https://github.com/AirtestProject/Airtest



击败排名	阴阳师						助	战		观战
1	祁画	0	0	0	0	0	0	2/2	0	
1	沉刀埋霜小楼庭	0	0	0	0	0	0	2/2	0	
1	寒风念雪	0	0	0	0	0	0	2/2	0	
1	「天音・寂」	0	0	0	0	0	0	2/2	0	
1	苏锦钰	0	0	0	0	0	0	2/2	0	
1	祁画	0	0	0	0	0	0	2/2	0	

Fig. 2. Example of a background return widget.

14,442 GUIs and 227,221 widgets in total. The distribution of each category is summarized in Table I. The annotations are cross-checked by the aforementioned developers and co-authors to reduce the risks of introducing bias and manual mistakes.

TABLE I
DISTRIBUTION OF WIDGETS.

Widget type	# widgets
Background Return	448
Button	182,608
Checkmark	1,253
Dropdown (Spinner)	7,750
Horizontal ScrollView	3,563
InputField (Editable Text)	852
LiveStream	15
Locked Button	17,336
ScrollView	10,511
Scrollbar	1,443
Slider	234
Toggle Button	1,208
Total	227,221

III. EVALUATION OF GUI WIDGET DETECTION FOR INDUSTRIAL MOBILE GAMES

In this section, we discuss the detailed evaluation results of seven representative GUI widget detection techniques. All the experiments were conducted on a workstation with two E5-8160, four RTX 3090, and 384 GB RAM, running on Ubuntu 20.04.

A. Experimental Setup

1) Dataset: We use the collected gaming dataset as the dataset for training all the deep learning-based widget detection models. Since the screenshots have different resolutions, we resize all the images to a fixed resolution of 2,560*1,440. We split the 14,442 images into train/validation/test datasets with the ratio of 8:1:1. Following previous work [6], the UIs of a single mobile game will not be split into different datasets.

2) Model training and configurations: For Faster R-CNN, YOLO family, and CenterNet family, we use their implementation on GitHub [26, 27, 28, 29, 30] and re-train these models on our collected game dataset. All the deep learning models are trained for 45,000 iterations to ensure that the models have been sufficiently trained. For Xianyu and UIED, we adopt their implementation from previous work [6]. Note that the latest version of UIED has replaced the original OCR technique EAST with Google OCR for easier deployment and better performance. Therefore, we use the UIED with Google OCR and configure its parameter according to the author's recommended settings for mobile applications in order to obtain its best performance.[2]

B. Metrics

Similar to previous works [6, 7, 10], we adopt *Precision*, *Recall* and *F1-score* to evaluate the performance of the selected techniques. Specifically, for each predicted bounding box b on a given screenshot s, we calculate its IoU with ground-truth bounding boxes and find the one that has the largest IoU with b and this IoU needs to be greater than a predefined threshold. If we can successfully find a ground-truth bounding box for b, we call this b as *True Positive* (TP). Otherwise, this b is a *False Positive* (FP). *False Negative* (FN) is a ground-truth bounding box that can not be matched with any predicted bounding box. The *Precision* is calculated as $TP/(TP+FP)$, *Recall* as $TP/(TP+FN)$ and *F1-score* as $2 \times Precision \times Recall/(Precision + Recall)$.

C. Analysis on Experimental Results

1) Performance of GUI Widget Detection.: The performance of the selected techniques is summarized in Table III.
Comparison between deep learning-based techniques. From the results, we can see that Faster R-CNN achieves the highest identification accuracy among all the deep learning-based techniques with IoU at 0.5. CenterNet2 follows Faster R-CNN to be the second best on identification accuracy. Two-stage techniques (Faster R-CNN and CenterNet2) achieve better effectiveness than one-stage techniques due to their explicit stage of proposing bounding boxes. One thing worth noting is that the two most recent proposed techniques, CenterNet2 and YOLOv5 (both proposed in 2021) fail to outperform Faster R-CNN (proposed in 2015) on identification accuracy. Between the two-stage models, Faster R-CNN outperforms CenterNet2 at lower IoU thresholds, but is outperformed at higher IoU thresholds (IoU$>$0.9). The reason may be that CenterNet2 is an anchor-free model, which enables it to suffer less from the size and shape of the UI elements. However, the flexibility will harm its performance when IoU threshold is low since the UI elements of mobile games are often put closely together. This issue has also been spotted in previous study [6].
Comparison between old-fashioned techniques. For old-fashioned techniques, UIED outperforms Xianyu, since UIED is empowered by deep learning-based OCR techniques and the idea of detecting non-text and text elements separately. Xianyu struggles to detect highly sophisticated GUI elements in mobile games.
Comparison between deep learning-based and old-fashioned techniques. While UIED has achieved the most competitive effectiveness in most scenarios in previous studies [6, 7], it does not keep ahead of its competitors in industrial mobile games. The rationale is twofold: 1) the different roles of text GUI elements in general applications (e.g., Taobao, Amazon) and mobile games. In a general application, text-elements are commonly designed to be clickable. However, the text-elements in mobile games are often displayed to introduce the background information of a gaming scene and are not clickable, which will lead UIED to produce FPs and undermine its performance. As shown in Figure 3, although UIED correctly detected two widgets on the right-bottom corner, it also generates lots of FPs. 2) the traditional computer vision techniques used by UIED struggle to detect highly sophisticated GUI widgets in mobile games. Although UIED outperforms deep learning-based techniques on general applications [7], its poor performance in mobile games makes it unsuitable to be the backbone GUI widget detection technique for automated visual testing for industrial mobile games.

