Reducing False Positives in Automated OpenCV-based Non-Native GUI Software Testing

Masato Yamamoto
University of Aizu
Aizu-Wakamatsu, Japan
s1220105@u-aizu.ac.jp

Evgeny Pyshkin
University of Aizu
Aizu-Wakamatsu, Japan
pyshe@u-aizu.ac.jp

Maxim Mozgovoy
University of Aizu
Aizu-Wakamatsu, Japan
mozgovoy@u-aizu.ac.jp

ABSTRACT

This paper is aimed at improving mobile game non-native GUI testing. We follow an approach to use OpenCV image recognition algorithms for detecting and accessing the hand-drawn GUI elements on the screen, in order to interact with them from within automated test scripts. In the previous work, we experienced the problem that some tests fail not due to the defects of the tested software itself, but because of the false positive results of template matching. It means that the high scores are sometimes elicited for the best match, though the requested GUI element is actually not present on the screen. In this contribution we investigate the possibilities of image filtering in order to reduce the number of such false positive cases. We describe our experiments with two algorithms supported by OpenCV library, a selection of GUI elements and mobile game scenes, and a number of image filtering methods. We demonstrate that using Canny edge detection filters can significantly improve the accuracy of recognizing false positive cases without affecting the true positive situations. Our conclusions can be helpful for improving hand-drawn GUI based mobile software testing reliability.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in ubiquitous and mobile computing**; • **Software and its engineering** → *Software testing and debugging*; • **Computing methodologies** → Image processing;

KEYWORDS

Software testing, non-native GUI, image recognition, false positive

ACM Reference Format:
Masato Yamamoto, Evgeny Pyshkin, and Maxim Mozgovoy. 2018. Reducing False Positives in Automated OpenCV-based Non-Native GUI Software Testing. In *The 3rd International Conference on Applications in Information Technology (ICAIT'18), November 1–3, 2018, Aizu-Wakamatsu, Japan.* ACM, New York, NY, USA, Article 9, 5 pages. https://doi.org/10.1145/3274856.3274865

ICAIT'18, November 1–3, 2018, Aizu-Wakamatsu, Japan

ACM ISBN 978-1-4503-6516-1/18/11…$15.00
https://doi.org/10.1145/3274856.3274865

1 INTRODUCTION

Many mobile applications (for example, mobile games) are based on non-native graphical user interface (GUI). The GUI elements in such applications may be hand-drawn. That is why they could not be as easily accessed from the test scripts as regular GUI controls which rely on native GUI components supported by an underlying operating system. Native GUI testing procedures and pitfalls (including mobile GUI testing) have been described in many tutorials and reports [2, 5, 7, 9].

In the case of non-native GUI, the problem of GUI element identification cannot be solved by finding a perfect match of a bitmap image inside a game scene, because of the following reasons:

- Onscreen objects may be rendered differently due to the device characteristics and/or rendering quality settings;
- Device screens have different resolutions and significantly vary in dimensions, hence we need to scale patterns, which might cause distortions;
- Onscreen objects can be overlapping, therefore some part of the image might be "hidden".

Thus, the only feasible solution is to rely on approximate matching. In a number of related studies, the authors discuss the process of image matching with the use of OpenCV [1] library (and its *matchTemplate()* method) [6, 8, 11]. In [12] the authors reported the problem of false positive cases, when matching high scores are sometimes elicited even if there is no template image existing in the source image.

In our work we particularly address the extensive software testing process in the *Unity* based mobile game "World of Tennis: Roaring '20s" [2] which serves as a good example of mobile application with rich hand-drawn UI using a large diversity of UI elements, game scenes and characters. Figure 1 illustrates an example of successful recognition of the template image within the game scene. Such a case is frequent in the process of testing, when, before proceeding with different GUI tests, we need to check whether the screen is oriented in desired position (not rotated).

However, we realized that some tests fail after false positive template detection (as Figure 2 shows), neither due to the defects of the tests, nor the tested software itself. The standard *OpenCV* pattern matching algorithms might yield the score, which is unacceptably high for a GUI element that, in fact, is not present on the screen. Thus, there is an interesting question on how to struggle with such false positive cases affecting the reliability of mobile software testing automation process.

