AG3: Automated Game GUI Text Glitch Detection Based on Computer Vision

Xiaoyun Liang
liangxiaoyun.duo@bytedance.com
Bytedance
Shenzhen, China

Jiayi Qi
qijiayi@bytedance.com
Bytedance
Beijing, China

Yongqiang Gao
gaoyongqiang@bytedance.com
Bytedance
Shenzhen, China

Chao Peng
pengchao.x@bytedance.com
Bytedance
Beijing, China

Ping Yang
yangping.cser@bytedance.com
Bytedance
Beijing, China

ABSTRACT

With the advancement of device software and hardware performance, and the evolution of game engines, an increasing number of emerging high-quality games are captivating game players from all around the world who speak different languages. However, due to the vast fragmentation of the device and platform market, a well-tested game may still experience text glitches when installed on a new device with an unseen screen resolution and system version, which can significantly impact the user experience. In our testing pipeline, current testing techniques for identifying multilingual text glitches are laborious and inefficient. In this paper, we present AG^3, which offers intelligent game traversal, precise visual text glitch detection, and integrated quality report generation capabilities. Our empirical evaluation and internal industrial deployment demonstrate that AG^3 can detect various real-world multilingual text glitches with minimal human involvement.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Scene anomaly detection**.

KEYWORDS

Visual Test Oracle, Software Testing, Deep Learning

ACM Reference Format:
Xiaoyun Liang, Jiayi Qi, Yongqiang Gao, Chao Peng, and Ping Yang. 2023. AG3: Automated Game GUI Text Glitch Detection Based on Computer Vision. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3613867

1 INTRODUCTION

Graphical User Interface (GUI) refers to the graphical display of the device operation user interface and is mainly consists of graphical buttons and text, which significantly contributes to the user experience of the application [26]. Due to issues such as device compatibility, various GUI glitches may frequently occur in gaming apps. Text glitches are one of the noticeable types of glitches, which usually appear in multilingual games and have a severe impact on the user experience when the glitch appears on game instructions, direction labels, and other critical parts. With the global expansion of the gaming industry and the increasing demand for high-quality games, the number of multilingual games has significantly increased. Therefore, detecting text glitches is critical to ensuring the user experience and preventing potential user loss.

In the industrial environment, game quality assurance mainly involves manual inspection and automated GUI exploration. However, they face several challenges with multilingual text glitch detection. Firstly, with the diversity of mobile devices and the accelerated update frequency of game versions, manual inspection requires significant manpower to deal with compatibility issues between multiple versions, device models, operating systems, languages, and screen resolutions. Furthermore, for multilingual GUI issues, subtle glitches may be overlooked due to human carelessness.

Secondly, the vast majority of existing automated GUI exploration tools only provide crash and functional bug detection capabilities [4, 16, 18, 19, 24, 33–35, 37–39, 42, 43, 47, 52]. To the best of our knowledge, only [12, 31, 32, 45, 46, 53] support GUI glitch detection, and among these tools, OwlEyes [32, 53] and GLIB [12] include text glitch detection. However, these tools primarily focus on graphical glitches and do not pay enough attention to detecting text glitches in multilingual games. Glib generates GUI glitch data by modifying the source code of the game, which is not feasible when there is no access to the source code and build tools. Moreover, GLIB does not mention the game traversal algorithm they use. On the other hand, OwlEyes is designed for regular Android apps and employs DroidBot [29] for app exploration by querying standard widgets through the Android Debugging Bridge (ADB) and operating on available widgets shown on the current screen. However, game screens are usually rendered using game engines, and operable GUI elements are not detectable through ADB. Other GUI glitch detection tools [31, 45, 46] focus on using reinforcement learning and imitation learning to interact with key objects in a game, rather than covering more textual scenarios in the game.

To address the aforementioned challenges, we present AG^3 (Automatic Game GUI Text Glitch detection), which provides the following capabilities:

(1) **Image-based decision-making.** AG^3 extracts operable GUI elements for game exploration and detects text glitches based on computer vision. This technique is non-intrusive to the game under test and has advantages such as no source code needed, minimal dependency, and high scalability.
(2) **Intelligent game traversal.** AG^3 uses an intelligent traversal algorithm that prioritizes exploring scenes with more text and only requires a simple configuration before testing any multilingual game to reduce human intervention.
(3) **Multilingual text glitch detection.** AG^3 supports the detection of text overlap and text overstep in multilingual games. We conduct an in-depth study of the causes and characteristics of these issues and design the optimal detection algorithm for them by deep learning methods.
(4) **Test report generation.** After each round of testing, AG3 automatically generates a comprehensive report that contains the number, screenshots, and repair suggestions of text glitches. This report is useful for game developers to track and fix issues, and ensures the overall quality of the game.

To evaluate the effectiveness of AG^3, we conduct an empirical evaluation of the tool's game traversal and text glitch detection capabilities. Experimental results show a high precision rate. Furthermore, after two years of industrial deployment, AG^3 uncovered over 2000 real text glitches in our games and its automatically generated test reports have helped our developers reduce the time spent on fault localization and debugging significantly. The experimental results and daily deployment of AG^3 demonstrate its usefulness in detecting and reporting text glitches accurately and efficiently.

In summary, this paper presents a new automated multilingual game testing framework with intelligent traversal and optimal text glitch detection capabilities. The main contributions are:

- We design detection algorithms based on deep learning for two common types of game text glitches, namely text overlap and text overstep.
- Our approach significantly reduces the manpower required for game testing teams and improves the overall efficiency of the testing process.
- We conduct a series of experiments and report the daily deployment of our framework, demonstrating its effectiveness in automatically playing games and detecting text glitches with high precision.
- We discuss our reflections and lessons learned from the implementation and deployment of our approach in the industrial environment.

2 BACKGROUND

In this section, we provide a brief introduction to the fundamental concepts of game testing and text glitch detection.

