Automated Device Testing for Windows Phone 7
Initial Research Report
Axelerate Solutions, inc

Abstract

The purpose of this document is to detail the initial results of the research for automating manual testing of Windows Phone 7 devices and multi-device testing. The document is technical in nature and provides a general overview, as well as a detailed walkthrough, of the associated algorithms, dependencies, implementation details, and an included demonstration application.

Introduction

Each Windows Phone 7 device must be validated against over 180 automated and manual tests for quality assurance. Some of these tests require a significant amount of time, which cannot be reduced without some form of automation. With automation, more tests can be introduced, especially for user interface and inter-phone operations testing.

Each device can be tested by treating the screen as a two-dimensional surface. Using a robotic arm, it can move to pre-set positions and simulate human input by pushing down on user interface components (either GUI elements or physical buttons). Since each device is unique, computer vision algorithms are used to detect the screen location and size, as well as graphical components of the screen (GUI buttons, etc). Since the goal is to automate testing as much as possible, certain screen elements are saved or converted to text through optical character recognition algorithms during normal testing procedures or on errors. This can then help developers working on quality control by having higher-quality error reports. Some tests can also be repeated over and over again without the fear of increased human error over time.

Automated testing for multiple devices is a difficult task to complete, due to the growth of complexity from testing inter-phone operation. Much like a single-device automated tests, multiple phones can be tested in parallel by treating the system as a whole unit. Each device may have several queued events that must execute (e.g. push button, accept call, etc.), but can be managed by one robotic arm based on speed of movement and a simple process-scheduling algorithm. The use of multiple robotic arms is possible, though not cost effective, and introduces complex problems of instruction ordering and process synchronization.

A proposed solution to automate multi-device testing would be to use a robotic arm, modern computer visions algorithms, GUI extraction methods, and optical character recognition in combination with a scripting language to executing a testing procedure. A scripting language, or other procedure description medium, must be created to help list a series of instructions to complete a single automated test. All testing instructions must be intuitive to the testing procedure developer, in which instructions such as “make call from phone A to B” and “phone B accepts call from C” are easy to describe and develop.

Solution Overview

The initial research was to investigate several approaches for the general problem of automating device testing. One of the biggest problems to solve first was to find a method of graphical user interface feature extraction. Sub goals included methods of correct screen detection as well as text recognition.

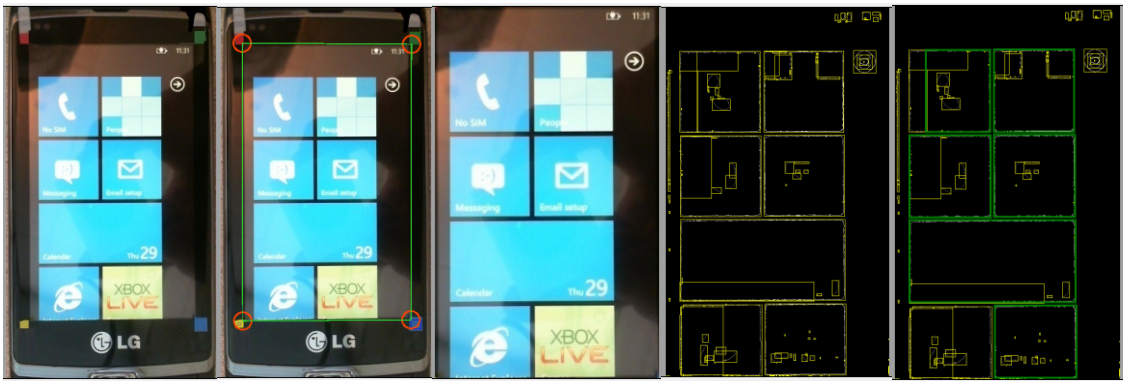
The best solution found, and implemented as a demonstration application, was to take a given image of a device (as a video source), find the device's source screen, then split it down to graphical user interface primitives, and finally categorize them into GUI element locations, sizes, and text. Interphone communication is not discussed in this initial report due to the need to solve the base problem first.

Other possible solutions exist, but with critical weaknesses. The most naïve solution is to blindly repeat user input events and validate the resulting image by doing a full image-to-image comparison. Though this can work for simple cases, there are significant issues with reliability and maintainability. Other ideas, and their faults, are discussed in the "Different Solution Approaches" section.

