Associating the Visual Representation of User Interfaces with their Internal Structures and Metadata

Tsung-Hsiang Chang
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
vgod@mit.edu

Tom Yeh
University of Maryland
College Park, MD 20742, USA
tomyeh@umd.edu

Rob Miller
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
rcm@mit.edu

ABSTRACT
Pixel-based methods are emerging as a new and promising way to develop new interaction techniques on top of existing user interfaces. However, in order to maintain platform independence, other available low-level information about GUI widgets, such as accessibility metadata, was neglected intentionally. In this paper, we present a hybrid framework, PAX, which associates the visual representation of user interfaces (i.e. the pixels) and their internal hierarchical metadata (i.e. the content, role, and value). We identify challenges to building such a framework. We also develop and evaluate two new algorithms for detecting text at arbitrary places on the screen, and for segmenting a text image into individual word blobs. Finally, we validate our framework in implementations of three applications. We enhance an existing pixel-based system, Sikuli Script, and preserve the readability of its script code at the same time. Further, we create two novel applications, Screen Search and Screen Copy, to demonstrate how PAX can be applied to development of desktop-level interactive systems.

ACM Classification: H5.2. Information interfaces and presentation: User Interfaces.

General terms: Human Factors, Design

Keywords: Pixel, accessibility API, text detection, text segmentation, graphical user interfaces

INTRODUCTION
Pixels are the most common characteristic and the ultimate elements produced by every application with a graphical user interface (GUI). In late 90's, researchers proposed using pixel-based methods for end-user programming and programming by example on existing user interfaces [9,10,17]. More recent work explores different possibilities with pixel-level interpretation for GUI automation, testing, and customization [3,4,5,16].

However, pixel-based methods have room for improvement in terms of speed and accuracy. For example, Sikuli Script [16] is not very fast, because it searches the screenshots of GUI elements across the whole screen and does not know how to narrow down the search space. Furthermore, it is hard to detect and extract the text content from pixel data.

UIST'11, October 16–19, 2011, Santa Barbara, CA, USA.

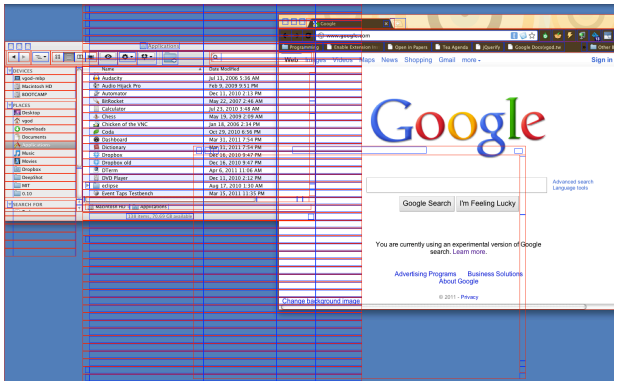


Figure 1 The internal structure of a GUI given by Accessibility APIs (AX) may not necessarily correspond to the actual visual representation of the GUI. Boxes above indicate the windows and widgets returned by AX even though they are not visible to users.

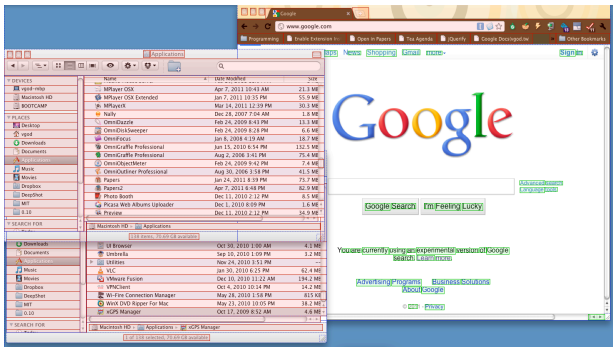


Figure 2 PAX combines pixels and Accessibility APIs for more accurate association between the visual representation and internal structure of a GUI. It filters accessibility information for only visible objects (red boxes) and also provides role, content, location, and size of objects detected by pixel-based methods (green boxes).

Current Optical Character Recognition (OCR) algorithms are designed for scanned documents, which are high-resolution with white background and simple column-based layouts, but not for on-screen text, which is low-resolution with colored background and could be randomly placed on the screen. Using current OCR algorithms on screen text would generate poor results [14,15].

Current pixel-based systems focus on using pixels as the sole source of input from the interfaces they operate. In order to maintain platform independence, these systems have intentionally neglected the other information that can be obtained from window managers or accessibility APIs. Unlike pixels, these extra sources of information are not necessarily available. For example, accessibility APIs are the standard hooks for exposing the internal structured metadata of a GUI to third-party assistive programs, such as screen readers, but to support such APIs requires engineering effort from individual software developers. As a result, accessibility hooks may be omitted or added later as the software matures. Fortunately, many popular commercial applications and built-in software on modern operating systems are accessibility-enabled. Therefore, why not leverage the accessibility metadata if the target applications provide them?

This paper introduces *PAX*, a hybrid framework combining *Pixels and Accessibility APIs*. PAX enhances the capabilities of current pixel-based systems and enables new interactive applications on top of existing interfaces. The key insight is that accessibility and pixel interpretations complement each other. Thus, both of these sources of information should be used together if possible.

PAX combines pixels with other information sources provided by GUIs, including accessibility metadata and low-level rendering data from the window manager, and associates the pixels with their internal hierarchical and structured attributes. As a result, PAX not only knows what is visible to the user on the screen but also understands the content and structures behind the pixels. We use accessibility metadata as a convenient and accurate source of widgets' information. If the accessibility metadata is not available, PAX automatically switches to pixel-level interpretation and still returns useful data. Furthermore, we use pixel-level methods to optimize the accessibility metadata. For instance, when accessibility APIs are not fine-grained enough to return the position of each word in a paragraph of text, we use a pixel-based segmentation algorithm, along with the known text for the whole paragraph obtained from the accessibility API, to locate the words with high precision.