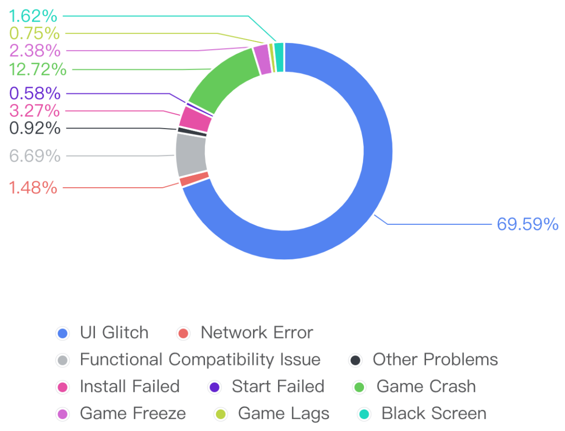


Figure 1: Compatibility Issue Distribution

2.1 Automated Game Testing

Manual game testing involves testers playing the game and performing traversal scripts, which are typically written in the form of a specification on how to play the game or recorded GUI action sequences for tools to replay and screen for visual bugs. Manual testing is usually labor-intensive, and writing comprehensive tests is challenging and time-consuming. This gives rise to automated testing tools, which can be classified into two categories:

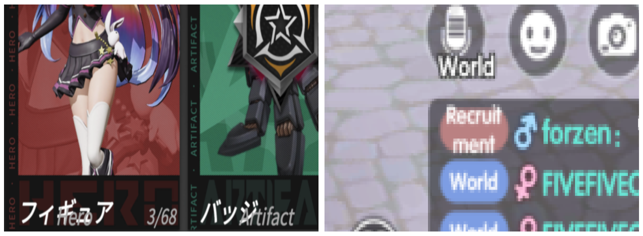
Source code instrumentation-based tools obtain the GUI tree of the game under test and inject executable operations into the process through the software development kit (SDK) provided by the game vendor. The major disadvantage of this method is that a poorly developed SDK may have side effects on the performance and stability of the host app, which can compromise the quality of the game. Additionally, this method may not be able to operate on system-level pop-up windows, which can lead to incomplete testing results.

Image-based tools utilize image recognition techniques to identify operable GUI elements displayed on the screen. As screens can be captured via screenshots, this method does not require any code or app instrumentation, and it does not affect the stability and performance of the game under test. This makes it a non-intrusive and scalable technique for automated game testing.

2.2 GUI Glitches and Text Glitch Detection

In 2021, a game testing company TestBird [54] conducted a study in which they collected and tested 32, 362 mobile game apps. The study reported a total of 1, 293, 809 compatibility issues across 10 different categories, with GUI glitches accounting for the largest proportion, as illustrated in Figure 1. Among the GUI glitches, text glitches were a major issue, and detecting them requires not only efficient detection algorithms but also practical and effective automated game exploration tools. As a result, the development of reliable and accurate tools for the detection of text glitches, as well as other GUI issues, has become increasingly important in the gaming industry.

Among the multilingual text glitches found in games, text overlap and text overstep are the most common issues. Thus, we focus on detecting these two types of issues. Text overlap occurs when multiple text lines overlap, as shown in Figure 2a, while text overstep happens when the text exceeds the GUI box in which it is located,



(a) Text Overlap **(b) Text overstep**

Figure 2: Examples of Text Glitches

as shown in Figure 2b. These two issues usually arise due to device compatibility and language translation problems. The length of words and sentences with the same meaning varies greatly between different languages, leading to text spilling over the GUI box or overlapping with other text in the original GUI box.

3 OUR APPROACH

In this section, we introduce AG^3, a non-intrusive multilingual game testing tool, which aims to achieve two primary objectives: high game scenario coverage and effective glitch detection. AG^3 consists of two main modules: Intelligent Traversal (Section 3.2) and text glitch detection (Section 3.3).

3.1 Overview

Automated code instrumentation for glitch detection is not practical for industrial game apps as different game studios have customized building pipeline, project structure and native dependencies. In additional, we cannot obtain complete source code from all game studios due to confidential reasons. As a result, we recognize the feasibility of image-based techniques in supporting the quality assurance of our games. Through discussions with our development and testing teams, we have identified that text overlap and text overstep bugs often occur in multilingual game screens. However, due to the complex nature of game screenshots and the small areas affected by text glitches, it is challenging to locate these bugs even with manual viewing. Moreover, there has been no dedicated work to define and study multilingual game text glitches. Therefore, we define and study these two types of game text glitches in this paper and develop detection algorithms for them.

The automation testing workflow of AG^3 is shown in Figure 3. In this workflow, the Intelligent Traversal module receives the screenshot of the current game screen, encodes, analyzes, stores, and decides on the next operation, and then completes the interaction with the device until the entire game traversal is completed. The end of the game traversal is primarily determined by the task configuration. Simultaneously, the tool automatically detects text glitches on the game screenshots and saves the detection results into the test report. The report is sent to the tester's mailbox automatically after the game traversal is complete. The test report includes the test case name and elapsed time, as well as the number of text overstep and text overlap detected and screenshots of text glitches (like Figure 6). The time interval for text glitch detection is generally set to the average time interval of adjacent interactions, and it is set to 1 second, empirically. Overall, the automation testing workflow of AG^3 aims to provide a comprehensive and efficient automated testing solution for multilingual games.

3.2 Intelligent Traversal

To enable scene classification, game GUI detection, and OCR engine, we use a combination of traditional digital image processing methods and deep learning methods. The powerful target localization and state perception technology of computer vision is leveraged to construct a scene traversal graph, which enables the generic game automated traversal for supporting the automated game text glitches detection.

Intelligent Traversal comprises four core modules: State Perception, Status Recorder, Observer, and Manager (described in detail below). The primary goal is to dynamically construct a directed game traversal graph and complete a full traversal of that graph, i.e., to operate on all GUI on each scene node. Our game traversal graph uses scene screenshots as nodes and the operation information between two different scene screenshots as their edges. It is important to note that the image similarity of the two screenshots is calculated to determine if they are the same node. The image similarity calculation method refers to the cosine similarity [58] between two image encoding features.

3.2.1 *State Perception.*

To facilitate efficient and accurate automated traversal of game scenes, Intelligent Traversal analyzes the GUI layout of the current screenshot and encodes the screenshots. The analysis results are sent to State Recorder, Observer, and Manager, respectively. For image encoding, we spread the last layer of convolutional features of VGG16 [55] into a one-dimensional vector as the encoding feature of the screenshot, which is used for image similarity calculation.

The GUI layout analysis includes multi-category GUI detection, multilingual OCR engine, and game screenshot scene classification. The multi-category GUI detector is responsible for locating and classifying GUI elements on the game screenshot into three categories: "Button", "Cancel", and "Pop-ups". Our multilingual OCR engine can support 44 languages and is responsible for the detection and recognition of multilingual text. Game screenshot scene classification is implemented using a simple deep learning classifier that classifies game screenshots into three categories: normal game screen, non-game screen (the screen outside the game APP), and game trap screen (the screen that does not need to be traversed in the game APP, such as advertising screens).

The above-mentioned GUI locations, OCR results, screenshot scene types, and image encoding features collectively form the features of each scene node in the game traversal graph. Overall, by leveraging state-of-the-art computer vision and deep learning techniques, Intelligent Traversal provides a comprehensive analysis of screenshots, allowing for efficient and accurate game scene traversal and the detection of text glitches in multilingual games.

3.2.2 *Observer.*

The Observer module is responsible for monitoring the traversal state and detecting exceptions such as being out of the game interface and game screen stuck. When such exceptions occur, the Observer module synchronizes the exception message to the Manager and Status Recorder.