TABLE II
RESULTS OF WIDGET DETECTION (IoU@0.75) AND RUNNING EFFICIENCY ON GENERAL APPLICATION.

Techniques	*F1-score*	*Avg. Time*
CenterNet	0.351	1.14s
CenterNet2	0.410	1.21s
YOLOv3	0.192	0.95s
YOLOv5	0.235	0.89s
Faster R-CNN	0.481	1.33s
UIED	0.513	2.8s
Xianyu	0.105	0.83s

D. Generalization on General-Purpose Applications

To test the generalizability of the selected widget detection techniques, we evaluate the performance of the trained models on general-purpose applications. We select two general applications, one is NetEase Cloud Music, which is a leading online music platform developed by NetEase, with 181.7 million monthly active users. The other is Weipinhui, which is an online shopping application. Both of these two applications are not included in our game GUI dataset. We take 100 screenshots from these two applications and label buttons in the screenshot by following the same procedure in Section II-D. Then, we evaluate the performance of the widget detection techniques on this dataset built upon general applications. The results are shown in Table II. We observe that although deep learning-based models suffer from performance drop on general applications, the performance drop is comparatively less than previous studies [10], indicating the effectiveness of our collected game dataset. Faster R-CNN and CenterNet2 can still maintain a competitive performance on general applications compared to UIED since their processing time on a single image is much shorter than UIED. Empowered by combining both deep learning and traditional computer vision techniques, UIED outperforms all the other widget detection techniques on general-purpose applications.

[2]Parameter for mobile applications: https://github.com/MulongXie/UIED/blob/4557d00ad462445d00952c74458c5455d0f7e13c/run_single.py\#L47

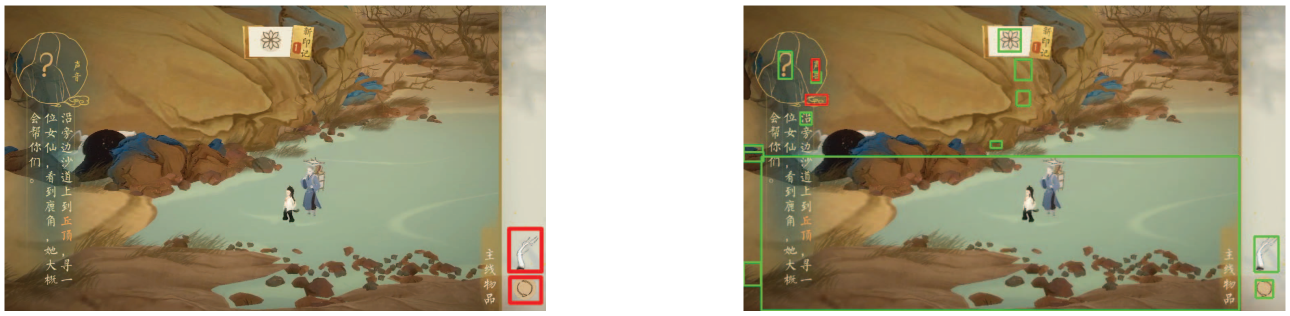


Fig. 3. Example of detection results of UIED on game GUI. The left figure is the ground-truth. And the right figure shows the results generated by UIED.

TABLE III
PERFORMANCE OF THE SELECTED TECHNIQUES.

