An Empirical Study of GUI Widget Detection for Industrial Mobile Games

Jiaming Ye
Kyushu University
Japan

Ke Chen
Fuxi AI Lab of Netease
China

Xiaofei Xie*
Kyushu University
Japan

Lei Ma*
University of Alberta
Canada

Ruochen Huang
University of Alberta
Canada

Yingfeng Chen*
Fuxi AI Lab of Netease
China

Yinxing Xue
University of Science and Technology of China
China

Jianjun Zhao
Kyushu University
Japan

ABSTRACT

With the widespread adoption of smartphones in our daily life, mobile games experienced increasing demand over the past years. Meanwhile, the quality of mobile games has been continuously drawing more and more attention, which can greatly affect the player experience. For better quality assurance, general-purpose testing has been extensively studied for mobile apps. However, due to the unique characteristic of mobile games, existing mobile testing techniques may not be directly suitable and applicable. To better understand the challenges in mobile game testing, in this paper, we first initiate an early step to conduct an empirical study towards understanding the challenges and pain points of mobile game testing process at our industrial partner *NetEase Games*. Specifically, we first conduct a survey from the mobile test development team at *NetEase Games* via both scrum interviews and questionnaires. We found that accurate and effective GUI widget detection for mobile games could be the pillar to boost the automation of mobile game testing and other downstream analysis tasks in practice.

We then continue to perform comparative studies to investigate the effectiveness of state-of-the-art general-purpose mobile app GUI widget detection methods in the context of mobile games. To this end, we also develop a technique to automatically collect GUI widgets region information of industrial mobile games, which is equipped with a heuristic-based data cleaning method for quality refinement of the labeling results. Our evaluation shows that: (1) Existing GUI widget detection methods for general-purpose mobile apps cannot perform well on industrial mobile games. (2) Mobile game exhibits obvious difference from other general-purpose mobile apps in the perspective GUI widgets. Our further in-depth analysis reveals high diversity and density characteristics of mobile game GUI widgets could be the major reasons that post the challenges for existing methods, which calls for new research methods and better industry practices. To enable further research along this line, we construct the very first GUI widget detection benchmark, specially designed for mobile games, incorporating both our collected dataset and the state-of-the-art widget detection methods for mobile apps, which could also be the basis for further study of many downstream quality assurance tasks (e.g., testing and analysis) for mobile games.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

GUI Detection, Game Testing, Deep Learning

ACM Reference Format:
Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. 2021. An Empirical Study of GUI Widget Detection for Industrial Mobile Games. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473935

1 INTRODUCTION

Mobile games are continuously gaining popularity with the advancement of mobile devices over the past decade. According to medias [11], the global market share of game industry is estimated to be more than 85 billion dollars annually, a large portion of which run on mobile devices. Such large potential leads to stiff global competition among game companies. Being supported with modern visualization technology and hardware acceleration of mobile devices, game producers often design mobile games with fabulous and charming visual experiences via graphical user interfaces (i.e., GUIs), towards attracting more users. Similar to traditional software, a mobile game can evolve and often be updated even

*Xiaofei Xie (xiaofei.xfxie@gmail.com), Lei Ma (ma.lei@acm.org) and Yingfeng Chen (yingfengchen2016@163.com) are the corresponding authors.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473935

more frequently. For example, inside our industrial partner NetEase Games, one of the largest game companies in the world, a typical mobile game usually experiences at least 3 version updates per day for various purposes, e.g. visual/audio feature enhancement, performance optimization, bug fixing, etc. However, the non-trivial amount of update changes can inevitably introduce new bugs, that can greatly affect the player experience upon being uncovered by users. Thus, quality assurance of mobile games is of great importance, and systematic testing is often under very tight pressure of frequent version updates.

Although some techniques have been developed for testing the mobile applications, ranging from simple random method Monkey, to more advanced methods such as Stoat [37], Sapienz [26] and Espresso [15], etc., they can still be limited for automated mobile game testing due to the unique and highly dynamic characteristics (e.g., heavy user interactions, difficult task accomplishment). Consequently, industrial mobile games still mainly rely on manual testing (i.e., playing games) and semi-automatic testing (i.e., manually written scripts), which are labor-intensive, inefficient and expensive, becoming the bottleneck of game testing process for better quality. To this end, some recent works attempted to apply the machine learning-based techniques for mobile app testing [17, 42]. Nevertheless, it is still unclear about the challenges and pain points of industrial mobile game, the understanding of which can be very helpful for better quality assurance of mobile games.

To bridge this gap, we first conduct a comprehensive survey inside our industrial partner *NetEase Games*. In particular, in the first step, we performed scrum interviews with two testing experts from the Quality Assurance (QA) department to gain the big picture to better understand the process of industrial mobile game testing at *NetEase Games*. The interview results show that, in *NetEase Games*, the mobile game product is usually tested in terms of the *usability* and the *compatibility*, where *usability* testing aims to detect function bugs and *compatibility* testing ensures that the game can be played smoothly across different devices. Currently, many of these testing tasks are still mainly completed by human testers. To boost the efficiency of mobile game testing, developers or testers are also actively exploring and developing some automated techniques (e.g., POCO [19], monkey [18]). Based on our understanding from the scrum interview, we continue to design a questionnaire to answer **RQ1** — What are the challenges and pain points in industrial mobile game testings? What could be the research opportunities?

Eventually, 50 mobile game testers answered our questionnaire, based on which we identified two main challenges in mobile game testing: a) how to precisely detect the *clickable* GUI widgets of games especially when the game is deployed on variant end mobile devices, which is of great importance for the following automated testing and analysis tasks and b) how to achieve high coverage especially for the large-scale mobile game with complex logic (e.g., some hard game tasks). As for the second challenge, some attempts [17, 41, 43] have been made to improve the coverage and detect bugs. While the detection of GUI widgets (the first challenge) in mobile games still lacks in-depth investigation, it would be the focus in the rest of this paper.

It is not until recently the machine learning techniques have been applied for GUI widgets detection of Android apps. For example, Liu *et al.* [24] propose to utilize deep learning models to detect visual issues in mobile GUIs and locate the regions of them; Chen *et al.* [9] propose to combine text-based and non-text-based models together to improve the overall performance in detecting regions of GUI widgets. However, these techniques are mainly designed and evaluated for conventional mobile apps. Due to the highly dynamic visual effects and interactive nature of game apps, it is still unclear that to what extent existing techniques can be adapted to be useful in the context of mobile games.