Out of the game interface exceptions are determined by the screenshot types from State Perception. If the similarity of two

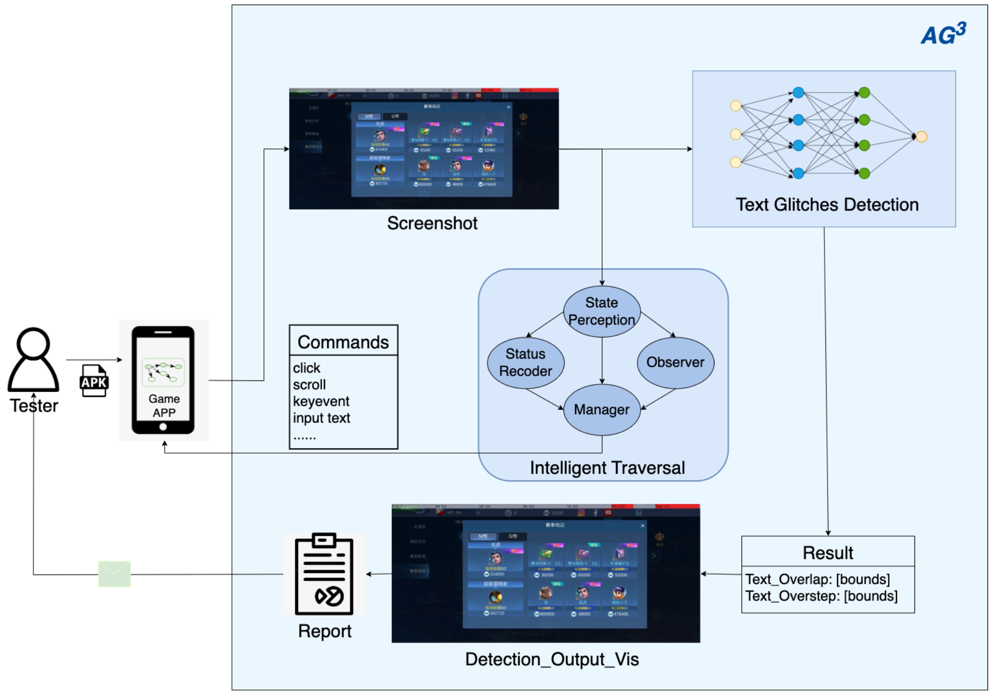


Figure 3: AG^3 **'s Workflow in Testing**

consecutive screenshots is high, it is indicative of a game screen that is stuck and not responding.

3.2.3 Manager. The Manager module analyzes the current node information from State Perception, the exception information from Observer, and the game traversal graph to make a comprehensive decision on the next operation. It then sends the decision command to the device to complete a single round of interaction and updates the traversal graph. It also sends the decision information to the Status Recorder to record the traversal progress.

3.2.4 Status Recorder. The Status Recorder module is responsible for recording all input information, including all node characteristics output by State Perception, exception information monitored by Observer, decision and traversal graph output by Manager.

Overall, by leveraging computer vision and machine learning techniques, the Intelligent Traversal module enables efficient and accurate automated traversal of game scenes, facilitating the detection of text glitches in multilingual games.

3.3 Vision-Only Text Glitches Detection

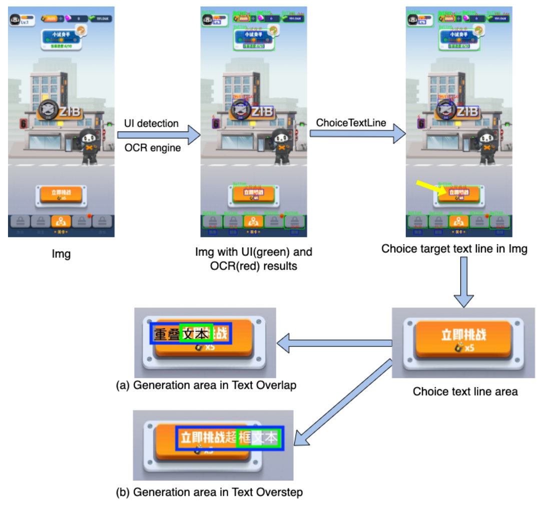
As an non-intrusive tool, AG^3 relies solely on visual information to detect text glitches based on computer vision. There are two main approaches in computer vision: digital image processing and deep learning. Digital image processing involves using computer algorithms to manipulate digital images, while deep learning relies on artificial neural networks to learn features directly from data. In our work, we focus on deep learning methods and use a combination of digital image processing and deep learning to synthesize training samples. Specifically, we use CNNs and transformers to detect and classify text glitches, enabling AG^3 to automate the process and improve the efficiency and effectiveness of game testing.

3.3.1 Data Collection. To train our deep learning models, we first preprocessed the collected screenshots by resizing them to a fixed size and converting them to grayscale images. In order to develop text glitches detection algorithms with strong generalization capabilities, we collected as many real historical bug screenshots as possible from our testing platform and the game studio quality assurance team, the specific amount of evaluation data is shown in Table 1. These screenshots were used to analyze the pattern of anomalies and for subsequent detection evaluation. Also, to evaluate the effect on real normal screenshots, real normal images from more than 100 games were collected from different sources.

As text glitch screenshots are rare in comparison to normal screenshots, text glitch data synthesis is necessary for this task. To generate the text glitch data, a generation algorithm is designed based on digital image processing. The pseudo code for this generation algorithm is shown in Algorithm 1, and its process is illustrated in Figure 4. During the data generation process, the location of the overall abnormal text line area ($[x, y, w, h]$, depicted by the blue box in the generation image in Figure 4) and its start and end coordinates ($[x_{start}, y_{start}, x_{end}, y_{end}]$, depicted by the green box in the generation image in Figure 4) are recorded for model training.

Overall, our text glitch detection models are trained using a large dataset of both synthesized and real glitch samples, ensuring that they have strong generalization capabilities for detecting various types of text glitches in multilingual games.

3.3.2 Text Abnormal Detection Algorithm. There are two common tasks of computer vision: image classification and object detection.



UI detection
OCR engine
ChoiceTextLine
Img
Img with UI(green) and OCR(red) results
Choice target text line in Img
Choice text line area
(a) Generation area in Text Overlap
(b) Generation area in Text Overstep
重叠文本
立即挑战超框文本

Figure 4: Data Generation Process

Image classification takes an image as input and outputs with all the probabilities that the image belongs to each category. Object detection takes an image as input and outputs the target objects and their locations and confidence scores respectively.

We introduce 2 different models for text overlap and text overstep detection. for text overlap detection, features of the text overlap anomalous regions are relatively subtle compared to the complex game background and are more similar to the normal samples, a single image classification or object detection model cannot achieve plausible results. So we use a combination of image classification and object detection models to detect text overlap issues. For text overstep, anomalous regions are more visible and a single object detection model is sufficient.

