# Facilitating Monkey Test by Detecting Operable Regions in Rendered GUI of Mobile Game Apps

Chenglong Sun
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
suncl@ios.ac.cn

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

Bo Jiang
School of Computer Science and Engineering
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

W. K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Graphical User Interface (GUI) is a component of many software applications. Many mobile game applications in particular have to provide excellent user experiences using graphical engines to render GUI screens. On a rendered GUI screen such as a treasury map, no GUI widget is embodied in it and the operable GUI regions, each of which is a region that triggers actions when certain events acting on these regions, may only be implicitly determinable. Traditional testing tools like *monkey test* do not effectively generate effective event sequences over such operable GUI regions. Our insight is that operable regions in a rendered GUI screen of many mobile game applications are given with visible hints to catch user attentions.**

**In this paper, we propose Smart Monkey, which uses the fundamental features of a screen, including color, intensity, and texture, as visual signals to detect operable GUI region candidates, and iteratively identifies and confirms the real operable GUI regions by launching GUI events to the region. We have implemented Smart Monkey as a testing tool for Android apps and conducted case studies on real-world applications to compare it with a peer technique. The empirical results show that it effective in identifying such operable regions and thus able to generate functional event sequences more efficiently.**

***Keywords— game testing; rendered GUI, monkey test***

## I. INTRODUCTION

Software programs are widely used in our digital living. To present good usage experience to users, many software developers use Graphical User Interface (GUI [26]) in their projects. GUI is one of the most important bridge between users and software applications. However, bugs existing in GUI-based programs present more and more serious problems in the real world. For example, after experiencing a few occurrences of flash back or black screen when using a mobile application, a user may choose to uninstall the problematic application.

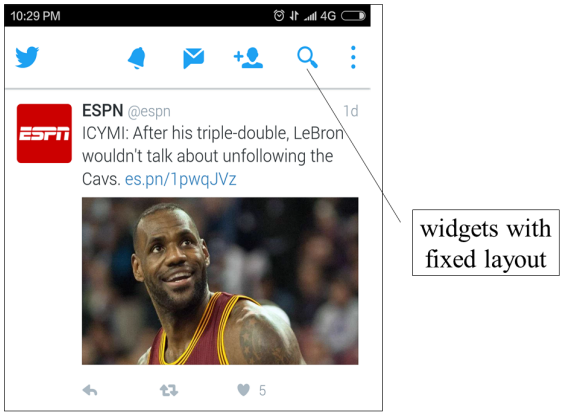
GUI testing is an integral part of many software development projects. There are a large number of testing tools and frameworks such as Ascentialtest [13], Sikuli [33], Appium [27], MonkeyRunner [3], UIAutomator [5], Robotium [14], and monkey [4] for various platforms and development environments.

On one hand, some testing tools are specifically designed to be used by professional programmers and test engineers. For example, automatic testing tools like MonkeyRunner and Robotium launch GUI events on GUI widgets and depend on composed test scripts to simulate user behavior [4][14]. To use these tools, mastering the corresponding script languages like Python [25] or testing frameworks like JUnit [18] are pre-requisites, making them not accessible to less skillful programmers or engineers. On the other hand, many applications (like games) use graphic engines like OpenGL [19] and others [12][29] to render images and use these images as the GUI in the background [19]. Many testing tools do not effectively handle such rendered GUIs because no widget information is correlated with them. Further, composing test scripts by developers can be difficult because the regions of a "background" that can receive GUI events, which we refer such a region to as an ***operable region***, may not be statically determinable.

Monkey test [4] is an easy-to-use testing strategy, which is able to operate a software application with rendered GUI. However, existing monkey test tools may only hit an operable region over a rendered GUI with a low probability, or may be inefficient in revealing software bugs with limited "blind trials" [8]. We thus ask a research question: Is it possible to detect operable regions in a rendered GUI and thus make a blind monkey test *smarter*?

Our insight is based on the following observation. A basic design principle is that a GUI may give user visible hints of operable regions to draw the attention of users. In this paper, we formulate our technique on top of this insight to detect operable regions in a rendered GUI. Inspired by the psychological studies that human perception system is sensitive to the contrast of visual signals, such as color, intensity and texture, we propose to detect the salient regions of a rendered GUI through color, intensity, and texture to infer ***operation candidates***. To validate our technique, we implemented it as a tool "*Smart Monkey*" on the Android framework to automate GUI testing for Android apps. We conducted case studies using real-world applications to evaluate the feasibility of our technique, compare it to peer techniques to assess its effectiveness, and evaluate its efficiency. The experiment on nine real-world

**Figure 1. Monkey test applied on simple GUI (the Twitter app)**




widely used mobile applications show that our technique is more effective than Monkey of Android by 33% in correctly hitting the operable regions on screens of these mobile applications and can expose the first crashing fault from applications earlier.

The contributions of this work is threefold. First, this is the *first* work targeting at handling rendered GUIs to detect operable region candidates, confirm them through concrete GUI events, and generate test scripts accordingly. Second, we show the feasibility of our technique by implementing it as a testing tool for Android app testing and share it on GitHub[1]. Third, we report an experiment, which shows our technique feasible to detect operable region candidates in a rendered GUI, confirms these candidates by concrete GUI events, effectively boosting the probability of generating a sequence of events that can trigger the GUIs to take actions.