Therefore, in this paper, we continue to conduct an empirical study towards understanding the usefulness of existing object detection techniques in detecting GUI widgets of industrial mobile games. To the best of our knowledge, it still lacks a game GUI widget detection benchmark of mobile games to enable systematic study. To further facilitate research along this line, we made large efforts to construct a mobile game GUI widget detection benchmark. Specifically, we develop an automatic technique to collect game GUI dataset (i.e., screenshots and corresponding widget labels). Since the automatic labelling may introduce inaccurate results, we further adopt a heuristic-based data cleaning strategy to improve the data quality. Finally, we create a game GUI dataset that contains a total of 2,993 GUIs with 38,776 widgets. Then, we integrated state-of-the-art GUI widget detection methods with our dataset, which together forming the very first benchmark to enable the research of GUI widget detection of mobile games.

Based on the constructed benchmark, we performed an empirical study towards to investigate **RQ2**: How effective are existing object detection methods across various mobile applications? Due to the diversity (i.e., designs) of mobile games, we aim to study how this difference affects model performance. In other words, can the detector trained on some game GUIs be generalized on other games or the general-purpose apps?

Furthermore, we performed in-depth manual analysis to figure out **RQ3**: What are the challenges for the detection of GUI widgets in mobile games?

In summary, the contribution of this paper is as follows:

- We conduct a survey in *NetEase Games* to investigate the urgent challenges and pain points for mobile game testing.
- We develop an automated method to collect and label the game GUI dataset, based on which, we create the GUI widget dataset for mobile games. We further integrate state-of-the-art mobile GUI widget detection methods, together forming the first benchmark specially designed for GUI widget detection research of mobile games.
- We conduct the empirical study to better understand the current status of GUI widget detection of mobile games, to identify challenges and opportunities. We make our benchmark publicly available[1] to enable reproducible study and facilitate further research on the downstream tasks such as mobile game testings.

2 GUI REGION DETECTION OF MOBILE APPS

In this section, we briefly introduce some recent representative methods of GUI region detection for mobile apps.

[1] https://sites.google.com/view/gamedc/

2.1 GUI Region Detection

Some earlier attempts [27–30, 38] have been made to analyze characteristics (e.g., design style, types of marked tags, usage scenario classification) of GUIs. In recent works, more attention shifts to GUI region detection of mobile apps, which could be the basis for many downstream tasks analysis. For example, Liu *et al.* [24] propose to utilize deep learning models to detect visual issues in GUIs and locate the regions of them; Chen *et al.* [9] propose to combine text-based models and non-text-based models together to improve overall performance in detecting regions of GUI widgets. However, they are implemented on general-purpose mobile (i.e., Android) apps instead of mobile games. It is still unclear to what extent existing GUI detection methods applied in the context of mobile games.

2.2 Models for GUI Region Detection

Faster RCNN and YOLOv2 are currently state-of-the-art deep learning models for GUI widget detection of mobile apps, which are widely adopted in recent studies [9, 24]. To be specific, Faster RCNN is a two-stage anchor-box-based deep learning technique for object detection. It adopts a novel region proposal network (RPN) to predict region proposals with a wide range of scales and aspect ratios. RPN accelerates the generating speed of region proposal because it shares full-image convolutional features and a common set of convolutional layers with the detection network. For each box, RPN then computes a score to determine whether it contains an object or not and regresses it to fit the actual bounding box of the contained object.

YOLOv2 is a one-stage anchor-box-based object detection technique. It treats GUI widget detection as a regression problem and extracts features from input images as a unified architecture. Different from the manually defined anchor box of Faster-RCNN, YOLOv2 uses the k-means method to cluster the ground truth bounding boxes in the training dataset, and takes the box scale and aspect ratio of the k centroids as the anchor boxes. For each grid of the feature map, it generates a set of bounding boxes. For each box, it regresses the box coordinates and classifies the object in the bounding box.

2.3 Metrics for GUI Region Detection

IoU (Intersection over Union) [1] is an important metric to approximate the performance of GUI widget detection. An IoU threshold indicates the requirement of the precision of prediction. The selection of IoU threshold largely affects 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.

3 INDUSTRY SURVEY AT NETEASE

To identify the challenges in industrial mobile game testing, in this section, we conduct a two-phase survey on mobile game testing at *NetEase Games*, the workflow of which is summarized in Figure 1. In particular, we first draft an initial set of interviews for gathering the key topics relevant to mobile game testing in industries to deepen our understanding. Next, based on the gathered concerns and questions, we design a structural questionnaire and distribute it to the game developers and testers at NetEase. The questionnaire is mainly designed to understand the pain points in current industrial mobile game testing, and further identify the potential opportunities for mobile game testing.

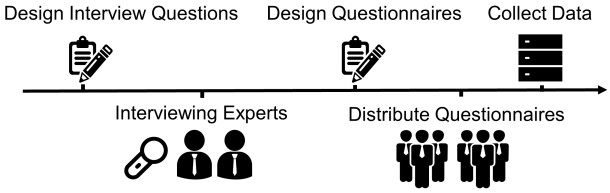

Figure 1: The overview of workflow of our survey.

Table 1: The interview questions regarding mobile game testing for two experts at NetEase. Questions that are marked green and red box consider the usability testing and compatibility testing of mobile games, respectively.

ID	Question
I1	Mobile testing routines introduction
I2	Team size
I3	How do we allocate works
I4	Do we cooperate with automated techniques
I5	What are the current limitations
I6	How to ensure high coverage rate by using test case
I7	How much slow down do automated techniques introduce
I8	How do the device difference affect our testing
I9	What is the current alleviation regarding the limitations

3.1 Interview of Mobile Game Testing Experts

At the beginning, to gain the big picture of industrial mobile game testing and understand the testing process in industrial games, we conduct scrum interviews of 2 industry experts who are in the lead position of mobile game testing at NetEase, i.e., the director of the *NetEase Testing Center* (E1) and the leader of the *NetEase Mobile Testing Lab* (E2), respectively.

As a part of the formal procedure of the interview, we have designed 9 questions that are summarized in Table 1, which intends to help us better understand the current status of mobile game testing at NetEase.

When the interview starts, we first invite the interviewee to give a basic introduction about the mobile game testing teams (I1, I2). Then, we ask questions about the testing routines of mobile games (I3) and the current automated techniques adopted in mobile game testings (I4). We also ask about the limitations captured by developers (I5). Then, we prepare specific questions (I6 to I9) for different interviewees, based on the mobile testing tasks they mainly lead. For E1, we focus on usability testing and ask about the current solutions and limitations (I6, I7). For E2, we prepare questions about the technical details in compatibility testings (I8, I9). Both interviews are arranged in 30 minutes. We record the interviews and also intend to make them publicly available after the internal approval of NetEase.