Text Overlap Detection. From a global view of the page, the anomalous region is often very subtle, which can have a significant impact on the object detection model that takes the whole page as input. To improve model effectiveness, we further classify the candidate local regions detected by the object detection model. Since each character in the anomalous region will correspond to a normal or anomalous label, this is somewhat similar to tasks such as text classification in natural language processing. Given the recent success of Transformers in natural language processing, we chose to build our classifier based on the Vision Transformer (ViT) [21].

Firstly, YOLOX [23] is used to detect and locate suspected abnormal text, and the text image is cut out from the original image based on coordinates. The image is then sent to the transformer classifier to perform binary classification for each token. The transformer classifier is introduced with the example of overlapping text, and its structure is shown in Figure 5. Refer to the ViT (Vision Transformer) [21] and the characteristics of abnormal text images, we divide the whole image into 16 small image patches according to the horizontal direction. Then, these linear embedding sequences and a learnable embedding vector cls-token are inputted to the transformer-encoder, and each encoding image patch is binarized classified by MLP (Multilayer Perceptron) [50]: normal or abnormal. Finally, when the cls-token patch is classified as abnormal, and three or more consecutive patches are also classified as abnormal, the area itself is classified as abnormal.

Algorithm 1 Text Abnormal Image Generation

```
Input: Normal Image $Img$, generated object $GenerateProcess$
1:  $TextLineList \leftarrow$ the list of all text lines in $Img$ obtained by OCR engine. Each text line includes coordinate information ($x$, $y$, $w$, $h$) and text content $Text$
2:  $ButtonUIs \leftarrow$ the list of UI with "Button" label of $Img$ by UI detector
3:  $Button \leftarrow$ the button from $ButtonUIs$, and a text line is surrounded by it
4:  $TextPixels \leftarrow$ the foreground text pixel value of a text line
5:  $TextOverlap \leftarrow$ the text overlap data generation process
6:  $TextOverstep \leftarrow$ the text overstep data generation process
7:  $AbImg \leftarrow$ the synthetic image obtained after $Img$ is written into abnormal texts
8:  if $GenerateProcess$ is $TextOverlap$ then
9:      $ChoiceTextLine \leftarrow$ the target text line, which randomly selected from $TextLineList$
10:     $AbnormalArea \leftarrow$ the area to be written in $ChoiceTextLine$, that is, the coordinates of the text overlapping area ($x_{start}$, $y_{start}$, $x_{end}$, $y_{end}$)
11:     Write a text with random $Text$ and font in $AbnormalArea$ of $Img$ to get $AbImg$
12: end if
13: if $GenerateProcess$ is $TextOverstep$ then
14:     $ChoiceTextLine \leftarrow$ the target text line, which randomly selected from $TextLineList$ conditional on it has surrounding $Button$
15:     $AbnormalArea \leftarrow$ the area to be written in $ChoiceTextLine$, that is, the coordinates of the text exceed its surrounding $Button$ ($x_{start}$, $y_{start}$, $x_{end}$, $y_{end}$)
16:     Get the $TextPixels$ of $ChoiceTextLine$
17:     Write a text with random $Text$ with $TextPixels$ in $AbnormalArea$ of $Img$ to get $AbImg$
18: end if
19: Update the abnormal text line location:($x$, $y$, $w$, $h$)
Output: Synthetic text abnormal image $AbImg$ with abnormal text line location ($x$, $y$, $w$, $h$) and abnormal area coordinates ($x_{start}$, $y_{start}$, $x_{end}$, $y_{end}$)
```

The synthetic abnormal images mentioned earlier are used to train the whole YOLOX model. Locations of the abnormal text regions and the specific starting and ending coordinates of the abnormal regions are used to construct positive samples of the training data for the classification model. Meanwhile, we use OCR technique to select a portion of normal sample regions from the training set of the object detection model as negative samples.

Text Overstep Detection. We use a simple YOLOX model for text overstep detection. The training dataset is also constructed using the same data synthesis method mentioned before.

3.4 Implementation and Usage

AG^3 is implemented in Python and compiled into a wheel package. After setting up the configuration file to specify the installation file

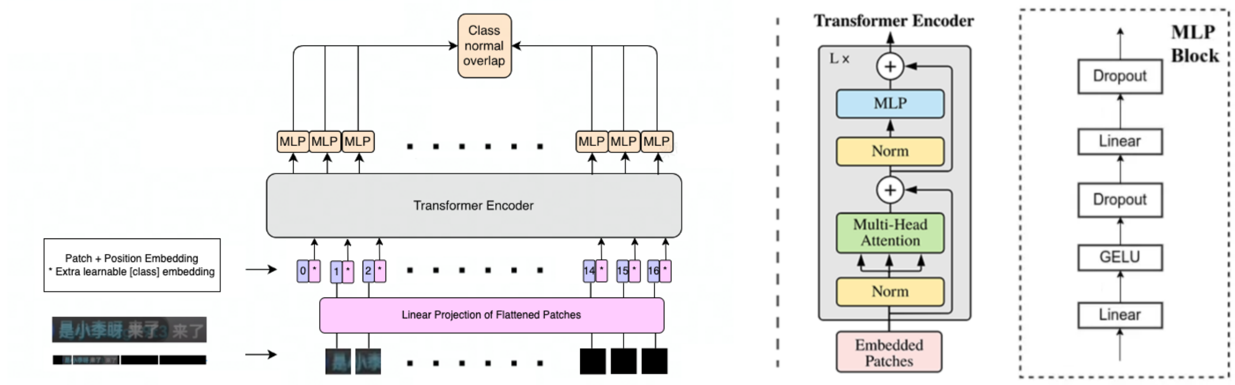
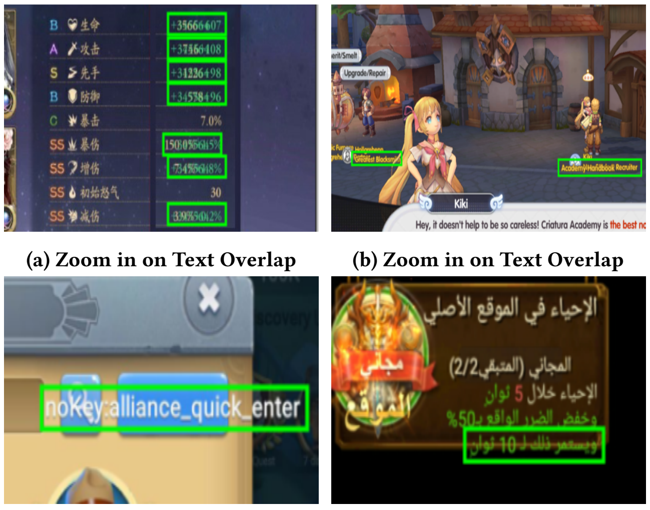


Figure 5: Transformer Classifier Structure



(a) Zoom in on Text Overlap (b) Zoom in on Text Overlap

(c) Zoom in on Text Overstep (d) Zoom in on Text Overstep

Figure 6: Detection Result Examples

of the game, device connection configuration and time duration, the tester can quickly start Intelligent Traversal and Text Glitches Detection by a single command. After the automatic traversal of the game, the text glitches detection report will be automatically sent to the tester's mailbox. In addition, modules of AG^3 are provided as independent functions, making the tool customizable by writing a new test script and calling these functions.