The organization of this paper is as follows. The motivation of this work is presented in Section II. Our technique is presented in Section III. The case studies that validate our work are given in Section IV. Related works are reviewed in Section V. Section VI concludes the paper.

## II. MOTIVATION

Many software applications developed in recent years have graphical user interfaces (GUIs). One of the easiest way for a user to interact with such software applications is through the provided GUIs.

### A. Monkey Test

Figure 1 shows the GUI of a well-known app Twitter. In the left plot (the app Twitter), buttons for general actions like "Home", "Alert", "Email", "Search", and so on are always placed at the top of the GUI. At the bottom of each post, the "Forward", "Friend Link", and "Like!" buttons can be found.

The record-and-replay [14] is a popular strategy for testing such an app, which has a traditional Android GUI layout. For example, Figure 2 lists out a test script to operate the GUI widgets in Figure 1. A button named "Add note" is located in the GUI and triggered. Then that the script checks whether a follow-up window page named "NoteEditor" is opened. After that, it inserts two notes "The First Test Case" and "The Second Test Case" in the editing widget. Finally, it surfs the "NoteList" page and validates whether the two insertions have been successfully completed.

### B. The Problem

Figure 3 shows a car racing scenario of the game *asphalt8*. To give a player a good playing experience, the GUI is not designed in a conventional GUI architecture as seen in Figure 1. Instead, the whole image is rendered using a graphics engine and is periodically refreshed. There are buttons at fixed positions in the GUI, such as the "Pause" button at the left-top corner. There are also dynamic operable objects, such as the "gas" bonus in the center of the screen. It only appears sometimes, and may be shown in different positons on the GUI at different moments. Furthermore, the latter elements can be more important and interactive for a game application.

Operable objects, whose appearances, numbers, positions, and other properties cannot be statically known or

**Figure 2. The record-and-replay test script for a static GUI**

```
    public void testAddNote() throws Exception
1   {
2       solo.clickOnMenuItem("Add note");
3
4       // Assert that NoteEditor activity is opened
5       solo.assertCurrentActivity("Expected NoteEditor activity", "NoteEditor");
6
7       // In text field 0, add Note 1
8       solo.enterText(0, "The first Test case");
9       solo.goBack();
10
11      // Clicks on menu item
12      solo.clickOnMenuItem("Add note");
13      // In text field 0, add Note 2
14      solo.enterText(0, "The Second Test case");
15
16      // Go back to first activity named "NotesList"
17      solo.goBackToActivity("NotesList");
18
19      // Assert that Note 1 & Note 2 are found
20      assert(solo.searchText("The first Test case"));
21      assert(solo.searchText("The Second Test case"));
22  }
```

[1] https://github.com/sunchenglong/smartmonkey

**Figure 3. Dynamic operable objects in the GUI of game asphalt8**


dynamically queried, are difficult to handle in a record-and-replay test by existing tools. Monkey is an Android stress testing tool that can automatically and randomly operate an Android application to test it against some generic oracles (e.g., crash). However, it has no knowledge about the locations of operable regions of the game under test, which may limit its effectiveness. Furthermore, there are also enhancement versions of Monkeys such as PUMA [15], which uses UI-Automator to query the GUI widget structure at runtime. However, it cannot query the rendered GUI information through UI-Automator (or DUMP for earlier versions of Android). As a result, mobile game applications that heavily use rendered GUIs cannot be handled gracefully by existing automated testing tools.

### C. *Our Insight*

There is a popular design principle that operable objects in a GUI screen should be attractive to the visions of human players so as to catch players' attentions more readily. For an ordinary human player watching a video sequence on such an application applying this design principle, the player may be attracted not only by the interesting events but also by the interesting objects in still images. This is referred as the spatial attention [34]. Moreover, based on the psychological studies, the human perception system is sensitive to the contrast of visual signals, such as color, intensity and texture. Based on the above insights, we propose to capture such signals to facilitate the identification of dynamic operable objects in a rendered GUI.

## III. OUR TECHNIQUE – SMART MONKEY

In this section, we first introduce the architecture of our technique "Smart Monkey", then explain the algorithm, and discuss its time complexity.

### A. *Architecture*

Our technique "Smart Monkey" is designed to test software applications, especially game apps, which have rendered GUIs (still images with regions on the images that actions are attached when some events are triggered). To do that, we propose to identify salient regions in a GUI using a computer vision approach as operable region candidates, and confirm these candidates as real operable regions by generating monkey tests to send GUI events to identified operable GUI widgets.

Figure 4 is the blueprint of our technique. Our technique performs the same randomized and iterative GUI state exploration of the application under test to generate event sequences. In each iteration, it first uses *adb* or the *http* protocol to get the screenshot (as well as the class name of current Activity) of the application under test from a mobile phone to the component of our technique running on a host computer. Then, if the operable regions (or region candidates) of a GUI state have been recognized in a previous iteration, our technique generates an event based on previous indexed recognition results. If a new GUI state is encountered during the current iteration, our technique invokes a region detection algorithm to detect operable region and store the recognition results by indexing the GUI state.