Performance / Model	*Prec.*	*Recall*	*F1-score*
CenterNet (IoU@0.5)	0.304	0.464	0.367
CenterNet (IoU@0.75)	0.293	0.453	0.356
CenterNet (IoU@0.95)	0.218	0.405	0.283
CenterNet2 (IoU@0.5)	0.51	0.44	0.472
CenterNet2 (IoU@0.75)	0.45	0.42	0.434
CenterNet2 (IoU@0.95)	0.39	0.37	0.38
YOLOv3 (IoU@0.5)	0.623	0.29	0.396
YOLOv3 (IoU@0.75)	0.594	0.206	0.306
YOLOv3 (IoU@0.95)	0.103	0.08	0.09
YOLOv5 (IoU@0.5)	0.64	0.441	0.522
YOLOv5 (IoU@0.75)	0.55	0.391	0.457
YOLOv5 (IoU@0.95)	0.195	0.103	0.135
Faster R-CNN (IoU@0.5)	0.81	0.58	**0.68**
Faster R-CNN (IoU@0.75)	0.57	0.40	0.47
Faster R-CNN (IoU@0.95)	0.118	0.084	0.098
UIED (IoU@0.5)	0.334	0.351	0.347
UIED (IoU@0.75)	0.293	0.26	0.276
UIED (IoU@0.95)	0.28	0.21	0.24
Xianyu (IoU@0.5)	0.239	0.091	0.132
Xianyu (IoU@0.75)	0.167	0.12	0.14
Xianyu (IoU@0.95)	0.103	0.007	0.013

IV. Evaluation of GUI widget detection-based Testing method.

A. *Action Planning Strategy*

Since the testing framework is supposed to be non-intrusive, we assume that the testing framework only uses screenshots as the source of information and thus can not rely on any precise information. As a result, we follow the setup in [7] and adopt the randomized exploration strategy as the test-action planning strategy. More specifically, at each round, the framework randomly selects one UI widget from the detected UI widgets and clicks on it.

As a consequence, we decide to set testing parameters as follows: before each round, with probability $\alpha = 0.005$, we restart the game; otherwise, the platform randomly chooses a detected widget and clicks on it.

B. *Evaluation of GUI Widget Detection-based Testing*

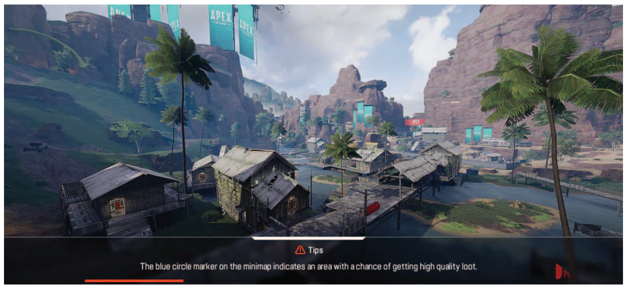
We select a famous mobile game named *Apex Legends Mobile* [31], an RPG game named GI4, and an action game named D10 to evaluate the effectiveness of the testing method. GI4 and D10 are currently under development inside NetEase Games and all these three mobile games are not included in our collected gaming dataset.

Baseline Techniques. We compare the effectiveness of our testing method with the aforementioned widget detection techniques. We denote the testing method instantiated with CenterNet, CenterNet2, Faster R-CNN, UIED, YOLOv5, YOLOv3, Xianyu as **C-Test C2-Test**, **F-Test**, **U-Test**, **Y3-Test**,**Y3-Test**, **X-Test**. Besides, we also recruit a testing developer to perform random clicking, denoted as **H-Test**, and compare these techniques with Monkey, a widely used tool for testing. For Monkey, we adopt its default settings as the previous works [7] and set the throttle to 200ms.

Environmental Settings. Since mobile games do not have a specific definition for scenarios used in previous work [7], we use unique UIs covered in unit time (one hour) to evaluate the effectiveness of widget detection-based testing. Specifically, after each round, we take a screenshot and compare its hash distance with historical screenshots using perceptual hashing [32]. If the hash distance is smaller than 0.5, we consider this screenshot to be a unique UI. To reduce the threats that are caused by randomness, we run each test three times. We run the mobile game on a Samsung S20 FE running on Android 12 with Android Debug Bridge (ADB) enabled. The captured screenshot will be transferred and processed on a server with E5-8160 CPU and one Nvidia A6000 GPU. We use Airtest introduced in Section II-C to capture the screenshot and perform the click action.

C. *Experimental Results*

Table IV summarizes the average unique UIs covered by the testing framework instantiated with different techniques in one hour along with the corresponding execution rounds on three selected games. The results show that the achieved F-test outperforms all the techniques except H-test, and covers three times more UIs than Monkey. All the widget detection-based testing techniques have shown to be more effective than Monkey, indicating the ability to recognize widgets can significantly improve the effectiveness of automated testing for mobile games, which is consistent to [7, 9].