The potential impact of PAX lies in its ability to improve existing pixel-based systems and to enable implementation of novel interaction techniques on top of existing interfaces. We validate this claim with concrete examples: the enhancement of pixel-based GUI automation (i.e., Sikuli Script [16]) and the implementation of two novel applications: *Screen Search* and *Screen Copy*. Screen Search is an attempt to allow users to conduct text-based search across multiple applications over the entire desktop rather than being limited to a particular window. It is applicable to any GUI components with text on the screen (e.g. on a title bar or in a tooltip of a button), even if it is occluded by other windows. Screen Copy is an attempt to allow users to copy the text content of a GUI component as well as the component itself. Users can paste the copied text into a text editor or reuse the copied component in a GUI designer application. The tool can be applied even when the text is not selectable or when the source code of the GUI is not available.

We make the following contributions in this paper:

- A hybrid framework that demonstrates how pixel analysis and accessibility metadata can be used together and complement each other.
- A text detection algorithm that finds text in screenshots, even with colorful backgrounds.
- A text segmentation algorithm that segments an image of a paragraph of text into individual word images, given known text.
- Validation of the framework in two novel applications and one enhancement of an existing system.

RELATED WORK

The idea of pixel-based interface interpretation and interaction emerged in the late 90's. Potter [9] developed a system, Triggers, to support application-independent end-user programming purely based on direct pixel access. Zettlemoyer and St. Amant [17] developed VisMap and VisScript to interpret pixels as high-level and structured representation of user interface objects, and then provided a set of mouse and keyboard commands (e.g. *mouse-move, single-click,* and *double-click*) to operate on the interpreted objects. WinCuts allowed users to cut a sub-region of an existing window and create an independent live view of the source, but did not interpret its content [12].

Recently, more pixel-based work has emerged. ScreenCrayons allows annotation on screen pixels [8]. Mnemonic Rendering determines the visibility of applications and shows motion trails of the changes when the hidden parts of windows are being revealed [1]. Sikuli Search allows users to search documents using the screen shot of GUI widgets [16]. Sikuli Script allows users to script user interfaces by taking screen shots of the targets the user wants to control [16]. Sikuli Test uses computer vision algorithms to analyze the pixels to generate testing scripts automatically [3]. Prefab interprets the pixels of a GUI and generates a high-level model of the widgets and their hierarchy [4,5]. The characteristic common to all this prior work is that it is completely focused on the pixel level. Instead of pure pixel methods, our work uses a hybrid approach that leverages pixels, the accessibility API, and the other useful information from the operating system to boost the robustness and performance of existing work and to enable new applications.

Most modern operating systems and GUI toolkits support Accessibility APIs, which were originally designed to be a standard hook for assistive technology applications, such as screen readers, or for GUI automation tools to communicate with a user interface programmatically. In addition to the assistive use of accessibility information, Stuerzlinger *et al.*'s User Interface Façades uses such information to allow users to customize an interface with copy-and-paste [11].

However, accessibility APIs are not widely available in every application and GUI widget. Hurst et al. reported that the Microsoft accessibility API can only correctly identify 74% of targets in eight popular applications on Windows [7]. Thus, they developed a hybrid approach that leverages the visual representation (i.e. the pixels) of a user interface to improve the accuracy of accessibility APIs for target identification. However, their approach does not deal with content; it is mainly for post-analysis of interaction logs to identify what targets the users might have clicked. In contrast, our approach is designed for real-time use to associate GUI widgets' internal metadata and their pixel representation. Chang *et al.*'s Deep Shot was a framework and an interaction technique that mixed pixel-based methods and accessibility APIs [2]. Deep Shot used visual features to identify which portion of the screen the user is looking at through a camera, and then found the application in that area using accessibility APIs. In addition, accessibility information was a backup channel if the target application of interest did not support the Deep Shot framework. Our work is a new abstraction layer that provides extra accessibility information to pixel-based systems and also expands the coverage for accessibility-based applications by providing the pixel interpretation of an interface with the same API.

PIXEL ANALYSIS VERSUS ACCESSIBILITY API

In this section, we describe the advantages and disadvantages of pixel-based methods and how we can use accessibility APIs to enhance the capabilities and performance of those methods.

Pixel Analysis

Pixels are the most common output medium of current computing devices. Every GUI application is eventually rendered as pixels on a screen. Unlike the pixels perceived from the real world and generated by a digital camera, the pixels generated by a computer itself have no noise, no distortion, and no other source of interference. Thus, early systems were able to use naïve bitmap matching to find targets on a screen [9,17]. Furthermore, Prefab has demonstrated that a UI model can be built from pixels in real-time so that researchers can build new interaction techniques on top of existing interfaces [4,5].

In contrast to understanding the UI model behind pixels, Sikuli Script takes a different approach to automate existing interfaces. In order to let users use loosely-bounded screenshots of automation targets, Sikuli uses template matching to fuzzily find the screenshots on the whole screen.

While they hold promise, the effectiveness of pixel-based methods can be challenged by four factors:

1. **Visibility constraints.** Invisible information, such as the items out of the current scrolling area or even the targets that are occluded by other windows, cannot be detected by pixel-based methods.
2. **Visual variations.** The accuracy of pixel analysis depends heavily on the look of target interfaces. If the user makes dramatic changes to the color scheme or the application theme, neither Prefab's trained prototypes nor Sikuli Script's fuzzy template matching screenshots can deal with the visual variations that result from such changes.
3. **Exhaustive screen search**. Pixel analysis is a potentially expensive operation, especially in high-resolution and multiple monitor environments with many millions of pixels. However, existing pixel-based methods such as Sikuli and Prefab often need to consider every pixel on the screen indiscriminately in order to locate certain targets, unless there is an external mechanism to direct their attention to specific regions on the screen.
4. **Low-resolution text.** The text content of an interface is hard to extract purely from pixels. Existing OCR engines are designed for high-resolution (150 to 300 DPI) scanned documents with white background and simple column-based layouts, but not for low-resolution screens (72 or 96 DPI) with colorful backgrounds and arbitrary layout. Simply plugging an existing OCR engine into a pixel-based system does not immediately work.

Accessibility API

Accessibility APIs are the standard interfaces built in modern desktop operating systems for assistive applications, such as screen readers, to access the low-level information of a user interface. Accessibility metadata is hierarchical, structured, and precise. For example, for an "OK" button in a confirmation dialog, we can get its role (AXButton), role description for humans, title, help message (the tooltip), enabled or disabled state, parent component, parent window, position, size, etc.

