Automatically Identifying Targets Users Interact with During Real World Tasks

Amy Hurst *Scott E. Hudson* *Jennifer Mankoff*
Human Computer Interaction Institute, Carnegie Mellon
5000 Forbes Ave, Pittsburgh, PA 15213
{akhurst, scott.hudson, jmankoff}@cs.cmu.edu

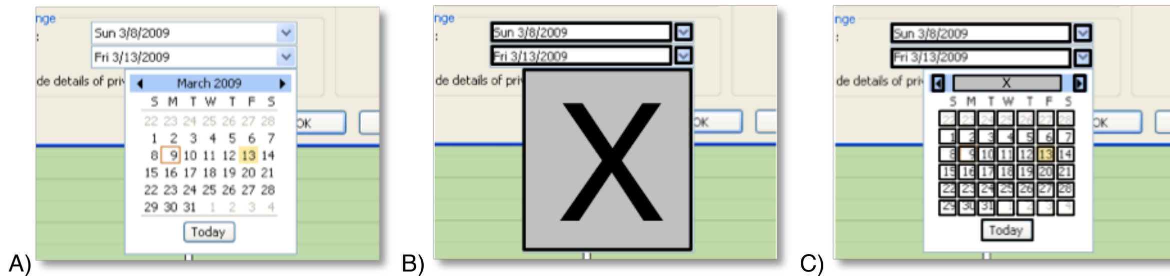

Figure 1: Our hybrid technique is able to identify significantly more targets than the Accessibility API alone. **A)** A screenshot of an open dropdown calendar object in Microsoft Outlook 2003. Interactive targets visible in this image include: two textboxes (with dates); two dropdown handles next to the textboxes; arrows on either side of the month; the month and year; any of the dates in the calendar; and the Today button. **B)** The Microsoft Accessibility API can identify 4 targets correctly (shown with a framed rectangle) and cannot identify 46 targets (shown with a filled gray rectangle). **C)** Our hybrid technique can identify all but one of targets in this example. All of the days of the month and the "Today" button change appearance when clicked on – something we can capture by calculating the difference image; and the arrows on either side of the month can be found with template matching.

ABSTRACT

Information about the location and size of the targets that users interact with in *real world settings* can enable new innovations in human performance assessment and software usability analysis. Accessibility APIs provide some information about the size and location of targets. However this information is incomplete because it does not support all targets found in modern interfaces and the reported sizes can be inaccurate. These accessibility APIs access the size and location of targets through low-level hooks to the operating system or an application. We have developed an alternative solution for target identification that leverages visual affordances in the interface, and the visual cues produced as users interact with targets. We have used our novel target identification technique in a hybrid solution that combines machine learning, computer vision, and accessibility API data to find the size and location of targets users select with 89% accuracy. Our hybrid approach is superior to the performance of the accessibility API alone: in our dataset of 1355 targets covering 8 popular applications, only 74% of the targets were correctly identified by the API alone.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IUI'10, February 7–10, 2010, Hong Kong, China.
Copyright 2010 ACM 978-1-60558-515-4/10/02...$10.00.

ACM Classification Keywords: H5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces.

General Terms: Design, Human Factors

Author Keywords: Computer Accessibility, Usability Analysis, Target identification, Pointing Input.

INTRODUCTION

The ability to analyze user actions in any software environment is necessary to answer research questions regarding software usability, human performance, and how computers are used in daily life. Furthermore, the size and location of targets used in any real world application is crucial to this analysis. Target size and location is necessary to assess the pointing performance of an individual with motor impairments [14], perform usability evaluations of real world software [13], and to compare preference and performance of multiple interactor designs [7]. Many researchers either collect this data in controlled laboratory settings with custom or adapted software [7,14,15] which is easy to analyze, or deploy studies in the real world and capture and hand code the interaction [13]. While frequently more difficult to analyze, data about real world computer use can provide a welcome balance to the more artificial data gathered in laboratory studies. We have developed a technique that enables evaluators to automatically collect the size and location of the full range of targets a user interacted with during real world use.

Accessibility APIs such as the widely deployed *Microsoft Active Accessibility API (MSAA API)* [20] are designed to provide information about interactors in Graphical Users Interfaces and are available on many operating systems and programming platforms. While these APIs can be extremely accurate at identifying some targets, in practice many real world targets are not supported by these accessibility APIs [2]. Unfortunately, this includes targets in commonly used, popular applications such as Microsoft Outlook (Figure 1). Frequently, tools that need this information avoid API limitations by operating in constrained settings where targets are known or have been manually defined. For example, the Eggplant Functional Tester [23] has experimenters manually define a target they are interested in and uses computer vision (template matching) to find all occurrences of that target in a given interface. Another approach is to ignore the API entirely and use only visual information about the interface to identify targets. Seminal work in the area of visual analysis for identifying user interface targets has been done by Amant *et al.* [2, 3, 28] to support cognitive modeling and programming by example tools. Unfortunately, we cannot compare the accuracy of our hybrid technique to their vision-only approach because the accuracy numbers for their evaluations are not available. Recently, Yeh *et al.* [27] have explored using icon matching to identify user-selected targets in a GUI.

