Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets

Morgan Dixon[1], Gierad Laput[1,2], James Fogarty[1]

[1] Computer Science & Engineering
DUB Group, University of Washington
{mdixon,jfogarty}@cs.washington.edu

[2] Human-Computer Interaction Institute
Carnegie Mellon University
gierad.laput@cs.cmu.edu

ABSTRACT

Pixel-based methods offer unique potential for modifying existing interfaces independent of their underlying implementation. Prior work has demonstrated a variety of modifications to existing interfaces, including accessibility enhancements, interface language translation, testing frameworks, and interaction techniques. But pixel-based methods have also been limited in their understanding of the interface and therefore the complexity of modifications they can support. This work examines deeper pixel-level understanding of widgets and the resulting capabilities of pixel-based runtime enhancements. Specifically, we present three new sets of methods: methods for pixel-based modeling of widgets in multiple states, methods for managing the combinatorial complexity that arises in creating a multitude of runtime enhancements, and methods for styling runtime enhancements to preserve consistency with the design of an existing interface. We validate our methods through an implementation of Moscovich et al.'s Sliding Widgets, a novel runtime enhancement that could not have been implemented with prior pixel-based methods.

Author Keywords

Pixel-based runtime modification; Prefab; Sliding Widgets; hybrid touch and mouse interaction; real-world interfaces.

ACM Classification Keywords

H.5.2. [Information interfaces and presentation]: User Interfaces;

INTRODUCTION

Pixel-based methods for runtime interface modification offer great potential for democratizing interaction. Because all graphical interfaces ultimately consist of pixels, these methods enable modifying applications without their source code and independent of their underlying toolkit implementation. Pixel-based systems have demonstrated a wide variety of promising enhancements to existing interfaces, including contextual and video-based tutorials [2,31,43], automated visualization retargeting [33], improved window managers [39], interface testing frameworks [7], ink annotation overlays [29], systems for exploring document workflow histories [17], and web content retargeting [22].

CHI 2014, April 26 - May 01 2014, Toronto, ON, Canada.

http://dx.doi.org/10.1145/2556288.2556979

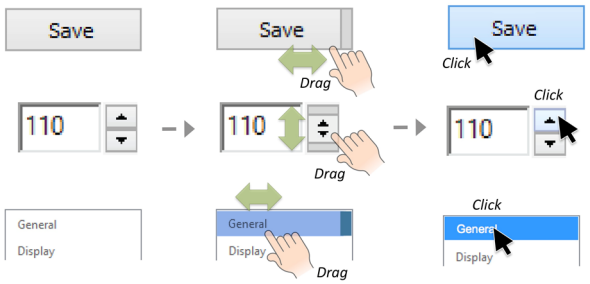


Figure 1: We present new pixel-based methods for modifying existing interfaces at runtime, and we use our methods to explore Moscovich et al.'s Sliding Widgets in real-world interfaces. We overlay Sliding Widgets throughout Microsoft Windows 8, replacing standard mouse-based elements to improve interaction with hybrid touch-and-mouse devices.

Despite the promise of pixel-based systems, they have been limited to relatively simple overlays that primarily point at or highlight existing elements. At least two key challenges have limited prior enhancements: (1) modeling a widget's *behavior*, and (2) capturing an interface's *style*. Structured support for these two tasks would enable more advanced modifications to a variety of existing interfaces.

Although current systems support the interpretation of pixels in an individual screen capture, the behavior of widgets in an interface is difficult to model because many potential enhancements require an understanding of how an interface changes across multiple frames (e.g., if a checkbox becomes checked, if a slider thumb has moved). In addition to monitoring these changes, more advanced enhancements also require manipulating the interface (e.g., sending click events to set a checkbox, dragging a slider). As a result, developers are faced with implementing their own complex frame-to-frame analysis of interfaces together with low-level input redirection mechanisms.