Accessibility APIs provide a convenient way to access many low-level attributes of existing software. However, to support such an API, application developers have to put in extra engineering effort to correctly expose the internal data. As a result, not every application and GUI widget supports accessibility APIs.

Fortunately, there are many applications that already accessibility-enabled. Hurst *et al*. reported that 74% of the widgets in eight popular applications were correctly identified by the accessibility API [7].

We conducted our own investigation into the current accessibility APIs for Mac OS X and Microsoft Windows (Microsoft Active Accessibility) regarding their capabilities and application compliance. We identified five challenges:

1. **Indifference to visibility.** Accessibility APIs does not know if a window or a GUI component is visible or not (see Figure 1). The location and the size of a minimized window would be returned as its original place and size before minimizing. If items are out of a scrolling area, they still can be reached by the accessibility API, and there is no way to tell which of them are visible to the user. This can lead to excessive and incorrect information when the developer just wants information about those the visible objects.
2. **Point-based object access.** Accessibility APIs support hit testing to gain access to an interface object shown on the screen. However, this ability is point-based and is limited to a single object; it is not possible to access a group of objects in a given region.
3. **Incomplete support.** Even applications that support accessibility APIs may not do so completely. An application may include new or complicated widgets that do not provide any accessibility metadata because they are not in the standard widget set.
4. **Coarse granularity.** The granularity of accessibility metadata may not fulfill the developer's needs. For example, the text content of a document may be returned as a block of text, where an novel interface technique may need the location of each individual word.
5. **Inconsistent text.** The text shown on an interface is not necessary consistent with the accessibility metadata, as it can be reformatted in an unknown way. For example, a date in accessibility metadata is "Friday, April 15, 2011 10:48:27 PM ET" but could be displayed as "Today, 10:48PM" on the screen.

Recognizing the respective strengths and weaknesses of pixels and accessibility APIs, we believe it would be ideal to combine them in a complementary manner, offering both generality from pixels and precision from accessibility metadata at the same time.

PAX: A HYBRID FRAMEWORK

We propose PAX, a Pixel+Accessibility hybrid framework that associates the high-level visual representation of GUI widgets with their low-level structured and hierarchical information. PAX has three sources of input: pixels, accessibility APIs, and window managers. PAX consolidates the information from these three sources into a new API that allows developers to easily access not only the pixels of the GUI widgets on the screen but also their internal information.

PAX associates the pixel representation of each GUI element on the screen with its underlying attributes, such as its role (which could be a button, a text field, etc.) and its text content or value. The underlying attributes are retrieved from the accessibility API if available. If not, PAX attempts to infer the role of the element from its pixel representation using template matching and uses pixel-based algorithms to detect and recognize the text (see Figure 3).

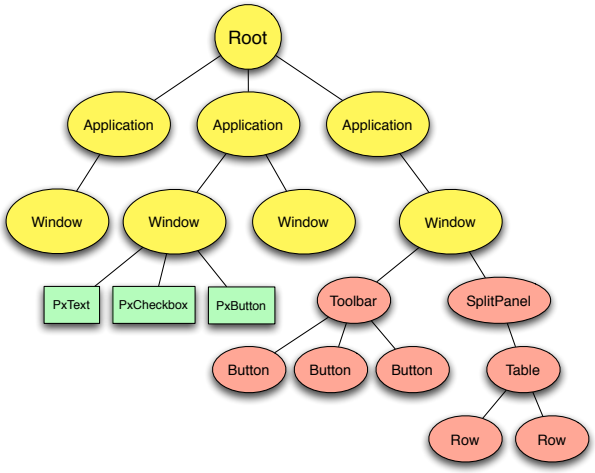
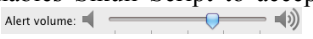


Figure 3 The tree of PAX UI elements. The yellow nodes, created using accessibility and window manager information, do not require any support from application developers. The red nodes are the elements exposed using accessibility APIs, while the green ones are reverse engineered from pixels.

PAX can improve the effectiveness of existing systems that are based purely on accessibility APIs or on pixels. On the one hand, systems based on accessibility APIs can benefit from the knowledge of the GUI's apparent visual representation to resolve certain ambiguities. On the other hand, pixel-based systems can use PAX to improve speed and accuracy. For example, PAX enables identification of a particular UI element using an XPath. Sikuli Script can store the XPath to a target UI component when taking the screen shot of it along with the screen shot image (e.g. in the header of the PNG file) and later use that path to locate the component without running template matching on the whole screen, or to constrain the search in a smaller region. Furthermore, PAX enables Sikuli Script to accept simple commands, e.g. `find(`  `).value()`, to easily retrieve the value of a slider without calculating the position of the thumb to infer it.

Designing PAX

There are three design goals for PAX:

1. The framework should enable existing pixel-based systems to easily access internal widget information given a region or a point on the screen.
2. The framework should simplify the difficulty and complexity in implementing novel interaction techniques on existing interfaces.
3. The framework should automatically give the most accurate results from available resources (either from accessibility metadata or pixel reverse engineering).

PAX maintains a tree of *UIElement* objects shown on the screen. The root of this tree is a virtual node, which does not represent any physical GUI elements. The root returns

all running applications as its children and can be obtained by calling a global function `getUIElementRoot`(). Each application is also a UI element, which returns its opened windows. Each window recursively contains the same hierarchical structure of its title bar, tool bar, and all the other UI elements.

PAX distinguishes three different visibility levels for a UI element:

- **Visible.** This element can be seen on the screen.
- **Virtually visible.** This element is meant to be visible on the screen but is partially or entirely occluded by other windows in front of it because of limited screen space; if the screen was infinitely large and no windows needed to overlap, this element would be visible.
- **Invisible**. This element is not meant to be seen by users because it is scrolled out of view, in a minimized window, or hidden by design.

Therefore, each UI element has three different methods to get its children according to their visibility: *getVisibleChildren()*, *getVirtuallyVisibleChildren()*, *getChildren()*, where they return the visible children, virtually visible children, and all children, respectively. Because a UI element may be partially invisible, PAX also provides two methods, *getVisibleBounds*(), which returns the bounds of the visible part of a UI element, and *getBounds*(), which returns the original bounds.