Tips

The blue circle marker on the minimap indicates an area with a chance of getting high quality loot.

Fig. 4. Example of a loading screen.

TABLE IV
UNIQUE UI COVERAGE AND EXECUTION ROUNDS.

Baseline	*Avg.* #Unique UI	*Avg.* Execution Rounds
C-Test	217	2,181
C2-Test	235	1,901
F-Test	**308**	1,310
Y5-Test	159	2,695
Y3-Test	132	2,403
U-Test	163	1,107
X-Test	121	2,809
Monkey	99	17,909
H-Test	811	✗

Among all the widget detection-based techniques, techniques with similar accuracy exhibit generally consistent performance. However, F-Test is significantly more effective than other techniques due to its high identification accuracy, which is different from the findings in previous work [7]. In the previous research, the authors claim that although Faster R-CNN has higher UI-element identification accuracy, the execution time of Faster R-CNN leads to relatively lower scenario coverage. However, the processing time of a specific technique is not as important in mobile game testing as in general applications. Since industrial mobile games are much more sophisticated than general applications, mobile games may often require dynamic loading resources which can take more than ten seconds. During the loading period, all the actions are invalid. Figure 4 is an example of a loading screen. The dynamic loading characteristic of mobile games makes the identification accuracy more important than the processing speed of a single round of execution.

Finally, all the automated testing techniques are not comparable to H-Test, exhibiting the potential for future research.

V. SURVEY

Unlike traditional mobile applications, mobile games have unique characteristics, as their UIs are more complicated and have unique behaviors like dynamic loading. While the evaluation results of the widget detection-based testing method have proven its effectiveness, whether this testing method is practical in the industrial environment is still unclear. To obtain a more complete picture of the widget detection-based testing method and identify challenges in industrial mobile game testing, we have conducted a survey at *NetEase Testing Center*.

A. Participant Recruitment

In order to obtain a balanced and wide view of the problems occurring in adopting widget detection-based testing methods, we involved two groups of developers: script-based testers and manual testers. In the former group, the developers' main duty is to write testing scripts that test the runtime status and functionality of mobile games. The second group mainly includes developers who test the usability and functionality of mobile games through manual clicking.

We recruited participants at *NetEase Testing Center* and *NetEase Mobile Testing Lab*, all the participants are required to have used the widget detection-base testing method for at least one month. As a result, we successfully received 45 positive responses from 21 script-based testers and 24 manual testers. To balance the ratio between these two groups, we removed three participants from manual testers whose working experience is less than half a year.

Overall, we recruited 42 participants to conduct the survey, equally divided into script-based testers and manual testers. Among the recruited participants, the testing experiences range from 1.5 years to 10 years with a median value of 3.4.

B. Questionnaire

To reveal the opinion of testing developers towards widget detection-based testing methods, we design a detailed questionnaire for participants. As shown in Table V, Q1-Q3 are common questions while Q4-Q6 are designed to reveal the effectiveness of widget detection-based testing methods in practice and challenges encountered during the adoption of widget detection-based testing methods. Q7 is designed to investigate the difference between testing general mobile applications and mobile games.

The results of the questionnaire show that 73% (21 from the manual-tester group, nine from the script-tester group) claim that the widget detection-based testing method can save at least eight hours per week for them. However, we observe that a great portion (57%) of the script-tester group complains that the widget detection-based testing method can not perform the tasks they want. Their daily work is to draft testing the script that performs a series of actions in the game and observe whether the actions can lead to a designed result (e.g., obtain an item, clear the current stage), which can test the functionalities of mobile games. Since the widget detection-testing method can not click on a designated button, making it unable to assist the functional testing.

Furthermore, the result of the questionnaire reveals a problem with the widget detection-based method in practice. Seven developers from the manual tester group report that the new testing method may easily get stuck into a scenario during long-term testing. Due to the complex logic of mobile games, there exist scenarios that need to click a specific button or perform a certain series of actions to get out of a scenario. Since the current widget detection-based testing method can only perform random clicking, it may easily get into a scenario and get stuck in it, preventing it from exploring deeper scenarios.

Additionally, 82% of the developers regard compatibility and easy-to-use as the advantage of the widget detection-based testing method. Since 77% of the developers need to test with more than 6 devices for a project, the robustness of the widget detection-based testing method on resolution alleviates them from migrating scripts between devices with different resolutions.