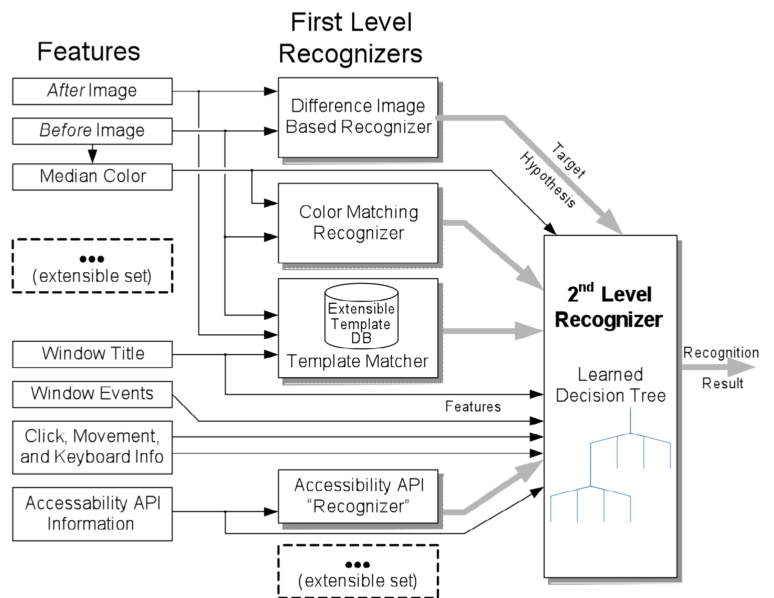

Figure 2: The architecture of our two-level recognizer. Thick gray lines represent target hypotheses, while thin black lines are features. Boxes at left represent feature generators. Middle column boxes are first level recognizers. Right hand box is the second level recognizer (classifier).

The primary contribution of this paper is a technical approach to the problem of finding the size and location of interactive targets from real world interactions that can step in where accessibility APIs leave off. This solution is flexible in that it can work across any application because it leverages visual cues that are ubiquitous across interfaces. We developed this technique using Microsoft Windows XP, but since it requires information that is relatively easy to acquire, it can be extended to other platforms. In this paper we describe how our technique, illustrated in Figure 1, successfully detects targets with 89% accuracy. We use a realistic dataset of 1355 targets gathered from a small representative set of real-world applications. Only 74% of the target selections in this dataset are accessible from the standard accessibility API. In comparing the difference between the size of targets found by the Accessibility API alone to those found by our hybrid technique, we show that the height of the targets found by the hybrid technique are significantly closer to the real size of the targets than the those found by the Accessibility API alone with an average difference of 7.2 pixels (or 19.6% of the average height of all targets in our dataset).

Approach: Leveraging "free" visual cues

When a mouse or other *locator* device is used to select a target, most graphical user interfaces (GUIs) provide standard visual cues (*e.g.,* change the appearance of the item once selected, such as the rectangle around the number 9 in Figure 1A). It is also common to visually change the appearance of the target during the interaction (*e.g.*, change the color of a button when a mouse button is pressed, such as in Figure 3). These visual cues can help the user to interpret the function of the targets or catch errors and misunderstandings.

These cues can also be used to automatically find targets that are not available through the Accessibility API by analyzing screenshots of the interface with computer vision techniques. For example, the change in a button's shadow when a user clicks on it is easily detectable using *Difference Images*. *Template Matching* [6] can use samples of common targets to detect others with similar shapes. *Color Matching* uses a target's color to estimate its boundaries.

Our approach leverages an extensible combination of techniques including computer vision, input event data and accessibility data. Machine learning is used to combine the information provided by these techniques. This allows us to take advantage of their respective strengths, while avoiding their weaknesses, and performs better than any of the individual techniques alone.

Overview

In this paper we first present our hybrid technique to target identification that leverages an accessibility API and three computer vision algorithms. Next we present our evaluation of this hybrid technique on real world interfaces. Finally, we discuss some potential uses for our technique and conclude with future work discussing how this technique could be used in other platforms including mobile devices.

RECOGNITION ARCHITECTURE

Our approach defines *targets* as interactive elements that the user clicks on. This fits the structure of the vast majority of GUIs, but does not support advanced interaction activities such as gesture based interaction or crossing activities [1]. Further, our current implementation does not support dragging (where button press and release events are widely separated). However, the techniques developed here should be able to support more advanced interactions in the future.

As stated earlier, our work leverages computer vision to find targets. Screen images are relatively noise free and well suited to even simple image processing techniques. However, no single technique is capable of identifying all targets with high accuracy. Instead, as shown in Figure 2, we have implemented an extensible set of techniques each of which produces a guess about a probable target (a *target hypothesis*). We then use machine learning techniques to make a choice between these candidate targets.

There is a range of well understood machine learning techniques [21] that can be used to intelligently choose the most likely target from multiple target hypotheses. Generally, these techniques make use of *labeled training data* to create (or *learn*) a statistical model. This learned model *predicts* which hypothesis is correct based on information that describes a particular interaction. This information is expressed in terms of a set of *features* derived from the interaction (such as the window title, input event data about user motions towards the target, or an image of the screen just before the click on the target).

The feature information and a ground truth label indicating the correct size and location of the target are needed for each interaction instance used in training. A learning algorithm processes this training data and produces a *classifier* – an executable entity that can take the features from a new interaction and use those to produce a predicted target size and position.

In our case we have created a two-level classification scheme. At the first level we use a set of classifiers each of which identifies some types of targets well, but others less well. We refer to these as *recognizers* to distinguish them from the *learned classifier* that intelligently picks from among these results based on features from the interaction. We use ADABoosting [8] on a decision tree (created with a variant of the well known C4.5 algorithm [21]) to implement this second level classifier. Our architecture is sufficiently general that first level recognizers can be added over time. Some examples of first level recognizers we have currently implemented include:

Accessibility API: This returns the size and location of the lowest level object under the cursor that is returned from the accessibility API.

