Using Image Recognition for Testing Hand-drawn Graphic User Interfaces

Improving Mobile Game GUI Tests with OpenCV Pattern Matching Methods

Maxim Mozgovoy, Evgeny Pyshkin
School of Computer Science and Engineering, Division of Information Systems
University of Aizu
Aizu-Wakamatsu, Japan
E-mail:{mozgovoy, pyshe}@u-aizu.ac.jp

***Abstract*—This paper discusses the use of image recognition for constructing automated GUI tests for the applications with hand-drawn user interface components, such as mobile games. Specifically, this contribution addresses the use of OpenCV pattern matching algorithms and the choice of most appropriate combinations of methods for GUI testing of the upcoming Unity-based mobile game "World of Tennis: Roaring 20's". Our idea is to classify UI elements (including buttons, game control elements, static and movable objects) with respect to their appearance in different type of scenes present in the game, as well as to find pattern recognition methods providing the best similarity values to increase UI element recognition quality.**

Keywords-software testing; GUI; image recognition; similarity; mobile game.

I. INTRODUCTION

Human-centric systems and the systems based on human-computer interaction (HCI) technologies are substantially multidisciplinary [1]. Through the prospective of the HCI interdisciplinary analysis, we make the observation that models and methods originally developed in one research area (not necessarily "human-centric") are often transferred and applied to a completely new distinct application domain [2]. In this work, we make an effort to examine a good example of such a transdisciplinary connection, which is a nontrivial case of using image recognition algorithms for improving software non-native graphic user interface (GUI) testing automation process. Mobile game development is a particular area where such an approach can be useful.

Indeed, in mobile games (such as ongoing project "World of Tennis: Roaring 20's" where we are involved in [3] (see Figure 1)), GUI is often designed with using hand-drawn components. It makes difficult developing standard automated GUI tests and basic functional smoke tests since all screen elements are in fact plain graphical images, in contrast to "classic" native GUI control elements that we can easily access programmatically nearly in the same way as users do, in test scripts [4]. Furthermore, non-native GUI elements can change their position on the screen and might look differently on different devices with different resolution.

The paper has the following structure. In Section II we describe our approach in general. Section III describes how the experiments were organized. In Section IV we examine a number of problems to be resolved while implementing test scripts using pattern recognition methods. In Section V we briefly summarize the current state of this project and introduce the primary tasks for future work.

II. APPROACH

In our previous work, we demonstrated that identifying objects of interest on the screen (such as GUI elements or game characters) could not be completely reduced to the task of perfect matching of a bitmap image inside a screenshot [5]. There are several reasons:

- Onscreen objects may be rendered differently for different GPU/rendering quality cases;
- Screens vary in dimensions, so patterns might need scaling;
- Onscreen objects often intersect with each other, so it happens that one object hides another one or can be distorted because of such an interaction.

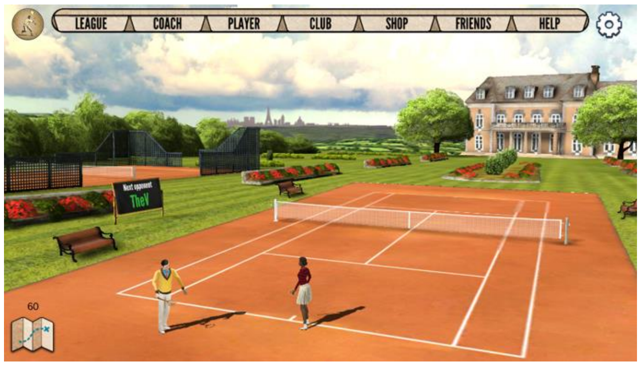


Figure 1. Actual screen of the "World of Tennis: Roaring 20's" mobile game.