Overall, we obtain valuable high-level understanding from interviews. For example, in compatibility testing, there are around 10

Table 2: The questionnaire questions. Questions in green box are for usability testing and questions in red box are for compatibility testing.

ID	Question
Q1	Years of testing experience
Q2	Gender
Q3	Job responsibility
Q4	Averaged number of monthly testing scripts
Q5	How much testing scripts are needed in a project?
Q6	Time for writing a testing script
Q7	Whether use automated techniques to aid testing?
Q8	Challenges during testing
Q9	Averaged number of weekly testing projects
Q10	Whether use automated techniques to aid testing?
Q11	Averaged number of devices in one testing
Q12	Time for a particular testing
Q13	Averaged number of groups in compatibility testing
Q14	Dream tools

test developers who are responsible for testing device compatibility. They often play the same interaction behaviors on multiple devices. To achieve that goal, the testing developers mainly adopt OneToMany [18] techniques. They first record interaction behaviors on one mobile device, and then replay the sequence of behaviors on other devices. This technique is somehow efficient but can be limited to the device differences, especially caused by resolution difference. That is, when they replay the interaction behaviors, the action may be replayed at an incorrect location, and thus affect the testing efficiency. The current solution is to categorize the mobile devices with similar resolutions into one group to eliminate the impact of resolution diversity. However, as the number of groups increases, it is still not efficient and becomes the bottleneck for continuous testing pressure of frequent updates.

In usability testing, more than 50 test developers concentrate on testing mobile game usability tasks. Usability testing of mobile games relies on writing scripts to test the runtime status of games. To ensure the scripts cover most game scenes, the scripts are usually double-checked by experienced developers. To facilitate testing, they often utilize POCO [19] to extract GUI information (e.g., clickable region coordinates) that can help write testing scripts. However, POCO has following limitations: 1) POCO is limited to game developing engines (i.e., Unity3D, Cocos2dx), and 2) POCO often misleads the testing by reporting wrong coordinates due to ignoring visual effects. For example, some clickable widgets are overlapped by other widgets. They are actually not clickable for users but POCO still reports them.

In summary, we gain the big picture of industrial mobile game testing and better understand the testing process from the scrum interviews. Currently, many of these testing tasks are mainly completed by human testers, which may lead to inefficiency. To improve this and automate the testing process, developers and testers are also actively exploring and developing automated techniques (e.g., POCO [19], monkey [4]).

3.2 Questionnaire

To further reveal the challenges in the way of automated game testing and understand the potential opportunities, we continue to design a fine-grained questionnaire for the developers and testers. The questionnaire is generally based on our interviews, but has more detailed questions, as shown as Table 2. Questions in white cells are common questions, questions in green cells are for usability testing developers and questions in red cells are for compatibility testing developers. We prepare specific questions for different testing tasks in order to discover the in-depth needs of developers. The questionnaires are anonymously distributed to forefront developers in *NetEase Testing Center* and testers in *NetEase Mobile Testing Lab*. At last we recieved 53 questionnaires. 44 of them are about usability testing, and 9 are about compatibility testing.

The results of questionnaires show that 52% (23 of 44) of the developers are senior ones who have more than 3 years experience. Among them, 77% (34 of 44) developers adopt semi-automated tool (i.e., POCO) in usability testing to assist writing testing scripts. We also find that 21 of the 34 developers spend about 9 to 35 hours one week to draft testing scripts. Additionally, we highlight 27 of the 34 developers complain that 1) POCO often misleads testing by ignoring visual effects (e.g., overlapping, shadowing) and 2) the time cost of using POCO is often unacceptable. We also observe some developers highlight that 1) POCO assisted testing script is not flexible for maintenance and 2) POCO is not stable that it often leads to program crashes. Finally, 85% (29 of 34) developers are convinced that applying widget detection methods in usability testing can increase at least 35% of testing speed.

The results of questionnaires also indite that 67% (6 of 9) compatibility testing developers adopt OneToMany [18] technique to assist testing such that the efficiency can be improved. 77% of the developers have more than 2 years experience in compatibility testing. 6 of them usually work on at least 3 projects in a week and 77% developers (7 of 9) have to test more than 8 devices in one testing task. Note that 67% (6 of 9) developers complain that testing procedures are suffering from large difference in resolutions of devices. 67% developers alleviate the effect of resolutions by grouping devices in similar resolutions into more than 3 categories. This alleviation can reduce the resolution difference in one group. However, the workload of testing procedures is largely increased since it has to be replayed among all groups. The resolution difference makes the compatibility testing inefficiency. In our investigation, 77% of them agree that applying widget detection methods in testing can definitely increase the efficiency of testing process by at least 50%.

Answer to RQ1: From our interviews, we know that existing mobile game testing tasks are mainly completed by human testers, which is very inefficient. Thus, automated testing is urgently needed for the industrial games. To this end, developers and testers are exploring and developing some tools to assist the testing. However, these techniques have severe limitations (e.g., inaccurate results, different resolutions). The response of questionnaires indicates that how to precisely detect the *clickable* GUI widgets of mobile games is of great importance for automating testing tasks.

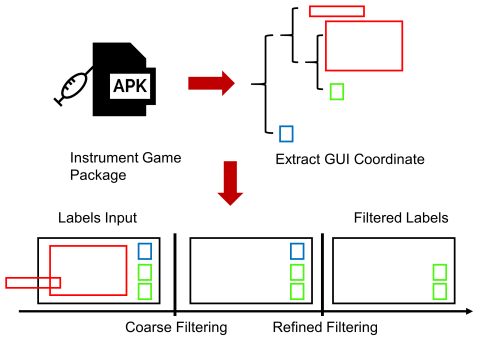

Figure 2: The workflow of building game dataset.

4 COLLECTION OF GAME GUIS

To the best of our knowledge, there is no game GUI benchmark now. To further facilitate mobile game testing, we made large effort to construct a mobile game GUI dataset. The overview of our workflow is shown in Figure 2. Specifically, we develop an automatic technique to collect game GUI images and label the widgets automatically in Section 4.1. Since the automatic labeling may introduce inaccurate results, we further adopt a heuristic-based data cleaning strategy to improve the data quality in Section 4.2.