Difference Image: Comparison of two images is a standard computer vision technique [6]. Visual feedback can be captured by comparing the screen image taken just before the locator button press event to a screen image taken just before the locator button release event. The bounding box around those screen pixels that are different is a potential target.

Color Matching: After blurring an image, the target often stands out as a region of color different from the surrounding image. This is especially so when the target is highlighted in response to the user's click. Standard computer vision techniques can be used to find the boundaries of regions of color that match the color under the cursor at the time of the click.

Template Matching: A known target can be used as a pattern or template by breaking it into its constituent corners and edges so that it can be compared to the image of an unknown target. This enables a single template to help find the borders of many similar targets of varying sizes.

IMPLEMENTATION

Data collection

The first piece of our target finding system is the capture component. CRUMBS (Capture Real-world User Mouse BehaviorS) captures information about interaction activities including (1) all locator, keyboard, and window events (2) any accessibility API information that is available and (3) two 300x300 screen captures that are centered on the cursor during a button event.

CRUMBS, written in C++, is based on DART (Disruption Awareness and Recovery Tracker) [11]. CRUMBS probes the MSAA API for the type and size of interactive targets. An API hook is used to take a 300x300 pixel screen capture (centered on the cursor) immediately before each button press event is dispatched, and immediately after each release event is dispatched. This limit is set for efficiency reasons, and because it is large enough to capture most targets (92% in our data set). If the cursor moves between press and release the second image is centered on the new cursor location. CRUMBS has been deployed in the field continuously since January 2008 and can also be used for laboratory data collection.

Two level classifier

Our implementation leverages the Weka toolkit [26], and we use the C4.5 Decision Tree learning algorithm [21] with binary splits (which suit our data well as it is a mix of numeric, nominal, and string features). The data used for training is hand labeled (details on the training data and results are given in the validation section). Our implementation currently runs offline, but could easily be augmented to run as data is collected. Below we describe various components of the implementation.

Feature Generation

We use features as a way to describe an interaction. Features are used by the first level recognizers to create their hypotheses, and potentially again by the second level classifier to help choose the correct target size or location from among these hypotheses. Features reduce the complexity of the decision problem while also highlighting the most important information. We extract the following features for each interaction:

Accessibility API features: The Accessibility API reports the size, location and type of the lowest level accessible object under the cursor.

Locator features: For each interactive sequence (starting from the previous sequence's release event and ending after the user presses and releases the locator button in the current sequence), the following features are reported:

Click features: Which locator button was pressed during the current press/release; click duration (time the locator button was held down between the current press and release); and distance slipped (amount of movement between locator button press and release).

Movement features: Euclidean distance moved (between the previous release and the current press); total distance moved (along the actual path of the locator during the same period); duration of movement (from the previous release until the current press); average velocity of movement (same period); and total sequence duration (time between the previous release and the first movement after the current release).

Keyboard features: Because it sheds light on what the target was used for, in the case of keyboard activity we are interested in what happens after the release rather than leading up to it. Consequently, keyboard features are measured from current release through the press of the next sequence. These include: a count of the number of keys pressed, a count of various special keys (including arrow, letter, number, enter, tab), a string of all keys pressed, and the time before first key press (after the click). Note that because this feature makes use of information occurring after the target is selected it will not be suitable for use in some real-time applications. As shown in our results (Figure 10), these features turned out not to be of high value for target finding, meaning that applications that operate in real time could operate without them.

Window event features: Like keyboard features, information about the window in use sheds light on what a target might have been selected for. The window title of the window currently in focus is reported as a feature. Window events that take place immediately after the locator button is released are also important indications of the user's goals. Any window events, including window title and whether it was minimized, maximized, gain focus, *etc.* are reported. These may occur anytime between the current release and the first move of the next sequence.

Additional possible targets: Each first level recognizer returns its best estimate for target size and position. However, additional features are generated indicating the presence of "second best" targets, the total number of targets found, and how many of the viable targets contain the point where there locator button press occurred.

Visual features: Features are generated indicating whether only one or many areas of visual change were present, the median RGB color value near the button press, the presence of a specific known interactor (as detected by a template), and whether that template was fixed or variable in size.

First Level Recognizers

The features described above feed into an extensible set of *first level recognizers*. Each first level recognizer is responsible for providing a hypothesized target size and position (or an indication that it finds no viable target) from those features. Details for each of our current recognizers follow.

Accessibility API recognizer

This first level recognizer simply returns a hypothesis matching the best information available from the Accessibility API. Specifically, the size and location of the lowest level accessible object under the cursor, during a pointing device's button press, is used as the target hypothesis.

For the work described here we make use of the *Microsoft Active Accessibility API (MSAA API)* and associated OS hooks. This is a cross-application Windows operating system level solution for getting low-level information about targets including push buttons, menus, textboxes and html links.

Limitations of the MSAA API: As previously stated, not all interactors are supported by the MSAA API. Examples of some of these unsupported targets include the entire editing area of Microsoft PowerPoint, the full playing area of Windows games such as Solitaire and Minesweeper, and specialty dialogs including the Character Map, custom color selector panels, and the Microsoft Paint controls. In addition, not all applications support the API in the same way. For example, two popular web browsers, Microsoft's Internet Explorer and Open-source Firefox, treat content in a very different way, limiting the API's access to them. Internet Explorer treats embedded Flash applets in such a way that the API is able to access most of the targets, however Firefox embeds Flash in such a way that the targets are not accessible through the API by clicking on them.

