

A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces

Morgan Dixon,¹ James Fogarty,¹ Jacob O. Wobbrock²
¹Computer Science & Engineering, ²The Information School

DUB Group, University of Washington

mdixon@cs.washington.edu, jfogarty@cs.washington.edu, wobbrock@uw.edu

ABSTRACT

We present a general-purpose implementation of a target-aware pointing technique, functional across an entire desktop and independent of application implementations. Specifically, we implement Grossman and Balakrishnan's *Bubble Cursor*, the fastest general pointing facilitation technique in the literature. Our implementation obtains the necessary knowledge of interface targets using a combination of *pixel-level analysis* and *social annotation*. We discuss the most novel aspects of our implementation, including methods for interactive creation and correction of pixel-level prototypes of interface elements and methods for interactive annotation of how the cursor should select identified elements. We also report on limitations of the Bubble Cursor unearthed by examining our implementation in the complexity of real-world interfaces. We therefore contribute important progress toward real-world deployment of an important family of techniques and shed light on the gap between understanding techniques in controlled settings versus behavior with real-world interfaces.

Author Keywords

Target-aware pointing, pixel-based reverse engineering, Bubble Cursor, Prefab, social annotation, real-world interfaces.

ACM Classification Keywords

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

INTRODUCTION

The human-computer interaction literature includes many promising techniques for *target-aware pointing*, including deep studies of specific characteristics of those techniques in controlled laboratory settings. This important family of techniques can improve pointing for a variety of people on a range of devices in many applications. Target-aware techniques can significantly outperform other pointing facilitation techniques, and they ultimately have great potential to improve the efficiency of interaction.

Despite the promise of these techniques, few have been deployed or evaluated in real-world interfaces. The impact of target-aware pointing, and our understanding of its

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.

CHI '12, May 5–10, 2012, Austin, Texas, USA.

Copyright 2012 ACM 978-1-4503-1015-4/12/05...\$10.00.

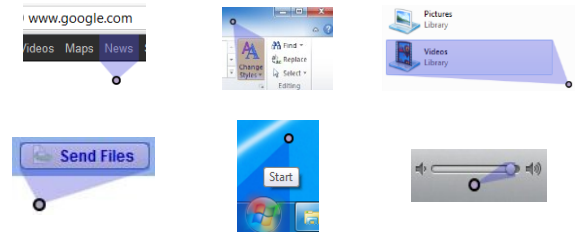


Figure 1: We present a practical implementation of a general-purpose target-aware pointing enhancement. Specifically we implement Grossman and Balakrishnan's Bubble Cursor across the Windows 7 desktop.

effectiveness, is currently limited by two challenges to *implementing* target-aware pointing in real-world interfaces.

First, external pointing enhancements often cannot obtain reliable information about the size and location of interface elements [9,22]. Accessibility APIs attempt to provide some of the necessary information, but are inevitably incomplete due to developer failures to implement the API. For example, Hurst *et al.* found 25% of widgets are completely missing from the accessibility API [22]. The API also exposes widget *models*, not necessarily their on-screen view (e.g., the pixel coordinates of a slider's thumb are intentionally encapsulated). Techniques for dynamic code modification can also often be made to work in a single application or toolkit [13,14,32], but are generally too brittle for enhancement of the full desktop. Code injection techniques also sometimes fail (e.g., when a skinnable application uses pre-rendered images of widgets instead of meaningful graphics operations). Failures at these levels of applications can only be corrected by their developers, so many interface elements remain opaque.

Second, even a complete enumeration of interface elements is insufficient for determining how a targeting enhancement should behave. Studies of pointing facilitation techniques generally treat an interface as a field of abstract targets (e.g., gray circles), but the notion of a "target" is often not well defined in real-world interfaces. Targeting ambiguities are presented by calendars, paint canvases, text fields, and many other standard and custom widgets [41]. It is difficult or impossible for the developer of a general-purpose enhancement to foresee and address all such ambiguities.

We address these challenges in a general-purpose implementation of Grossman and Balakrishnan's *Bubble Cursor* [18], an area cursor that dynamically expands to

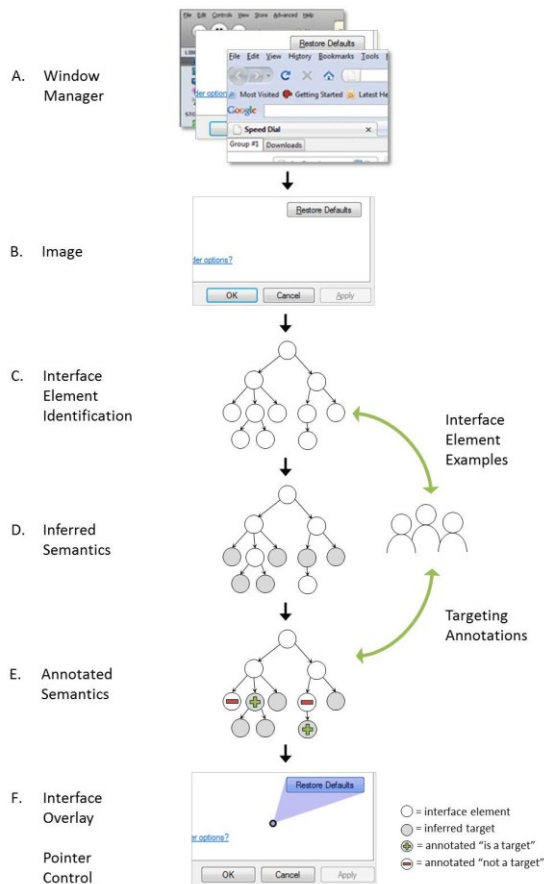


Figure 2: We implement a general-purpose Bubble Cursor in a novel architecture that combines pixel-based *identification* of interface elements with *interpretation* of their targeting. We emphasize a *human-driven* approach with interactive extension and correction of both implementation layers.

always capture the nearest target. Our implementation functions across the Windows 7 desktop, as illustrated in Figure 1. The methods we develop can be applied to any modern desktop, and they can be extended to support other target-aware techniques. Specifically, we architect our enhancement to separate *identification* of interface elements from *interpretation* of how to target those elements. We then implement identification using Prefab’s pixel-based methods for reverse engineering interface structure [11,12]. We implement interpretation by annotating interfaces with desired targeting behavior. We also develop interfaces for correcting errors in both levels, and we argue that *social* mechanisms for identification and interpretation are essential to any broad target-aware deployment.