Like accessibility APIs, PAX provides *getRole*(), *getText*(), and *getValue*(), for getting the role, the text, and the value of a UI element, respectively. A role represents the type of a component, e.g. a button or a list item. Text can be the label or the string value of a component. A value is a number that is only meaningful for some components, e.g. a slider or a check box. These values are retrieved from accessibility APIs if they are available; otherwise pixel reverse engineering techniques are used. Further, PAX also provides *getVisibleText*(), which returns the text actually shown to the user.

In addition to the standard accessibility attributes, each UI element has a *getScreenshot*() method for getting the screen shot of the element even it is only virtually visible, and *getXPath*(), which returns an XPath to the element, such as /Application[name="Word"]/Window[1]/TabGroup[1]/Group[text="Paragraph"]/MenuButton[text="Bulleted List"]. The *name* attribute in an XPath is only valid for Applications nodes, whereas *text* and *value* can be used in the remaining nodes. Developers can save the path and locate the same element quickly with a global function `locateUIElement(xpath)`.

To find particular elements, two methods can be used, by screen location or by content. PAX supports single-point hit testing with `getUIElementAtPoint(x, y)`, which returns the element at the given point, and `getUIElementsInRect(rectangle)`, which returns all the visible elements in the given rectangle on the screen. To find by content, each UI element has the three methods *findChildren(pattern), findVisibleChildren(pattern)* and *findVirtuallyVisibleChildren(pattern),* which return all, visible, and virtually visible children whose text content matches the given regular expression pattern.

Finally, to better support advanced interaction techniques, each UI element has a method *focus*(), which sets the keyboard focus to that element and also brings its parent window to the front at the same time. This is particularly useful when the developers need to perform further interaction on a UI element.

Bridging between Accessibility APIs and Pixels

One of this framework's goals is to automatically give the most accurate results from the available resources. Therefore, PAX constructs the UI element tree from all running applications and their corresponding accessibility handles. If an application has exposed all necessary accessibility hooks, its descendants in the PAX tree are simply copied from the accessibility tree and wrapped up with a *UIElement* interface. Sometimes a few widgets or even the entire application do not support accessibility APIs, and in this case, each of these widgets or windows looks like a black hole, with only a single node in the accessibility tree to record its boundaries.

When PAX walks down the accessibility tree and reaches a leaf node, it determines if there is a need to reverse engineer the pixel contents of the node with three simple rules: (1) if the node is a text component (e.g. a text label, a text field, or a text area), run our text segmentation algorithm to breaks the text into word component pieces; (2) if the node's role is not a container and not a text component (e.g. it is a check box, a radio button, etc.), do nothing; (3) otherwise run pixel reverse engineering methods on the node's screenshot. The text segmentation algorithm is useful for higher-granularity information about text components, and will be described later in this section.

With pixel reverse engineering methods, such as Prefab, we still can provide similar information to the accessibility metadata even if we reach a dead end in the accessibility tree. In our current prototype, we did not attempt to completely build the hierarchy of UI widgets from pixels using Prefab's method, but we used Sikuli's template matching to find a small set of GUI widgets (e.g. radio buttons, check boxes, sliders) instead. Furthermore, we developed a new algorithm for locating arbitrary text content in a complicated component, e.g. a web view.

A PAX tree is lazily generated on demand. Once parts of it are generated, the results are cached for fast response. The developer can explicitly request the cache to be updated. With proper event hooks that monitor the updates of the corresponding UI, the cache can be updated automatically after the UI is changed. For the reverse-engineered components, the tree can be automatically updated by comparing the consecutive screen shots. Comparing two 1680x1050 screenshots takes only 30ms on a modern PC; therefore, it is feasible to use this technique to continuously monitor the changes of a UI.

In the last parts of this section, we discuss how we have dealt with the challenges mentioned above as well as the text segmentation and detection algorithms.

Determining the Visibility of UI Elements

With only accessibility APIs, we cannot tell if a window or a component is visible or not. To address this problem, our solution is to request the *z-order* of each window from the *window manager*, and create "masks" to cover the occupied areas of each window from top to bottom. Thus, if a component is not fully covered by the masks and also intersects with its parent's visible bounds, it is visible from the user's point of view.

For virtually visible elements, we only care if a UI element intersects with its parent's bounds. If so, it is virtually visible; otherwise it is invisible.

Region-based Hit Testing

Accessibility APIs usually support hit testing, which is used for getting the accessibility information on a particular point on the screen. Unfortunately, this feature is limited to a single point and a single object.

To get multiple elements in a region, a naïve method is to run the single-point hit testing on each point in that region. However, this is inefficient because a region could have millions of points. Another method is to traverse the component tree and find all visible elements that intersect with the given region. But this is not possible with pure accessibility APIs, because they do not know if a component is visible or not.

Fortunately, PAX already has precise information about the visibility of each UI element. Therefore, PAX provides a function that enables developers to get the internal information of multiple objects in a given region on the screen.

Text Detection and Extraction From Pixels

Current pixel reverse engineering techniques, such as Prefab, can locate common GUI widgets and extract their textual content. However, Prefab's method requires text be located over predictable backgrounds that Prefab can model based on provided examples. If the text is on a background for which Prefab has not been trained, or a complicated background that Prefab cannot model (e.g., a photographic wallpaper), it will not find the text. Recently, computer vision researchers have conducted research on segmentation and recognition methods for small screen-rendered text and reported accuracy achieved of 99.2346% [13,14,15]. However, they assumed the position of the text is known and did not address the problem of text detection. To complement these pixel reverse engineering techniques, we have developed a text detection algorithm that locates text in arbitrary position in a screen image.

Given a screen image, the algorithm for converting it to words has three major steps: (A1) segment the image into disjoint blobs of pixels, (A2) merge character blobs into word blobs, and finally apply OCR to extract words.