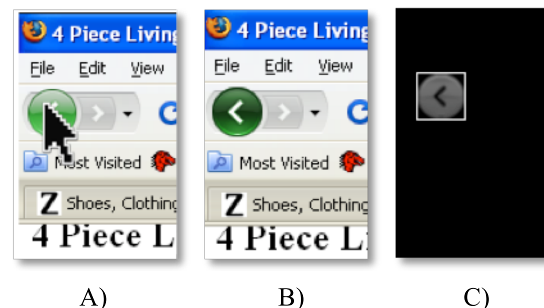

Figure 3: Selected Target found by calculating the difference between images taken during a locator button's press and release events. **A)** The user is pressing on the "Back" button in Firefox (the cursor here is for illustration only and not captured in our data). **B)** The image captured after the user releases the button. **C)** After subtracting A from B, the resulting difference is the correct target (shown with a bounding box).

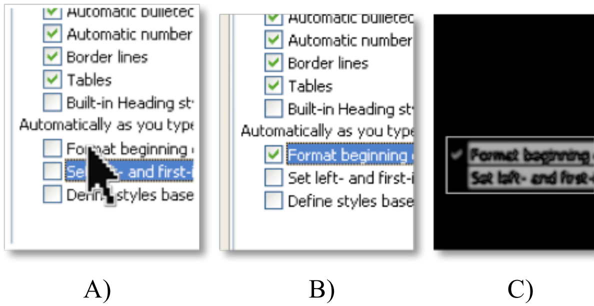

Figure 4: Overlap error with difference imaging: The difference image accidentally joins the targets because the top edge in **A)** overlaps with the bottom edge in **B)**. **C)** shows the resulting target which is too big.

Figure 5: Visual change error in difference imaging. New visual content appears that overlaps or is near the target. **A)** Tooltip in the press image very near target. **B)** Tooltip is not present in release image **C)** Resulting difference region includes both the target and the tooltip, which is incorrect.

ing. Difference imaging failed to find this target because images captured from the button down and button up events were identical. **A)** Original image. **B)** Image after median blur. The color over the cursor (dark blue) is selected for color matching **C)** The matching algorithm selects the smallest blob of that color over the cursor (the correct target).

Difference image recognizer

A difference image is a subtraction of all the pixel values in two images [6]. Targets that produce a salient visual change upon interaction may be detected by calculating the difference between the image before the press event of a click is dispatched and the one found after the release event is dispatched.

Recall that CRUMBS captures an image immediately before the mouse press and immediately after the mouse release in an interaction sequence (Figure 3 A and B). Difference imaging compares these two images. Since it is common for users to *slip*, or accidentally move the pointing device a few pixels during the click, we automatically align the images according to the amount and direction of motion between press and release. Any overlap is cropped so that both images are the same size. At this point, a Gaussian blur is applied to both images. Applying this blur helps minimize the impact of very small visual elements (such as letters) that are really part of a larger targets (such as a button).

After the images have been prepared, we iterate through each image and subtract the pixel values from one image to another at the corresponding locations. This subtraction yields an output image of only the differences between the two images. Next we apply a connected component filter [6] to group changes into contiguous regions (or *blobs*). The results of the connected component filter are analyzed to find the bounding box of the smallest sufficiently large blob (at least 5x5 pixels) that overlaps the cursor position. If no blobs are sufficiently large then the bounding box of all pixels that differed between the two images is returned.

This technique, along with template matching and color matching techniques, is written using a combination of the JAVA Advanced Image library [12] and ImageMagick [10].

Limitations of the difference image recognizer: If there was only one region of visual change (such as in Figure 3) and it is sufficiently large (either its height or its width is greater than 5 pixels), then the bounding box of this region is given as the hypothesis target size. However, sometimes multiple visual changes occur, and the boundaries of these targets overlap in the difference image. For example, Figure 4 shows how difference imaging fails when the boundaries of the two targets touch after blurring.

Difference imaging also has difficulty when the button press triggers additional animations or otherwise changes the targets. Examples include displaying tooltips, non-target interactors losing their focus, or visual changes unrelated to the pointing device occurring. Figure 5 illustrates when a tooltip window appears too close to a target.

Color matching recognizer

The color matching technique searches for a color region in the image associated with the current sequence press event that could indicate the bounds of a target (Figure 6). The image is prepared using a median blur filter of 10 pixels (Figure 5B). This has a similar effect to the Gaussian blur used in difference imaging – removing small differences – but tends to choose a single color rather than continuously varying one. Once the image is blurred, the color of the pixel directly under the cursor (at the center of the image) is used as the color to match. The match algorithm may return multiple regions. These regions are handled similarly to difference imaging (the bounding box of the smallest region greater than 5 pixels in either height or width is selected). If no blobs are sufficiently large, a bounding box around all matched pixels is used.

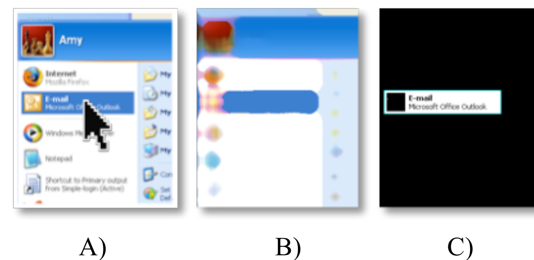

Figure 6: Selected target identified using color matching. Difference imaging failed to find this target because images captured from the button down and button up events were identical. **A)** Original image. **B)** Image after median blur. The color over the cursor (dark blue) is selected for color matching **C)** The matching algorithm selects the smallest blob of that color over the cursor (the correct target).