Figure 2 offers an overview of our system. We first query the window manager for images of targeted windows. We then *identify* elements using Prefab’s pixel-based methods. This requires a library of Prefab prototypes, each created from one or more example images. Prefab creates a tree containing a node for each interface element (e.g., a leaf for a text label, a leaf for a slider thumb, or an inner node with

children for a button and any interior icon or text). We then *interpret* the interface to determine potential targets, using annotations inferred from the interface structure and content together with annotations from prior interactive target correction. Finally, we walk the tree to determine what the Bubble Cursor should target and then overlay a translucent highlight over that target. Executing this process many times per second yields our general-purpose Bubble Cursor.

The specific contributions of our work include:

- An implementation of a general-purpose target-aware pointing enhancement. Specifically, we scale Prefab’s [11] implementation of the Bubble Cursor using pixel-level analysis to target interface elements throughout Microsoft Windows 7.
- A novel architecture for general-purpose target-aware pointing enhancements. Informed by the insight that knowledge of interface element locations and dimensions is insufficient for a general-purpose target-aware pointing enhancement, we separate pixel-level *identification* of interface elements from higher-level *interpretation* of how to target those elements. Prior work has assumed identifying all elements is sufficient. We are the first to develop such a layering of semantics upon pixel-based methods.
- Two interfaces for users to correct the behavior of a general-purpose target-aware pointing enhancement. We examine the requirements and design space for such interfaces and develop two concrete designs that explore complementary approaches.
- A discussion of implications for the design and evaluation of target-aware pointing techniques in the complexity of real-world interfaces. Examining our implementation with real-world interfaces reveals both unexpected targeting behaviors not addressed in prior literature and new potential approaches to overcoming the limitations of current target-aware techniques.

RELATED WORK

We focus on Grossman and Balakrishnan’s Bubble Cursor [18], but the Bubble Cursor is emblematic of a much larger body of target-aware techniques. This includes gravity wells [24], force fields [1], sticky icons [42], semantic pointing [5], area cursors [26], enhanced area cursors [15], bubble targets [10], object pointing [19], and drag-and-pop or drag-and-pick [3]. Such techniques offer great potential, but remain difficult to deploy in practice. The pixel-level analysis and social annotation methods we develop can help enable the broad deployment of these and future techniques.

The difficulty of deploying target-aware techniques motivates *target-agnostic* techniques, which aim to improve pointing without knowledge of targets. We are aware of only four such techniques: conventional pointer acceleration (cf., [6]), PointAssist [21], the Angle Mouse [41], and the Pointing Magnifier [25]. In addition, Hurst *et al.*’s [23] use of click history to approximate gravity wells can be considered a

target-agnostic technique [23]. Despite the ingenuity of these techniques, they are inherently limited by their ability to consider only mouse kinematics and clicks. Our current work provides a foundation for the future deployment of target-aware techniques. This is especially promising because no target-agnostic technique has demonstrated performance superior to the best target-aware techniques.

Edwards *et al.* [13] and Olsen *et al.* [32] implement runtime modification of existing interfaces by replacing the toolkit drawing object and intercepting commands (e.g., `draw_line`, `draw_string`). Stuerzlinger *et al.* [37] present advanced customizations in their User Interface Facades, many based on interface introspection using the accessibility API. Eagan *et al.* [14] dynamically load code into program space with Scotty, developing runtime modifications based on greater access to the underlying interface model. Toolkit introspection and injection techniques can indeed enable runtime modifications that would otherwise be difficult or impossible, but with the consequence that enhancements are limited to interfaces that provide the required support.

Because all graphical interfaces ultimately consist of pixels, pixel-based methods are motivated by their independence from underlying application or toolkit requirements. Classic work by Zettlemoyer *et al.* [45,46] examined widget identification in IBOTS and VisMap for interface agents and programming by example [34,35]. St. Amant *et al.* [36] developed Segman for cognitive modeling applications. In an interactive context, Olsen *et al.*'s [33] ScreenCrayons links ink annotations to arbitrary screen elements. Tan *et al.*'s [39] WinCuts interactively subdivides windows via a copy-paste metaphor. Yeh *et al.*'s Sikuli [40] uses template matching and voting based on invariant local features to identify targets in interface scripting and testing applications. Dixon *et al.*'s [11,12] Prefab demonstrates real-time modification using input and output redirection together with pixel-based reverse engineering of interface content and structure. Our current work is informed and inspired by this prior research, contributing to pixel-based methods through deep implementation of our Bubble Cursor.

The strengths and limitations of application introspection versus pixel-based methods motivates synergies between the two approaches. Hurst *et al.* [22] studied and addressed the coverage of accessibility APIs, finding approximately 25% of widgets are completely missing from the API's view of many common interfaces. They developed a hybrid technique that improves target boundary detection using pixel-based methods together with interaction traces. Chang *et al.* [7] explored several synergies in PAX, including use of Sikuli to obtain paths to elements in the accessibility API, pixel-level analyses to locate screen text, and the use of Sikuli to find elements in portions of the screen where the accessibility API's representation is incomplete. Neither system developed methods capable of supporting the real-time demands of an interactive Bubble Cursor. More importantly, the current paper highlights that even complete

identification of interface elements does not provide the *interpretation* required to enable target-aware pointing.

Dixon and Fogarty [11] present a rudimentary demonstration of target-aware pointing in their original Prefab research. Although they use pixel-based identification to target a handful of elements, they never examine the challenges of applying these techniques beyond the scope of simple, isolated interfaces. For example, they assume all necessary examples of interface elements have already been provided to Prefab, and they do not discuss the ambiguities of targeting in real-world interfaces. The current work is unique in its deep application of Prefab to a general-purpose Bubble Cursor. In addition to our direct contributions to this first deep implementation of general-purpose target-aware pointing, we also present the first tools for interactive correction and extension of Prefab's identification of interface elements. Our methods for interpretation of identified elements are also a clear advance over earlier Prefab research.