4 EVALUATION AND DEPLOYMENT

We investigate the following research questions (RQ) based on empirical evaluation and industrial deployment to study the effectiveness and usefulness of AG^3:

RQ1. How effective is AG^3 in terms of detecting multilingual game text glitches?
RQ2. How effective is AG^3 in terms of automated game traversal?
RQ3. What are the results achieved by AG^3 in practical industrial deployments?

We discuss evaluation metrics, results and analysis for each research question in the rest of this section.

Glitch Type	Normal	Glitch
Text Overlap	1,658	55
Text Overstep	1,658	288

Table 1: Evalutation Data Distribution

4.1 RQ1. Effectiveness of Multilingual Game Text Glitch Detection

Experiment Datasets. The evaluation dataset used in our study includes real historical text abnormal screenshots, also known as glitch images, provided by our quality assurance team, as well as over 1600 real screenshots without glitches, known as normal images, from more than 100 real games. The use of real-world data for the evaluation dataset enables us to better understand the effectiveness of our work in practical applications. The distribution of the dataset is presented in Table 1, which indicates that the proportion of normal and abnormal samples is significantly imbalanced, which is consistent with the distribution of the number of abnormal samples in the actual run.

Regarding the training dataset, we utilized the approach outlined in Algorithm 1 to process a variety of internal game screenshots (which did not intersect with the evaluation dataset) that were gathered by the QA team over an extended period of time, allowing us to obtain synthetic anomaly images. Both Text Overlap and Text Overstep were utilized, and the resulting training dataset consisted of samples of whole page screenshots that were generated via the synthesis algorithm, with a learning rate of 1e-3 and a batch size of 128 used for YOLOX model training. Additionally, we cropped several normal and abnormal text areas from this dataset in order to further train the text overlap classification model, which utilized a learning rate of 1e-5 and a batch size of 128. The quantity and distribution of the datasets can be seen in Table 2. Both text overlap and text overstep use 10000 game screenshots with generated text anomalies as training set, and the ViT classifier for text overlap uses 13800 generated text overlap images and 6900 normal text images as the training set.

Evaluation Metrics. To evaluate the effectiveness of our detection methods, we use four types of evaluation metrics. Three of them are commonly used metrics of image classification in the literature

Glitch Type	YOLOX	ViT Normal	ViT Abnormal
Text Overlap	10000	13800	6900
Text Overstep	10000	N/A	N/A

Table 2: Training Data Distribution

and we introduce an additional error rate as summarized below. We refer to glitch images as positive samples.

Accuracy reflects the models ability to make correct decisions on the test set. The model accuracy is calculated as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where a TP (true positive) is a positive outcome that is also predicted correctly by the model to be positive and a TN (true negative) is a negative outcome that is also predicted correctly by the model to be negative. On the other hand, an FP (false positive) is a negative outcome but predicted to be positive and an FN (false negative) is a positive outcome but predicted to be negative. The more correct samples the model predicts, the higher accuracy value it will have.

Precision is the proportion of the correct samples predicted as glitch (TP) among all samples predicted as glitch. The model precision is calculated as

$$Precision = \frac{TP}{TP + FP}$$

This can represent the feelings of use when reviewing manually suspected problems.

Recall indicates the proportion of correct samples predicted as glitch among all ground truth glitch samples and calculated as

$$Recall = \frac{TP}{TP + FN}$$

This metric reflects the ability to spot real GUI glitches.

Error rate (ER) is designed by ourselves, which indicates the proportion of error samples predicted as glitch (FP) in all normal images and calculated as

$$ER = \frac{FP}{TP + FP}$$

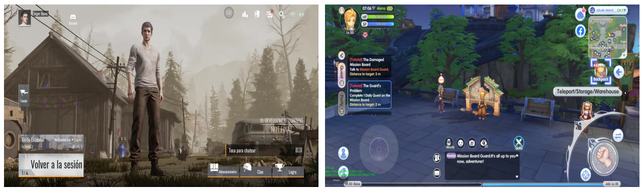
Because of the imbalance of our evaluation dataset, which is similar to the real-world situation, we need ER as the metric for reflecting the effect of FP in real application.

Comparison. After studying and analyzing others' work [12, 31, 32, 45, 46, 53], text glitches related detection is mentioned in GLIB [12] and OwlEyes [32, 53]. OwlEyes and GLIB are two tools focusing mainly on the detection of GUI glitches. Since OwlEyes does not open source their model and the application scenario is a non-game application, we only do a comparison of text glitches GLIB with AG^3 on the evaluation metrics mentioned above. Also, for text overlap detection, we will evaluate the improvement of the classification module on the detection effect.

Results And Analysis. The experimental result for text glitches are shown in the table 3 and 4. From the results, we can see that AG^3 achieves the best results in all four metrics and is not affected by the imbalanced distribution of normal and abnormal images. Meanwhile, the precision and recall of GLIB on multilingual text glitches is very low. The reason for this is that the magnitude of text glitches in GLIB's anomaly synthesis is small (2%) and differs significantly from the form of text glitches in multilingual scenarios.

Also, from the results, the addition of the classification model to the text overlap detection greatly improves the precision of the model while keeping the other metrics stable. Because the introduction of the posterior classification model does filter out many anomalous sample regions that are incorrectly recalled.

We present more detection examples as shown in Figure 6.



(a) Shooter Game **(b) Action RPG Game**

Figure 7: The screenshots of the two games

4.2 RQ2. Intelligent Traversal Effectiveness

Experiment Design. To measure the traversal effectiveness, techniques for ordinary apps usually use code or activity coverage as the criteria. However, code coverage requires access to the source code, and the game app is rendered using the game engine and does not distinguish between activities. Therefore, these coverage metrics are not suitable in our case. Moreover, the goal of our tool is to support multilingual game text glitches detection, so we are more concerned with how well the game traversal algorithm covers the text resources. Therefore, a new coverage metric, Text-Coverage is proposed in this paper. We apply AG^3 on two popular games in different languages: Shooter Game and Action RPG Game. Shooter Game is a first-person shooter (FPS) game, while Action RPG Game is an action-adventure game. Both of them are complex and large games developed using Unity 3D. The running time setting for a game traversal is 24 hours, and the resolution is (2400, 1080, 3). Figure 7 depicts the example screenshots of the two games.

Evaluation Metics. The Text-Coverage can be used to measure the degree of text coverage of the game intelligent traversal algorithm, and it is defined as:

$$TC = \frac{k}{n}$$

where n is the number of deduplicated text terms contained in the language pack (LP in short) of a game and k represents the number of matching LP terms found during Intelligent Traversal. The LP is maintained and provided by the game development team, which contains all the text terms in the game, and will also contain some redundant terms, such as: discarded words from old versions of the game, special triggered terms, etc. The partial terms of the LP for the Action RPG Game in English is shown in Table 5.