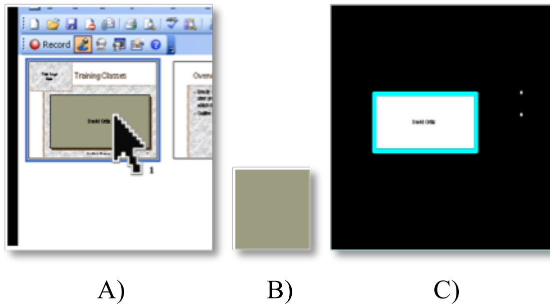

Figure 7: Example of false positive from color matching from Microsoft PowerPoint. Target in image **A)** was the thumbnail of slide 1 which is highlighted by the interface with a blue square. **B)** Color chosen to use for color matching after applying median blur to A. **C)** Results of color matching. In this example, choosing the color of the center pixel after applying the median blur yielded an incorrect result because the bounds of the selected color were much smaller than the target size.

Limitations of the color matching recognizer: This technique complements difference imaging because it works even when there is no visual change. However it currently does not support targets with a mutli-color background (such as a gradient or shading as found on the Microsoft Windows Start Button). Additionally, it does not support targets that are not surrounded by a differently colored background (such a list items or checkboxes that share the same background). Figure 7 illustrates an example where color matching failed to identify the correct target because it matched a color that was connected to a smaller shape within the target.

Template matching recognizer

Template matching is a relatively simple computer vision technique that matches a prepared set of pattern or *template* images against a source image (captured by CRUMBS) and finds occurrences of the template image within the source image (Figure 8). Given the image of a target, it can be used as a fixed sized image (or icon) template. It can also be processed into a variable sized template by automatically extracting the corners and edges of the icon. By convolving the source image with a template, it is possible to quickly identify all regions where template pixels match the target image.

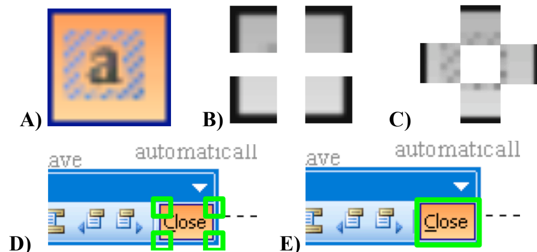

Figure 8: Example of corner template matching. Where template **A)** is used to identify target in **E)**. **B)** Corner templates: 5x5 pixels. **C)** Edge templates 6x6 Pixels. **D)** Source image with found corners marked with squares. **E)** Source image with line around found target. Note that this technique is size independent, as the aspect ratio in **E)** is different from **A)**.

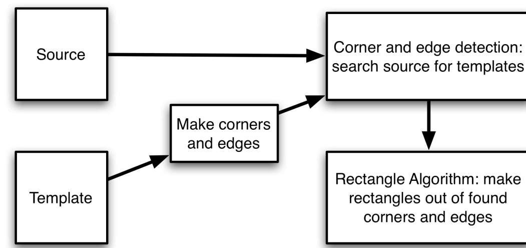

Figure 9: Corner and edge template matching algorithm takes in source and template images, extracts the edges and corners from template, and searches source for template corners and edges. Finally rectangles are formed from matched corners and edges.

Systems using the techniques described here should be delivered with a set of templates designed to match many common toolkit components appearing on various platforms. Templates can then be added to the template set on an as needed basis. We envision this happening when users detect and correct errors made by the system. In particular, when errors are reported, the user might be asked to correct or disambiguate [16] the system by drawing a bounding box around the correct target. The system may also ask the user to indicate which if any part of the current window title string is indicative of the application being used (see discussion of application groupings below). The training and validation section below will discuss our investigation of the effects of adding more templates on accuracy.

Template matching is a multi-step process. First we check for an exact (icon) template match on both screen captures associated with the interaction activity. Next, the template matching algorithm searches for variable size matches by comparing each corner and edge of the template to the corresponding region around the spot where the locator button click occurred (Figure 9). Additional corners are created by connecting any two strongly matching adjacent edges even where a corner was not found. The algorithm takes all corner template matches and corners created from adjacent edges, and searches this set for possible resulting rectangles. Finally, the smallest rectangle is returned as the resulting target hypothesis.

The sets of possible templates to match against are grouped by application or application class. Applications are determined by examining window titles (which frequently include the application name and the active file name) using application indicator strings delivered with the system or subsequently identified by end users. For efficiency reasons, we restricted the list of potential templates matched based on the indicated application. Window titles that do identify a recognized application are put into an “unknown title” category.

Limitations of the template matching recognizer: Fixed size (*icon matching*) works best on icons or unique targets that don’t have a border. Variable size (*corner and edge matching*)

Application	% of data
MS Outlook	39.9%
Web Browsing (IE and Firefox)	23.5%
MS Word	12.2%
Windows Explorer (files & customization)	9.3%
Media Player	7.6%
MS PowerPoint	7.6%

Table 1: Distribution of test data by application

works best on targets with clear borders and is able to handle items that differ in size from the original template, something icon matching cannot do. Thus the two techniques complement each other. However, both have difficulty finding the correct target in cases where an animation has not completed when the screenshot is taken, because in that case the screenshot may not contain the final image that the template matches.

TRAINING AND ACCURACY VALIDATION

Recall that the creation of a classifier depends on a training stage in which labeled data is needed. In our case, the data is example interaction activity, and the labels are the correct target size and location for each activity. Here we will describe how we acquired this training data, how the classifier was trained, and present results of accuracy tests.

Data acquisition

Although the CRUMBS data acquisition component of our system has been field deployed for a year as a part of a related project, the deployment setting is not Internet enabled and targets a population of users with motor impairments who spend much of their time playing games on the computer. While this data set is interesting for many reasons, we did not feel that it was sufficiently diverse to demonstrate our technique.