We draw inspiration from extensive interface customization research, including integration of sharing and other social mechanisms. Most of this research is limited to the web, where the DOM provides a model of interface elements. Classic examples are ChickenFoot [2] and CoScripter [29], and systems like Highlight [31] extend these ideas to task-centric re-authoring for mobile devices. Clip, Connect, Clone [17], d.mix [20], and Vegemite [28] demonstrate end-user mash-up methods. In the desktop context, Chapuis and Roussel's [9] UIMarks allow creation of macros invoked by targeting special marks within an interface. Tagaki *et al.* [38] develop social annotation methods for improving web content accessibility. Hurst *et al.*'s [23] use of click history also suggests a role for sharing that history.

A REAL-WORLD BUBBLE CURSOR

Among the variety of target-aware pointing techniques, we implement the Bubble Cursor for several reasons: (1) we believe it is the fastest general technique in the literature, (2) it can be implemented as an overlay, without modifying targeted elements, and (3) it is exemplary of the family of target-aware techniques, so the methods we develop should have broader relevance. Our discussion revisits this, emphasizing our work as a starting point for the real-world design and deployment of target-aware techniques.

Our methods are inherently *human-driven*, in that we focus on allowing people to improve targeting by interactively correcting erroneous behavior. We also expect any broad deployment will be *social*, with people sharing interactive corrections and receiving updates based on the corrections of others. Interfaces are procedurally generated, so their pixel-level appearance rarely changes. Familiar interfaces will therefore be thoroughly annotated and appear to "just work". But interactive correction will remain important, both to ensure individuals can correct private, niche, or unpopular interfaces and to allow communities to quickly annotate newly-released interfaces.

This section presents the primary technical components of our system: (1) *identification* of elements using Prefab’s pixel-based methods, (2) *interpretation* of those elements via interactive and automatic annotation, and (3) *targeting* according to those annotations. We then briefly comment on details of our current Windows 7 implementation. This section focuses on responsibilities of each component and how they combine to enable a deployable Bubble Cursor. The next section then considers inevitable errors in such a system and introduces interfaces for interactively correcting those errors in terms of these underlying components.

Identifying Interface Elements

We identify interface elements using Prefab’s pixel-based methods for reverse engineering interface structure [11,12]. Prefab’s methods are a strong fit for several reasons, but the most important is they can be *corrected* and *extended* by providing additional examples of interface elements. This sharply contrasts accessibility APIs, where there is generally no recourse if an element is not correctly exposed. To enable our human-driven approach, we develop the first interactive tools for providing Prefab with examples at runtime.

More specifically, Prefab identifies interface elements via a library of *prototypes*. A prototype describes an arrangement of pixels, and each is learned from example images. Prefab uses two high-level strategies to identify interface elements: (1) exactly matching prototype pixels against an image, or (2) modeling prototype background and differencing pixels in an image to identify foreground interface elements. Prefab realizes these high-level strategies by varying how it generalizes from example images according to *models* of the *parts* of interface elements. Parts can be *features* (exact patches of pixels) or *regions* (methods for painting areas of variable size, such as gradients or repeating patterns). Figure 3 illustrates three prototypes selected to show a range of complexity in their underlying models.

The simplest are *exact-match* prototypes, which consist of a single feature exactly matching the pixels of an example. These do not generalize, but many interface elements also do not vary in appearance (e.g., checkboxes, icons, radio buttons). For example, the left prototype in Figure 3 identifies all unchecked Windows 7 Steel checkboxes.

A more complex *slider* prototype uses multiple parts to account for the variable length of the slider trough and the variable position of the thumb. Five parts characterize the appearance of the slider’s thumb, the left and right ends of the trough, and a repeating trough pattern on either side of the thumb. The middle prototype in Figure 3 was generalized from the single illustrated example and identifies all occurrences of the standard Mac OS X slider. Prefab performs this generalization by searching for an assignment of pixels from the example to parts in the model minimizing the number of pixels in the resulting prototype. If multiple examples are provided, Prefab searches for the minimal prototype consistent with all examples. Additional discussion of this search is available elsewhere [11,12].

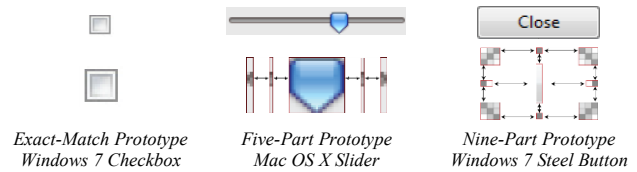


Figure 3: Prefab uses examples of interface elements to generalize prototypes of the appearance of families of widgets.

A *nine-part* prototype adds the ability to model background and use runtime pixel differencing to identify unpredictable foreground elements. For example, the prototype at the right of Figure 3 identifies all Windows 7 steel buttons and any text or icons painted over their gradient background. It was generalized from the single illustrated example button. Nine-part prototypes are first identified by matching their four corners and four edges, similar to matching the five parts of a slider. Prefab then uses the interior *content region* to identify elements painted over the interior background. Additional discussion of content regions, including how Prefab generalizes a background from example images that include foreground elements, is available elsewhere [12].

Differencing the pixels between content regions and their backgrounds allows Prefab to identify elements that are not in its prototype library. For example, a nine-part prototype of a tab pane can allow identification of all elements within that pane (e.g., buttons, icons, text). If Prefab lacks prototypes for these elements, it can still identify them as connected sets of foreground pixels, represent them as children of the tab pane, and thus make them available for interpretation as potential targets.

Prefab generalizes the idea of containment within a content region by organizing the entire interface into a hierarchy. The root corresponds to the processed image, and identified elements are added as children to any element in which they are spatially contained. This *spatial hierarchy* is not strictly the same as an interface’s logical hierarchy, but represents visible containment (e.g., buttons, group boxes, tab panes). The next subsection uses the hierarchy to robustly annotate identified interface elements with targeting information.

Interpreting Interface Elements

The above subsection described how Prefab identifies a hierarchy of elements, but any hierarchy is by itself insufficient for targeting. Even complete compliance with a typical accessibility API does not enable an effective Bubble Cursor. This is ultimately due to mismatches between available metadata and the needs of an external enhancement. Framework and application developers cannot foresee all potential external enhancements, so they cannot provide all relevant metadata. For example, current accessibility APIs are designed to provide access to the underlying data in each widget (i.e., the widget *model*). They therefore expose the value to which a slider is set, but not the screen location of the thumb (precluding targeting of that thumb). General targeting behavior is undefined for other elements (e.g., calendars, paint canvases, text fields),

and will vary among techniques. We believe this mismatch between available metadata and the needs of enhancements is inherent, requiring a human-driven approach.