Before calculating the Text-Coverage achieved by Intelligent Traversal on a game, it is necessary to record the whole traversal process and split the frames. According to the adjacent operation time interval, we set the frame rate to 2fps. Therefore, we can obtain all the screenshots in the traversal process, and put them through the OCR engine to get all the traversed terms. The TC is calculated mainly by the Length Adaptive Levenshtein Distance to

Model	Precision	Recall	Accuracy	Error Rate	Distribution (Normal/Glitch)
GLIB	14.29%	7.27%	95.62%	85.71%	1658/55
AG^3 w/o ViT	31.45%	90.91%	94.30%	68.55%	1658/55
AG^3	94.34%	90.91%	99.60%	5.66%	1658/55

Table 3: Text Overlap Results

Model	Precision	Recall	Accuracy	Error Rate	Distribution (Normal/Glitch)
GLIB	4.00%	0.35%	84.02%	96.00%	1658/288
AG^3	81.03%	97.92%	96.40%	18.97%	1658/288

Table 4: Text Overstep Results

Term
Character pre-creation phase. Unable to enter games.
Invalid order.
PERSISTENCE FAILED
Insufficient items.
User or target does not meet the requirement.
Reached item usage limit.
This invite has expired.
Unable to continue upgrading.
Please bind cashback character first.
The target has been banned. Cannot use this feature.
They are too far away
Currently on interaction invitation cooldown
No mount selected to ride
Deliver
You guys took so long!
Let's go help her!
Here, it's done. Try it on!
The ship is docked!
What is this letter on the floor?
This... this is...?!

Table 5: Partial LP of Action RPG Game in English

match traversal terms and LP terms, so as to obtain the number of matching terms k. The pseudo code is shown in algorithm 2.

Results and analysis. Due to the large manpower requirement for a full collection of game LP, we conduct Text-Coverage statistics for only 2 games in 3 languages. Also Intelligent Traversal is compared with Random Traversal (Randomly click on the detected buttons within the game app), and the experimental results are listed in Table 6. We can see that Intelligent Traversal can effectively support traversal of multilingual games and the Text Coverage of the Intelligent Traversal is 15% more than Random Traversal in all 3 game traversals. Figure 8 shows the Text-Coverage achieved by Intelligent Traversal and Random Traversal on the Action RPG Game in English, Shooter Game in Spanish and Shooter Game in Portuguese respectively. As the traversal proceeds, our Intelligent Traversal (red) is continuously exploring new regions and covering more terms, exceeding the results of the Random Traversal (green) in all 3 languages for both games. Figure 8 also indicates that 24 hours of traversal time is sufficient and that continuing to extend the traversal time will not cover a larger amount of new terms.

Algorithm 2 Get Text-Coverage

```
Input: Deduplicate LP's terms $LPTs$, Deduplicate traversal terms $TTs$
1:  $n \leftarrow$ number of $LPTs$;
2:  obtain $[LPTs_1, LPTs_2, LPTs_3]$ which is divided according to the length of an term in $LPTs$, the length range are: (0,6],(6,15],(15, +∞) separately;
3:  obtain $[TTs_1, TTs_2, TTs_3]$ which is divided according to the length of an term in $TTs$, the length range are: (0,6],(6,15],(15, +∞) separately;
4:  set distance threshold of the first length range $thres_1 = 0.75$;
5:  set distance threshold of the second length range $thres_2 = 0.7$;
6:  set distance threshold of the third length range $thres_3 = 0.6$;
7:  $LD \leftarrow$ Levenshtein Distance [59];
8:  set current number of matching terms $k = 0$;
9:  for $i \leftarrow 0$ to 3 do
10:     for $LPT$ in $LPTs_i$ do
11:         for $TT$ in $TTs_i$ do
12:             $d = LD(LPT, TT)$
13:             if $d \leq thres_i$ then
14:                 $k += 1$
15:                 Break
16:             end if
17:         end for
18:     end for
19: end for
Output: Text-Coverage: $TC = \frac{k}{n}$
```

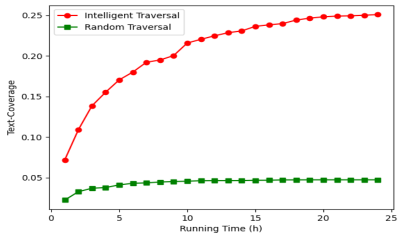
Game	Language	AG^3	Random Traversal
Action RPG Game	English	25.1%	4.7%
Shooter Game	Spanish	42.5%	27.6%
Shooter Game	Portuguese	44.4%	26.8%

Table 6: Text-Coverage

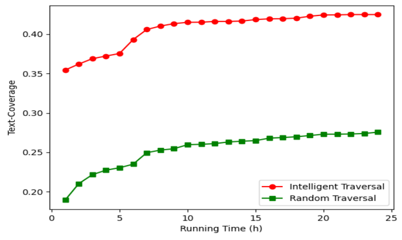
4.3 RQ3. Industry Deployment Study

In industrial game testing, testers are always concerned with the overhead created by the testing method and when deploying AG^3, there are two main concerns: (a) How much additional memory is required to support the test procedures? (b) How is the execution speed affected during runtime?

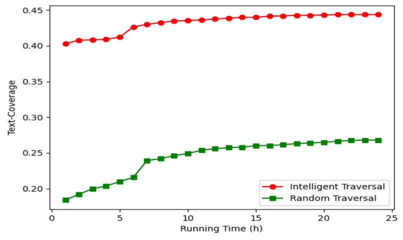
On the game side, there is no memory overhead because AG^3 only relies on screenshots of the game and all computer vision



(a) Action RPG Game in English



(b) Shooter Game in Spanish



(c) Shooter Game in Portuguese

Figure 8: The Text-Coverage Follow Game Running Time.

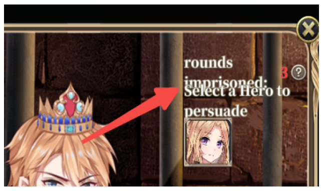
based services are used by an external process. When testing is enabled, execution of the game is affected in two aspects: (1) the transmission of screenshots to the host device, (2) the resource used by processes of data analysis, storage and decision-making. For (1), the average time spent on capturing screenshots and sending screenshot is about 0.2s, and the resolution of the mobile has no effect on the time consumed. For (2), the average time consumption of game scene classification GUI detection is about 0.5s, and the average time consumption of the OCR engine is about 0.6s, and these three intelligent services are processed in parallel, so the overall time consumption of this intelligent decision-making process is approximately 0.6s. Combining the time-consuming operation of the Android terminal and the time-consuming intelligent decision-making, the time interval between adjacent automatic operations of AG^3 is about 1 second, that is, the tool can complete 86,400 operations in one day, which can basically cover the overall traversal of an ordinary game.