4.1 Obtaining Game GUI Dataset

We develop a technique that can automatically locate the click widgets and outputs the region information of them, based on which we manually label the widgets. Specifically, we first instrument the game software by POCO such that, when a game is started, POCO can automatically instantiate and watch global members of game object [20]. After installing the instrumented game on the mobile devices, we connect the device to the computer and enable Android ADB to debug mode [3] on mobile devices. The screenshots, as well as GUI tree, are captured during the game playing. We then extract GUI widget coordinates based on the GUI tree and label the widget. However, there are a number of widgets that are irrelevant (e.g., widgets which are not clickable). We remove these widgets by filtering their element types (e.g., unclickable widgets are often with element type "Scene"). At last, we obtain 1135 GUIs with 17,808 elements from four games (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords and AllStar). Since mobile games are mostly played horizontally, the game screenshots are rotated for 90 degrees to keep the same size as the existing GUI dataset.

To enrich our dataset with games published by other companies, we spent three weeks to download games, take screenshots and manually label clickable widgets. We choose five games (i.e., Onmyoji, Arena of Valor, Princess Connect, Naruto and SevenDay) released by *NetEase Games*, *Tencent Games* and *Cygames*. Note that while the two games Onmyoji and SevenDay are developed by NetEase Games, they cannot be instrumented by POCO so we have to manually label them. After enough screenshots of a game (i.e., the screenshots could cover at least 80% play scenes) are collected, we label the clickable widgets by a third-party labeling tools [10]. From the above games, We obtain 1,849 GUIs and 20,968 widgets in total. The description of the game dataset is shown in Table 3. To avoid introducing manual mistakes, all labels are cross-checked by the co-authors.

Table 3: The composition of game GUIs in our dataset.

Game Name	#N of GUIs	#N of Widgets	Released by
Elysium of Legends	503	6,873	NetEase Games
Dream Chaser	224	3,838	NetEase Games
Butterfly Swords	307	5,160	NetEase Games
AllStar	110	1,937	NetEase Games
Onmyoji	693	8,050	NetEase Games
Arena of Valor	183	2,856	Tencent Games
Princess Connect	137	1,268	Cygames
Naruto	383	4,138	Tencent Games
SevenDay	453	4,656	NetEase Games
Total	2,993	38,776	

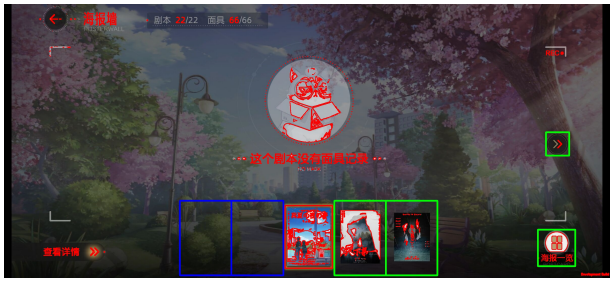

Figure 3: The example of applying contour filtering algorithm in GUI. The red dots denote detected text contours, the blue boxes denote boxes being filtered out and the green boxes denote boxes being reserved.

4.2 Data Cleaning

After collecting the 2,993 GUIs with 38,776 widgets, we observe that there are some inaccurate labels that may decrease the performance of models. Generally, the inaccurate labels can be grouped into two categories: (1) invalid labels and (2) incorrect labels.

The invalid labels include labels with negative coordinates and labels with coordinates outside the screen. These labels are mainly due to the limitation of the automatic labeling technique by extracting GUI information with POCO. Specifically, POCO automatically extracts GUI trees from screens. Without filtering, some illegally placed widgets (i.e., placed with negative coordinates or placed outside the screen) are included in labels. These invalid labels cause errors during training models. Therefore, they need to be cleaned before training.

The incorrect label denotes labels that are valid for training but harmful for training models. The incorrect label consists of two categories: (1) the irregularly large boxes and (2) empty boxes. Specifically, the irregularly large box denotes that the size of a box is far larger than regular box, these boxes produced by POCO development kit are ineffective for locating a clickable button. The empty boxes are incorrectly labeled by POCO due to the unawareness of visual overlaps among widgets. For example, in Figure 3, the two blue boxes are widgets in previous screens. They are buried under the current background and cannot be clicked. However, they are detected by POCO and are labeled incorrectly.

Algorithm 1: CoarseFiltering(): traversing all bounding boxes and filter out invalid coordinates.

```
   input  : P, all the screenshots
   input  : B, all the bounding boxes
   output : VB ← ∅, the set of valid boxes
1  foreach screenshot p ∈ P do
2      w, h ← p.getPitureSize()
3      //get screenshot width and height
4      foreach bounding box b ∈ B do
5          VB ← VB ∪ {b}
6          c ← b.getCoordinates()
7          // get values of the rectangle points
8          if hasNegtiveValues(c) is True then
9              VB ← VB − {b}
10         if hasOutsideScreenCoors(c, w, h) is True then
11             VB ← VB − {b}
12 return VB
```

To filter out such invalid and incorrect labels, we propose a label filtering method to remove dirty labels. Our method includes a coarse filtering and a fine-grained filtering.

Coarse filtering. We design the coarse filtering in Algorithm 1. The coarse filtering mainly focuses on invalid coordinates (i.e., coordinates with negative values or coordinates out of the screen). The input of Algorithm 1 includes the image source P and corresponding bounding boxes B. The output is a set of valid boxes. Algorithm 1 is composed of four steps: (1) obtaining width and height of the input screenshot at line 2; (2) checking if the coordinates of a box contain negative values at line 8; (3) checking if the coordinates are outside the screen at line 10 by comparing the width and height of the screen with the coordinates; (4) finally outputting a valid set of boxes of the screenshot. If the input set has n screenshots and m bounding boxes, the time complexity of line 2 is $O(nm)$.

Fine-grained filtering. The fine-grained filtering aims at filtering out irregularly large bounding box and empty box.

To remove the irregularly large bounding boxes, we propose to filter them by comparing the size of boxes with the size of the screen. That is, given a threshold factor λ, a bounding box whose size is larger than λ percent of size of the picture will be dropped. To find a proper value for λ, we randomly pick 10,134 bounding boxes from game dataset and collect the size of these boxes. Next, we compare the size of boxes with size of the screen. We find that 99.6% of bounding boxes has less than 10% of size of the screen, which means that almost all bounding boxes are small boxes in games. In fact, as we further investigate the 0.4% large boxes, we find that they are all unrelated background widgets and are not clickable. Thus they should be removed. In our implementation of this algorithm, the factor λ is set to 10.