We therefore interactively *annotate* which elements of an interface hierarchy should be targeted by a Bubble Cursor. Any node can be marked as a *target*, can implicitly not be targeted due to other targets in the hierarchy, or can be explicitly marked *not a target*. We store annotations using an XPath-like path descriptor based on properties of an element, its location in the hierarchy, and properties of its ancestors. A library of annotations therefore consists of path descriptors with associated metadata. Annotations can be quickly retrieved for an element, and entire libraries can be stored, combined, and shared to enable social annotation.

A path descriptor needs to identify a *specific* element in a *specific* interface while being robust to changes in element size or position. The root of a hierarchy corresponds to the entire image processed by Prefab, so the root needs to include how an image was captured. We currently set root attributes for the application executable name and top-level window class. If this became insufficient, additional root attributes could be added for problematic applications (e.g., including the URL for images captured within web browsers). The remainder of the descriptor is based upon the unique identifier for each prototype along the path from the root to the annotated element. Elements identified via background differencing do not have a prototype, so we use a content attribute based on a hash of their pixels (which allows differentiation among sibling content elements). Finally, we use an index for otherwise identical descriptors.

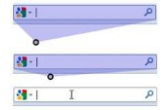
Direct manual annotation likely can be sufficient in a broad deployment leveraging social mechanisms, but also can be expedited by even minimal inference. We currently employ two simple mechanisms. The first assumes every leaf is a target. This heuristic yields good behavior for many widgets (e.g., checkboxes, radio buttons). The second generalizes annotations of inner nodes to other nodes with identical subtrees. Scrollbars provide one example of this, as their thumbs vary in size and usually contain a knurling graphic. A nine-part model represents the thumb as a node with a single child (i.e., the knurling). Preferred behavior is to target the entire thumb, and annotation of one scrollbar can then generalize over other occurrences of that scrollbar. Arbitrarily sophisticated inference could identify likely targets, and our XPath-like descriptors suggest relevance of the deep literature on wrapper induction [27]. But any inference mechanism will sometimes fail, so we see these techniques as a powerful complement to human annotation.

Choosing a Target

We choose a target for the Bubble Cursor in a pre-order traversal of the hierarchy of the window nearest the pointer. We consider each *target* node to determine which is closest to the pointer, defining distance to be zero if the pointer is within a target. Recursion ends at target nodes, because the spatial containment represented by the hierarchy means any

children will be further from the pointer. Recursion can therefore also end at non-target nodes that are further than the current best. Some arrangements require considering the possibility the nearest target is not in the nearest window, but other windows are typically ignored because their root is further from the pointer than the current best target.

Our design for real-world interfaces also required subtle refinement of the Bubble Cursor. One important refinement is our degradation into the standard point cursor whenever within a target (illustrated here with a storyboard of the cursor approaching a textbox). On one hand, this is aesthetically simpler than crosshairs used to illustrate the center of the original Bubble Cursor with abstract targets [18]. But it is also important because it creates a *graceful degradation* in the face of unknown interface elements or ambiguous targets. If Prefab fails to interpret the pixels in a portion of an interface, or if the appropriate targeting behavior is genuinely ambiguous (e.g., as with a text field), then normal point cursor behavior is automatically restored in that region without requiring a hotkey or other modifier. Pointing can therefore be improved in many interactions without necessarily being penalized in others. We believe such *conservative* strategies are promising for the design of deployable external interface enhancements.



The cursor must also be capable of *dragging* targets while moving or sizing elements (e.g., scrollbars, sliders, windows). Dragging is implicitly supported by our conservative strategy, as moving into a target allows use of the point cursor to drag the target. We also enable dragging from outside a target. When the drag is initiated, the Bubble Cursor latches onto the target. Subsequent movement drags the target, and mouse release resumes standard targeting behavior. We currently do not use knowledge of potential drop targets during the drag, but could imagine strategies for target-aware drops.

Implementation Details

This section has focused on novel methods and strategies in our system, as these can be applied and extended in future research on target-aware pointing or external enhancement of existing interfaces. For completeness, we briefly discuss relevant details of our current Windows 7 implementation.

At the start of each cycle, we query the Windows 7 Desktop Window Manager for the bounding box, application executable path, class name, and z-order for each visible top-level window. We construct a hierarchy with the desktop at the root and top-level windows as children, and we then capture pixels for the window closest to the pointer. If two windows are at the same distance (typically due to overlap), we choose the front. As a performance optimization, we ignore entire windows that are marked as non-targetable (e.g., visible but non-interactive windows). We also infer target annotations during traversal, and only when there is no existing human annotation (i.e., our inferred annotations are lazily evaluated).

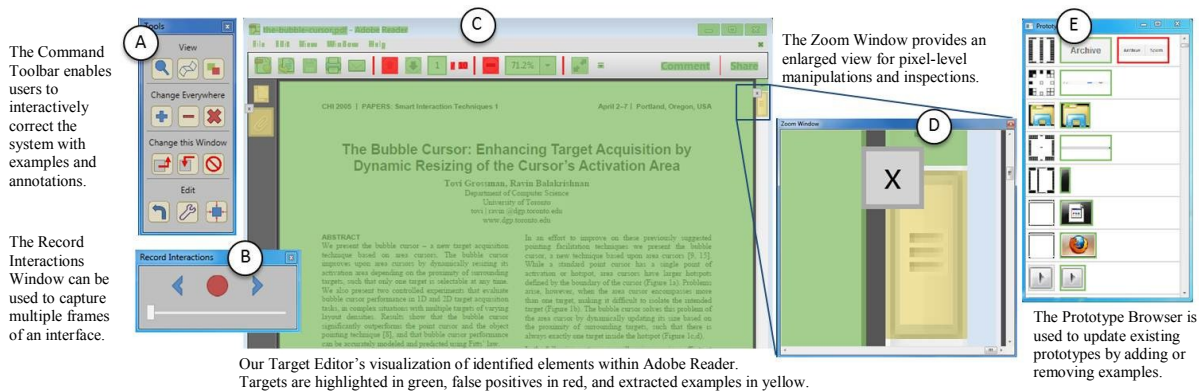


Figure 4: The Target Editor interface is used to extract examples, create and update prototypes, and manipulate annotations.