In terms of industrial deployment, AG^3 has been serving several game studios within Company A for two years.The total number of games tested so far is 27. The total number of bugs found by the tool and confirmed by the developer teams is 2790. Some of the real bugs found by AG^3 are in 9. For the ease of observation, we have zoomed in the anomaly region. Based on our framework, quality assurance teams have produced 74 multilingual reports of the game. According to the statistics provided by them, it used to take 12 man-days to produce one report without AG^3 but with our tool, only 1 simple testing configuration is required and a report can be automated generated a day later.Through these results, we conclude that AG^3 can be well used in real industrial applications to identify problems and significantly reduce the manpower consumption caused by multilingual game testing.

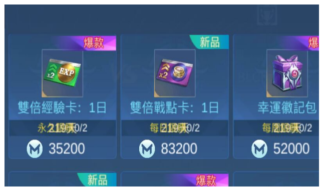
5 DISCUSSION & LESSONS LEARNED

In this section, we discuss lessons learned and practical experience gained from the design, implementation and deployment of AG^3 in the industrial environment.

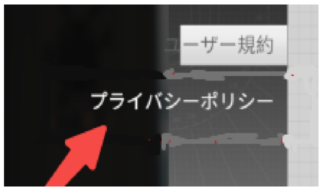
Multilingual game automation tool is important in real production. Our long-term data analysis by the quality assurance team concludes that text glitch issues in multilingual games are more frequent than in monolingual games and typical mobile apps. In real practice, the labor cost consumed in discovering these problems is very large without automated tools. A considerable number of multilingual text glitches can be found and intercepted by AG^3



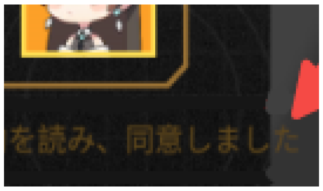
(a) Real Text Overlap Bug



(b) Real Text Overlap Bug



(c) Real Text Overstep Bug



(d) Real Text Overstep Bug

Figure 9: Examples of Real Bug Found by AG^3

before they are introduced to end users and the tool can significantly reduce the manpower in this job. In addition, the vision-only technique makes the automated testing process less costly to get started because it does not require code instrumentation and have no dependency on certain devices and platforms.

Synthetic data is critical in UI glitch detection. The long-tail distribution problem[60] is a very common issue in UI defect detection tasks. The academic community has proposed many approaches to solve the long-tailed machine learning problem. Algorithm-focused approaches include few-shot learning, zero-shot learning and adding some bias to the loss function. There are also solutions start from the perspective of the data, such as resampling, down-sampling, data augmentation, etc. Although these methods have achieved good results in the literature, in practice, it is usually difficult to achieve the desired practical results. From our work, as well as the work of GLIB [12] and OwlEyes [32, 53], we can see that data synthesis is still a very important step in actual production. Moreover, data synthesis requires a deep understanding of the data, which emphasizes the importance of data collection. Only when the synthesized data is close enough to the real anomaly, we can get better detection results.

Understanding the actual application scenario is more important than the optimizing algorithm itself. In industry, understanding real-world application scenarios can greatly reduce the cost of deploying tools and improve the efficiency of industrial deployment. For example, the design of the tool should take into account the actual users and the use environment. AG^3, which starts the automatic traversal of the game with simple configuration, is built into a Python package with minimum dependencies. It can greatly reduce the learning and configuration cost of game testers and facilitates testers to extend AG^3 to their existing test framework.

Algorithm optimization also needs to take into account real business scenarios, such as the design of key indicators and the direction of algorithm optimization. For example, in order to meet the needs of multilingual text glitch detection, we proposed a novel coverage metrics, text coverage to measure the effectiveness of the game traversal strategy. Moreover, our text glitch detection algorithm pays more attention to the recall rate, rather than balancing the precision and recall at the same time. Because in the text glitch detection task, the business requires as many uncovered bugs as possible, for the optimal user experience, and false positives can be easily removed from the test report manually. In this sense, we focused more on improving the recall rate of the algorithm, so that even if the precision rate of our initial version of the detection algorithm was only about 30%, many bugs can be found in time before they are introduced to the players.

6 RELATED WORK

6.1 GUI Testing

Automated testing is critical to the continuous delivery and continuous integration. It reduces testing time and human effort and effectively avoids testing errors caused by testers' inertial thinking. In recent years, research related to automated GUI testing is emerging rapidly [2]. The most important part of GUI testing is the design traversal algorithms. Currently, GUI traversal algorithms are mainly categorized into random, model-based, search-based and machine learning based techniques.

a) Random GUI Testing: Android Monkey [19] is the state-of-the-practice GUI testing tool that randomly performs actions available on the current GUI page. Based on empirical evaluation of open source applications [15] and commercial applications [57], Monkey is able to outperform some model-based tools in terms of coverage achieved and app crashes uncovered.

b) Model-based GUI testing: Model or search-based techniques are among the most popular traversal algorithms at present [27]. Model-based tools use predefined GUI model representations to guide action execution [6, 29]. Instead of using a fixed model, [25] dynamically switches between coarse and fine levels of GUI abstraction. [51] introduces a two-stage method involving construction of a finite-state model of weighted GUI exploration and Gibbs sampling of probabilistic models through iterative mutation.

c) Search-based GUI testing: A Pareto multi-objective approach through genetic evolution is used [36] for search-based testing. Record and Replay [20] by dumping and loading the state of the entire emulator in certain states to improve evolutionary efficiency.

d) Machine Learning based GUI testing: Machine learning based methods can be divided into deep learning and reinforcement learning. A deep learning approach to learning from human interactions is proposed [30], which suffers from heavy model and low efficiency problems. Reinforcement learning techniques are used to promote the effect of testing [1, 56] but not suitable to learn from existing user actions. By combining deep learning and reinforcement learning, [17, 49] explore functionalities that can only be accessed through a specific sequence of actions in Android GUI testing.

Besides these techniques that focus on exploring standard apps, [31, 45, 46] are based on reinforcement learning and imitation learning for playing against specific objects in the game. As for intelligent traversal for general games, there is no directly related work, to the best of our knowledge.

6.2 GUI Glitch Detection

GUI search [5, 7, 8, 10, 11, 48, 61] and GUI code generation [3, 9, 13, 14, 40] are studied in the literature. For example, [41] checks whether the GUI displayed violates the design by image similarity calculation based on computer vision, and its following work [44] further detects and summarizes GUI changes in evolving apps.

In terms of specific GUI issue detection, some work focuses on GUI rendering delay [22] and image loading [28]. GUI glitch detection algorithms based on a deep learning model are also proposed [12, 32, 53]. While they all mention text glitch detection support, they do not design model specifically for the characteristics of text glitch in game screenshot, but just uniformly use a CNN binary classification model to detect multiple types of GUI anomalous images.