As for the demanding requirements of an ideal testing method, the most demanding characteristic is compatibility (93% of the developers mentioned). Developers complain about suffering from drafting different testing scripts for different devices and manual testers also need to test new-added functions across different devices. Although there exists OneToMany [33] technique which can partially alleviate the load, the developers are far from satisfied. The second most mentioned characteristic is automated testing script generation. Due to the unique characteristics of mobile games, testing scripts are essential to conducting functional testing. There exists an urgent demand for automated script generation techniques. Furthermore, an ideal testing framework should be easy to deploy and the visualization can enable novices to perform testing procedures. There are few mentions of high scenario coverage and the ability to generate more readable reports.

Based on our survey results, we have discovered that utilizing widget detection-based testing methods can reduce the burden on manual testers when evaluating the usability and stability of mobile games. However, both sets of developers are dissatisfied with the performance of widget detection-based testing when it comes to functional testing. The reason being that the widget detection-based testing approach is unable to understand the significance of scenarios and buttons, making it inadequate for executing specific functional testing tasks.

C. Difference between General Applications and Mobile Games

Based on the results of the questionnaire, we have summarized the following unique characteristics of mobile games.

Higher image complexity of UI. Previous works [10] have found that the widget density and widget diversity of mobile games is significantly higher than in general applications. In this study, we further investigate this issue, and find that the image complexity of the UI of mobile games is also higher than general applications. To get a quantitative measurement for this problem, we adopt spatial information which is widely used to estimate the complexity of an image. We calculate the average SI_{mean} on our dataset and RICO dataset. The results show that the average SI_{mean} on game dataset (30.41) is 18.5% higher than RICO dataset (25.67), which can partially explain the significant performance drop of the traditional computer vision-based widget detection techniques on game dataset.

Sophisticated interaction logic. The interaction between the user and mobile games is more complicated than in general applications. The rationale is as follows: 1) the inner state of a mobile game is much more complicated than general applications. For example, playing or testing mobile games can be treated as a sequential decision process [11], while the general applications do not require such complicated interaction. 2) The interaction between users and the scenario in mobile games is more complicated. Due to the sophisticated interactions required by the mobile game, it is difficult for an automated testing method to achieve high scenario coverage in mobile games.

Dynamic loading. Also known as dynamic resource loading. With mobile games becoming increasingly complex, resources often need to be loaded dynamically while the game is loading or being played. Furthermore, the loading time consumed by dynamic loading is nondeterministic, which makes it difficult to implement automated testing methods.

TABLE V
QUESTIONNAIRE QUESTIONS.

ID	Question
Q1	Years of testing experience
Q2	Job responsibility
Q3	Average number of devices for one project
Q4	How much time can be saved per week after using widget detection-based testing method
Q5	Problem encountered during using widget detection-based testing method
Q6	Advantages and disadvantages of widget detection-based testing compared to your current workflow
Q7	Difference between testing general mobile applications and mobile games
Q8	Ideal tools

VI. DISCUSSION

In this section, we discuss our findings and describe several implications.

A. Industrial Deployment

Up to the present, WDTEST has been continuously tested in the "Quick and Intelligent Compatibility Testing" pipeline at *NetEase Games Testing Center*. More than 18 mobile games are required to be tested using WDTEST before release. We also received positive responses from the testing department. For instance, WDTEST has found 19 crashes since its deployment, with 16 of which have been confirmed as bugs and fixed. We plan to provide WDTEST as a commercial visual GUI testing service on the Fuxi Platform of NetEase, a renowned research institute that serves over 200 clients.

B. Issues with Vision-based Techniques on Mobile Games

Although widget detection-based testing has demonstrated its effectiveness and the advantage of being non-intrusive and easy-to-deploy. However, we observed some issues during the development and deployment of WDTEST. For deep learning-based object detection techniques, the main issue is that a widget detection model may fail to detect widgets when the widgets are closely put together (e.g, card deck, store), causing the model to produce FNs and undermines the performance of the whole framework. As for traditional widget detection techniques, although these techniques are generally fast and may even demonstrate the best average effectiveness in certain scenarios [34], these techniques may not be suitable for industrial mobile games. Our experience and experimental results have shown that the image complexity of industrial mobile games makes traditional computer vision techniques struggle to effectively detect widgets. At the very early stage of developing WDTEST, we tried to use GUI testing techniques based on traditional computer vision techniques such as image-matching based approaches [35, 36], since we can easily acquire original UI design of widgets from the UI team. However, after several rounds of experiments, the overall accuracy is lower than 3.2%. As shown in Figure 5, the image matching technique fails to match the correct position of the return button on the left top corner but finds a matched image with 0.98 confidence on an old-fashioned Chinese building inside the screenshot. We inspected some examples and figured out that the low accuracy of image-matching approaches can be explained as follows: 1) the provided widgets from the UI design team are the original image without background information, but the screenshot will have a background in practice. 2) The image-matching methods mainly utilize traditional computer vision techniques such as edge detection. These traditional computer vision techniques are ineffective when tackling mobile game UIs which are filled with fabulous visual effects.