We render our Bubble Cursor as a translucent window, and update its z-order to render above the targeted window. We redirect input using a low-level hook to intercept mouse events and update the center of our Bubble Cursor instead of the system pointer. We use each mouse event to move the center of our Bubble Cursor, and we clip the system cursor to the center of our targeted element. If the target is partially occluded by another window, we clip to the center of a visible region (and adjust the size of our overlay).

Our associated video was captured on a typical laptop. Prefab is mostly single-threaded and unoptimized, but still processes an interface every 100 msec. Many optimizations described in [11,12] could improve performance, however our video shows the cursor is responsive and we have not found performance to be a concern.

INTERACTIVE TARGETING CORRECTION

We previously noted that many familiar interfaces will seem to “just work”, but that interactive correction remains important to our approach. We identified six requirements for correcting target identification and interpretation:

- *Invoking Correction:* A person must be able to access an interface to correct erroneous targeting behavior.
- *Capturing an Interface:* Errors are caused by incorrect identification or interpretation of elements in images of an interface, so a person must be able to capture images that illustrate the failure condition.
- *Visualizing Identification and Interpretation:* A person must be able to understand what error occurred so that they can determine how it should be corrected.
- *Extracting Examples:* A person must be able to extract examples of the elements of an interface that should be identified by our pixel-based methods.
- *Creating Prototypes:* A person must be able to use extracted examples to create appropriate prototypes that can then be identified at runtime.
- *Authoring Annotations:* A person must be able to annotate desired targeting of an interface hierarchy.

These requirements imply a large design space. Because we focus on the first deep implementation of a general-purpose target-aware technique, we explore two initial points in the

design space. The first is a full-featured design, called the *Target Editor*. The second is a lightweight interface for in-context annotation, called the *Annotation Menu*.

Correcting Behavior with the Target Editor

Figure 4 presents screenshots of the Target Editor. Its major components include the *window image*, colored *highlights* of identified elements, and the *command toolbar*.

Invoking Correction. The editor is always accessible via a system tray icon and a keyboard shortcut. When a person encounters erroneous targeting, they invoke the editor and then click a window on the desktop to “edit” that window.

Capturing an Interface. Selecting a window as part of invoking the editor triggers capture of an image of that window. The image is processed to identify and interpret interface elements, then displayed in the editor.

Some errors only occur during interaction with a *dynamic* interface, so a captured static image may be insufficient. For example, the blinking text cursor in a textbox can lead to erroneous targeting. When present, it is identified as a leaf and targeted. When absent, the textbox itself becomes a leaf and is targeted. The result is a Bubble Cursor that jumps between the blinking caret and the larger textbox. To determine why this occurs, a person may need to inspect multiple images captured at different times.



The editor addresses this need with a “Record Interactions” tool to capture videos of interaction. A record button starts and stops recording, a slider supports skimming captured video, and buttons advance individual frames. The blinking text cursor behavior is fixed by annotating the textbox as a target, thereby overriding the inferred targeting of the text cursor leaf.

Visualizing Identification and Interpretation. Translucent highlights visualize system identification and interpretation of interface elements. Green highlights are placed over elements that will be targeted (i.e., the first *target* node encountered on every path from the root, whether inferred or annotated). Red highlights are placed over elements explicitly annotated as *not a target*. Identification is thus encoded via the presence of a highlight, while interpretation is illustrated via color. Interactive selection is illustrated via

a border, and elements can be manipulated individually or in batch using multiple selections with the toolbar.

Extracting Examples. Prefab often builds a good prototype from just a single example of an interface element. But it can also under-generalize, requiring additional positive examples to broaden its concept, or over-generalize, requiring negative examples to narrow its concept.


For example, this YouTube movie control causes Prefab to overgeneralize because its widgets are painted to share vertical edges. A single example creates a prototype that over-generalizes by identifying every group of k adjacent widgets (creating duplicate leaves and inner nodes that do not describe any true element). Providing any of these false detections as a negative example will correct the prototype, resulting in a prototype that detects the combination of the single-pixel edge and the adjacent interior pixel.


Examples can be specified by rubber-banding a yellow highlight. Each highlight can be moved, resized, or deleted. Because pixel-level selection can be tedious and error-prone, the editor provides “Zoom”  and “Snap”  tools.

“Zoom” opens an enlarged view of the image and highlights. The enlarged view receives input, so it supports pixel-level adjustment of highlights. As a shortcut, double-clicking in the image or on any highlight also opens the “Zoom” tool with its view centered at the double-click point.

“Snap” resizes highlights to tightly fit interface elements. Informed by a heuristic from the original Prefab work [11], it starts from the center of a highlight and uses gradient thresholds to search for possible bounding rectangles. Among the heuristically-identified rectangles, it chooses the one closest in size to the drawn highlight. This does not always snap exactly to interface elements, but is helpful for expediting extraction. When it fails, it usually snaps close to the desired size and can then be adjusted using “Zoom”.


Two categories of examples can be automatically extracted: (1) positive examples of elements already identified by background differencing within a parent’s content region, and (2) any negative examples. In both cases, Prefab has already identified the bounds of the element. There is no need to manipulate the highlight, so a person instead just adds the extracted element to a new or existing prototype.

Creating Prototypes. A new prototype is created using the “Add”  tool. A dropdown menu allows model selection (e.g., the exact-match, five-part, and nine-part models discussed previously). For each selected highlight, the editor creates a new prototype from the highlighted positive example. Automated creation of the prototype from the example takes a few seconds, after which the user can immediately see the impact of the new prototype on identification and interpretation within the interface.


Existing prototypes are updated with additional negative examples using the “Mark Incorrect”  tool. Note that the

editor already knows which prototype should be updated: the prototype that falsely identified the example.


Updating existing prototypes with additional positive examples requires the selection of the prototype to receive the new example. The “Prototype Browser” visualizes the prototypes in the current library, together with the positive and negative example images used to create each prototype. A person selects an existing prototype, then uses “Add” to provide it with additional examples.

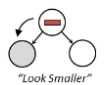
Bad examples or entire prototypes can be deleted via the “Prototype Browser”. Bad prototypes can also be deleted directly in the editor by selecting a highlight and clicking the “Remove”  tool. This removes the prototype that identified the highlight, and it is helpful for quickly deleting prototypes accidentally created from bad examples.


Authoring Annotations. Three tools are used together for annotation of the desired interpretation of targets.