[1] OpenCV – an open source image processing library that we use here for template matching [3]
[2] http://worldoftennis.com/

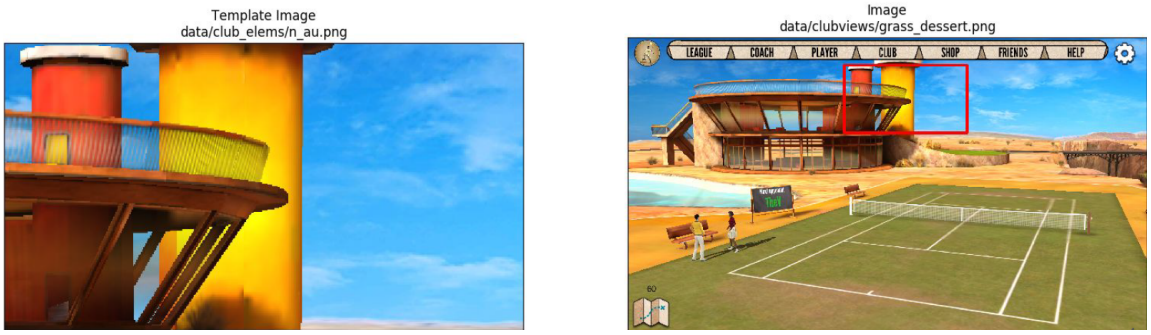

Figure 1: True positive case: reported score is 0.99 **(normal high score for the element existing on the screen)**

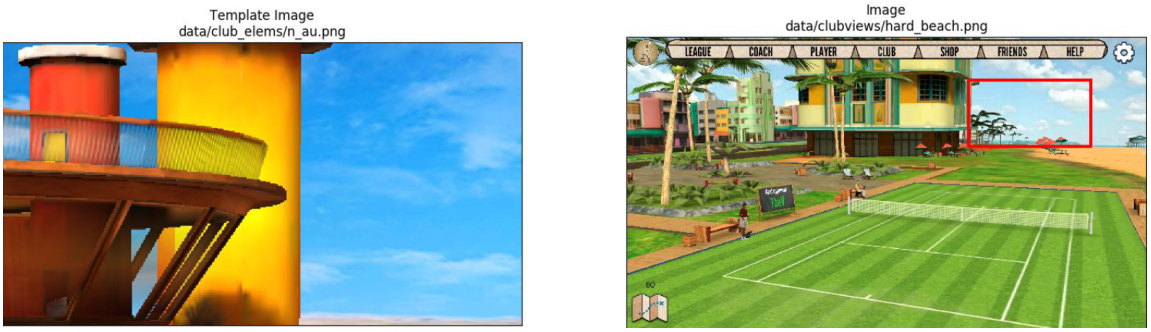

Figure 2: False positive case: reported score is 0.93 **(unacceptably high score for the element which is actually absent on the screen)**

In this research, we study two possible algorithms supported by OpenCV and investigate the possibilities of template and source image pre-processing aimed at reducing the number of false positive GUI element recognition cases.

2 OUR APPROACH

In this section, we describe an approach to decrease the number of false positive cases reported in test scripts using GUI element pattern matching. Our hypothesis is that some image transformations of both the source image (game screenshot) and the pattern can be helpful in order to reduce the number of false positive cases.

2.1 Pattern Matching Methods

Our experiments run with two pattern matching methods supported by OpenCV [3] used for recognizing game objects and hand-drawn Unity GUI elements in plain graphical data.

The first method used in our experiments is *TM_CCORR_NORMED* which is the normalized version of the correlation matching method that multiplicatively matches a template against the image and then maximizes the matched area. According to [1], the function slides through the image I, compares the overlapped patches with width w and height h against the template T stores the comparison results in result R. The summation is done over template and/or the image patch, namely: $x' = 0 \ldots (w-1)$, $y' = 0 \ldots (h-1)$. The result matrix of locations $R(x, y)$ can be described by the following equation:

$$R(x,y) = \frac{\sum_{x',y'} ((T(x',y') \cdot I(x+x', y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x', y+y')^2}} \quad (1)$$

The second method *TM_CCOEFF_NORMED* is the normalized version of the correlation coefficient matching method that matches a template against the image relative to their means and generates a matching score ranging from ..1 (complete mismatch) to 1 (perfect match):

$$R(x,y) = \frac{\sum_{x',y'} ((T'(x',y') \cdot I'(x+x', y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x', y+y')^2}} \quad (2)$$

In equation 2:
$T'(x', y') = T(x', y') - \frac{1}{w \cdot h} \cdot \sum_{x'',y''} T(x'', y'')$,
$I'(x+x', y+y') = I(x+x', y+y') - \frac{1}{w \cdot h} \cdot \sum_{x'',y''} I(x+x'', y+y'')$,
w and h are template's width and height correspondingly.