Thus, the most straightforward way is to rely on approximate pattern matching. There are several tutorials where an idea of using image matching in creating test scripts is discussed [6][7]. OpenCV library [8] provides a number of methods for pattern recognition and can serve as a typical tool used for searching and finding the occurrences of the given pattern in a larger image. Basic OpenCV pattern matching methods can be accessed by using *matchTemplate()* function with a parameter defining a specific method among the variety of supported pattern matching methods [9][10]:

1. CV_TM_SQDIFF: square difference matching minimizing the squared difference between the pattern and the image area;
2. CV_TM_SQDIFF_NORMED: normalized version of the square difference matching (normalized methods are typically used when the effects of lightning difference between a pattern and an image should be reduced [10]);
3. CV_TM_CCORR: correlation matching method multiplicatively matching a template against the image and then maximizing the matched area;
4. CV_TM_CCORR_NORMED: normalized version of the correlation matching method;
5. CV_TM_CCOEFF: 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); and
6. CV_TM_CCOEFF_NORMED: normalized version of the correlation coefficient matching method.

As we know from different sources (such as [11]) the *matchTemplate()* function slides a template over the given area and computes similarity value in a range of [0..1] for each pixel location, thus maximizing pattern matching similarity. The function yields the best value as the final recognition similarity, so we are able to analyze the result from the viewpoint of GUI elements recognition quality.

An automated test consists of the following steps:

- Take a game screenshot (which is relatively time-consuming process that might take up to several seconds depending on a target mobile device);
- Detect the presence of a certain GUI element (using image recognition);
- React properly;
- Check the expected application behavior or program state; and
- Repeat the process.

III. First Experiments

For the first implementation of test scripts, we used *matchTemplate()* function and the pattern matching method TM_CCOEFF_NORMED. After experimenting with a number of test scripts, we realized that pattern matching reliability significantly depends on a recognition task. For example, simple button-like GUI elements (buttons, menus, tabs) can be recognized with high degree of similarity (0.90..0.98), according to OpenCV reports. Similarity score decreases to (0.63..0.65) for certain elements interfering with the background like menu item placed against the sky with moving clouds. This makes perfect template matching impossible in principle. Worse similarity values might occur even for the objects that are not graphically complex, but contain patterns distorted during rescaling: Figure 2 shows an example of low similarity score achieved for a simple edit box component.

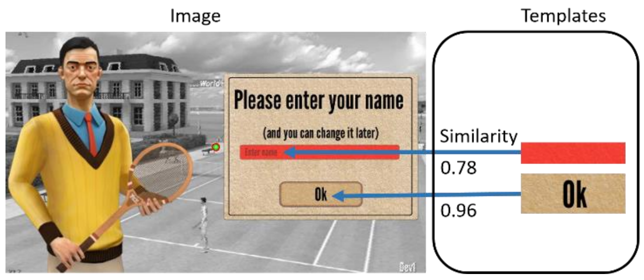
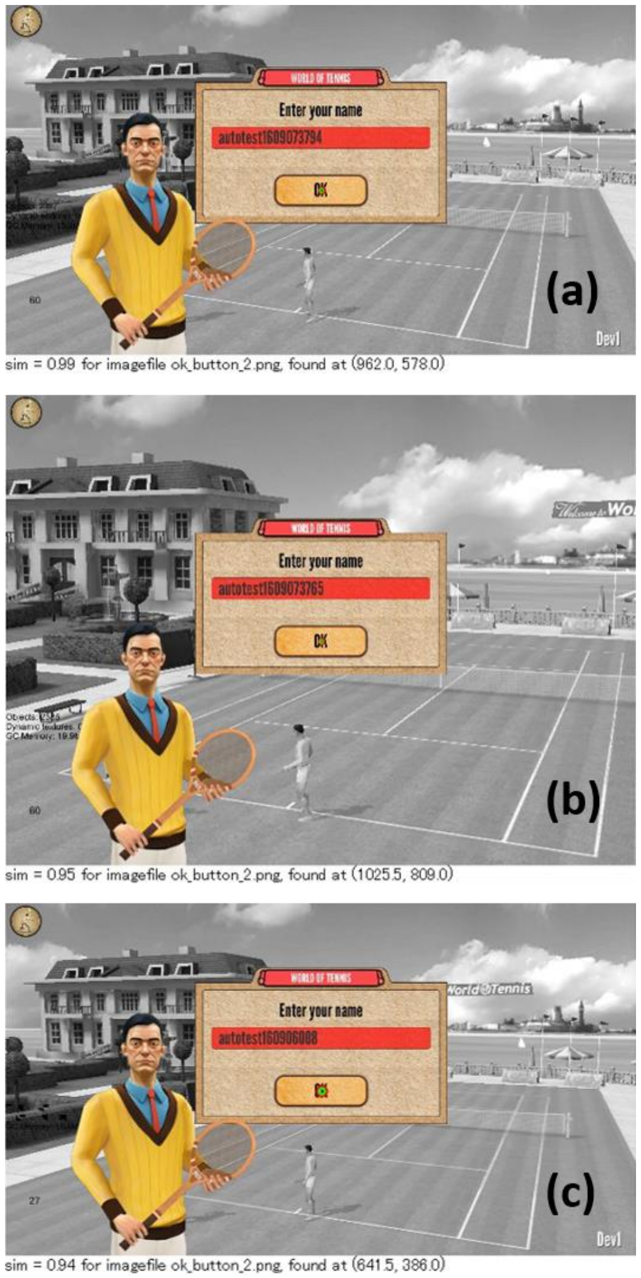