Note that the GUI state (current window) is represented by the current active activity class name, which is a simple string that can be queried with adb interface. In short, the Monkey tool randomly selects a point to send event during state exploration; and different from the Monkey tool, our technique first randomly selects a not-yet-selected operable region among the detected operable regions for the current GUI state, then it sends an event of a particular type (e.g., among touch, drag, or click) to a random coordinate within that operable region. Then the application will transit to the next state (which can be the same as the current one), and the Smart Monkey just repeats the event generation process.

We have implemented our technique on the Android framework and built the whole project using maven [28]. All the algorithms are run on a remote site (a computer).

**Figure 4. The schematic diagram**

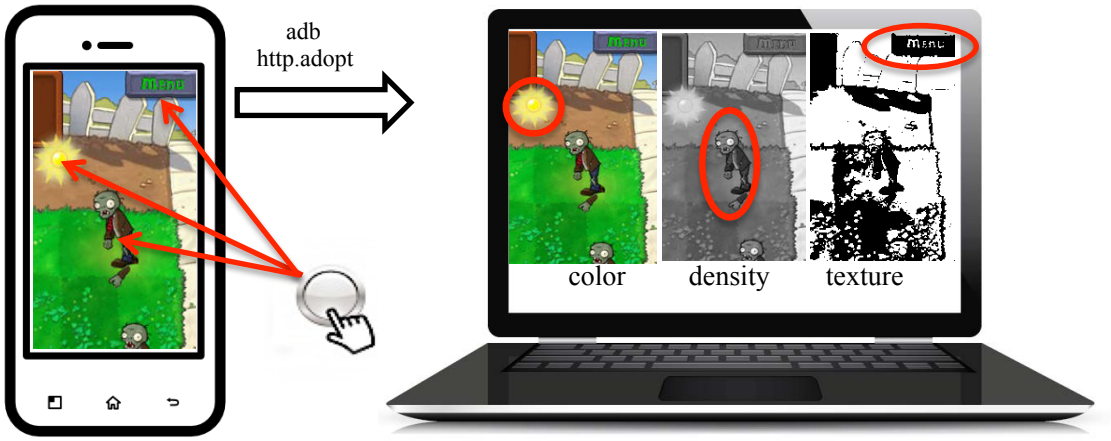


TABLE I. Visual features and their detection algorithms

| | | Saliency detection algorithms | | |
|---|---|---|---|---|
| | | ***GA*** [34] | ***CA*** [10] | ***SR*** [16] |
| visual features | density | √ | √ | |
| | color | | √ | |
| | texture | | | √ |

Unlike Lin et al. [20] that used camera to capture the images on android phones, our tool only uses the *adb* and the http protocol [3] to transfer image data.

### B. Visual Features

We choose three kinds of fundamental features, namely color, density, and texture, as visual signals to detect operable region candidates. They are listed in Table I.

Pixel space based saliency detection algorithms can be for gray spatial attention model (aka. the GA algorithm [34]) and color spatial attention model (aka. the CA algorithm [10]), and other models (e.g., the SA algorithm [16]). For instance, SR can be used to find saliency region in terms of texture but runs inefficiently; and CA can be used to find saliency regions in terms of color, and thus can help to find regions with different density. Our technique apply all these three saliency detection algorithms, namely GA [34], CA [10], and SR [16], to detect the high contrast regions in terms of the three visual features in a GUI.

### C. Saliency Detection Algorithm

Detecting the operable regions from a still image is the goal of saliency detection, and our algorithm SliencyDetect for this purpose is listed in Figure 5, which is implemented in our tool using the OpenCV framework and in Java.

According to the spatial attention model, the saliency map of an image is built upon the color contrast between image pixels. The saliency value of a pixel $I_k$ in an image I is defined as equation (1),

$$S(I_k) = \sum_{\forall I_i \in I} D(I_k, I_i) \quad (1)$$

This equation is expanded to have the following form,

$$S(I_k) = D(I_k, I_1) + D(I_k, I_2) + \cdots + D(I_k, I_N)$$
$$= \sum_{j=1}^{n} f_j D(c_l, c_i) \quad (2)$$

Let $I_k = c_l$, and equation (1) is further restructured, such that the terms with the same Ii are rearranged to be together, where $f_j$ is the frequency of pixel value an in the image. We calculate the frequency of the pixel value and store them in a map (lines 06 to 14), in order to decrease the complexity of the algorithm. In this part, the input is an image, gray or color are all available, and output is a saliency image. A saliency is a distance map, is the pixel value on the whole image gray or color value histogram weighted distance. The histogram calculates in lines 06 and 07. The two loops in lines 08 to 10 are calculating the weighted map, then, in lines 11 to 13, we can use this map to get the each map saliency value, decrease the loop level.

For a saliency image, we also need a function to choose the most operable points (lines 15 to 19), we use a random strategy. We have used a threshold to filter the points in saliency image to increase the hit ratio in line 17.