2.2 Organization of Experiments

We use the filtered dataset for both source images and template images. 4 types of transformations were applied: gray scaled images, edge detection, edge detection of the gray scaled images, and Canny edge detection [4]. We can transform the images by using any appropriate software, such as *ImageMagic* [3]. Here is the list of

[3] http://ftp.icm.edu.pl/packages/ImageMagick/beta/ImageMagick-6.8.5/index.html

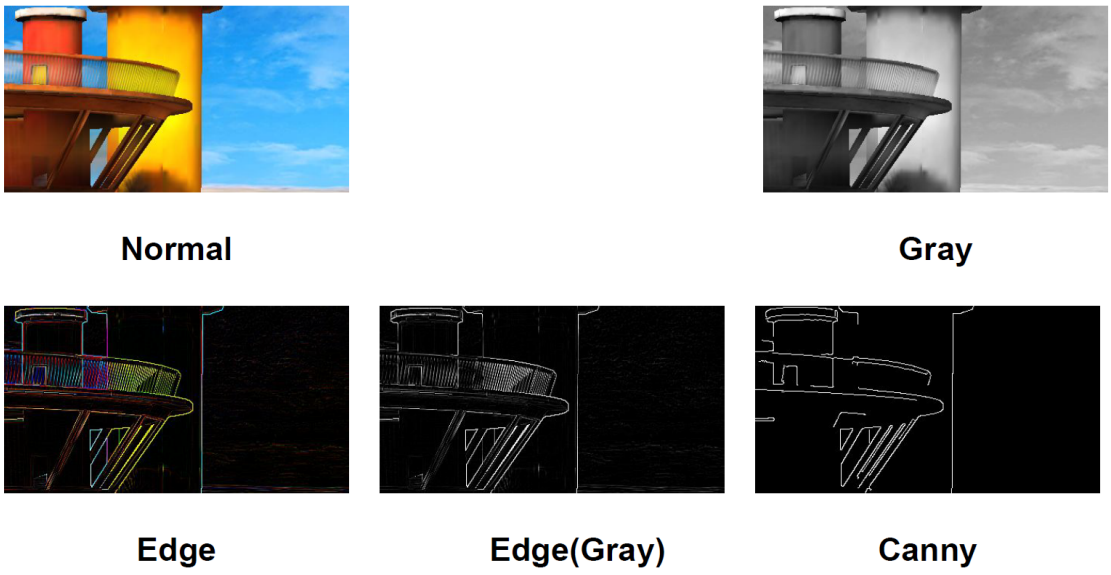

Figure 3: Template filtering.

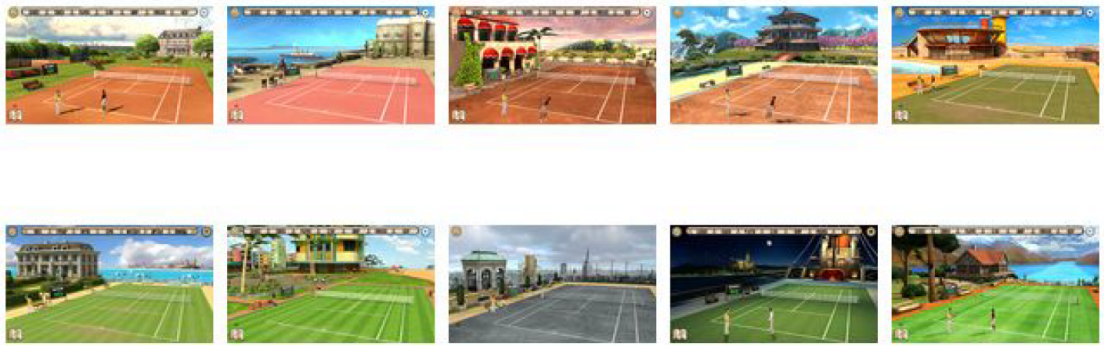

Figure 4: Source images used for experiments (club views from tennis game scenes).