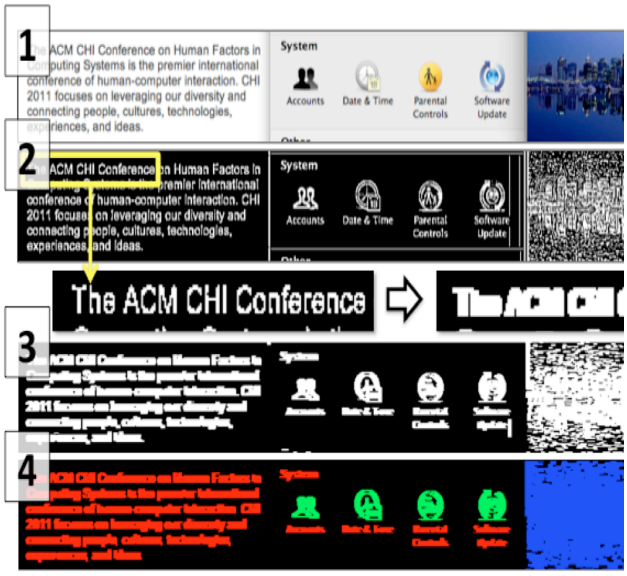


Figure 4 Text Detection process. (1) Given an image, (2) foreground pixels are extracted as small blobs. (3) Blobs are connected to form larger blobs, (4) which are classified into text (red), icon (green), or photo blobs (blue).

A1. Salient Component Detection

The goal of this step is to decompose a screen image into a set of salient components, each of which is composed of a blob of foreground pixels. Given a screen image as input, we first convert the image from color to grayscale. We apply an adaptive threshold to filter out low-contrast pixels. Figure 4-2 gives an example of the image after this process. Many container widgets such as panels have large areas of plain background pixels that can be easily filtered out in this way. From the high-contrast foreground pixels that are left, we detect long lines that might be window borders or grouping cues. These long lines are then removed so that components close to those lines would not be mistakenly interpreted as being connected by the lines. After this process, text elements turn into a set of blobs, each of which correspond to a character, whereas image elements turn into a set of disjoint parts.

A2. Text Extraction

Next, we merge blobs of individual characters into larger blobs of words and use OCR to extract text from each word blob. To merge blobs, we apply a dilation operator to expand the extent of each blob horizontally. If a blob is a character, horizontal dilation will connect it with the characters before and after, as long as the amount of dilation exceeds the amount of character spacing. This spacing depends on the font size, which can be estimated from the height of the blob. Figure 4-3 shows the output of this merging process. Then, given a string of connected blobs, we check two properties to decide whether it is likely to be text. First, we check if these blobs share a common height and baseline. Next, we check if the color variation among the foreground pixels is low, since GUI text tends to be

rendered in a single, solid color to improve readability. Blobs satisfying both conditions are considered to be text blobs (red pixels in Figure 4-4) and passed to the OCR engine to extract words from them.

We did not implement Wachenfeld's screen text recognition algorithms [13,14,15], but used the Tesseract OCR engine (http://code.google.com/p/tesseract-ocr/) in our current prototype instead. If it were given the whole screen image as input, the Tesseract OCR engine would perform poorly because it assumes the text is in a single column. If we segment screen images into blocks of words that are processed individually by the OCR engine, the overall performance is better.

Text Image Segmentation Given the Text

The minimum granularity returned by accessibility APIs is one UI component. This may be not enough if developers need the location and the bounds of each individual word or even each character in a text component. Therefore, we have developed an algorithm that segments the image of a text component into individual word blobs. Unlike other text segmentation algorithms, the text string is known from the accessibility API, so we have additional clues to locate each word more accurately. Furthermore, since this algorithm runs on the leaf nodes in accessibility trees, we can assume the text is on a GUI widget with a simple or gradient background for the sake of readability. If the text has a complicated background, our algorithm would not work.

Given that the text is already known (except for some inconsistent cases, e.g. reformatted dates), this problem is not as hard as the original text segmentation problem in OCR. We describe this algorithm in two steps: (B1) segment images into N blobs, where N is the number of words in the given text; (B2) sort and match each blob to its corresponding word in the text.

B1. Text Segmentation

By assuming the background of the text is a solid or gradient color, we look for a vertical or horizontal line for which each pixel is the same color, in a descendent or an ascendant order in the given region. Once we have found such a line, we create a background by repeating this line to fill the size of the image, and then subtract the original image with this background to get an image with pure text pixels.

We use a top-down approach that is modified from recursive X-Y cut [6] to break a text image into word blobs. We assume the text could be split into multiple lines, but no single word is broken with hyphens. The idea of this algorithm is to calculate the sum of the pixels in each horizontal and vertical line to produce a density graph of white space. This graph shows several peaks that define horizontal or vertical gaps between lines or words, which are also the cut points we need to segment the image into smaller pieces. Our algorithm finds the largest gap, defined by its number of pixels, in the image, and cuts the image horizontally or vertically until the number of pieces remaining equals the number of expected words.

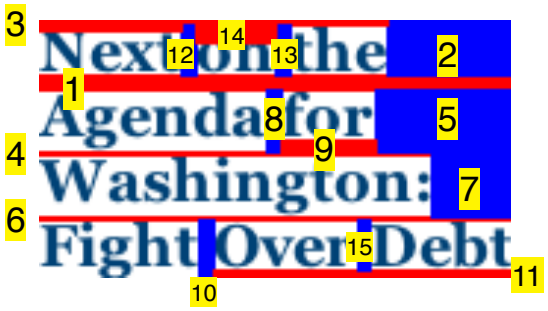


Figure 5 Text segmentation algorithm. Each number represents the order of the cut. (blue is vertical, red is horizontal)

B2. Matching each word with a blob

After the first step, we have N small blobs, each of which corresponds to a word. Next, we sort these blobs vertically and group them into lines with similar baselines. Blobs in a line group are then sorted horizontally to match the writing order of western text.

Matching Accessibility Metadata with What Users See

The *getVisibleText*() method of a *UIElement* should return the text that users see on the screen. One goal of PAX is to deliver the most accurate results. Therefore, we should use the text from the accessibility API if possible. However, the text shown on the screen is not necessary consistent with the one returned by the accessibility API.

A common example is automatic truncation when a string is too long. For example, "a very very very long file name.txt" may appear on the screen as "a very very…name.txt". Another example is time and date formation. For instance, "Friday, April 15, 2011 10:48:27 PM ET" could be shown as "Today, 10:48PM" on the screen.

To address this inconsistency, we compare the text from the accessibility API and the text from OCR. If the edit distance between these two strings is smaller than a threshold that is proportional to the length of the OCR string, we infer that no inconsistency exists and return the accessibility string as the visible text. Otherwise, when the strings are inconsistent, the OCR text is returned. Note the developer does not necessarily need the visible text, which is why we provide *getVisibleText*() and *getText*() for developers to choose according to their scenario.