The calculation of GA (lines 20 to 24) computes the distance between pixel and every pixel gray level as the saliency map each gray value. The calculation of CA (lines 25 to 28) uses the color value in $L{\times}a{\times}b$ color space as the

Figure 5. The pseudo-code of Saliency algorithm

```
function SaliDtc (method)
      @INPUT:  x[] - Image
      @OUTPUT: y[] - Points

01  switch method:
02    case GA: y ← call GA (x)
03    case CA: y ← call CA (x)
04    case SR: y ← call SR (x)
05  return y
```

```
function GenSaliTbl (x)
  @INPUT:  x[] — Image
  @OUTPUT: s[] - Saliency Image

06 for r, c in [1, nRow], [1, nCol]:
07   Hist[x[r][c]]++
08 for outlevel in Hist:
09   for inlevel in Hist:
10     Map[outlevel] ← d[r][c]+
         Hist[level]*call
             dist(outlevel,inlevel)
11 for r, c in [1, nRow], [1, nCol]:
12   for level in Hist
13     s[r][c] ← Map(x[r][c])
14 return s
```

```
function SaliCan (s)
  @INPUT:  s - Saliency Image
           THRESHOLD
  @OUTPUT: y[] - Points

15  while  y[] is not full
16     r, c ← ran(1, nRow), ran(1, nCol)
17     if s[r][c] < THRESHOLD
18        y ← call cancat(y, d)
19  return y
```

```
function GA
  @INPUT:  x[] - Image
  @OUTPUT: y[] — Points

20  nRow, nCol ← size of x
21  g ← call gray(x)
22  s ← call GenSaliTbl (g)
23  y ← call SaliCan (s)
24  return y
```

```
function CA
  @INPUT:  x[] - Image
  @OUTPUT: y[] — Points

25  l ← call Lab (x)
26  s ← call GenSaliTbl (l)
27  y ← call SaliCan (s)
28  return y
```

```
function SR
  @INPUT:  x[] - Image
  @OUTPUT: y[] - Points

29  g ← call gausian(x)
30  x' ← call fft(x)
31  x'' ← x' — g
32  s ← call ifft (x'')
33  y ← call SaliCan(s)
34  return y
```

value unit, and computation is like LC algorithm, the saliency map is conducted by the distance between every pixel value.

The difference is the color information to choose, which is the gray-scale transformation (line 21) in GA algorithm, and the $L \times a \times b$ transform (line 25) in CA algorithm. The rest step are all the same, using saliency map function first (lines 22 and 26), then use the operable points predict function (lines 23 and 27).

The calculation of SR (lines 29 to 34) is calculating of on frequency domain. The input and output are all the same to the algorithms GA and CA, the unique step is the saliency image generating.

First, we calculate the image Fourier spectrum (lines 30). Next, we calculate the saliency spectrum image via the source spectrum image minuses the Gaussian noise (lines 31). Finally, we calculate the image in pixel domain via inverse fast Fourier transformation [16] (lines 32).

### D. Time Complexity

The time complexity of GA, CA and SR algorithms are all $O(NXY)$ , where $N$ is the level number of histogram, and $X$ and $Y$ are the image height and width, respectively. For a gray image (GA), the maximum of $N$ is 25. For a color image (CA), it is 256×256×256; and it is 1 in the SR algorithm. The time complexity of getting the operable points from a saliency map is $O(K)$ (lines 15 to 19), where $K$ is the number of operable points. The complexity of the algorithm shown in Figure 5 is therefore $O(NXYK)$.

## IV. Experiment

In this section, we first explain our research questions, then introduce our experimental design, and finally show the results with some case studies.

### A. Research Questions

Smart Monkey is a technique closely similar to monkey, but is a smart one: it can "see" the rendered image on the screen, and select the ones most similar to a button or tag to click or drag. The computer vision algorithms were run on a server. The testing scripts only need the widget name and snapshot information. Therefore, a non-professional can easily write test cases.

For the smart monkey part, we mainly focus on the amount of valid operations generated per minute, and for auto testing based image script part, we also want to assess the test script execution time and the results.

**RQ1**: Is our technique effective in identifying operable widgets in mobile game screenshots?

**RQ2**: Is our technique more effective for testing mobile game than the built-in monkey of Android?

### B. Design and Environment of the Experiment

We compare the effectiveness of our technique to the built-in monkey tool of Android on a suite of representative real-world applications.

The subject programs are listed in Table II. Every row is an application, and the column *type* is the application type such as social network, game and so on. The third column is the application *version*; the last column is the application *source and the size*. We have chosen nine applications from Google play and Samsung play, which include three kinds of applications: social network, game and shopping. The minimal size of application is 6.68Mb and the maximal size is 92.48Mb. These applications are all popular applications with large user base. Note that we include both game apps having rendered GUI and game apps of widget structure for a comparison.

We have used three mobile devices to run the subject programs, which are Samsung GALAXY J7 (Android OS 5.1), HTC X920e (Android OS 5.1), and XiaoMi 4s (MIUI 7). A host computer is also configured to execute our computer vision component to identify operable regions and events generated to be sent to these mobile device. The host computer is a Mac mini (OS X Yosemite 10.10.5), which contains a 2.6 GHz Intel Core i5 CPU, and 16GB RAM. JDK, OpenCV and Android SDK are installed on the host as our implementation rely on these frameworks. The screen capture used the standard Android screen capture interface. We have used the *adb* tool to transfer the screenshot and the commands between the monkey part and the computer vision part of our technique.