General Approach

Solution Introduction

The general solution approach investigated was to break down the problem into four steps to solve individually: Screen detection, projection transformation, GUI component extraction, and finally text string extraction. The first and second steps are to correctly identify the source screen, for better image processing and to convert real-world screen coordinates to coordinates a robotic arm can touch. This also helps with removing any configuration overhead by the user who is starting the testing procedure. The third step is to extract the GUI components through image processing. The final step is to convert these extracted primitives into useful data, such as text or icons.



The above is a series of images showing the transformation between an original source image to GUI decomposition. Yellow marks possible GUI components, while green identifies the most likely images.

Screen Detection

The first step, screen detection, is done by attempting to find all four corners of the phone's screen, resulting in a four-point polygon. This is done to help isolate the screen contents, and not to accidently detect background elements, or elements found on the phone such as a button icon or the status LED. It is also important to dynamically detect screens as each device may have a different screen resolution, physical size, or screen ratio. Detecting the screen dynamically avoids having the tester reconfigure the entire system for each unique device. The camera's lens may also not be parallel to the screen, causing a distortion where the lower half of the screen is seen as wider than the top half, distorting GUI components. Another large benefit of this is that any movement of the phone, camera, or environment does not break the testing process, as the source screen can be quickly found again as set intervals.

The best method of this dynamic screen detection is to have four unique identifiers on each corner, such as a small dot of a specific color. For the demonstration application, the top-left is defined as a red square, top-right is green, bottom-right is blue, and bottom left is yellow. By associating a color with a corner, we can quickly find the source image. These four colors at each corner are unique primary colors so that they are easily found. The demonstration application applies a color filter for each target corner color (such as top-left is red), then sorts which blobs of this color are most likely the corner identifier by using blob detection and filtering rules. These rules include ratio measurement (round shapes usually have a square ratio of 1:1) and a minimum and maximum threshold. Though it is very possible to confuse an on-screen color with the corner color, the demonstration application picks the best fitting blob closest to the corner of the device. There are other run-time issues, such as camera auto-color, which is discussed in detail at the bottom of this paper.

Other approaches to screen detection exist, but have their own issues. For example, it is better to detect colors over shapes, simply because shape detection is more computationally intense and these shapes must be relatively large to be detected. Other methods, such as edge detection, may fail as the background of the screen could be the same color of the device chassis. The naïve approach would be to ask the testing user to manually pick the four corners, but is not dynamic enough to deal with positional changes of position over time.

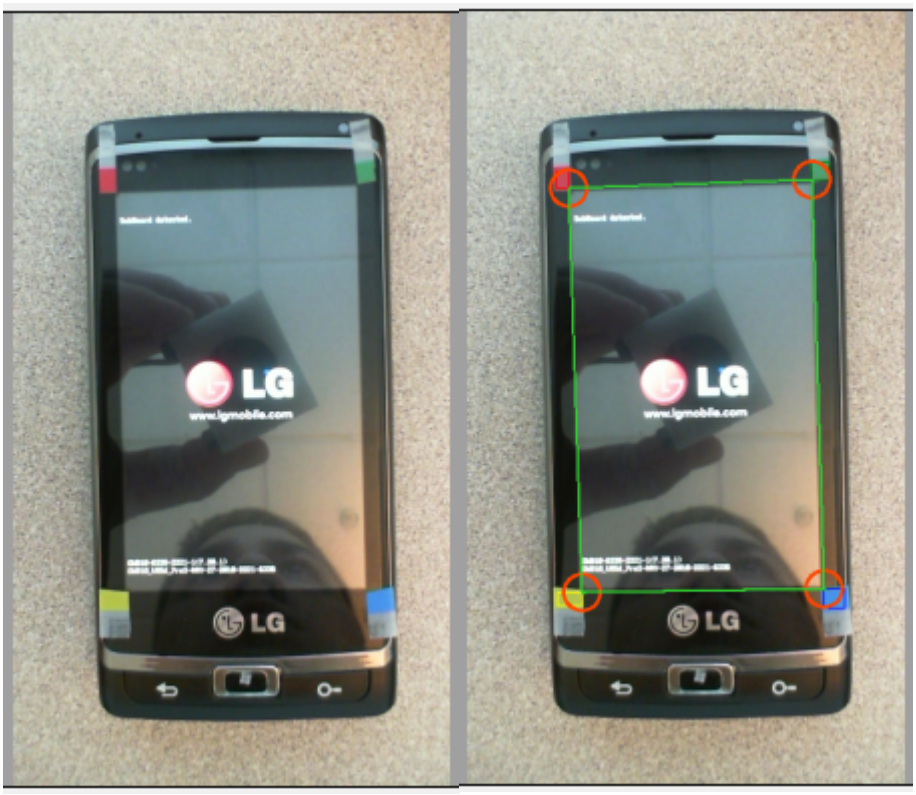
Projection Transformation

Once the screen position and properties are known, a projection transformation is applied. Simply, this means that the screen itself is zoomed in and adjusted such that it appears as though we are looking at a per-pixel image of the source screen. This is to help with better GUI detection since we warp the image back into a correct rectangle. Even slight angular differences between the surface of the screen and the view-frustum of the camera can create dramatic perspective warps in the image.