To filter empty boxes, we apply the Suzuki's Contour Tracing Algorithm [39] that reports points of textures, text and contours of widgets. We empirically find that most empty boxes contain no contours. Therefore, we remove boxes that contain no points. For example, in Figure 3, the reported points from our algorithm are denoted as red dots. As the blue boxes contain no red dots, these boxes are deemed as empty boxes and are removed from labels. Differently, all of the green boxes contain red dots and they are deemed as valid labels.

Algorithm 2: FinegrainedFiltering(): traversing all bounding boxes and filter out irregularly large boxes and empty boxes.

```
   input  : P, all the picture source
   input  : B, all the bounding boxes
   input  : λ, the factor for determining the size of large boxes
   output : VB ← ∅, the set of valid boxes
1  foreach picture p ∈ P do
2      w, h ← p.getPitureSize()
3      S ← w × h
4      // calculate the size of screen
5      Con ← p.findAllContours()
6      // find contours by Suzuki's Contour tracing
        algorithm
7      foreach bounding box b ∈ B do
8          VB ← VB ∪ {b}
9          S_b ← b.getBoxSize()
10         // calculate the size of bounding box
11         if S_b > λ × S is True then
12             VB ← VB − {b}
13         foreach contour c ∈ Con do
14             if b.contain(contour) is True then
15                 break;
16             VB ← VB − {b}
17 return VB
```

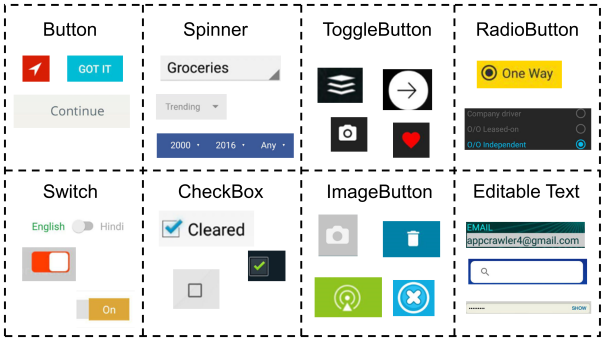

Figure 4: The examples of GUIs used in training models.

Algorithm 2 shows the method that is composed of four steps: (1) getting the size of the input image at line 2; (2) finding all contours in the picture by using Suzuki's Contour tracing algorithm at line 5; (3) calculating the size of bounding box and compare it with image size at line 9 to line 12; (4) checking if there exists a contour point within the bounding box at line 13 to line 16. If the input set has in total n images, m bounding boxes and we find c contours in a picture, the time complexity of Algorithm 2 is $O(nmc)$.

5 EVALUATION

5.1 Experiment Preparation

5.1.1 Dataset. To adopt existing methods [9, 24], we follow previous works [9] to download and preprocess the well-known RICO [12] dataset. The RICO GUI dataset contains 66,261 GUI screenshots and 199,830 GUI widgets. We remove widgets which are not clickable

(e.g., text widgets, images), not visible for users (e.g., overlapped widget, shadowed widget). After this, we obtain 61,906 widgets. We further filtered out 31,985 GUI widgets with incorrect coordinates (e.g., coordinates outside the screen). Finally, we collect 30,011 widgets. Examples of widgets used in our dataset can be found in Figure 4. Since the GUI screenshots have different image size, we resize them into the fixed resolution of 1440*2560. We split 30,011 elements RICO dataset into train/validation/test dataset with a ratio of 8:1:1 (24K:3K:3K). For game dataset, we use the data that is automatically labelled as training set and use the data that is manually labelled as the testing set because the manual labels are more accurate for the evaluation.

5.1.2 Model Training. We follow the training method in previous works [9, 24]. For Faster RCNN and YOLOv2, our training is based on their models which are pre-trained on COCO object detection dataset. We keep the default batch size of 256 and use SGD optimizer for Faster RCNN. Meanwhile, we use the default batch size of 64 and use adam optimizer for YOLOv2. Faster RCNN uses VGG16 [36] as the backbone while YOLOv2 utilized Darknet19 [32] as the backbone. Both models are trained for 45,000 iterations to ensure they are sufficiently trained. As the models may predict duplicated bounding boxes regarding the same object, we use non-maximum suppression (NMS) to remove redundant boxes and keep the best one.

5.1.3 Metrics. We adopt the metrics used in previous works [9, 24] to evaluate the performance of our models. Specifically, we use the precision, recall and F1 scores to evaluate the performance of models. The intersection over union (IoU) indicates the intersection of two regions A and B divided by the of the union region of them. The IoU threshold affects the precision of prediction. In previous works, the threshold ranges from 0.3 to 0.9. In order to better evaluate the model precision (i.e., the IoU threshold is not too low or not too high), we set the IoU threshold as a comparatively strict value 0.8 in our experiments. A true positive prediction (TP) is the prediction that satisfies both confidence threshold and IoU threshold, while a false positive prediction (FP) is the prediction that only satisfies confidence threshold. The false negative (FN) is the region missed by models. We calculate the precision rate by $TP/(TP + FP)$ and the recall rate by $TP/(TP + FN)$. We further compute the F1 score as: $F1 = (2 \times Precision \times Recall)/(Precision + Recall)$.

5.2 Effectiveness of Filtering

5.2.1 The Filtered Dataset. The labels filtered by coarse filtering and fine-grained filtering are shown in Table 4. In this table, "negative" and "outimg" denote coordinates that have negative values and coordinates outside the screenshot. The "large box" represents the bounding boxes with irregularly large size and the "empty box" represents the boxes containing no widgets.

We observe that only one game (i.e., Elysium of Legends) has negative coordinates and four games (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords, AllStar) have incorrect coordinates outside the screen. The reason is that the labels in these four games are automatically generated by our GUI information extraction technique (See Section 4.1). Since the extracted widgets are filtered by rules, some widgets with the out-of-GUI coordinates may be missed by rules and are included in our dataset. The model training cannot start when these invalid coordinates exist. Therefore, we leverage the coarse filtering to eliminate labels with invalid coordinates. After coarse filtering, 639 coordinates with negative values and 2,234 coordinates out of screen are removed.

The fine-grained filtering filters 336 irregularly large boxes. We observe that the games that are automatically labeled (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords and AllStar) tend to contain more dirty labels (e.g., 189 in Elysium of Legends and 103 in Dream Chaser). Our method also helps filter out empty boxes, and the results show that most game labels contain a number of empty boxes. Recall that the empty boxes are ones that are incorrectly labeled by POCO due to the unawareness of visual overlaps among widgets. For example, in Figure 3, the blue boxes are widgets in previous screens. They are buried under the background and not clickable in the current screen.