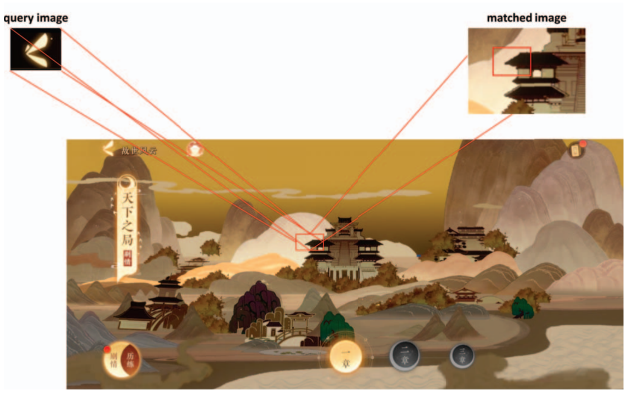


Fig. 5. Example of a failed image-matching.

C. Potential for Adopting Reinforcement Learning in Visual GUI Testing

Currently, the action planning strategy of WDTEST is a randomized strategy, which is simple but effective. Recently, reinforcement learning has proven its effectiveness in testing games [4, 11]. Since the key challenge in game testing is that game testing requires playing games as a sequential decision process, there exists the potential for adopting reinforcement learning in visual GUI testing for mobile games and improving the testing performance. For example, we could store the historical data of the previous test runs and use these data to construct a probabilistic model $\mathcal{M}$. Since $\mathcal{M}$ can only provide one-step guidance information, one can utilize reinforcement learning techniques to expand one-step information to multi-step guidance information and guide the visual GUI testing.

D. Needs for Automated Test Generation Support on Functional Testing for Mobile Games

From the previous discussion in Section V, we have noticed that developers are not satisfied with the performance of WDTEST on functional testing. Currently, functional testing for mobile games in NetEase Games is conducted by manually writing scripts, which is labor-intensive. Given the importance of functional testing for quality assurance, it is critical and urgent to have more efficient techniques that can provide automated test generation support in order to make mobile games more robust. Based on the results of the questionnaire in Section V, the most demanding testing tool is automated testing script generation.

VII. THREATS TO VALIDITY

In this section, we will discuss the threats to the validity of our work.

Selection of data source and object detection models. In this paper, we adopt 33 mobile games to construct our GUI dataset. The selection of games could be biased. To counteract this bias, we cover 11 diverse types of mobile games published by 12 different mobile game companies. Furthermore, the selection of object detection models could also be biased. To minimize this bias, we adopt all the object detection models used in the previous similar studies [6, 7], plus two most recently proposed models (CenterNet2, YOLOv5).

Subjectiveness of the labeling process. This study involves manual labeling and inspection during the construction of the GUI dataset, which may introduce bias and present threats to the validity of our dataset. To reduce this threat, we stick to the internal quality control process of *NetEase* to counteract this issue. However, minor issues could still be introduced. To alleviate this, we try our best and employ eight developers along with multiple authors of this paper to crosscheck the labels three times.

Randomness in the testing process. Another threat could be the randomness in the evaluation process of WDTEST, to counteract this threat, we repeat each testing configuration three times.

VIII. RELATED WORK

In this section, we discuss the following lines of closely related research.

Deep learning-based object detection models. With the adoption of deep learning techniques, object detection models have greatly evolved and have been applied to many cutting-edge areas [37]. Based on the difference in workflow, current objection detection techniques can be divided into two categories, one is the two-stage detector, such as Faster R-CNN [16], CenterNet2 [20]. The other is the one-stage detector such as YOLO [17, 18]. However, the aforementioned models are usually designed for detecting real-world objects and can not be directly applied to GUI widget detection tasks. Deka et al. [25] first build a dataset, RICO, including 72K UI screenshots, as well as 10K user interaction traces from 9,772 Android apps, which can be used to help design mobile app UIs and train widget detection models. Liu et al. [38] propose a method that utilizes convolutional neural networks to detect UI components in mobile apps and then mine design and interaction data. However, both RICO and the method proposed by Liu et al. are for general applications and do not generalize well on mobile games.