This projection transformation is a simple linear equation in which all we need are the coordinates of all four corners of the image, as defined through the previous process, to solve. After this

projection transformation, all GUI components have now been warped to their original shape primitives. Since many GUI components are rectangles or specific icon images, it is much easier to compare and find a sub-image within the source screen.

Note that for this transformation to work, we must give as much detail as possible before the transformation takes place. Even if the video capture device is high-resolution, we need to rotate it in such a way to see as much of the device as possible. If the device is a vertically held handset, we must have the video capture device record at a 90 degree angle such that the width of the camera is running parallel to the length of the device.



The above images show that perspective transformation may actually help solve the issue of camera distortion. Notice the testing user's face is clearly visible, as well as the blue dot from the recording camera device, which may introduce errors.

Graphical User Interface Extraction

One of the most complicated components of the automated testing process is the GUI element extraction. The core of this project is to detect GUI components, which can be achieved through several methods. These include image-to-image comparisons, image registration, image feature extraction, or other image processing and computer visions algorithms. Since the fundamental goal of this project is to

repeat user actions and confirm there were no failures, the best solution must be both reliable and fast.

In general, the Windows Phone 7 graphical user interface style is both clean and simple. Many of the GUI primitives are rectangular in shape and include a small unique image or icon. This can be quick for a computer to find, as edge-detection algorithms can be applied to find the edges of each rectangle, followed by blob detection to find all possible rectangles. Other scenes provide different challenges: menu navigation, which is text-heavy, has much smaller and harder to identify icons. Some scenes, such as picture navigation, are near impossible to navigate because of the overlays of various images. For this project to work, we must create a specific parsing system for each of the scenes and a mock-UI system. The home menu can be quickly parsed by only looking for squares and specific shapes. Other menus, such as the above mentioned pictures browser, will require full image-to-image comparison to work.

Much of the "real-world" implementation details are discussed in the next section titled "Robotic Interface & Scripting". The following text is how the current demonstration application implements its own GUI detection, and the general approach that should be taken in the final solution.

First, we convert the source image into grayscale. This is both to eliminate the costly overhead of implementing image processing algorithms through all three channels (red, green, and blue) but also to remove unneeded details that would interfere with some of the computer vision algorithms. The next step is to blur the image. Certain icons on the home screen have complex graphics that may confuse computer vision algorithms and convert that single button to several smaller buttons. For example the Facebook graphic icon has several squares that animated by changing colors over time. The XBOX Live icon a character portrait and the multi-colored logo. These two examples can easily confuse the next steps and generate many smaller buttons rather than the correct single button each example was in.

After the blur, we apply an edge detection algorithm to find all the borders of the GUI buttons. In this step, the background must be black, or at least significantly different than the user's theme color. This is to prevent incorrect edge detection. Because of the blur, we do not have to worry about images within each button, as their edges are blurred together. This blur is not strong enough to blur the large buttons with the background, which is why we need a large color difference with the background.

Though corner detection can be used instead of edge detection, most corner detection algorithms are overly sensitive and give too many false-positives. If the GUI button is not straight or curves slightly, many more corners may be detected than just the expected four. In the above graphic we can see that the edges of buttons are not necessarily smooth, which leads to incorrect corner detection. A more substantial issue with corner detection is how we relate each corner found with other corners. Since it is possible to detect a false corner, we might accidentally join it with another to form a non-existent button.

Once the edges of each button are found, we then apply a blob-filter. This filter finds all buttons, as well as several false-positives due to natural noise in the image as well as because of images within buttons. To choose which button is best, we apply a minimum and maximum size limits for each button. For each menu in the GUI system, we need to apply specific rules. In the home screen, there are several large buttons and one wide button that have very similar geometric properties. In this case, we remove

all blobs that are wider than 50% of the image width, or smaller than 35% of the image width. We apply the same logic, but with different ratios, with the button heights. Though this quickly removes any false positives, we also remove valid buttons that may be more unique, such as the Calendar button which is much wider than all other buttons. Other logic may be used with filtering to better choose what is considered a "probable" button, though the above logic works well for most situations that are not significantly dynamic.