To account for this, we generated what we believe to be a realistic and representative training and test corpus. We carried out the exact steps specified in a set of tutorials for Windows and Microsoft Office products (called the Step By Step Tutorials). These tutorials give clear, step-by-step instructions for how to accomplish common tasks. Because they provide instructions for what to do, they remove any potential bias on the part of the experimenters in selecting targets. Because they focus on common tasks for common products, they provide reasonably representative data. We used portions of the following tutorials to create our initial dataset: Microsoft Office Outlook [17], Microsoft Office: PowerPoint 2003 [18], Microsoft Office: Word 2003 [19], and Learning Windows XP [24].

1. Window title
2. API result type
3. Width of target found by Template Matching
4. Height of target found by Template Matching
5. Area of target found by Template Matching
6. Color used for color Matching
7. Technique (connected component, or bounding box) used to analyze blobs in difference image
8. Velocity before button press
9. Distance moved before button press
10. Time elapsed between last button release and next movement
11. Count of valid targets found by difference image
12. Area of target found by difference image
13. Width of target found by difference image
14. Height of target found by difference image
15. Count of occurrence of a window focus event immediately after the button release.

Figure 10. Top 15 features used as by the second level classifier, sorted by information gain.

Data from Internet interactions was added to this set by having our colleagues interact with several of the websites on the Alexa.com Top sites in the United States. Our participants selected websites they used normally from our list of top websites, and performed tasks they would normally do on them. They chose to use 7 of the 10 most popular sites (as of March 2009): google.com, yahoo.com, facebook.com, youtube.com, Wikipedia.com, eBay.com, craigslist.com. The relative proportion of interactions from each application is shown in Table 1.

Next we segmented the data into interaction sequences ending with a click. Since our initial implementation explicitly does not support drag interactions, and training on them would produce anomalous results, we also removed any interaction activity with a movement between press and release longer than 10 pixels. (We should be able to remove this limitation and consider drags in a later release.) Clicks that could not be assigned to a specific target after careful human examination of the data were also removed; these were frequently focus clicks or accidental clicks. The result was a data set of 1355 interaction sequences. These were processed offline to generate the features needed for the training phase.

Labeling of ground truth

To establish labels for training, our dataset was hand coded by one of the authors by selecting the "click sensitive" screen region for each target within the images captured with our data. The distribution of interactor types which make up targets in our data is shown in Table 2.

Training

The Weka machine learning system [26] was used to create the classifier using the ADABoost.M1 algorithm [7] with 10 iterations using the C.45 decision tree algorithm [21] as its base classifier. Overall, on average, the decision tree for each iteration from training on our interactive target labels contained 136.2 decision nodes. Figure 10 indicates the top 15 features used in the resulting classifier, ordered by information gain.

Interactor Type	% data	Interactor Type	% data
pushbutton	50.1%	tree view item	1.0%
menu item	9.8%	outlook date item	0.9%
list item	8.4%	table cell	0.5%
text link (webpage)	7.2%	window (focus click)	0.5%
image link (webpage)	6.3%	combo box	0.4%
text box (1 line)	4.4%	icon	0.4%
check box	3.2%	radio button	0.4%
scroll bar arrow	2.5%	desktop	0.2%
tab	1.3%	equation	0.1%
text (single word)	1.1%	spin box	0.1%
text area (multi-line)	1.0%		

Table 2: Distribution of interactor types acting as targets in the training dataset.

Accuracy results

In assessing accuracy we considered the size and position of each classification made by the system. We considered a classification to be correct if it contained the click point of the interaction and was within 15% or 5 pixels (whichever was larger) of both the width and height of our human labeled ground truth.

To estimate the overall accuracy of the resulting classifier we used the common 10-fold stratified cross validation method – we estimate the accuracy of the final classifier as the average accuracy found in 10 classifiers, each built from 90% of the test data and tested against the remaining 10%. Our tests indicate an overall accuracy of 89% (kappa = .8), and Figure 11 illustrates the overall accuracy of our hybrid approach compared to the accuracy of each individual recognizer.

As mentioned earlier, the current implementation of CRUMBS logging software only captures images that are 300x300 pixels. Not only does this cover 92% of our targets, but for a number of applications, the details of large targets are of less interest (*e.g.,* they are easy to select). However, this limitation does cause some misclassification, since in the end 7.9% of the targets in our dataset ended up being larger than 300 pixels in some dimension (and so are clipped in the images our classifier uses). Of these, 2.6% are actually misclassified (with the remaining 5.3% being correctly identified by the API without the need for images). As a result, it may be possible to improve overall accuracy by capturing larger images if the performance issues regarding taking full-sized screenshots can be resolved.

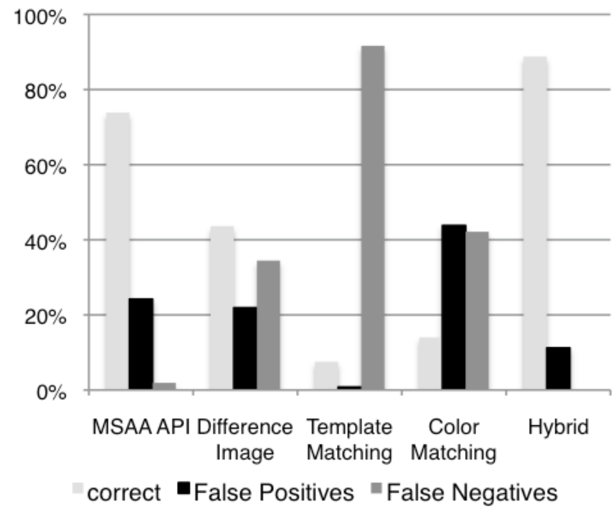

Figure 11: Raw accuracies of four first level recognizers (API, Difference Image, Template Matching, Color Matching) and the overall accuracy of our hybrid technique.