We have designed two experiments as follows.

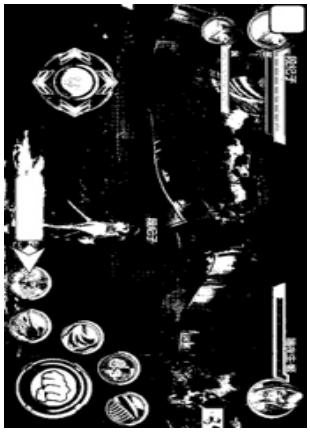
In our first experiment, to examine the effectiveness of Smart Monkey, we compare it to the build-in monkey of Android. For each technique, we have run nine applications on three devices (the applications detail is in Table II) for 5 times each to calculate the mean number of effective operations (e.g., click, tap) within a 3-minute duration. We further want to compare the relative effectiveness of the three saliency detection algorithms. Similar to the first experiment, we have run the same nine applications on the same three devices for 5 times each to calculate the mean number of effective operations.

**Table II. The examine applications**

| | *Type* | *Version* | *Source* | *Size* |
|---|---|---|---|---|
| Twitter | Social Net | 5.96.00 | Google play | 62.39Mb |
| Instagram | Social Net | 7.17.0 | Google play | 42.00Mb |
| Flickr | Social Net | 4.0.7 | Google play | 17.20Mb |
| Wechat | Social Net | 6.3.15 | SamsungMarket | 34.70Mb |
| GuitarTune | Game | 3.0.3 | Google play | 29.16Mb |
| Deadlyracing | Game | 1.0 | Google play | 46.45Mb |
| Temple Run | Game | 1.6.14 | SamsungMarket | 92.48Mb |
| GameDev | Game | 1.0.6 | GooglePlay | 6.68Mb |
| Jingdong | Shopping | 5.0.0 | SamsungMarket | 77.32Mb |

**Figure 6. Some saliency detect in several android applications. The a, c, e, g pictures all are the source image and the green circles are the saliency point, the b, d, f, h pictures all are the binary saliency map.**

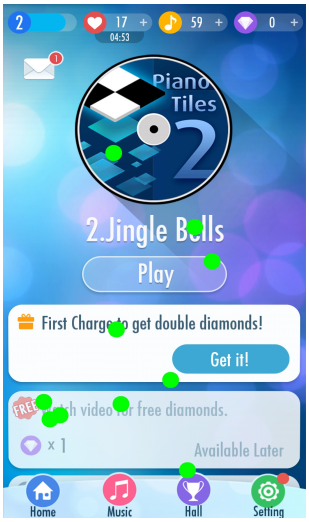
(a) Screen shot of *Final Fight* fighting scene



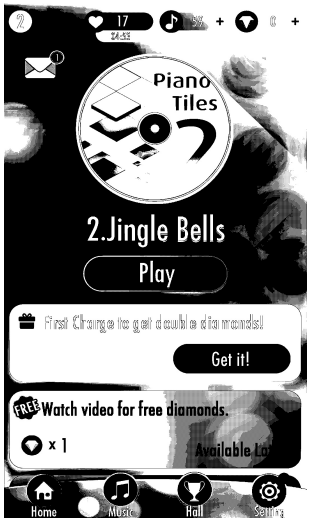
(b) Saliency image of *Final Fight* fighting scene

(c) Screen shot of *Final Fight* choose hero scene

(d) Saliency image of *Final Fight* choose hero scene

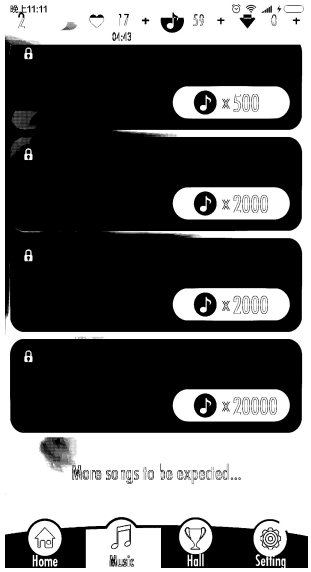



(e) Screen shot of *Piano Tiles* start scene




(f) Saliency image of *Piano Tiles* start scene


(g) Screen shot of *Piano Tiles* music choose scene




(h) Saliency image of *Piano Tiles* music choose scene

In the second experiment, we want to compare the testing effectiveness of our tool and the built-in Monkey tool to test several game applications. We record the time to the first occurrence of crash using each tool.

### C. *Result and study cases*

In this section, we list out the experiment results together with the observations.

#### 1) *Answering RQ1*

Figure 6 shows four examples. The plots (a), (c), (e), (g) are the source images, and he plots (b), (d), (f), (h) are the saliency map binary images. In each plot, there are green points. They represent the saliency detection results. Let us take the first one for example to illustrate.