The “Look Bigger”  tool annotates the *parent* of a selected element as a *target*. The Bubble Cursor then targets that parent, thus no longer targeting the element. For example, this can be used to target a button instead of its text, a canvas instead of its content, or an entire window instead of any interior icons. The tool does not modify any annotations of the selected element or its children. Such annotations have no effect, as our Bubble Cursor targets downward from the root. Leaving them intact allows “undo” via “Look Smaller”.



The “Look Smaller”  tool annotates a selected element as *not a target*, which will cause the Bubble Cursor to consider its descendants as possible targets. Because we heuristically infer leaf elements to be targets, this is typically used to undo the “Look Bigger” tool. But it can also correct inferred targeting of inner elements and would likely see more use together with more extensive inference of targets.



The “Not a Target”  tool annotates a selected element and all descendants as *not a target*. This is used to ignore an entire subtree of non-targetable elements, such as labels or disabled buttons.



Lightweight Targeting with the Annotation Menu

The Annotation Menu examines a design that is less capable than the Target Editor, but can also be used without leaving the context of an interaction. Figure 5 storyboards the menu, which is embedded in the Bubble Cursor and invoked after a dwell of the pointer within a target.

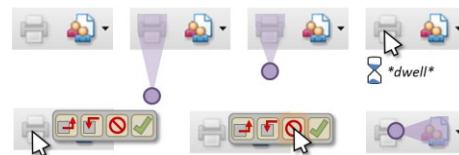



Figure 5: A storyboard of the Annotation Menu. It appears when the center of the Bubble Cursor dwells on a target.

Interface *capture* and *visualization* are implicit, as the Bubble Cursor itself conveys the current identification and interpretation. Editing is limited to *authoring annotations* using the same “Look Bigger,” “Look Smaller,” and “Not a Target” functionality from the editor. Each tool makes its edits to the annotations, closes the menu, and updates the Bubble Cursor to target the new nearest target.

The menu also provides a “Confirm”  tool. When clicked, it applies a *confirmed* annotation to the selected interface element. The Annotation Menu then does not invoke itself upon dwell over *confirmed* elements.

Discussion of Interface Design

Figure 6 recaps our two designs in the context of our six requirements. They take different approaches to correction, but both are helpful and suggest additional possibilities. For example, we use simple visualizations of the identification and interpretation of elements, but other tree visualizations could be considered. Our *confirmed* annotation was developed for the Annotation Menu, but it would be a sensible addition to the Target Editor. Rapid batch confirmation of targeting would then prevent the Annotation Menu from appearing in interfaces that are already known to be correct. Greater collection of explicit confirmations would also provide additional data for use in training learning systems to automatically infer targets.

Our requirements and designs also suggest opportunities for fundamentally different approaches. Image *capture* might be automated through passive observation of interface use (e.g., as in Hurst *et al.*'s prior work [22]), with an interface *invoked* later to review targeting corrections inferred from the observed usage. Instead of focusing on the current interface, a design might focus on *creating prototypes*. Such a design could be organized around review, creation, and modification of prototypes and could retrieve images of specific interfaces from a large usage history to illustrate the targeting that results from the current prototype library. A crowdsourcing interface might focus on local *invocation* and *capture*, with other requirements addressed by remote workers. Because our designs require an understanding of pixel-based methods, other designs might explore a range of required *expertise*. Our initial focus has demonstrated effective interfaces, but a variety of additional approaches can be developed upon our underlying technology.

EXAMINING BEHAVIOR IN REAL-WORLD INTERFACES

The Bubble Cursor and other target-aware techniques are often designed, discussed, and evaluated using fields of abstract targets and distractors. Because our implementation provides a unique ability to deploy such techniques, we sought to examine what new insights we could gain from examining the Bubble Cursor in real-world interfaces. We identified and annotated elements in a variety of interfaces, including Microsoft Office applications, Mozilla Firefox, Adobe Reader, instant message clients, several web pages, and the file browser. We studied a total of 31 applications, creating a library of 754 prototypes and 714 annotations.

Requirement	Target Editor	Annotation Menu
Invoking	System Tray Keyboard Shortcut	Dwell
Capturing	Static Frame on Launch Record Interactions Tool	Implicit
Visualizing	Colored Highlights of Elements	Implicit
Extracting	Rubber-Band Zoom Snap Automatic via Prefab	N/A
Creating	Add / Remove / Mark Incorrect Model Selection Prototype Browser	N/A
Authoring	Look Bigger Look Smaller Not a Target	Look Bigger Look Smaller Not a Target Confirm

Figure 6: Our interfaces highlight two initial points in the design space of correctional interfaces for our system.

Our findings with real-world interfaces include insights into two important challenges: (1) the limitations of pointer proximity as a proxy for user intent within an interface, and (2) conflicts between target-aware behavior and the intentional design of interfaces for typical point cursors.

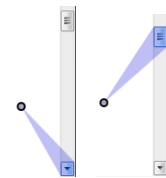
Pointer Proximity and User Intent

The Bubble Cursor and other target-aware techniques use pointer proximity as a proxy for user intent to manipulate interface elements. It is understood to be an imperfect proxy (hence the notion of distractors), but examining behavior in real-world interfaces gives new insight into limitations.

One example we found is that a person may expect similar elements to have similar targeting behavior. Rows of elements provide a good example, as the arrangement suggests each item is equally relevant. But placement of a menu bar next to a toolbar can create different effective targets for otherwise similar elements. The Voronoi overlay shows “Page Up” is more difficult to target than “Page Down”, and “Help” behaves differently than the other menu items. The behavior is correct, but can be jarring.



The dynamics of real-world interfaces also expose gaps between pointer location and user intent. Consider using a Bubble Cursor to click a scrollbar arrow. A person’s intended focus is likely to remain upon the arrow after a click, but the movement of the scrollbar toward the cursor may now mean the thumb is the nearest target. Latching onto the thumb is the correct behavior for the Bubble Cursor, but it may be unexpected.



A third case emerges when interfaces act upon a notion of user intent that competes with the Bubble Cursor. For example, the Windows 7 taskbar contains a row of buttons for accessing windows of open applications. A hover over any opens a preview. Approaching the taskbar with the Bubble Cursor results in the cursor snapping to a button, invoking the preview, and then snapping to the preview. This is



helpful if the desired window is captured, as clicking will activate the window. But if an adjacent taskbar button was desired in the first place, then the benefits of the Bubble Cursor are destroyed by snapping to the preview.