GUI Testing. Mobile games are graphically-rich applications with attractive visual effects of GUI that bridge the gap between games and players. To ensure the end-user experience of graphically-rich applications such as mobile games, previous works have proposed methods to help GUI testing [7, 9, 39, 40, 41, 42]. Specifically, Liu et al. [42] propose a deep learning-based technique named OWLEYE that can model visual information from the GUI screenshot thus enabling it to detect GUI display issues and locate the corresponding detailed region of the issue. Chen et al. [39] propose GLIB, a code-based data augmentation technique to detect GUI glitches in mobile applications. Thomas et al. [9] first propose a method that can identify GUI widgets in screenshots using machine learning techniques and then use this information to guide random testing of mobile applications, which can significantly improve the branch coverage in 18 of 20 applications. Ran et al. [7] first develop an industrial visual testing framework named VTEST, which only takes device screenshots and performs automated test generation. However, both OWLEYE and GLIB are designed to detect GUI display issues. While VTEST is developed for industrial mobile applications and can deal with non-standard UI elements, it still does not generalize well on industrial mobile games. Since industrial mobile games often use a game engine to render GUIs instead of traditional GUI widgets, and the interaction logic of mobile games is much more complicated than general-purpose applications [11].

Game testing. Games become increasingly popular and complex due to users' ever-growing demands for entertainment. As a result, testing games has exhibited its unparalleled importance to ensure user experience. However, as pointed out by [43], even the most popular games on the market are not sufficiently tested. Aleem et al. [44] summarize the main reason for this unbalanced development to be the absence of automated testing techniques for games. As for mobile games, the existing research is still preliminary. Lovreto et al. [45] first report their experience of testing 18 mobile games using exploratory testing by manually drafting testing scripts to test the functionality of mobile games and point out limitations of existing testing methods. Khalid et al. [46] conduct a study by mining user reviews from 99 free mobile games and find that most negative reviews come from a limited subset of devices. Zheng et al. [11] first propose a composite testing technique, WUJI, by combining reinforcement learning and multi-objective optimization algorithms, which is the first automated testing for real-world games and outperforms the state-of-the-practice. Li et al. [47] propose GBGALLERY, which is the first public bug database for game testing. Ye et al. [10] first conduct an empirical study to investigate the performance and effectiveness of widget detection techniques on industrial mobile games and release the first GUI dataset for mobile games, inspiring us with the potential of applying widget detection techniques on testing mobile games. However, the size of the dataset published by Ye et al. is relatively small (nine mobile games) and they do not perform widget detection-based testing on mobile games. WUJI can hardly be applied to mobile games in practice due to the non-trivial amount of version updates of mobile games. The exploratory testing through writing testing scripts used in [45] is labor-intensive and is not fully automated. To the best of our knowledge, WDTEST is the first systematic and automated testing framework for industrial mobile games.

IX. CONCLUSION

In this paper, we report our experience and practice (at NetEase Games) of the development and deployment of the widget detection-based testing framework, WDTEST, for industrial mobile games. We performed a systematic study to evaluate the performance of existing widget detection techniques. To achieve this, we first built by far the largest GUI dataset for industrial mobile games consisting of 33 mobile games of 11 types. Then, we evaluate the performance of seven GUI detection techniques on our game GUI dataset. Based on the evaluation results, we implemented our widget detection-based testing framework called WDTEST, outperforming the widely used random testing tool Monkey with three times more covered unique UIs. To obtain a complete picture of widget detection-based testing in industrial settings, we conduct a survey in NetEase Games. The results show that widget detection-based testing can alleviate the burden of testing developers on compatibility testing and stability testing but is not fully satisfying for functional testing. In the end, we discuss our implications for developers and researchers in order to identify the challenges and opportunities in developing automated testing techniques for industrial mobile games.

ACKNOWLEDGMENT

This work was supported by JST, the establishment of university fellowships towards the creation of science technology innovation, Grant Number JPMJFS2132. This work was also supported in part by JSPS KAKENHI Grant No. JP19H04086 and No.JP20H04168, JST-Mirai Program Grant No.JPMJMI20B8, as well as Canada CIFAR AI Chairs Program, the Natural Sciences and Engineering Research Council of Canada (NSERC No.RGPIN-2021-02549, No.RGPAS-2021-00034, No.DGECR-2021-00019).

References

[1] "Mobile gaming global market report 2022," https://www.globenewswire.com/news-release/2022/09/12/2513796/0/en/Mobile-Gaming-Global-Market-Report-2022.html, accessed October, 2022.

[2] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, accessed October, 2022.

[3] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021.

[4] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022.

[5] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl Eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 398–409.

[6] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li, "Object detection for graphical user interface: Old fashioned or deep learning or a combination?" in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1202–1214.

[7] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, "Automated visual testing for mobile apps in an industrial setting," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 55–64.