EVALUATION OF TEXT ALGORITHMS

In this section, we describe the evaluation of our text detection and segmentation algorithms in PAX.

Text Detection Algorithm

To test the performance of our text detection algorithm, we constructed a dataset that consists of six high-resolution screenshots downloaded from the Internet. This dataset covers a variety of GUI widgets and text content on three major platforms (Mac OSX, Ubuntu Linux, and Windows 7). Each word in the screenshots was located and labeled manually as ground truth. The total number of visible words in this dataset is 1141. The number of visible win-

dows is 16. This dataset was held back while we were developing the text detection algorithm; we used screenshots of our own computers for training purpose and preserved this collection only for testing.

During testing, we manually cropped the images of the 16 windows (since window bounds are available in PAX) and applied our text detection algorithm to each image. Our algorithm made 1236 detections. We compared the results to the ground truth and found our algorithm was able to achieve a precision of 84.39% (1043/1236) and a recall of 91.41% (1043/1141). The most common errors were isolated digits that were too small and were repeatedly mistaken by the algorithm as noise for 32 times (2.81%).

Text Segmentation Algorithm

To evaluate the performance of our text segmentation algorithm, we built a dataset of 331 images each of which is a tightly bound block of text. This dataset was split into training and test sets to prevent overfitting when developing the algorithm. The former has 546 words in 57 images and the latter has 2046 words in 274 images. Each image block has at least two words. These images were collected from our own Mac computers and covered popular desktop applications and web sites (e.g., Microsoft Word, twitter.com, cnn.com). We applied the text segmentation algorithm to each image and manually verified the results. We found only 30 words out of 2046 words were incorrectly segmented, which represents an accuracy of 98.55%.

VALIDATION THROUGH EXAMPLE APPLICATIONS

To validate the usefulness of PAX, we present three novel applications enabled by PAX: enhanced Sikuli Script, Screen Search and Screen Copy.

Sikuli Script

Sikuli Script, developed by Yeh et al., is a pixel-based approach to GUI automation that lets users take a screenshot of a target to direct mouse or keyboard inputs to that target [16]. It was deployed in early 2010 and a sizable user community has grown since. The discussions on its user forum suggest some difficulties of using Sikuli in practice. First, full-screen matching leads to ambiguity and slow performance. Users need an efficient way to constrain the search space to a certain application. Second, screen matching fails if the target window is occluded by other windows. It would be better if the matching worked even when the target window is only virtually visible. Finally, users need a reliable method to read the text from applications and can accept poor OCR results as better than nothing.

To demonstrate the validity of PAX, we enhanced Sikuli Script by addressing the above issues using the system. First, we added a new *App class,* which manages the information about an application and its window. App provides methods to open, close, and switch focus to a certain application by giving its name or a disk path. Once an application has opened, the corresponding App instance saves a reference to the *UIElement* of that application. Thus, a user may call *app.window(n)* to get the *n*th window as a Sikuli region and then all subsequent pixel computation can be constrained to this region for improved efficiency. This App class uses applications' accessibility information provided by PAX to enable Sikuli Script to constrain the matching area within an application window dynamically instead of using a fixed region on the screen, and also addresses the performance and ambiguity problem.

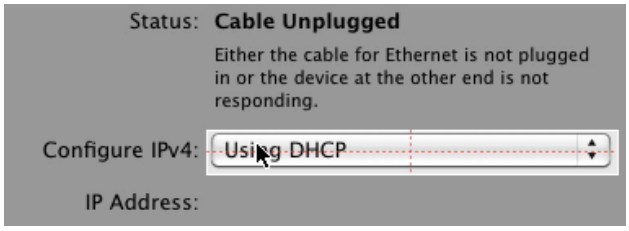


Figure 6 As the cursor moves, the boundaries of the target can be identified automatically in Sikuli Script. After the user clicks to capture a screenshot, the XPath to this target is also stored with the screenshot image for future use.

Second, we enhanced Sikuli's screen capture interface and searching algorithm with PAX. In the screen capture interface, we use PAX's hit testing to automatically identify the target's boundaries as the mouse cursor moves (see Figure 6). A user can simply click on a target to take its screenshot, or use the original method of dragging out a rectangle around the target. When a screenshot is taken, the XPath to the target's UIElement is also saved along with the screenshot (as metadata of the PNG file). Later, when the script is executed, Sikuli Script attempts to find the target with its XPath using PAX first, and then uses the original template matching method if the XPath fails or is unavailable.

Unlike the first enhancement that requires explicit use, this implicitly speeds up the time spent searching for a target and removes the ambiguity. If the complete XPath is not available, because the target widget does not support accessibility APIs, the enhancement still helps because we can at least know which application the target belongs to and constrain the search within the region of that application. This enhancement also addresses the second issue to allow Sikuli to search virtually visible windows, whose screen content can be seen from PAX even when they are overlapped by others.

Finally, we enhanced Sikuli's region-based operation by linking each region with the corresponding UIElements and propagating the value and text from leaf components to their logical group. This allows a script to read text from a region or get the value of a component. For example, *find(*

).value() can be used to read the value of a slider. Similarly, text also can be read by *region.text()*. Although PAX tries to unify pixels and accessibility metadata so that Sikuli users can be unaware of PAX, there are some notable differences when using different source of underlying information together. In the slider example we just mentioned, if the slider exists in the accessibility tree, PAX simply returns its absolute value. How-

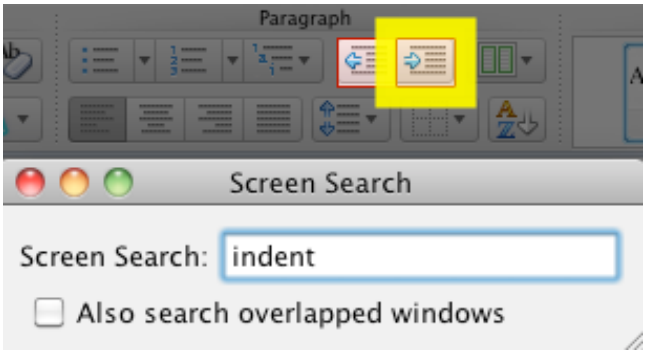


Figure 7 The two buttons with only image icons "Decrease Indent" and "Increase Indent" can be searched with Screen Search.