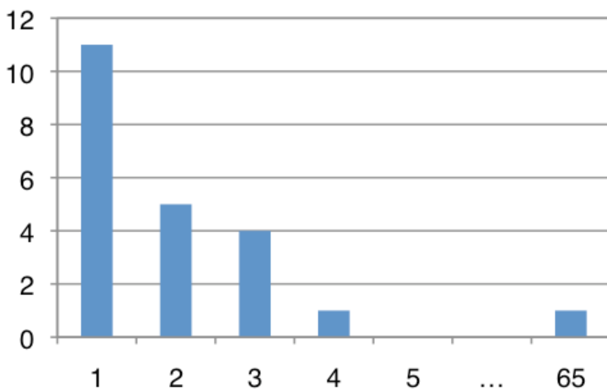

Figure 12: Distribution of number of targets covered by a single template. Most templates (11) found only one target (the one they were created from). The most successful template of the 22 we created found 65 targets, and the rest found four or fewer targets. While it is true that a relatively small number of targets is found by each template, the cost of creating each one is small.

Because templates may be added over time, it is useful to consider the impact of templates on overall accuracy. Our intuition was that a fairly small number of templates will provide sufficient accuracy benefits, and that this will allow us to distribute a manageable library of initial templates with a system and still get high accuracy from the beginning.

To create an initial set of templates, we considered each target which failed the preliminary version of the system without template matching. For each of these failures we semi-automatically created a template. In particular, we manually outlined the bounding rectangle of the target -- much as a user might in providing correction feedback -- then the system automatically extracted corner and edge images to create a template. As we progressed through 154 initial error cases (in random order) we eventually created 22 distinct templates. These templates successfully covered 102 of the initial 154 error cases, illustrated in Figure 12.

DISCUSSION

Based on systematic tests with realistic data, our results demonstrate the viability of a hybrid approach to finding the location and size of targets. Even though our algorithm is relatively computationally simple, it managed to correctly identify most of the targets in our dataset. As illus-

trated in Figure 11, our hybrid approach was more accurate than any of our first level recognizers -- only missing 11% of targets, compared to lower accuracy for all of the component first level recognizers it is built from, including the accessibility API. This suggests that our overall approach might be further improved by adding additional first level recognizers, even if those recognizers don't themselves perform better than our current system. For example, we might include rule-based techniques such as those of Amant *et al.* [2,3,28].

Another addition to our technique would be to design first level recognizers to handle targets that demonstrate systematic types of errors. For example, one common error in our data set involved targets with multiple unconnected visual components, such as checkboxes with adjacent clickable labels. When the user clicked on either the checkbox or the label, the label received a blue highlight, and a check was added to the box. Our difference image algorithm had difficulty with these targets because it tended to see them as separate visual elements. Template matching is best suited to find icons, so it could find the checkbox but not its accompanying label. Finally, color matching identified the label, but missed the checkbox. Based on these findings, a combined target hypothesis created from the results of two first level recognizers may be a better hypothesis than either recognizer can provide alone.

Another error that happens systematically in our data is caused by the choice to capture only 300x300 pixel images. Although we did this for efficiency reasons, as mentioned earlier this choice is responsible for 2.6% of our errors. By adjusting the capture size based on runtime information from the API to extend to the full width of the proposed API target, we could make it possible to eliminate many of these errors.

APPLICATIONS AND FUTURE WORK

Identifying the size and location of targets interacted with during real world use can enable novel additions to existing technology. Below we present three areas that can benefit from accurate target identification of user interfaces.

Improving Computer Accessibility

Having more reliable information about the size and location of targets a user interacts with provides opportunities to assess and adapt to performance metrics during real world use. One approach to assess pointing accuracy is to use metrics drawn from characterizations of data with respect to Fitts' Law [5]. Historically this metric has been explored in the laboratory, but our hybrid technique could expand on research that looks at this metric during real world use [4]. Additionally, there are many target-specific pointing performance metrics that have yet to be studied out of the laboratory [13,15]. Finally, knowing the size and location of targets interacted with could greatly expanded the ability to understand real world pointing performance of motor impaired individuals [9].

We are interested in using automatic assessments of pointing performance to suggest adaptations to a computer environment to increase computer accessibility. For example, accurate information about target selection performance could help identify when a user slips off a target during a click. With accurate assessment of this pointing problem, a software adaptation that froze the locator position during a click, such as Steady Clicks [24], could be deployed.

Supporting Automatic Extraction of a Task Sequence

In usability evaluations, having a list, or *task sequence*, of the targets a user interacted with to achieve a goal is desirable. Task sequence can be analyzed to reveal differences in the steps a user performed and a pre-defined sequence, or to compare the actions of multiple users. Experimenters typically generate a task sequence through logging software built into a custom application [7] or by hand coding video logs [13]. Our technique could help usability specialists by automatically extracting these sequences from real world use, and be used in existing video annotation tools such as Transana (http://www.transana.org).

Automatically Scripting Common Actions

During real world use, it is common to encounter repeated sequences of UI interactions. Tools such as Automise (http://www.automise.com) enable users to easily generate scripts to perform multiple UI tasks. We envision automatic target identification being incorporated into these tools to automatically identify targets in frequently performed task sequences and suggest scripts to automate these tasks. Yeh *et al.* have begun to explore this domain with Sikuli, which allows users to graphically write GUI automation scripts [27].