[8] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI screenshots for search and automation," in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 183–192.

[9] T. D. White, G. Fraser, and G. J. Brown, "Improving random gui testing with image-based widget detection," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 307–317.

[10] J. Ye, K. Chen, X. Xie, L. Ma, R. Huang, Y. Chen, Y. Xue, and J. Zhao, "An empirical study of GUI widget detection for industrial mobile games," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1427–1437.

[11] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.

[12] "The public dataset and supplementary materials." https://sites.google.com/view/wdtest-for-mobile-games/, accessed October, 2022.

[13] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[14] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[15] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, "Deep Learning vs. Traditional Computer Vision," in *Advances in Computer Vision*, K. Arai and S. Kapoor, Eds. Cham: Springer International Publishing, 2020, pp. 128–144.

[16] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf

[17] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[18] "Yolov5," https://github.com/ultralytics/yolov5, accessed October, 2022.

[19] X. Zhou, D. Wang, and P. Krähenbühl, "Objects as points," *CoRR*, vol. abs/1904.07850, 2019.

[20] X. Zhou, V. Koltun, and P. Krähenbühl, "Probabilistic two-stage detection," in *arXiv preprint arXiv:2103.07461*, 2021.

[21] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "Uied: A hybrid tool for gui element detection," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1655–1659.

[22] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union:

A metric and a loss for bounding box regression," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 658–666.

[23] T. Yu, Z. Ren, Y. Li, E. Yan, N. Xu, and J. Yuan, "Temporal structure mining for weakly supervised action detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 5522–5531.

[24] H. Yu and S. Winkler, "Image complexity and spatial information," in *2013 Fifth International Workshop on Quality of Multimedia Experience (QoMEX)*, 2013, pp. 12–17.

[25] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 845–854.

[26] "A faster pytorch implementation of faster r-cnn," https://github.com/jwyang/faster-rcnn.pytorch, accessed October, 2022.

[27] "Yolov3," https://github.com/ultralytics/yolov3, accessed October, 2022.

[28] "Yolov5," https://github.com/ultralytics/yolov5, accessed October, 2022.

[29] "Objects as points," https://github.com/xingyizhou/CenterNet, accessed October, 2022.

[30] "Probabilistic two-stage detection," https://github.com/xingyizhou/CenterNet2, accessed October, 2022.

[31] "Apex legends mobile: Hyperbeat," https://www.ea.com/games/apex-legends/apex-legends-mobile?isLocalized=true, accessed October, 2022.

[32] C. Zauner, "Implementation and benchmarking of perceptual image hash functions," 2010. [Online]. Available: http://phash.org/docs/pubs/thesis_zauner.pdf

[33] "Solopi," https://github.com/alipay/SoloPi/blob/master/README_eng.md, accessed October, 2022.

[34] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated assertion generation via information retrieval and its integration with deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 163–174. [Online]. Available: https://doi.org/10.1145/3510003.3510149

[35] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 1535–1544.

[36] J. Tuovenen, M. Oussalah, and P. Kostakos, "MAuto: Automatic mobile game testing tool using image-matching based approach," *The Computer Games Journal*, vol. 8, no. 3, pp. 215–239, 2019.

[37] H. M. Gan, S. Fernando, and M. Molina-Solana, "Scalable object detection pipeline for traffic cameras: Application to tfl jamcams," *Expert Systems with Applications*, vol. 182, p. 115154, 2021.

[38] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *The 31st Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '18. New York, NY, USA: ACM, 2018, pp. 569–579. [Online]. Available: http://doi.acm.org/10.1145/3242587.3242650

[39] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "GLIB: Towards automated test oracle for graphically-rich applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1093–1104.

[40] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (gui) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, p. 1679–1694, oct 2013.

[41] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 987–992.

[42] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 398–409.

[43] D. Lin, C.-P. Bezemer, and A. E. Hassan, "Studying the urgent updates of popular games on the steam platform," *Empirical Softw. Engg.*, vol. 22, no. 4, p. 2095–2126, aug 2017.

[44] S. Aleem, L. F. Capretz, and F. Ahmed, "Critical success factors to improve the game development process from a developer's perspective," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 925–950, 2016.

[45] G. Lovreto, A. T. Endo, P. Nardi, and V. H. S. Durelli, "Automated tests for mobile games: An experience report," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 2018, pp. 48–488.

[46] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 610–620.

[47] Z. Li, Y. Wu, L. Ma, X. Xie, Y. Chen, and C. Fan, "GBGallery: A benchmark and framework for game testing," *Empirical Softw. Engg.*, vol. 27, no. 6, nov 2022.