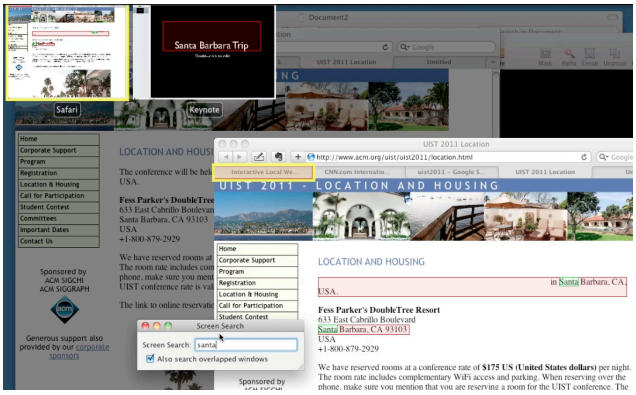


Figure 8 Screen Search finds the given keyword in multiple applications and shows the matched components and their windows at the same time.

ever, if it does not exist, there is no way to read its absolute value from the pixels. In this case, PAX returns a value between 0 and 1 by measuring the distance from the thumb to the two ends of the slider.

Our enhancements have addressed several practical issues in Sikuli Script. At the same time, the readability and the learnability of Sikuli code are preserved.

Screen Search

Search is a common and important feature in almost every application. However, it is usually limited to the application's text content. There is no general method to search the GUI components in a user interface. For example, the text in a text field or in a text area is searchable, but the buttons on a toolbar or the text label on a check box are not. As GUI applications become large and complicated, searching GUI components is especially useful for the users who are not familiar with an interface. For instance, toolbars are widely used in many applications, but the image icons on them are not necessarily easy to understand. In these cases, a user must move the mouse cursor to hover over each button and wait for the tooltip to learn its meaning. The ability

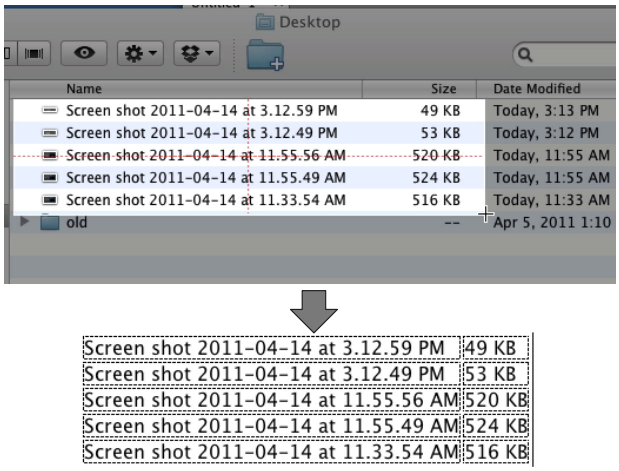


Figure 9 Screen Copy can be used to select and copy columns of a table or a list view.

to search components by their label or description would be a solution for this problem.

We have created Screen Search as a sample application to demonstrate how PAX can support building new interaction techniques on existing user interfaces. Screen Search enables a user to search not only text but also all GUI components on the screen by keywords.

Unlike the usual search bound to a single application, Screen Search is a global function that searches the content and UI of multiple applications at the same time. Furthermore, it allows a user to quickly navigate or switch the keyboard focus to any components found by Screen Search. In other words, this feature enables the use of a keyboard to navigate a user interface in a non-sequential order. For example, a user can search for "indent" to locate the "Increase Indent" and the "Decrease Indent" buttons on a toolbar, and hit the Enter key to select the highlighted one (Figure 7).

Screen Search has two modes: searching only the visible objects (the ones that can be seen on the screen), or virtually visible objects (the ones can be seen or are just overlapped by other windows). Each mode has a different way to present matched objects. Because the matched objects in the first mode are visible on the screen, we simply highlight them using a yellow box. In the second mode, matched objects are not necessarily visible, so we cannot just draw a box at each position. Instead, we draw a thumbnail of each window that has matched objects in a row and a big preview window with the current selected object. The user can press the Tab key to switch the focus among all matched objects and also bring their parent window to the preview position.

With PAX, the implementation of Screen Search is straightforward because PAX has decided the best source for obtaining the metadata of a component. A naïve implementation is calling *findVisibleChildren* or *findVirtuallyVisibleChildren* of the root of the *UIElement* tree, de-

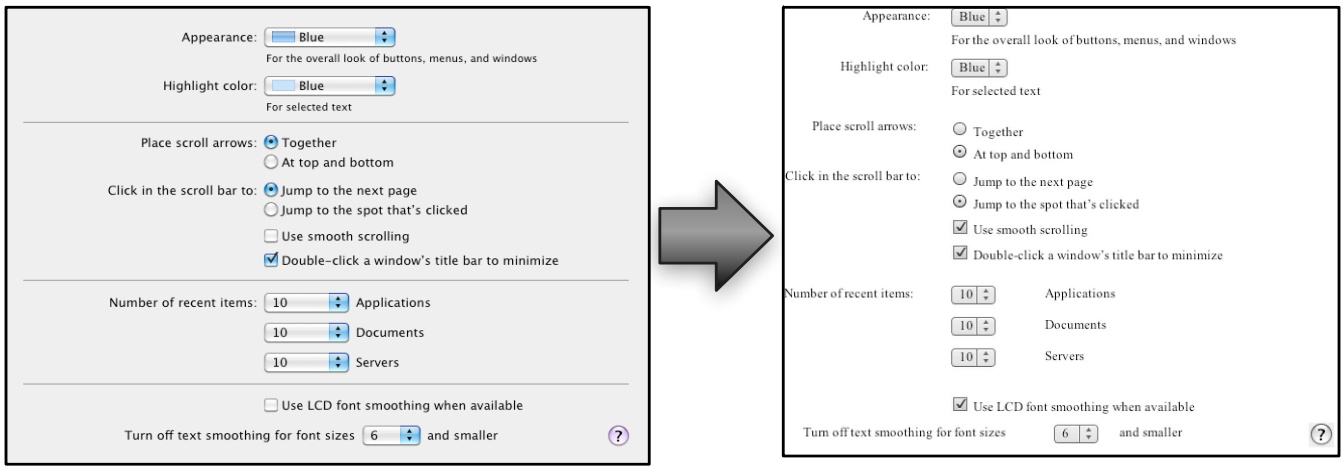


Figure 10 With Screen Copy, one can copy a Mac OS X user interface and paste it into a WYSIWYG HTML editor to create an HTML version of the same interface.

pending on the search mode, to retrieve the matched components in each window. However, to support incremental search, we traverse all nodes in the *UIElement* tree and build an index of the text in each node in a background thread periodically. This requires more complexity but provides a better interface that suggests how many and which objects are matched as a user types.