Situations where there are lots of strings of text, such as the sub-menu reached from the home screen, can give many false-positives because the text may be interpreted as small buttons. To prevent this issue, it is important for these characters to be removed before the blob detection is applied. This aspect requires OCR to detect the characters, and may be computationally intense. This issue is discussed in the "Issues" section with several suggested possible solutions. Once all of these buttons are found, we then attempt to categorize the graphical user interface components.

GUI Categorization

In this final stage of the automated testing system, we attempt to start linking visual data from the source image to data that can be quickly used in the scripting system. First, we attempt to pull out the image of the button, so that it can be quickly tested against another image. Then, any strings of text into digitized text to be used in the testing script or bug reporting. This is done through OCR, or optical character recognition, which may be computationally intense and not always consistent. Once text is converted from an image format to a digital format, we attempt to categorize UI components, linking certain buttons with certain names. For example, we can attempt to do GUI validation by making sure the first icon on the sub-menu list is the "Alarms" application, by checking the name and icon against what is currently seen by the robot.

Virtual GUI Screen

After all of the device screen GUI components have been categorized, the robot may finally compare its actions: it can check the resulting device screen GUI components with that of an expected screen generated by the testing application. This generated GUI screen is not graphical in nature but simply contains expected button sizes and locations, as well as is aware of the context the phone is in (home screen, sub-menu, etc). If the robot does an action that fails, then the automated testing application knows of the failure by comparing expected GUI rectangle locations and seeing the differences are above a preset reasonable threshold.

General Approach Conclusion

The best solution found for this initial proof of concept was a mix of image processing and computer visions algorithms. Given an image of a device to test, we extract the screen and attempt to remove any distortions to help the system better detect graphical user interface components. After this is done, the image is turned to grayscale and then blurred, to remove any high-fidelity components that may confuse the GUI detection system. Then we apply an edge filter and blob-detection, to find the possible locations of buttons. Once we filter those locations to a set of very probable button locations,

we then categorize these buttons by attempting to convert any images or text into searchable or indexed data.

Robotic Interface & Scripting

Much of this initial research was to build a proof of concept of the GUI detection system. After finding a good approach and testing with a proof of concept , it is possible to connect this system to a robotic interface and simulate test cases with a scripting language.

To best simulate a testing scenario, the scripting language will have to be parsed by a "virtual user interface" system, in which we do not compare the device screen to an expected screen image, but simply compare button positions and content versus what is found on screen. If the scripting language tells the robot to scroll through 150 pixels down, the virtual GUI system should know which buttons should exist where, if the screen where to scroll down 150 pixels. After the physical movement is applied to the phone with the robotic arm, the vision software can quickly verify button placement, rather than do image-to-image comparison which may be not only slow, but very unreliable and not maintainable.

This virtual GUI system is only meant to generate an "excepted screen" in terms of button placement and general layout. The comparisons done after a test procedure is only between the visual button placements and sizes versus what is expected through this screen. Though the scripting language is not yet defined, doing such a system is quick, reliable, and maintainable.

Software Dependencies

Dependencies

Many of the above mentioned algorithms used in the general solution, are implemented in most image processing and computer vision libraries. Unfortunately most, if not all, of these libraries are open source, and no legitimate commercial-equivalent exists. OpenCV, the most commonly used computer vision library in the market, has been open sourced by Intel since 2000. The reason is that many of these topics are commonly used in either the academic field (commonly tied with Open Source) or within private uses that do not sell their own computer vision libraries.

The "core" algorithms used in the general approach are:

- RGB to Grayscale Filter
- Gaussian Blur Filter
- Canny Edge Detection
- Gaussian Blob Detection

Though there exists many implementations of each of the above mentioned algorithms, most of them are found in open source libraries which may or may not be allowed for private commercial use based on the associated license. There does not exist any related libraries found within the Microsoft research group, but there are several smaller projects related to computer visions libraries found within

Microsoft's internal site CodeBox:

- Sunflower – Image feature extraction
- C15 – Visual Verification
- VisionTools – Image processing

Another solution would be to re-implement these functions from scratch using the original academic papers. Though this may take time, these specific algorithms have been chosen in the general solution to be easy to maintain and work with, and are thus not unreasonable to re-implement. Most of these functions are either linear equations or statistical functions. Note that many of these libraries are written in a variety of different languages as well as may include their own dependencies which possibly have other conflicting licensing.