In Figure 6, plot (a) is a screenshot of an Android game app *Final Fight*, and the screen is horizontal. Plot (b) is the saliency image of plot (a). After processing, six regions are marked as saliency regions. Then we randomly place a green dot[1] within each region in plot (a). Plots (c) and (d) are another test on the same app. In plot (c), ten saliency regions are recognized. The plots (e) to (h) are from the American mobile game *Piano Tiles 2* and the screen is vertical. We notice that the principal color has changed from (e) to (g), and the saliency images are accordingly inversed. However, the operable regions do not change much. For instance, the left bottom *Home* button and *Hall* button are always recognized. It shows that the saliency detection method is robust. All of the computation costs less than 800ms. We deem the performance acceptable for an interactive system.

We also observe that not all the recognized saliency regions reflect realistic operable regions. For example, in the test of plot (a), three of the six dots valid input regions. In plot (c), more saliency regions are recognized, but some of them are invalid (false positives).

[1] We apologize that the green dots may be not easy to see in a gray-scale printing.

TABLE III. THE COVERAGE RATIO ON APPLICATIONS

| | *Image Num* | *Total Operable Objects* | *Hit Num* | *Hit Num Uniq* | *Hit Ratio* |
|---|---|---|---|---|---|
| Twitter | 3 | 39 | 23.67 | 10.67 | 27.35% |
| Instagram | 3 | 27 | 21.00 | 9.67 | 35.80% |
| Flickr | 4 | 25 | 23.00 | 9.67 | 38.67% |
| Wechat | 4 | 49 | 27.33 | 17.67 | 36.06% |
| GuitarTune | 5 | 55 | 28.33 | 18.33 | 33.33% |
| Deadlyracing | 5 | 23 | 14.67 | 12.67 | 55.07% |
| Temple Run | 3 | 22 | 17.33 | 4.67 | 21.21% |
| GameDev | 4 | 32 | 8.00 | 7.67 | 23.96% |
| Jingdong | 5 | 103 | 35.33 | 26.33 | 25.57% |

We further measure the percentage of effective hits for each application, as summarized in Table III. The first column is the name of application under test, the second column is the number of images for each application, and the third column is the total number of operable objects in each application. The fourth column and fifth columns are the average number of hits and unique hits by operable position in three times. The last column is the hit ratio, which is calculated as total number of operable objects divided by the unique hit number, which means that the third column divided by fifth column.

For instance, the first row is the GA algorithm of our tool running on the Twitter 3 screenshots. There are 39 operable objects in total; and our tool identifies 33 operations. On average 23.67 of them hit operable objects, and 10.67 operations are unique. So, the hit ratio is 27.35%.

Our tool attains the best results on *Deadlyracing* with a hit ratio of 55.07%, while the worst results on *Temple* with a hit ratio 21.21%. The average hit ratio in these nine applications is 33.00%. Generally, the result shows that our tool is effective.

We further compare with the Monkey tool to know the relative effectiveness of our tool. The results are shown in Table IV. Every row corresponds to an application, and the first column is application name, and the second and third columns are the number of valid operations of our tool and Monkey, respectively, the last column is the incremental ratio of our tool compared with Monkey. In this table, the number of valid operations is calculated during three minutes by running our Smart Monkey and monkey tools. The table is easy to understand that the higher the number, the more effective the tool is.

TABLE IV. THE VALID OPERATIONS IN 3 MINUTES

| | *Smart Monkey* | *monkey* | *Incremental Ratio* |
|---|---|---|---|
| Twitter | 12.4 | 8.2 | 51.22% |
| Instagram | 14.0 | 9.6 | 45.83% |
| Flickr | 16.0 | 14.0 | 14.28% |
| Wechat | 11.2 | 5.6 | 100.00% |
| GuitarTune | 14.6 | 8.2 | 78.04% |
| Deadlyracing | 12.6 | 10.8 | 16.67% |
| Temple Run | 11.6 | 11.2 | 3.57% |
| GameDev | 10.2 | 4.8 | 112.50% |
| Jingdong | 15.8 | 6.4 | 146.88% |

For instance, in the first row on *Twitter*, our tool performs 12.4 valid operations within 3 minutes. Meanwhile, the result of monkey is 8.2. Our tool performs better than Monkey on this application with the increase ratio 51.22%. The other rows can be interpreted similarly. From Table IV, we observe that our tool performs best on the application *Jingdong*, the worst on *Temple Run*. The increase ratios are all positive with the mean value 63.22%.

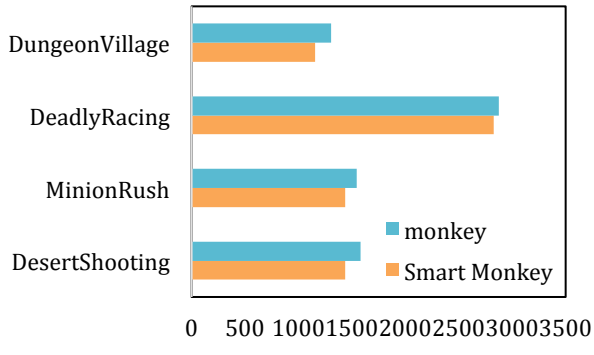
Thus, we can answer *RQ1* that our technique is effective to identifying operable widgets from the rendered images of mobile game screenshots.