5.2.2 Model Performance on Filtered Dataset. In the evaluation, we use the games that are automatically labeled (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords and AllStar) as training dataset, and the manually labeled games (i.e., Onmyoji, Arena of Valor, Princess Connect, Naruto, SevenDay) as testing dataset. The results are shown in Table 5. In this table, the "Before Filtering" denotes performance of models on dataset only with coarse filtering, and the "After Filtering" denotes performance of models on dataset which has been processed by fine-grained filtering. Note that the dataset used here is already processed by the coarse filtering because negative or out-of-GUI coordinates can largely affect the performance. Hence, we mainly evaluate the model performance before/after the fine-grained filtering.

We observe that the performance of all models increases after we apply the fine-grained filtering. Specifically, Faster RCNN with default settings has the best performance with the highest precision (17.3%) and recall (16.4%) on the filtered dataset. Compared with the model with the same settings but on unfiltered dataset, the improvement of dataset leads to around 4.6%, 4.6% and 4.7% increase in precision, recall and F1 score, respectively. Faster RCNN with customized settings keeps the same unsatisfactory performance as in the previous empirical study. The performance of this model slightly increases on the filtered dataset. YOLO with k value 5 obtains the largest improvement on the filtered dataset. The processed dataset leads to more than 7% improvement on precision and at least 5% improvement on recall and F1 score. Comparatively, the performance of YOLO with k value 9 is improved on the filtered dataset for at least 2%. In summary, the filtered dataset effectively improves the model performance in terms of precision, recall and F1.

5.3 Model Performance

In this section, we conduct the experiments to evaluate the model performance on the RICO and game dataset. For models adopted in our experiments (i.e., Faster RCNN and YOLOv2), we prepare two sets of hyper-parameters to evaluate their performance. Specifically, for Faster RCNN, we change the original three anchor scales (8, 16, 32) to larger scales (16, 32, 64). We make this change in consideration that larger anchor scales may better fit the objects in GUIs. The Faster RCNN with larger scales is represented by "Faster RCNN

Table 4: Game GUI labels after filtering. The labels filtered by coarse filtering are in red cells and labels filtered by fine-grained filtering are in green cells.

	Original	negative	outimg	After Coarse Filtering	large label	empty label	After Fine-grained Filtering
Elysium of Legends	6873	639	976	5258	189	112	4957
Dream Chaser	3838	0	1187	2651	103	546	2002
Butterfly Swords	5160	0	37	5123	18	1159	3946
AllStar	1937	0	34	1903	26	391	1486
Total	17808	639	2234	14935	336	2208	12391

Table 5: The performance of models on filtered dataset.

Setting	Before			After		
	P	R	F1	P	R	F1
FR Default	12.9%	11.7%	12.2%	17.3%	16.4%	16.9%
FR Customized	0.5%	0.1%	0.1%	0.8%	0.2%	0.3%
YOLOv2 k=5	0.3%	≈0%	0.1%	7.6%	5.5%	6.4%
YOLOv2 k=9	4.4%	2.3%	3.0%	6.5%	5.0%	5.7%

Figure 5: The example of applying fine-grained filtering method to filter out dirty labels in manual labeled dataset.

Customized". For YOLOv2, we use two sets of k values (i.e., 5 and 9) in consideration that more anchors may improve the model performance. The two values are also adopted in previous works [31] for evaluations. The dataset for our evaluations includes RICO and game dataset.

5.3.1 Model Performance on Game Dataset. We compare the performance of deep learning models on both RICO and game dataset. We train our models with different settings on RICO dataset and test our models on game dataset. The results are listed in Table 6. The "P" and "R" in this table denote precision score and recall score, respectively. The "FR" represents the deep learning model Faster RCNN and the "FR Customize" means the model with changed anchor sizes. We also evaluate YOLOv2 model with different k values to study the impact of the parameters on model performance.

We observe that the precision and recall of models drop sharply on game dataset. Faster RCNN has 55.9% F1 score on RICO dataset but only 7.7% on game dataset. Specifically, the precision of Faster RCNN drop half on games and the recall drop from 59.1% to 4.6%. A recall under 5% is rather low for a model and denotes that the model is ineffectively to cover most GUI widgets. We also observe that the Faster RCNN with customized settings perform worse than model with default settings, and perform even worse on game datasets

Table 6: The performance of models (IoU > 0.8)

Trained On RICO Tested on RICO and Game

Setting	On RICO			On Game		
	P	R	F1	P	R	F1
FR Default	52.9%	59.1%	55.9%	23.4%	4.6%	7.7%
FR Customized	22.9%	10.4%	14.3%	1.1%	0.2%	0.3%
YOLOv2 k=5	50.7%	37.9%	43.4%	9.8%	1.3%	2.3%
YOLOv2 k=9	49.5%	36.5%	42.0%	10.0%	1.2%	2.2%

Trained On Game Tested on RICO and Game

Setting	On RICO			On Game Dataset		
	P	R	F1	P	R	F1
FR Default	4.1%	3.7%	3.9%	12.9%	11.7%	12.2%
FR Customized	0.1%	0.1%	0.1%	0.5%	0.1%	0.1%
YOLOv2 k=5	≈0.0%	≈0.0%	≈0.0%	0.3%	≈0.0%	0.1%
YOLOv2 k=9	≈0.0%	≈0.0%	≈0.0%	4.4%	2.3%	3.0%

with an F1 score of 0.3%. Recall that we attempt to change the anchor size and expect the model to precisely match bounding boxes labels. However, the experiment results show that our configuration with a larger anchor size leads to worse performance.

YOLO models perform close to Faster RCNN regarding the precision of 50.7%. However, the recall of YOLO is far worse than Faster RCNN. We observe that the performance of YOLO also sharply drops when tested on game dataset, with an F1 score decrease from 43.4% to 2.3%. Recall that we change the model setting of YOLO to a larger value of k (i.e., 9) and expect the YOLO model to generate more anchors to match bounding box labels. The experiments show that the configuration of a larger of k results in a worse performance.