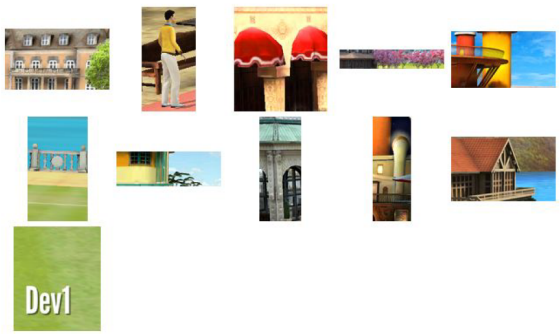

Figure 5: Template images (UI element fragments).

command samples that can be used in order to proceed with a desired filtering in *ImageMagic* (see Figure 3):

```
convert input.png -type GrayScale output.png
convert input.png -edge 1 output.png
convert input.png -colorspace Gray -edge 1 output.png
convert input.png -canny 0x1+10%+30% output.png
```

Figure 4 demonstrates 10 mobile game screenshots selected as source images (they are different tennis game club views used in test suite). As templates, we used 10 fragments cut from the source images, as well as one more image, which is not present in any of source images (see Figure 5). Thus, we try the possible combinations of 10 screenshots (as source images) and 11 templates, investigated against 2 pattern matching algorithms and 5 different filtering methods (including 4 above mentioned transformations and the case when there is no filter applied). For visualizing the experimental results in a way shown in Figures 1, 2 and 6, we

Table 1: Score Distribution (TM_CCORR_NORMED, True positive cases)

Filters	*Score ranges*									
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
No filter	0	0	0	0	0	0	1	1	8	0
Grayed	0	0	0	0	0	0	0	0	7	3
Edge detection	0	0	0	0	0	0	1	1	8	0
Grayed Edge detection	0	0	0	0	0	0	1	1	8	0
Canny	0	0	0	0	0	0	1	1	8	0

Table 2: Score Distribution (TM_CCOEFF_NORMED, True positive cases)

Filters	*Score ranges*									
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
No filter	0	0	0	0	0	1	0	1	8	0
Grayed	0	0	0	0	0	1	1	0	5	3
Edge detection	0	0	0	0	0	1	0	1	8	0
Grayed Edge detection	0	0	0	0	0	1	0	1	8	0
Canny	0	0	0	0	0	1	0	1	8	0

Table 3: Score Distribution (TM_CCORR_NORMED, False positive cases)

Filters	*Score ranges*									
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
No filter	0	0	0	0	0	0	0	19	81	0
Grayed	0	0	0	0	0	0	0	10	81	0
Edge detection	7	84	8	1	0	0	1	1	8	0
Grayed Edge detection	14	80	6	0	0	0	1	1	8	0
Canny	89	10	1	0	0	0	1	1	8	0

Table 4: Score Distribution (TM_CCOEFF_NORMED, False positive cases)

Filters	*Score ranges*									
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
No filter	0	14	19	32	22	13	0	0	0	0
Grayed	0	15	18	32	23	12	0	0	0	0
Edge detection	87	12	1	0	0	0	0	0	0	0
Grayed Edge detection	92	8	0	0	0	0	0	0	0	0
Canny	95	5	0	0	0	0	0	0	0	0

developed a tool displaying the template image, source image, their relative paths in the system, pattern matching method used, and the best match score. The region corresponding to the best match is shown by a red bordered frame within the area of the source image.

2.3 Results

Tables 1–4 list the experimental results for the earlier described combinations. Each cell in the tables reports the number of tests for a filtering method (table row) with the pattern match score within the corresponding interval (table column). The first column indicates the score interval from 0.0 to 0.1, the second column corresponds to the scores greater than 0.1 but less or equal than 0.2, the third column serves the scores greater that 0.2 but less or equal than 0.3, and so on.

From Tables 1–2 we can see that image transformations do not affect the true positive cases: all 10 tests (for each algorithm) corresponding to the possible cases, where the template **can** be found in the source images, have the scores, which are high enough.

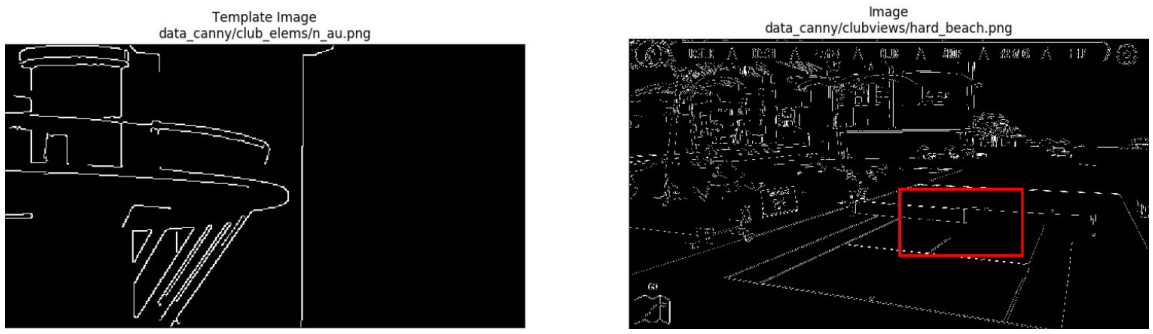

Figure 6: Struggling with false positive case: reported score is 0.10 (satisfactory low score for the element which actually absent on the screen)