Figure 2. Template matching similarity varies for different UI elements.



sim = 0.99 for imagefile ok_button_2.png, found at (962.0, 578.0)

sim = 0.95 for imagefile ok_button_2.png, found at (1025.5, 809.0)

sim = 0.94 for imagefile ok_button_2.png, found at (641.5, 386.0)

Figure 3. Similary scores might differ depending on the device.

Even in the simplest cases, the similarity scores might differ depending on the device where the code is running. As Figure 3 illustrates, the scores for OK button range from 0.94 to 0.99 for different devices even if there is no any recognition complication such as "bad" background, surrounding or changing objects, etc. (apparently due to different screen resolutions and screen image scaling distortions). Table I summarizes this experiment.

TABLE I. EXPERIMENTING WITH OK BUTTON

Case	Description of the test case			
	Device	*Screen*	*Tap size*	*Similarity*
Figure 3 (a)	Xiaomi Redmi Note 3 Pro	1920x1080	1920x1080	0.99
Figure 3 (b)	iPad Air	2046x1536	1024x768	0.95
Figure 3 (c)	Doogee X5 Max Pro	1280x720	1280x720	0.94

There are also false positive cases, when the pattern matching algorithm detects the presence of a certain UI element, actually not shown on the screen.

Typically, such a false positive case might happen if some similar-looking graphical elements are confused with each other, especially when there are surrounding moving objects or complex background. One way to struggle with such cases is to try to match larger regions in order to include more context to a search request. For example, in the "World of Tennis: Roaring 20's", the Skip button is always placed next to a checkbox, so we can try to match the whole button/checkbox region. If there are several possible candidate elements, we can naturally report one having the highest similarity ratio with its identified match.

In principle, for test engineers, there is no much importance in achieving high similarity scores: we do not have to know whether a GUI element exists on the screen or not. We know that it *supposed* to be there. Otherwise, the test will fail (no expected element found). However, we believe that improving GUI element recognition will definitely facilitate the process of writing reliable application tests.

IV. PROBLEM STATEMENT

In matter terms, we face a purely interdisciplinary problem: the procedures for non-native GUI based software testing automation require the combined use of several technologies including traditional automated feature tests, functional testing frameworks, information retrieval, and image recognition.

As it follows from the observations mentioned in Section III, an important problem is to find optimal parameters of image recognition algorithms to maximize GUI elements recognition reliability, and therefore, to decrease the number of automated tests that might fail, not because of the software bugs, but due to the UI elements recognition defects.