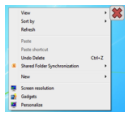
Conflicts Due to Intentional Design for a Point Cursor

Modern interfaces are carefully and intentionally designed for use with a typical point cursor. This section presents examples where the Bubble Cursor conflicts with that design, together with initial thoughts on how such conflicts could be resolved through extensions to our system.

Designers often use small elements for minor or infrequently-accessed functionality. The Bubble Cursor can greatly distort the importance of such minor elements. For example, the Windows 7 file browser includes several minor icons beneath its search box. These eclipse targeting of the search box itself when approaching from below, which is the typical direction of approach. Selection of the search box is therefore more difficult than intended. Extending the methods in this work, it is interesting to consider a *look past* annotation. This might indicate an element should be targeted only if the pointer is directly overhead. The Bubble Cursor could then look past these elements to allow easy targeting of the search box, but would still latch onto minor elements when the pointer was directly overhead. This seems preferable to a hotkey or some other method for disengaging the Bubble Cursor.



Another example is the design of typical context menus, which are dismissed by clicking in non-interactive space. Such a design is incompatible with a technique like the Bubble Cursor that always targets the nearest interactive element. This problem motivated creation of DynaSpot [8], which adjusts the maximum reach of a Bubble Cursor based on pointer speed. Examining this in the context of our system suggests additional strategies. For example, a *dismissable* annotation might overlay a close icon at the upper-right of an element. Upon selection, the system could use its identification and interpretation of the surrounding interface to click in non-interactive space.



A third example is the scrollbar. As pictured elsewhere, it is easy to imagine the utility of a Bubble Cursor attached to a scrollbar thumb. But the trough is also interactive, meaning the thumb can only be grabbed from a narrow horizontal channel. In our own use, we therefore annotate the trough as *not a target*. This allows easy capture of the thumb, but also prevents targeting the trough. It is interesting to realize that the same *look past* annotation discussed above would remove the need for this tradeoff (i.e., latching onto the thumb from outside, but targeting the trough from within). This also suggests a more general distinction between “major” versus “minor” targets, a theoretical perspective that can inform the design of future techniques. Such future techniques could be evaluated with a combination of

laboratory studies and insights gained from examination in real interfaces using the methods we have developed here.

DISCUSSION AND CONCLUSION

Although we describe our approach to implementing a Bubble Cursor, we also provide a general strategy for implementing additional interface enhancements. Our separation of identification from interpretation, together with our approaches to each, can enable and inform a variety of future enhancements. For example, Prefab prototypes interactively created with our Bubble Cursor can be shared and re-used with any Prefab-based application. Similarly, our Bubble Cursor can benefit from Prefab prototypes created in other applications. At the interpretation level, target-aware enhancements may be able to directly share and re-use annotations (e.g., the notion of a *target* for Sticky Icons [42] may be the same as a *target* with our Bubble Cursor). Other techniques may have different interpretations of targets based on additional annotations, perhaps bootstrapped through inference based upon annotations collected with our Bubble Cursor or some other technique. Finally, our strategies also apply to non-pointing enhancements. For example, Findlater *et al.*'s [16] Ephemeral Adaptation could be implemented using analyses of interaction history together with explicit annotation of element groupings and other preferences.

This work is the most extensive application of Prefab's pixel-based methods to date. Our experiences validate Prefab's methods, but also suggest opportunities to advance pixel-based systems. For example, we found that Prefab would benefit from additional sophisticated models of the pixel-level appearance of complex elements (e.g., it could not deeply model complex text panes found in Microsoft Visual Studio 2010). We also sometimes found it tedious to create multiple prototypes for different appearances of the same interface element (e.g., multiple states of a button, checkbox, or radio button). Pixel-based methods could therefore benefit from additional modeling of the dynamics of interface elements. Our work has also not addressed the case where changes to a prototype might orphan existing annotations (i.e., might change the descriptor needed to retrieve existing annotations). We previously noted that wrapper induction techniques may help to infer descriptors that are robust to change [27], and we also believe we could explicitly migrate annotations between hierarchies obtained with different prototype libraries (e.g., by checking that they refer to the same spatial region in the captured image). Overall, we believe that deep applications like that pursued in this paper have a unique potential to inform future development of underlying pixel-based methods. Building out this application has also given us a better understanding of effective and sustainable abstractions for developing with Prefab, and we are currently exploring the best way to make this functionality available.

Looking forward, we believe the methods and findings of this work suggest future research opportunities in both

interaction techniques and underlying systems. Our hope is to help more interaction research in escaping the laboratory, putting it into the hands of end-users who stand to benefit from the field's rich innovation.

ACKNOWLEDGMENTS

We thank Jon Froehlich and Shaun Kane for discussions related to this work. This work was supported in part by the NSF under awards IIS-1053868 and IIS-0811063, by Intel, and by the UW College of Engineering Osberg Fellowship.