5.3.2 Model Performance Across Various Mobile Applications. We also train models on game dataset and evaluate their performance on RICO and game dataset, respectively. The models are with same settings of models trained on RICO. The results are shown in Table 6. We observe that the models perform much worse on test dataset. Specifically, comparing the models tested on game and RICO dataset, the performance of them drops sharply. For Faster RCNN, the precision rate on test dataset drops from 12.9% to 4.1% and the recall rate decreases from 11.7% to 3.7%. The performance of RCNN with larger anchor sizes (i.e., customized settings) also drops from 0.5% to 0.1%. The Faster RCNN with the default setting performs better transferability on different datasets. For YOLO models, the performance of models with the two settings (k= 5, 9)

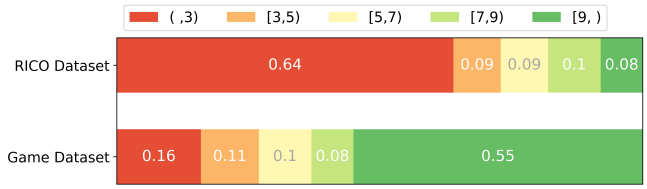

Figure 6: Distribution of the number of elements per GUI on RICO and game dataset.

Figure 7: A game GUI example of high widget density. The blue boxes denote labels and red boxes denote model predictions. The overlooked boxes are pointed by green arrows.

drops to less than 1% regarding precision, recall and F1 score. The YOLO models fail to produce an effective prediction on different dataset.

5.4 Root Cause Analysis

5.4.1 Game GUI Element Density. We first take a deep look at the UI density difference between regular application GUIs from RICO dataset and game GUIs from our collected dataset. The basic distribution of GUI density is summarized at Figure 6. We observe that 55% of game GUIs have more than 9 elements, and in contrast, only 8% of GUIs in regular applications contain have more than 9 widgets. Most regular applications contain no more than three widgets in one interaction screen. Differently, game applications intend to offer more interaction options to attract users. As the game GUI elements are often placed side by side and separated by only small padding, detecting GUI regions under high density is challenging for models. For example, Figure 7 is a screenshot of character selection. There are 50 elements in this GUI and each element is placed closely to others. Our models predict most regions of these elements, but missed 5 boxes.

5.4.2 Art Style Diversity in Game. The game GUIs are subject to art design needs. In fact, in some typical kinds of games (e.g., Massive Multiplayer Online Role-Playing Game), the mechanics of gameplay are rather complex. Unlike regular applications aiming at providing a serious of services, game developers intend to build a virtual world to attract users to explore it while enjoy gaming. To achieve this goal, game developers design widgets in different shapes and colors. For example, after we investigate GUIs in a game *Elysium of Legend* and GUIs in a hotel booking application of RICO dataset. We collect the number of heterogeneous widgets and regular widgets, the

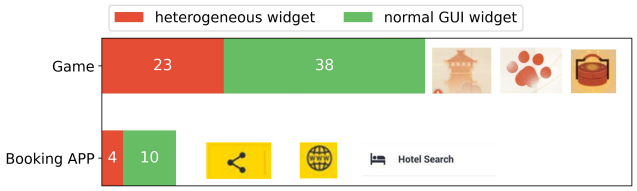

Figure 8: Comparing the number of GUIs categories in a game and a Android application. Followed by the examples of heterogeneous GUI elements in this game.

Figure 9: Return widgets in different games. Widgets in different game are separated by solid lines.

data is shown in Figure 8. In this booking application, there are 14 GUI elements in total and only 4 heterogeneous GUIs (e.g., sharing widget, map widget). In game software, there are more than 60 types of widgets, and 23 of them are heterogeneous widgets. These irregular widgets are often in various shapes (ancient buildings, cat's claws), as shown in Figure 8. Moreover, these widgets are often placed upon background pictures, making it difficult for models to separate background textures from widgets. In our experiments, all models missed the three irregular widgets.

Art style difference also causes huge diversity between games. We investigate the return widgets in nine games in our dataset, and results are shown in Figure 9. We surprisingly find that there exist no two similar return widgets. Some return widgets are even reshaped in uncommon style (e.g., the return widget in the left part of the second row), which confuse models to properly identify them. As the art style gap between games is often large, not to mention the gap between regular applications and games, and this also explains the reason why the performance of models trained on RICO dataset but tested on game dataset sharply drops.

5.5 Summary of Findings

5.5.1 Answers to the Research Questions. Through the analysis of experiments above, we could find that:

- **Answer to RQ2:** The models (i.e., Faster RCNN and YOLO) cannot generalize well across game and RICO dataset. The experimental results show that the performance of models drops sharply on game dataset. Meanwhile, models trained on game datasets are not accurate for detecting regular GUI widgets in non-game applications.
- **Answer to RQ3:** The models achieve unsatisfactory performance because: (1) widgets are compactly placed in game GUIs, making the density of GUI widget in game far larger than that in regular applications, (2) heterogeneous GUI shapes and high diversity of

GUI styles between games make it difficult for models to detect widget regions.

5.5.2 Research Direction Highlight. Based on our findings, we provide insightful suggestions for future research directions.

Enhancing Dataset. In our study, we automatically and manually labeled 2,993 screenshots of 9 games. Compared with previous works, RICO dataset includes more than 60K screenshots. Additionally, in order to assist better game GUI testing in industries, the dataset only provides coordinates of clickable widgets. This can be further enhanced by adding support of multiple types of widgets (e.g., text widgets, picture widgets) for facilitating future research on game GUIs. Additionally, the dataset can be enhanced by collecting various types of games (e.g., action games, adventure games, first-person shooting games).

Game Style Comparison. Recall that the model performs badly across different datasets. In fact, some mobile games share similar art styles with particular games. For example, the art style of the game Arena of Valor is similar to game Onmyoji Arena (not included in our dataset in this paper but also famous on Google Play Store). If we find a metric to evaluate similarity between the styles, we can summarize a minimum set of games which covers most game styles. In further training, models could be trained on this minimum set to save considerable amount of time.

Applying GUI Detection In Industry. As we have discussed with testing developers in *NetEase Games* about applying widget detection method in industry game testings, the developers agreed with that GUI widget detection techniques are important for game designs and game testings. Further, a game GUI dataset detection technique can help a freshman engineer rapidly become familiar with developments in new games. Currently, we are engaged in the implementation of this service and applying it in industry game testing in near future.