Future Work

In future work, we plan to leverage our results to improve computer access in real world interactions. Specifically, we plan on using our hybrid technique to identify the size and locations of real world targets encountered in daily life to automatically assess the pointing performance of individuals with pointing problems.

Our solution is currently limited to Windows platforms because CRUMBS only works with the MSAA API. However, the visual characteristics of interaction that we leverage are not limited to the Windows operating system. Other platforms (including mobile phones) utilize the same visual affordances our technique is designed to detect: strong visual changes on interaction, color highlights, common borders, and repeated graphics. As future work we would like to develop our technique to other platforms, using their respective accessibility APIs.

CONCLUSIONS

We have presented a new application independent technique to identify targets a user interacts with. This technique uses a hybrid of accessibility API information and of several computer vision techniques to identify these targets. In a dataset that was collected from realistic interactions with real applications, our technique was able to identify 89% of the targets, and the underlying accessibility API could only identify 74% of these targets.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants EEC-0540865, IIS-0713509, IIS-0325351, IIS-0205644, the first author's NSF Graduate Student Research Fellowship; IBM Research; and the Pennsylvania Infrastructure Technology Alliance.

REFERENCES

1. Accot, J. and Zhai, S., (2002), More than dotting the i's – foundations for crossing-based interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems,* ACM Press, pp. 73-80.
2. St. Amant, R., Lieberman, H., Potter, R., and Zettlemoyer, L., (2000), Programming by example: visual generalization in programming by example. In *Communications of the ACM*, ACM Press, 43, 3, pp. 107-114.
3. St. Amant, R. and Riedl, M. O., A perception/action substrate for cognitive modeling in HCI. In *International Journal of Human-Computer Studies*, Elsevier, 55(1), pp. 15-39.
4. Chapuis, O., Blanch, R., and Beaudouin-Lafon, M., (2007), Fitts' Law in the Wild: A Field Study of Aimed Movements. Technical Report n.1480, LRI, Univ. Paris-Sud, France, December 2007, 11 pages.
5. Fitts, P.M., (1954), The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. In *Journal of Experimental Psychology,* 47, pp. 381-391.
6. Forsyth, D. and Ponce, J., (2002), *Computer Vision – A Modern Approach,* 1st edition, Prentice Hall.
7. Findlater, L. and McGrenere, J., (2004), A comparison of static, adaptive, and adaptable menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, pp. 89-96.
8. Freund, Y. and Schapire. R.E., (1996), Experiments with a new boosting algorithm. In *Proceedings of Machine Learning*, Morgan Kaufmann, pp. 148-156.
9. Hurst, A., Mankoff, J., and Hudson, S. E., (2008), Understanding pointing problems in real world computing environments. In *Proceedings of the ACM SIGACCESS conference on Computers and Accessibility*, ACM Press, pp. 43-50.
10. ImageMagick software suite: http://www.imagemagick.org (Accessed 09/17/09)
11. Iqbal, S. T. and Horvitz, E., (2007), Disruption and recovery of computing tasks: field study, analysis, and directions. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM Press, pp. 677-686.
12. JAVA Advance Imaging, Template Matching https://jaistuff.dev.java.net/ (Accessed 09/17/09)
13. Kaufman, D. R., Patel, V. L., Hilliman, C., Morin, P. C., Pevzner, J., Weinstock, R. S., Goland, R., Shea, S., and Starren, J., (2003), Usability in the real world: assessing medical information technologies in patient's homes. In *Journal of Biomedical Informatics,* 36, 1/2, pp. 45-60.
14. Keates, S., Hwang, F., Langdon, P., Clarkson, P. J., and Robinson, P., (2002), Cursor measures for motion-impaired computer users. In *Proceedings of the ACM SIGACCESS conference on Computers and Accessibility*, ACM Press, pp. 135-142.
15. MacKenzie, I. S., Kauppinen, T., and Silfverberg, M., (2001), Accuracy measures for evaluating computer pointing devices. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems,* ACM Press, pp. 9-16.
16. Mankoff, J., Hudson, S. E., and Abowd, G.D., (2000), Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, ACM Press, pp. 11-20.
17. *Microsoft Office Outlook 2003 Step By Step*, Microsoft Press, 2004.
18. *Microsoft Office Powerpoint 2003 Step By Step*, Microsoft Press, 2004.
19. *Microsoft Office Word 2003 Step By Step*, Microsoft Press, 2004.
20. Microsoft Active Accessibility API, http://msdn2.microsoft.com/en-us/library/ms697707(VS.85).aspx, accessed 9/17/09.
21. Mitchell, T., (1997), *Machine Learning*, McGraw-Hill.
22. Quinlan, J.R., (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufmann.
23. *Testplant Software,* Eggplant Functional Tester, http://www.redstonesoftware.com/, accessed 9/17/09.
24. Trewin, S., Keates, S., and Moffatt, K., (2006), Developing steady clicks: a method of cursor assistance for people with motor impairments. In *Proceedings of the ACM SIGACCESS conference on Computers and Accessibility*, ACM Press, pp. 26-33.
25. *Windows XP Step By Step,* 2nd edition, Microsoft Press, 2005.
26. Witten, I.H. and Frank, E., (2005), *Data Mining: Practical machine learning tools and techniques,* 2nd Edition, Morgan Kaufmann.
27. Yeh, T., Chang, T., and Miller, R. C., (2009), Sikuli: using GUI screenshots for search and automation. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, ACM Press, pp. 183-192.
28. Zettlemoyer, L. S. and St. Amant, R., (1999), A visual medium for programmatic control of interactive applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, pp. 199-206.