Screen Copy

Screen Copy is a novel application we have built using PAX. Screen Copy allows a user to select a rectangular area on the screen and copies not only the text but also the GUI widgets within that area into the system clipboard. Figure 10 shows an example where the interface for setting the appearance on Mac OS X can be copied and pasted into a WYSIWYG HTML editor.

Screen Copy does not simply copy GUI widgets independently, but preserves the hierarchy of widgets and their logical grouping. In the example shown in Figure 10, the two sets of radio buttons are correctly grouped together. Thus, only one button in each group can be selected at a time.

Screen Copy is useful for copying an existing interface without its source code and converting it into another format of representation. For example, we could train a library of Flex/Flash widgets using template matching or methods based on machine learning such as Prefab, and then use Screen Copy to convert a Flex/Flash interface to HTML.

Screen Copy also provides a rectangular selection model to existing programs that only have the common text selection implementation. For example, in a web browser, it is very hard to select and copy any non-text objects, such as a table or a form with pictures. The selection is constrained by the flow of text and the underlying structure of HTML. Thus, one can not easily select only one column of a table or two objects across their different containers. However, these can be achieved with the rectangular selection model provided by Screen Copy (see Figure 9).

Screen Copy is straightforward to implement with PAX. Using PAX's rectangular hit testing, the corresponding UIElements within the selected area can be retrieved easily. With the selected objects, Screen Copy transforms each UIElement to HTML tags according to its role and content. Finally, the HTML is copied into the system clipboard and a proper MIME type of the data is set so other applications can then convert it into their own format.

Screen Copy only copies the static interface of an application. It does not copy the dynamic behavior or animated effect on the interface. Additionally, some items that require more interactions to see (such as a drop-down box or a context menu) cannot be copied, so the drop-down boxes in Figure 9 were populated with only the one item that was visible at that time. Currently, as a tool implemented to demonstrate PAX, it does not copy widgets that cannot be represented in standard HTML tags. However, in the future, more complex transformations can be implemented to support non-standard widgets.

CONCLUSION AND FUTURE WORK

We have described PAX, a hybrid framework that uses pixels and accessibility metadata to complement each other. We proposed and evaluated two new algorithms for detecting text on screen and segmenting a text image into word blobs assuming the text is known. We validated our framework by implementing three applications: improving Sikuli Script, Screen Search, and Screen Copy. While promising, PAX has several limitations for future work:

First, in our prototype system, we used template matching to identify GUI widgets from pixels as a proof of concept. The modular design of our framework makes it possible to integrate other powerful pixel reverse engineering methods such as Prefab in the future.

Second, the developers who use PAX need to be aware of the uncertainty due to OCR errors. As future work, more robust OCR algorithms can be integrated with PAX to minimize this uncertainty.

Third, PAX currently uses the accessibility APIs provided by each OS. However, other sources of hierarchical UI representation, such as DOM, can also be integrated nicely with PAX to further improve coverage. For example, on Mac OS X, the built-in browser Safari has implemented a transformation that converts a DOM to an accessibility tree, making this integration possible.

Lastly, PAX currently uses the most common set of accessibility metadata for maximum compatibility on each platform. If more platform-specific metadata and APIs can be used, this opens the door for a diverse body of research. For example, we can add more actions, such as push buttons, or open a drop down menu, to each UIElement, so PAX could be a UI automation framework that automatically uses accessibility APIs or Sikuli Script as its backend. PAX could also support UI customization that provides set methods on each UIElement and draw a customized UI on top of the existing one to show different layout or effects.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and UID group for great suggestions and feedback. This work was supported in part by by the National Science Foundation under award number IIS-0447800 and by Quanta Computer as part of the TParty project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Bezerianos, A. and Dragicevic, P. Mnemonic Rendering. *UIST 2006.*

[2] Chang, T. and Li, Y. Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras. *CHI 2011*.

[3] Chang, T., Yeh, T. and Miller, R. GUI Testing Using Computer Vision. *CHI 2010*.

[4] Dixon, M. and Fogarty, J. 2010. Prefab: Implementing Advanced Behaviors using Pixel-based Reverse Engineering of Interface Structure. *CHI 2010*.

[5] Dixon, M., Leventhal, D. and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *CHI 2011*.

[6] Ha, J. and Haralick, R. Recursive XY Cut using Bounding Boxes of Connected Components. *ICDAR 1995*.

[7] Hurst, A. and et al. Automatically Identifying Targets Users Interact with during Real World Tasks. *IUI 2010*.

[8] Olsen, D.R., Jr, Taufer, T. and Fails, J.A. ScreenCrayons: Annotating Anything. *UIST 2004.*

[9] Potter, R.L. Pixel Data Access: Interprocess Communication in the User Interface for End-User Programming and Graphical Macros. Ph.D. Thesis, University of Maryland at College Park. 1999.

[10] St Amant, R., Lieberman, H., Potter, R. and Zettlemoyer, L. Visual Generalization in Programming by Example. *Communications of the ACM*. 43, 3 (2000), 107-114.

[11] Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*.

[12] Tan, D.S. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *CHI 2004.*

[13] Wachenfeld, S. and Fleischer, S. A Multiple Classifier Approach for the Recognition of Screen-Rendered Text. *Computer Analysis of Images and Patterns*, Vol 4673, 921-928, 2007.

[14] Wachenfeld, S., Fleischer, S. and Klein, H. Segmentation of Very Low Resolution Screen-Rendered Text. *ICDAR* 2007.

[15] Wachenfeld, S., Klein, H.-. and Jiang, X. Recognition of Screen-Rendered Text. *ICPR 2006.*

[16] Yeh, T., Chang, T. and Miller, R. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009.*

[17] Zettlemoyer, L. A Visual Medium for Programmatic Control of Interactive Applications. *CHI 1999*.