*2) Answering RQ2*

We have run several applications via our tool and Monkey on HTC X920e, to detect faults in practice. In order to test our tool in terms of fault exposing capability, we have designed an experiment to compare the time to expose the first fault our tool to monkey. The result is in Figure 7. The horizontal axis is the time to first fault (e.g., crash or exception in our context) in seconds of an application running on the real device with either tool, which is calculated as the mean of five runs. For each application, the lower (in orange) bar is our tool and the upper (in blue) one is the monkey. Out of three of the four applications, i.e., *DungeonVillage*, *MinionRun*, and *DesertShooting*, our tool finds a crashing bug quickly, 8.00% mean increase ratio compared to that of Monkey. On *DeadlyRacing*, our tool is 1.80% higher than Monkey.

Thus, we can answer *RQ2* that our technique is more effective to expose a crashing (vulnerability) fault than Monkey for testing mobile game applications with rendered GUIs.

**Figure 7. The crash time (unit: second) of application running on our Smart Monkey and monkey seperately.**



Threats to validity of the results observed in the experiment may include different performance measurement and use of external subject programs.

## V. Related Work

### A. GUI Testing

Test case generation is one topic of GUI testing. Amalfitano et al. [1] developed a technique based on a crawler that automatically builds model for the application GUI and obtains test cases that can be automatically executed. Memon et al. [23] presented a technique to automatically generate test cases for GUI systems. Their key idea is that the test designer is likely to have a good idea of the possible goals of a GUI user and it is simpler and more effective to specify these goals than to specify sequences of events that the user might employ to achieve them.

In another work [22], they described "GUI Ripping", a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The extracted information is then verified by the test designer and used to generate test cases automatically. Huang et al. [17] developed a method to repair automatically GUI test suites feasibly, using a genetic algorithm to evolve new test cases that increase our test suite's coverage while avoiding infeasible sequences.

Meanwhile, many researchers focus on simplifying the test cases and abstract the testing. Vieira et al. [30] offered two ways to manage the number of tests. First, custom annotations and guards use the Category-Partition data that allows the designer tight control over possible, or impossible, paths. Second, automation allows different configurations for both the data and the graph coverage. Brooks et al. [8] used "usage profiles" to develop a probabilistic usage model of the application, which is used to ensure that a new version of the application will function correctly. Rauf et al. [26] have presented a GUI testing and coverage analysis technique centered on genetic algorithms, then exploited the event driven nature of GUI.

Computer vision is a field that includes methods for acquiring, processing, analyzing, and understanding images and, in general, high-dimensional data from the real world in order to produce numerical or symbolic information, e.g., in the forms of decisions [31]. Different from the above conventional approaches, Chang et al. [9] tried to use computer vision techniques to aid GUI testing. They implemented a testing tool named Sikuli for desktop applications. Sikuli scripts describe target screenshot and the IDE can identify the graphic elements on the desktop and operate them accordingly.

### B. Mobile Testing

Mobile devices and mobile apps are very popular. As a result, mobile testing becomes more and more important. Appium [27] and UIAutomator [5] are open source test automation frameworks for use with native, hybrid and mobile web apps. Robotium [14] and MonkeyRunner [3] can handle android applications only.

At the same time, there are many researches focusing on test case generation via the widgets and component. N. Mirzaei et al. [24] used symbolic execution methods to implement android testing. T. Azim et al. [6] developed a strategy named Depth-first Exploration that mimics user actions for exploring activities and their constituents in a slower, but more systematic way. All these tools use the information of widgets.

For the situation of no widget information catched, Lin et al. [20] used a USB camera on an android device to collect image information and identify the widgets, and then used the USB and *adb* communication tools to transit the test instructions conducted on the Oracle servers, realize the test function. Monkey [4] is also an android test that needs no information of the application. However, it is reported costly in revealing software bugs [8]. Different from Monkey [4], we in this work employ computer vision techniques to find operable regions and trigger GUI events accordingly.

### C. Saliency Detection

Visual saliency is the perceptual quality that makes an object, person, or pixel stand out relative to its neighbors and thus capture our attention. Detection of visually salient image regions is useful for applications like object segmentation, adaptive compression, and object recognition.

There are a lot of other saliency detection algorithms, divided into two kinds, pixel space or spectral based [33][34], and feature based [23][32]. Xie et al. [32] exploited low- and mid- level cues, and used Bayesian framework to classify saliency region. Hou and Zhang [16] developed a saliency detection method using the image magnitude in the frequency domain minus the average between thousands of image magnitude in the frequency domain. Feature based algorithms use the special point or location to descript the image, and then use the classifier to find the saliency location.

## VI. Conclusion

Mobile game apps with rendered GUI widgets cannot be effectively tested by existing automated mobile testing tools. In this paper, we proposed a technique to apply saliency detection algorithms to recognize those rendered operable regions within mobile game applications as operable region candidates. Our technique generates concrete events to confirm whether these candidates are real operable regions. We have implemented our technique as an Android testing tool to test mobile game applications with the information on the detected operable regions. The experimental results on real-world widely used mobile applications show that our technique is effective to detect real rendered operable regions within mobile game applications. Furthermore, it is also generally effective in terms of the time to expose the first crashing fault.