REFERENCES

- Ahlström, D., Hitz, M. and Leitner, G. An Evaluation of Sticky and Force Enhanced Targets in Multi Target Situations. *NordiCHI 2006*. 58-67.
- Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. Automation and Customization of Rendered Web Pages. *UIST 2005*. 163-172.
- Baudisch, P., Cutrell, E., Robbins, D., Czerwinski, M., Tandler, P., Bederson, B. and Zierlinger, A. Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch- and Pen-Operated Systems. *INTERACT 2003*. 57-64.
- Baudisch, P., Cutrell, E., Hinckley, K. and Gruen, R. Mouse Ether: Accelerating the Acquisition of Targets Across Multi-Monitor Displays. *CHI 2004*. Extended Abstracts. 1379-1382.
- Blanch, R., Guiard, Y. and Beaudouin-Lafon, M. Semantic Pointing: Improving Target Acquisition with Control-Display Ratio Adaptation. *CHI 2004*. 519-526.
- Casiez, G., Vogel, D., Balakrishnan, R. and Cockburn, A. The Impact of Control-Display Gain on User Performance in Pointing Tasks. *Human-Computer Interaction 23 (3)*. 215-250.
- Chang, T., Yeh, T., and Miller, M. Associating the Visual Representation of User Interfaces with their Internal Structures and Metadata. *UIST 2011*. 245-256.
- Chapuis, O., Labrune, J., and Pietriga, E. DynaSpot: Speed-Dependent Area Cursor. *CHI 2009*. 1391-1400.
- Chapuis, O. and Roussel, N. UIMarks: Quick Graphical Interaction with Specific Targets. *UIST 2010*. 173-182.
- Cockburn, A. and Firth, A. Improving the Acquisition of Small Targets. *HCI 2003*. 181-196.
- Dixon, M. and Fogarty, J. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *CHI 2010*. 1525-1534.
- Dixon, M., Leventhal, D., and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *CHI 2011*. 969-978.
- Edwards, W.K., Hudson, S.E., Marinacci, J., Rodenstein, R., Rodriguez, T. and Smith, I. Systematic Output Modification in a 2D User Interface Toolkit. *UIST 1997*. 151-158.
- Eagan, J.R., Mackay, W.E., and Beaudouin-Lafon, M. Cracking the Cocoa Nut: User Interface Programming at Runtime. *UIST 2011*. 225-234.
- Findlater, L., Jansen, A., Shinohara, K., Dixon, M., Kamb, P., Rakita, J., and Wobbrock, J.O. Enhanced Area Cursors: Reducing Fine Pointing Demands for People with Motor Impairments. *UIST 2010*. 153-162.
- Findlater, L., Moffatt, K., McGrenere, J. and Dawson, J. Ephemeral Adaptation: The Use of Gradual Onset to Improve Menu Selection Performance. *CHI 2009*. 1655-1664.
- Fujima, J., Lunzer, A., Hornbæk, K. and Tanaka, Y. Clip, Connect, Clone: Combining Applications Elements to Build Custom Interfaces for Information Access. *UIST 2004*. 175-184.
- Grossman, T. and Balakrishnan, R. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *CHI 2005*. 281-290.
- Guiard, Y., Blanch, R. and Beaudouin-Lafon, M. Object Pointing: A Complement to Bitmap Pointing in GUIs. *GI 2004*. 9-16.
- Hartmann, B., Wu, L., Collins, K. and Klemmer, S.R. Programming by a Sample: Rapidly Creating Web Applications with d.Mix. *UIST 2007*. 241-250.
- Hourcade, J.P., Perry, K.B. and Sharma, A. PointAssist: Helping Four Year Olds Point with Ease. *IDC 2008*. 202-209.
- Hurst, A., Hudson, S.E. and Mankoff, J. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI 2010*. 11-20.
- Hurst, A., Mankoff, J., Dey, A.K. and Hudson, S.E. Dirty Desktops: Using a Patina of Magnetic Mouse Dust to Make Common Interactor Targets Easier to Select. *UIST 2007*. 183-186.
- Hwang, F., Keates, S., Langdon, P. and Clarkson, P.J. Multiple Haptic Targets for Motion-Impaired Computer Users. *CHI 2003*. 41-48.
- Jansen, A., Findlater, L. and Wobbrock, J.O. From the Lab to the World: Lessons from Extending a Pointing Technique for Real-World Use. *CHI 2011*. Extended Abstracts. 1867-1872.
- Kabbash, P. and Buxton, W. The "Prince" Technique: Fitts' Law and Selection Using Area Cursors. *CHI 1995*. 273-279.
- Kushmerick, N., Weld, D.S. and Doorenbos, R. Wrapper Induction for Information Extraction. *IJCAI 1997*.
- Lin, J., Wong, J., Nichols, J., Cypher, A. and Lau, T.A. End-User Programming of Mashups with Vegemite. *IUI 2009*. 97-106.
- Little, G., Lau, T.A., Cypher, A., Lin, J., Haber, E.M. and Kandogan, E. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *CHI 2007*. 943-946.
- McGuffin, M. and Balakrishnan, R. Acquisition of Expanding Targets. *CHI 2002*. 57-64.
- Nichols, J. and Lau, T.A. Mobilizing by Demonstration: Using Traces to Re-Author Existing Web Sites. *IUI 2008*. 149-158.
- Olsen, D.R., Hudson, S.E., Verratti, T., Heiner, J.M. and Phelps, M. Implementing Interface Attachments Based on Surface Representations. *CHI 1999*. 191-198.
- Olsen, D.R., Tauffer, T. and Fails, J.A. ScreenCrayons: Annotating Anything. *UIST 2004*. 165-174.
- Potter, R. Triggers: Guiding Automaton with Pixel to Achieve Data Access. A. Cypher, eds. *MIT Press*.
- St. Amant, R., Lieberman, H., Potter, R. and Zettlemoyer, L.S. Visual Generalization in Programming by Example. *Communications of the ACM 43(3)*. 107-114.
- St. Amant, R., Riedl, M.O., Ritter, F.E. and Reifers, A. Image Processing in Cognitive Models with SegMan. *HCI 2005*.
- Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*. 309-318.
- Takagi, H., Kawanaka, S., Kobayashi, M., Itoh, T., and Asakawa, C. Social Accessibility: Achieving Accessibility through Collaborative Metadata Authoring. *Assets 2008*. 193-200.
- Tan, D.S., Meyers, B.R. and Czerwinski, M. Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *CHI 2004*. 1525-1528.
- Yeh, T., Chang, T.-H. and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009*. 183-192.
- Wobbrock, J.O., Fogarty, J., Liu, S., Kimuro, S., and Harada, S. The Angle Mouse: Target-Agnostic Dynamic Gain Adjustment Based on Angular Deviation. *CHI 2009*. 1401-1410.
- Worden, A., Walker, N., Bharat, K. and Hudson, S.E. Making Computers Easier for Older Adults to Use: Area cursors and Sticky Icons. *CHI 1997*. 266-271.
- Zellweger, P.T., Bouvin, N.O., Jehøj, H., and Mackinlay, J.D. Fluid Annotations in an Open World. *Hypertext 2001*. 9-18.
- Zhai, S., Morimoto, C. and Ihde, S. Manual and Gaze Input Cascaded (MAGIC) Pointing. *CHI 1999*. 246-253.
- Zettlemoyer, L.S. and St. Amant, R. A Visual Medium for Programmatic Control of Interactive Applications. *CHI 1999*. 199-206.
- Zettlemoyer, L.S., St. Amant, R. and Dulberg, M.S. IBOTS: Agent Control Through the User Interface. *IUI 1998*. 31-37.