7 CONCLUSION

In this paper, we have presented AG^3, an automated testing tool for text glitches in multilingual games that includes game traversal, text glitch detection and report generation. As an image-based game testing tool, AG^3 does not rely on intrusive code instrumentation but is shipped in a ready-to-use testing tool with minimal configuration required. The effectiveness of AG^3 is demonstrated through experiments and daily industrial use. In sum, this paper makes the following contributions:

Game traversal We designed an intelligent traversal strategy, with more than 15% test coverage compared with the baseline.

Text glitch detection A novel test glitch detection algorithm based on deep learning is proposed after analyzing characteristics of text glitches in games. The experiment shows that AG^3 is able to achieve 90% recall rate which means that it can spot most of real-world multilingual text glitch issues in games. Through error and precision rate, we conclude that suspected problems can be filtered out from the large number of original screenshots to a great extent, minimizing the workload of test engineers.

Practical application As of now, AG^3 has been integrated into internal games. A total of 74 multilingual games have been tested by AG^3 and over 2,790 confirmed bugs are found. Meanwhile, the labor time consumption to produce one report was reduced from 12 man-days to 1 man-days.

ACKNOWLEDGMENTS

We would like to thank all the Reviewers for taking the time and effort necessary to review the paper. We sincerely appreciate all valuable comments and suggestions, which helped us to improve the quality of the manuscript.

REFERENCES

[1] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce. 2018. Reinforcement learning for android gui testing. In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2–8.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. 2012. Using GUI ripping for automated testing of android applications. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 258–261.

[3] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui. In *30th IEEE/ACM International Conference in Automated Software Engineering (ASE)*. 248–259.

[4] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.

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

[6] N. P. Borges, J. Hotzkow, and A. Zeller. 2018. Droidmate-2: a platform for android test generation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 916–919.

[7] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, Shengdong Zhao, and Linda Liu. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. In *the ACM on Human-Computer Interaction, Volume. 4, No. CSCW.*

[8] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, and Shengdong Zhao. 2019. Gallery DC: Design Search and Knowledge Discovery through Auto-created GUI Component Gallery. In *ACM on Human-Computer Interaction 3, CSCW (2019)*. 1–22.

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

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

[11] 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[12] Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. 2021. 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*. 1093–1104.

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

[14] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. In *IEEE Transactions on Dependable and Secure Computing (2019)*. IEEE.

[15] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated test input generation for android: Are we there yet?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.

[16] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2017. Scripted gui testing of android apps: A study on diffusion, evolution and fragility. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–32.

[17] F. Y. B. Daragh and S. Malek. 2021. Deep gui: Black-box gui input generation with deep learning. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 905–916.

[18] Giovanni Denaro, Luca Guglielmo, Leonardo Mariani, and Oliviero Riganelli. 2019. GUI testing in production: challenges and opportunities. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. 1–3.

[19] Google Developers. 2023. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey.

[20] Z. Dong, M. Bohme, L. Cojocaru, and A. Roychoudhury. 2020. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 481–492.

[21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[22] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. 2017. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In *2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[23] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2021. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430* (2021).

[24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

[25] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su. 2019. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

[26] Bernard J Jansen. 1998. The graphical user interface. *ACM SIGCHI Bulletin* 30, 2 (1998), 22–26.

[27] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein. 2018. Automated testing of android apps: A systematic literature review. In *IEEE Transactions on Reliability*. IEEE, 45–66.

[28] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lu. 2019. Characterizing and Detecting Inefficient Image Displaying Issues in Android Apps. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 355–365.

[29] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[30] Y. Li, Z. Yang, Y. Guo, and X. Chen. 2019. Humanoid: A deep learningbased approach to automated black-box android app testing. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[31] Guoqing Liu, Mengzhang Cai, Li Zhao, Tao Qin, Adrian Brown, Jimmy Bischoff, and Tie-Yan Liu. 2022. Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation. In *2022 IEEE Conference on Games (CoG)*. IEEE, 237–244.

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

[33] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*.

[34] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated generation of reproducible test cases for android apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. 99–104.

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[36] K. Mao, MG. Harman, and Y. Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *25th International Symposium on Software Testing and Analysis*. 94–105.

[37] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*. 280–290.

[38] 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)*. IEEE, 1479–1480.

[39] 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)*. IEEE, 559–570.

[40] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *arXiv preprint arXiv:1802.02312*.

[41] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *40th International Conference on Software Engineering*. 165–175.

[42] 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)*. IEEE, 15–18.

[43] 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)*. IEEE, 505–506.

[44] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. 2018. Detecting and Summarizing GUI Changes in Evolving Mobile Apps. *arXiv preprint arXiv:1807.09440*.

[45] Alfredo Nantes, Ross Brown, and Frederic Maire. 2008. A framework for the semi-automatic testing of video games. In *Proceedings of the AAAI Conference on*

Artificial Intelligence and Interactive Digital Entertainment, Vol. 4. 197–202.

[46] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2021. Automated game testing using computer vision methods. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 65–72.

[47] Chao Peng, Zhao Zhang, Zhengwei Lv, and Ping Yang. 2022. MUBot: Learning to Test Large-Scale Commercial Android Apps like a Human. In *38th International Conference on Software Maintenance and Evolution (ICSME 2022)*.

[48] Steven PReiss, Yun Miao, and Qi Xin. 2018. Seeking the User Interface. In *Automated Software Engineering*. 157–193.

[49] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. 2021. Deep reinforcement learning for black-box testing of android apps, In 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). *arXiv preprint arXiv:2101.02636*.

[50] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.

[51] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Yao Y., G. Pu, and Y. Liu. 2017. Guided, stochastic model-based gui testing of android apps. In *2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[52] 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.

[53] Yuhui Su, Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2021. OwlEyes-online: a fully automated platform for detecting and localizing UI display issues. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1500–1504.

[54] TestBird. 2023. Mobile Internet Testing Expert. https://www.testbird.com/.

[55] Torchvision. 2023. The pre-training model weights of VGG16. https://download.pytorch.org/models/vgg16-397923af.pth.

[56] T. A. T. Vuong, S. Takada, S. Koppula, and R. Bryce. 2018. A reinforcement learning based approach to automated testing of android applications. In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 31–37.

[57] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie. 2018. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 738–748.

[58] Wikipedia. 2023. Cosine similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

[59] Wikipedia. 2023. Levenshtein Distance. https://en.wikipedia.org/wiki/Levenshtein_distance.

[60] Wikipedia. 2023. Long tail. https://en.wikipedia.org/wiki/Long_tail.

[61] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: vision-based workflow action recognition from programming screencasts. In *41st International Conference on Software Engineering (ICSE)*. IEEE, 350–361.

Received 2023-05-18; accepted 2023-07-31