In the future, we will use our technique on other platforms such as iOS, mobile web applications. It is also interesting to study a cloud-testing environment to apply our technique to test large-scale mobile games.

## Acknowledgment

This work was supported by grant from the National Key Basic Research Program of China (no. 2014CB340702), grant from the National Natural Science Foundation of China (no. 61379045), and grants from the General Research Fund of Hong Kong (nos. 11200015, 11201114, 111313, 125113, and 123512).

## REFERENCES


[1] D. Amalfitano, A. R. Fasolino & P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW 2011), 2011 IEEE Fourth International Conference*, pages 252-261, 2011.

[2] D. Amalfitano, A. R. Fasolino & P. Tramontana. A toolset for GUI testing of Android applications. In Software *Maintenance (ICSM), 2012 28th IEEE International Conference*, 650-653, 2012.

[3] Android Developer Website. "Monkeyrunner". Available: http://cs.szpt.edu.cn/android/tools/help/monkeyrunner_concepts.html.

[4] Android Developer Website. "The tool android monkey". Available: http://cs.szpt.edu.cn/android/tools/help/monkey.html.

[5] Android Developer Website. "Uiautomator". Available: http://wear.techbrood.com/tools/help/uiautomator/.

[6] T. Azim, I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, 48(10): 641-660, 2013.

[7] H. Bay, T. Tuytelaars, L. Van Gool. Surf: Speeded up robust features. In *Computer vision*, ECCV 2006.

[8] P. A. Brooks, A. M. Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333-342, 2007.

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

[10] M. Cheng, N. J. Mitra, X. Huang, P. H. Torr, S. Hu. Global contrast based salient region detection. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(3): 569-582, 2015.

[11] M. M. Cheng, Z. Zhang, W. Y. Lin, P. Torr. BING: Binarized normed gradients for objectness estimation at 300fps. In *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR 2007, pages 3286-3293, 2014.

[12] Cocos Play. "Cocosplay". Available: http://play.cocos.com.

[13] Dr.Dobbs.com. The Best Testing Tools. In *Jolt Awards 2014*, Retrieved June 2014.

[14] Github, Inc. "robotium". Available: http://code.google.com/p/robotium.

[15] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services* (MobiSys '14). ACM, New York, NY, USA, 204-217, 2014.

[16] X. Hou, L.Zhang. Saliency detection: A spectral residual approach. In *Proceedings of 2007 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR 2007, pages 1-8, 2007.

[17] S. Huang, M. B. Cohen, A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of Third IEEE International Conference on Software Testing, Verification and Validation*, ICST 2010, pages 245-254, 2010.

[18] Junit4. "Junit". Available: http://junit.org.

[19] Khronos Group. "Opengl". Available: https://www.opengl.org.

[20] Y. D. Lin, E. T. Chu, S. C. Yu, Y. C. Lai. Improving the accuracy of automated GUI testing for embedded systems. In *IEEE Software*, 31(1): 39-45, 2014.

[21] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of 7th International Workshop on Automation of Software Test*, AST 2012, pages 22-28.

[22] A. Memon, I. Banerjee, A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings. 10th Working Conference on WCRE*, 2003, pages 260-269.

[23] A. M. Memon, M. E. Pollack, M. L. Soffa. Hierarchical GUI test case generation using automated planning. In *IEEE Transactions on Software Engineering*, 27(2): 144-155, 2001.

[24] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, R. Mahmood. Testing android apps through symbolic execution. In *ACM SIGSOFT Software Engineering Notes*, 37(6): 1-5, 2012.

[25] Python software foundation. "The python language". Available: https://www.python.org.

[26] A. Rauf, S. Anwar, M. A. Jaffer. Automated GUI test coverage analysis using GA. In *Proceedings of Seventh IEEE International Conference on Information Technology: New Generations*, ITNG 2010, pages 1057-1062.

[27] Sauce Labs. "Appiunm". Available: https://saucelabs.com/appium.

[28] The Apache Software Foundation. "Maven". Available: http://maven.apache.org.

[29] Unity Technologies. "Unity3d". Available: http://unity3d.com.

[30] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, J. Kazmeier. Automation of GUI testing using a model-driven approach. In *Proceedings of the 2006 ACM international workshop on Automation of software test*, pages 9-14.

[31] Wikimedia Foundation, Inc. "computer vision". Available: https://en.wikipedia.org/wiki/Computer_Vision.

[32] Y. Xie, H. Lu, M. H. Yang. Bayesian saliency via low and mid-level cues. In *Image Processing, IEEE Transactions*, 22(5): 1689-1698, 2013.

[33] T. Yeh, T. H. Chang, R. C. Miller. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183-192, 2009.

[34] Y. Zhai, M. Shah. Visual attention detection in video sequences using spatiotemporal cues. In *Proceedings of the 14th annual ACM international conference on Multimedia*, pages 815-824, 2006.