Optical Character Recognition (OCR)

Optical character recognition is key for image to text transformation in the GUI categorization. If an error were to occur on-screen, it is important to be able to automatically document this text to help automate the process, which is one of the mail goals of the project.

Microsoft has had several OCR-enabled products in the past, most notably OCR support in Word 2007. As of 2010, many of these features have either been removed or deprecated and are no longer supported. Microsoft research in the past has also worked on OCR, but with no formal library or known publication. Though in the internal Microsoft open-source project hub there are several visions libraries, there are no known OCR tools that significantly work.

Known Issues & Limitations

Heuristic Algorithm Usage

This general solution can work, and has been proven to work in the proof-of-concept demonstration application, but many of the algorithms (notably blob detection) have "heuristic" values. It is unknown yet how to best decide which series of rectangles on-screen are the "correct" buttons. Validation against the "virtual GUI system" may be the best approach, so that we can always track between the visibly seen buttons and the expected buttons. Each on-screen rectangle, possibly a GUI button, can be compared to other rectangles, in the virtual GUI system, and have their root mean squared values tested against a certain threshold.

Run-Time Speeds

In this general solution, there are some algorithms that take significantly longer than others. Most significantly is the "blur" algorithm. Though this blur can be seen as a heuristic factor in the process, as it lowers data resolution, it is critical to the rest of the image processing system because of its ability to blend unimportant data, but leave the edges of each GUI button visible. To resolve this, certain functions can be repeated less frequently than others. For example, screen detection only needs to update ever second or two, while GUI detection should be running all the time.

Camera Reflection & Varying Light

Many device screens have a reflective coating or finish, which may reflect parts of the camera or light source into the image. To prevent this, it is recommended that the robot operates in a low-light environment or in a dark room, where the only light source is the screen itself. The light from the recording camera may reflect enough on-screen to report false data. Modern cameras, including web camera,

Manual Setup Needed

Though the goal of this project is to automate testing as much as possible, it still requires human oversight and manual setup during test initialization. This is because as the lights change, device change, or other factors the testing user may have to reconfigure the corner locations (by setting the corner colors). Most notably, when the phone is being setup, it needs to be well placed in the camera's field of view as best as possible. Even if this general solution solves for any placement or angle errors, it is best to prevent any need for a projection transformation.

Importance of Screen Corners

It can be argued we do not need the four unique colors in the corner as there are other methods of setting up the source image. The user can be forced to manually choose the four corners through a setup application using a mouse and an image of the device in question. The user may also be forced to move the device in question to fit a certain "target zone" for the camera. Overall, it is still better to have a simplified automated corner detection system to prevent any errors over time as the robot manipulates the phone.

Office 2010 Deprecated OCR

As mentioned, there is no more support for OCR through Microsoft Office, or any other Microsoft product. Office 2007 did have this feature as an API, but it is now officially deprecated. Other solutions exist as proprietary software or as open source libraries.

Complex & Animated Icons

Animated icons can give false data, as the icon changes image over time and may appear as different values, button types, or even button count. Some icons, notably XBOX Live, are drawn with too much detail and can cause the same issue as with an animated button.

Icon Comparison

Icon comparison, or simply image-within-image searching, may be a needed component of this general solution, when searching for a specific image on-screen. This can be done with two well-known methods: Brute-force root mean square (RMS) searching, or through feature extraction. RMS, though easy to implement, runs slowly and have not give a good solution based on the resolution of the input image. Feature extraction, comparing features between two images rather than pixels, is another

method which is slightly slower, but can give more accurate search results. Ultimately, if we must compare images, the feature extraction method is more appropriate for this automated testing system.

Conclusion

The goal of this initial report was to investigate into how realistic it is to create an automated testing system for repeating test procedures both consistently and quickly. A general solution was proposed, to segment the given device screen into GUI components, and then compare it with a virtual screen after a robotic arm manipulates the physical device. After each action, these screens are compared based on a series of computer vision algorithms. Over time, based on testing needs or in a failure, some image components may be turned into text to better help with bug reporting. All of this is possible, though may require external image processing and computer vision libraries. Several other issues exist which must be resolved in a real-world implementation, such as camera-screen glare, lack of OCR support from Office, camera auto-adjustment, and more.

A demonstration application was developed as a proof-of-concept for GUI extraction. It shows the ability to segment the screen into multiple GUI components and text. Though the system can be automated, it may require user configuration upon initialization (such as corner-color adjustments). The robotic interface and scripting language has yet to be investigated, though the proposed virtual GUI system should allow for fast validation of resulting screens after a scenario, and allow for easy maintainability of the system.