5.6 Threats to Validity

We note that the randomness is an inevitable factor when applying deep learning models. To alleviate the effect of this factor, we repeat the experiments mentioned in our study for 5 times and record the average values. The selection of games could be biased. In our study, we adopt 6 games released by them. To counteract the bias, we adopt additional games released by other companies (e.g., *Tencent Games*, *Cygames*). Also, the models adopted in our can be biased. In previous works, other fancy models (e.g., CenterNet [44], EAST [45], Grad-CAM [35]) are adopted in their methods. We choose to select Faster RCNN and YOLO in our experiments because they are commonly adopted in most similar studies. On the other hand, the recall rate which is adopted as our metric may be a potential threat. Generally, the recall performance of a model is difficult to evaluate due to the lack of ground truth. In our study, the labels of clickable widgets are mostly processed by our authors and are prone to introducing incompleteness. To alleviate this, we make our best attempt to check the labels for 3 times. On this basis, despite our labels are unable to be 100% complete, our dataset however provides a convincing benchmark for evaluating model performance.

6 RELATED WORK

In this section, we discuss works that are most relevant to ours.

GUI testing. GUI plays a key role in bridging the gap between users and applications. Therefore, previous works have proposed methods to aid GUI development in GUI searching [6–8, 33, 40] based on image features and GUI testing [5, 17, 42] based on deep learning models. Specifically, Hu *et al.* [17] proposed to automatically generate input cases for GUI testing. They feed input to the application and analysis the running traces to find bugs. Zhao *et al.* [42] trained a deep learning model to predict workflow actions of applications, which provides valuable experience of applying deep learning to advance the efficiency of GUI testings. However, the above techniques are all developed for general-purpose applications (e.g., hotel booking applications, shopping applications). They do not generalize well for game GUIs.

Object detection deep learning models. The object detection techniques greatly evolve in the past five years. The mainstream of the proposed methods can be roughly categorized into two-shot detection [13, 14, 16, 34] methods and single-shot methods [23, 31], based on their workflows. Object detection models sketch a tight bounding box around the object and classify what the object is. However, the above models are trained to identify objects in the real world. They cannot be directly applied in detecting GUI elements. Deka *et al.* [12] have published a dataset with 72K UI screenshots including widgets, buttons and scrolls, etc. The downloaded screenshots are labeled into 27 categories. Liu *et al.* [22] train convolutional neural networks on this dataset to detect UI components, which offers us valuable experience of applying deep learning models to detect GUI elements.

Game testing. Games are becoming increasingly popular along with the rapidly developing Internet. One game should be well tested to eliminate bugs before being published. However, as surveyed by Lin *et al.* [21], even the most popular games are not sufficiently tested. The main reasons for this imperfection can be summarized as the absence of automated testing techniques (i.e., manual testing is still dominant in game testing), due to the survey of Alemm *et al.* [2]. For mobile games, the current works are still preliminary. Lovreto *et al.* [25] develops a method of writing scripts to test functions on 18 mobile games. The limitations of existing techniques are also discussed in their study. Zheng *et al.* [43] first propose composite game testing technique by enhancing reinforcement learning with multi-objective optimization algorithms and outperforms other state-of-the-arts.

7 CONCLUSION

In this study, we first conduct a survey in *NetEase Games*, including scrum interviews and questionnaires. The survey results show that applying object detection method to detect game GUI widget can be the pillar to boost game testing efficiency in practice. To this end, we develop a method to automatically collect GUI of industrial games and a method for data-cleaning. The evaluations show that, 1) existing general-purpose GUI methods cannot perform well on games and 2) the unsatisfactory performance of existing methods is mainly caused by the compactly placed GUI widgets and the diverse GUI shapes.

REFERENCES

[1] Adrian. [n.d.]. Intersection over Union. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. Accessed September, 2018.

[2] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. 2016. Critical success factors to improve the game development process from a developer's perspective. *Journal of Computer Science and Technology* (2016), 925–950.

[3] Android. [n.d.]. Android ADB Debug Mode. https://developer.android.com/studio/command-line/adb. Accessed September, 2018.

[4] Android. [n.d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. Accessed September, 2018.

[5] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. 2018. GUIfetch: supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 236–246.

[6] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. *Proceedings of the ACM on Human-Computer Interaction* (2020), 1–22.

[7] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery DC: Design search and knowledge discovery through auto-created GUI component gallery. *Proceedings of the ACM on Human-Computer Interaction* (2019), 1–22.

[8] 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 TOSEM* (2020), 1–31.

[9] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. 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*. 1202–1214.

[10] Darrenl. [n.d.]. A graphical image annotation tool. https://github.com/tzutalin/labelImg. Accessed September, 2018.

[11] Darrenl. [n.d.]. How Big is the Global Mobile Gaming Industry?. https://www.visualcapitalist.com/how-big-is-the-global-mobile-gaming-industry/. Accessed September, 2020.

[12] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. 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*. 845–854.

[13] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

[15] Google. [n.d.]. A lightweight, fast, and customizable Android testing framework. https://developer.android.com/training/testing/espresso. Accessed September, 2018.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence* (2015), 1904–1916.

[17] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 77–83.

[18] Alibaba Inc. [n.d.]. OneToMany: a wireless, non-invasive testing tool for automatic Android software testing. https://github.com/alipay/SoloPi. Accessed September, 2018.

[19] Netease Inc. [n.d.]. POCO: A cross-engine UI automation framework. https://github.com/AirtestProject/Poco. Accessed September, 2018.

[20] Unity Inc. [n.d.]. Unity Documentation. https://docs.unity3d.com/ScriptReference/GameObject.html. Accessed September, 2018.

[21] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. 2017. Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering* (2017), 2095–2126.

[22] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning design semantics for mobile apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 569–579.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. 21–37.

[24] 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)*. 398–409.

[25] Gabriel Lovreto, Andre T Endo, Paulo Nardi, and Vinicius HS Durelli. 2018. Automated tests for mobile games: An experience report. In *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 48–488.

[26] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[27] Atif M Memon and Myra B Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *2013 35th International Conference on Software Engineering (ICSE)*. 1479–1480.

[28] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 559–570.

[29] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 15–18.

[30] Kevin Moran, Mario Linares Vásquez, and Denys Poshyvanyk. 2017. Automated GUI testing of Android apps: from research to practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 505–506.

[31] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[32] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7263–7271.

[33] Steven P Reiss, Yun Miao, and Qi Xin. 2018. Seeking the user interface. *Automated Software Engineering* (2018), 157–193.

[34] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497* (2015).

[35] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[36] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[37] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[38] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[39] Satoshi Suzuki et al. 1985. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing* (1985), 32–46.

[40] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.

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

[42] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. [n.d.]. ActionNet: Vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st ICSE*. IEEE, 350–361.

[43] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. [n.d.]. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 772–784.

[44] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. 2019. Objects as Points. In *arXiv preprint arXiv:1904.07850*.

[45] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 5551–5560.