There is a number of issues deserving particular attention. A typical problem in the process of initiating interaction between a testing framework and a fullscreen mobile application is to detect whether the device screen is upside down (it happens sometimes, and is not always detected correctly without pattern matching). For example, the first screen visible to the user of the "World of Tennis: Roaring 20's" is a "club view", so we can take some fragments of clubs and try to find them in the screenshots. Examples of club view elements are presented in Figure 4. We can also try to search the rotated fragments in order to diagnose that the screen is not in the position required for testing. Preliminary experiments show that, for such a problem, false positive cases might be a significant issue.

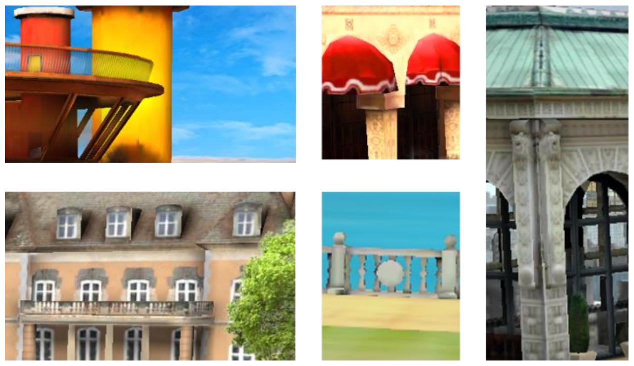


Figure 4. Club view template examples that can be used for checking screen orientation.

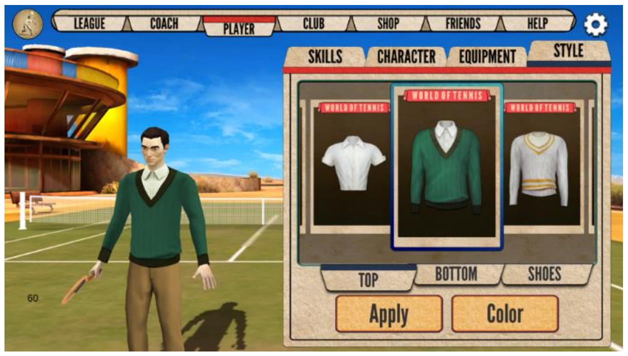


Figure 5. Standard UI controls: buttons, tabs, static images.

Even with relatively simple buttons (like those presented in Figure 5) there could be some problems:

a) Sometimes the game designers slightly change the buttons (in order to beautify them, make them slightly larger or smaller, change fonts, colors, etc.);
b) Sometimes buttons might be disabled, and the test scripts should be accurate enough to discern enabled and disabled buttons;
c) Sometimes there could be additional elements shown next to the button captions.

The challenge is to make sure that we still can match changing buttons in (a) and (c), but be able to distinguish them in (b).

There are also numerous moving objects on the screen. Suppose the test script needs to press on the character's head in the pictures shown in Figure 6. An animated head might make a perfect match difficult not only because of changes in object view itself, but also because of possible changes in the adjacent screen area (e.g., an airplane appeared in the sky, in our case). Hence, it might be required to work with a set of different images related to the same UI element and to perform a matching process for all of them. We have to consider a possibility to work with a larger region providing necessary context to avoid false positive recognition results.

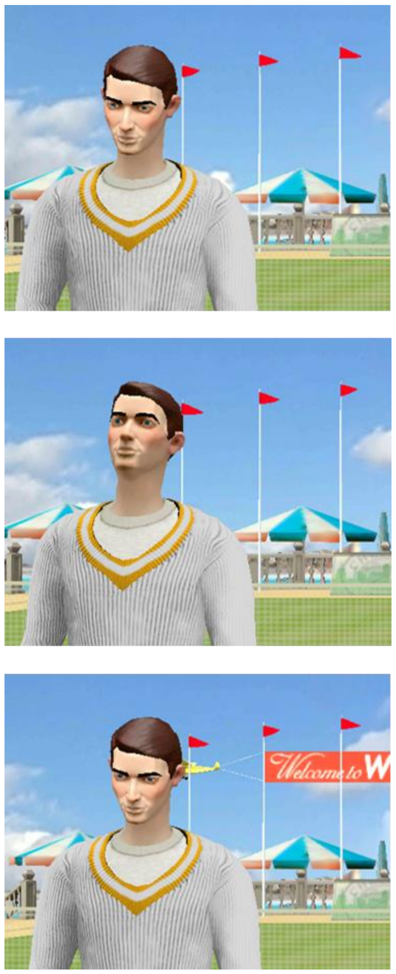


Figure 6. Moving objects: the object view changes and the surrounding area might change as well.