The tests corresponding to the case, where the template images **do not exist** in the source images, are much more interesting. Our dataset (11 templates against 10 screenshots) gives us 100 false positives for each algorithm (let us remind that 10 combinations of the total number of 110 experiments serve the true positives).

From Table 3 we can observe that for the *TM_CCORR_NORMED* algorithm, there is large number of tests, where the best match score is greater than 0.8, though the template does not belong to any of dataset screenshots.

The *TM_CCOEFF_NORMED* algorithm (Table 4) works better, but still there is a number of tests, where the best scores are not as low as we could expect for the elements which are not on the screen (for example, in 13 tests the scores are higher than 0.5).

Experiments with grayed images do not show any significant improvement. However, the scores go to the lower ranges if we use edge detection filters. Canny edge detection algorithm gives the most promising results. For *TM_CCORR_NORMED* with Canny edge detection filter, 89% of tests have the scores less or equal than 0.1; 100% of test have the scores less or equal than 0.3. For *TM_CCORR_NORMED* 95% of tests have the scores less or equal than 0.1; 100% of test have the scores less or equal than 0.2.

It means that the score for the best match is very low, so the probability of false positive template recognition can be significantly reduced without affecting the true positive cases.

Figure 6 illustrates the situation with the same input template and the source image as in Figure 2, but with using Canny edge detection algorithm: the best match has the score 0.1, which could not lead to the false positive UI element detection.

3 CONCLUSION

Our preliminary experiments described in this work show that using transformed images does not significantly improve recognition of UI elements in the case when an element is present on the screen. However, such transformations may be useful in struggling with the cases of false positive UI element detection.

Despite pattern matching issues are not frequently discussed within the context of software testing, the interest of researchers and software developers to this topic is increasing [10].

We believe that for software applications based on non-native GUI (including mobile applications as a particular case), improving image recognition algorithms can make the software tests more reliable and help to avoid or minimize manual work of test engineers, necessary in order to fix the bugs conditioned by the low accuracy of hand-drawn UI element recognition.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Vitaly Klyuev for his valuable comments and helpful suggestions.

REFERENCES

[1] [n. d.]. OpenCV 2.4.13.6 documentation, Object Detection, matchTemplate. https://docs.opencv.org/2.4/modules/imgproc/doc/object_detection.html?highlight=matchtemplate#matchtemplate Accessed: May 20, 2018.
[2] Emil Borjesson and Robert Feldt. 2012. Automated system testing using visual gui testing tools: A comparative study in industry. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 350–359.
[3] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc.
[4] John Canny. 1987. A computational approach to edge detection. In *Readings in Computer Vision*. Elsevier, 184–203.
[5] André MP Grilo, Ana CR Paiva, and João Pascoal Faria. 2010. Reverse engineering of GUI models for testing. In *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*. IEEE, 1–6.
[6] Ville-Veikko Helppi. 2016. Using OpenCV and Akaze for Mobile App and Game Testing. http://bitbar.com/using-opencv-and-akaze-for-mobile-app-and-game-testing Accessed: Nov 2, 2016.
[7] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 77–83.
[8] Szymon Kazmierczak. 2016. Appium with Image Recognition. https://medium.com/@SimonKaz/appium-with-image-recognition-17a92abaa23d#.oez2f6hnh Accessed: Nov 2, 2016.
[9] Kanglin Li and Mengqi Wu. 2006. *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons.
[10] Inês Coimbra Morgado and Ana CR Paiva. 2017. Mobile GUI testing. *Software Quality Journal* (2017), 1–18.
[11] Maxim Mozgovoy and Evgeny Pyshkin. 2017. Unity application testing automation with appium and image recognition. In *International Conference on Tools and Methods for Program Analysis*. Springer, 139–150.
[12] Maxim Mozgovoy and Evgeny Pyshkin. 2017. Using image recognition for testing hand-drawn graphic user interfaces. In *UBICOMM 2017, The Eleventh International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IARIA, 25–28.