Our hypothesis is that experimenting with different pattern matching algorithms will allows us to provide a number of recommendation for test script developers. These recommendations will provide hints, which algorithms are better to use in which test contexts.

V. Conclusions and Future Work

Let us note that image recognition algorithms are rarely discussed within the scope of software testing, so we believe that advancing and improving the quality of the proposed approach will provide a feasible solution to be used as a part of integration pipeline in software development and testing. The special focus of this approach is on developing mobile applications (including mobile games) characterized by the presence of hand-drawn or non-native GUI components (for example, applications developed with Unity).

Our primary task is to arrange a number of experiments with real UI elements of different kind, real game screens of different size and resolutions and integrated with the tests running on real devices.

A big variety of UI components designed for the "World of Tennis: Roaring 20's" allows us to classify them in a number of classes including the following UI types:

- Button-like elements: buttons, edit boxes or similar;
- Static images: player portraits, field, etc.;
- Dynamic objects: moving player figures or similar.

We have to run pattern-matching algorithms for significantly different usage contexts: for example, in player settings window, club selection window, game selection window, ongoing game window, etc. We expect that for every combination (algorithm, UI element, usage context), the reported similarity values could give us better understanding how to improve the quality of test scripts and, therefore, how to make the next steps toward building a testing automation framework for mobile applications based on hand-drawn or non-native GUI components.

References

[1] H. R. Hartson, "Human–computer interaction: Interdisciplinary roots and trends," Journal of Systems and Software, 1998 Nov 30, vol. 43(2), pp. 103-118.

[2] E. Pyshkin, "Designing human-centric applications: Transdisciplinary connections with examples," In Proc. of 2017 3rd IEEE International Conference on Cybernetics (CYBCONF), Exeter, UK, Jun 21-23, 2017, pp. 455-460.

[3] "World of tennis. project homepage," accessed: Jul 10, 2017. [Online]. Available: http://worldoftennis.com/.

[4] "Automating user interface tests," accessed: Jul 7, 2017. [Online]. Available:https://developer.android.com/training/ testing/ui-testing /index.html.

[5] M. Mozgovoy and E. Pyshkin, "Unity application testing automation with appium and image recognition," in Tools and Methods of Program Analysis (TMPA-2017), 3rd International Conference on, 2017, Springer CCSI, vol. 779, in press.

[6] V. V. Helppi, "Using opencv and akaze for mobile app and game testing," (January 2016), accessed: Jul 7, 2017. [Online]. Available: http://bitbar.com/using-opencv-and-akaze-for-mobile-app-and-game-testing.

[7] S. Kazmierczak, "Appium with image recognition," (February 2016), accessed: Jul 7, 2017. [Online]. Available: https://medium.com/@SimonKaz/appium-with-image-recognition-17a92abaa23d\#.oez2f6hnh.

[8] "OpenCV Library," accessed: Jul 8, 2017. [Online]. Available: http://opencv.org/.

[9] "OpenCV: Template Matching, accessed: Jul 8, 2017. [Online]. Available: http://docs.opencv.org/master/de/da9/ tutorial_template_matching.html.

[10] G. Bradski and A. Kaehler, "Learning OpenCV: Computer vision with the OpenCV library," O'Reilly Media, Inc., 2008.

[11] R. Laganière, "OpenCV Computer Vision Application Programming Cookbook," 2nd ed., Packt Publishing, 2014.