Capturing an interface's style is important because many runtime modifications do not fully alter the appearance of an existing interface, but instead directly overlay new elements onto the interface. Therefore it is important that these enhancements are styled to preserve consistency with

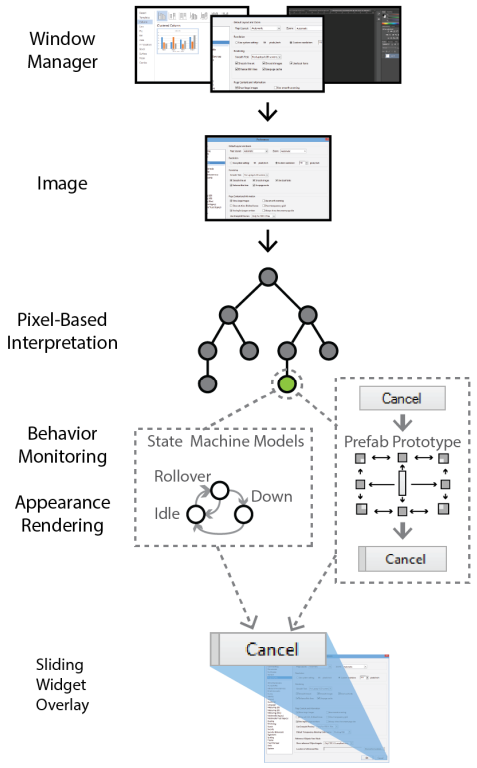


Figure 2: We build upon Prefab's methods for pixel-based identification of interface elements. Identified elements are replaced by overlaying a corresponding Sliding Widget. The original widget is then modeled and manipulated using state machine models. Finally, we style the appearance of overlaid Sliding Widgets by mapping elements of pixel-level appearance from Prefab's prototypes of the original interface.

the existing interface. Without proper support for capturing style, enhancements are likely to be jarring and unusable.

This paper addresses these limitations with novel methods for modeling the state and capturing style of individual widgets. Specifically, we build upon Prefab's pixel-based methods for interpreting interfaces [8,9,10,11]. We first introduce methods for real-time modeling of widgets in multiple states. We then describe methods for linking an original widget's state into the representation needed by a new surface enhancement. Finally, we introduce style mappers and show how they can be used to render entirely new widgets in an existing interface while maintaining a style that is consistent with the original design. We focus on these widget-level methods because they serve as building blocks for complex enhancements to entire interfaces.

We contextualize these contributions in an implementation of Moscovich et al.'s Sliding Widgets, touch widgets activated by sliding a moveable element [24]. Our implementation dynamically overlays Sliding Widgets on mouse-based interface elements throughout Microsoft Windows 8. For example, Figure 1 and our associated video present screenshots of our implementation in a variety of popular interfaces, including Microsoft Word 2013, Adobe Reader, Gmail in the Google Chrome Browser, and Windows Explorer. We replace elements in these applications using various types of Sliding Widgets, including Sliding Buttons, Sliding Spinners, Sliding Toggles, and Sliding Dropdown Menus. Sliding Widgets are well beyond the capabilities of prior pixel-based systems, and the implementation in this work informs and validates the design of our new methods.

In addition to validating our methods, we choose to implement Sliding Widgets in the context of existing interfaces because the implementation could directly benefit an emerging set of *hybrid* devices that support both touch and mouse input. Examples of these devices are shown in our associated video. Unfortunately, graphical interfaces for hybrid devices are either difficult to use or expensive to produce. This is because developers either: (1) provide standard mouse-based interfaces with small and densely arranged targets, or (2) implement an entire alternate interface optimized for touch. We therefore explore the approach of dynamically replacing existing mouse-based elements with touch controls. This approach can improve interaction with hybrid devices without the burden of designing and implementing two entire alternate interfaces.

Figure 2 overviews our system. We first query the window manager for images of windows, and then interpret their elements using Prefab's pixel-based methods [9,10,11]. Specifically, we use Prefab to recover an interface hierarchy containing a node for each interface element (e.g., a leaf for a text label, an inner node with children for a button and any interior content). We walk this tree to determine what elements to replace, and then we overlay our Sliding Widget. To maintain a style consistent with the source interface, our overlay renders each Sliding Widget using elements of the pixel-level appearance, as captured by Prefab's prototypes. This process modifies a single frame captured from an interface, and so we repeat this many times per second, synchronizing each Sliding Widget with its underlying widget. Specifically, we use state models to monitor and manipulate underlying widgets. Transitions in a state model cause the Sliding Widget to update its appearance and behavior, and interaction with a Sliding Widget generates input manipulating the underlying widget.

The specific contributions of our work include:

- Methods for pixel-based modeling of widgets in multiple states. Specifically, we introduce *abstract state models* for describing how to interpret and manipulate categories of widgets. Abstract models are then parameterized with pixel-level data specifying the appearance of a specific widget in each of the modeled states.
- Methods for managing the combinatorial complexity that arises in creating a multitude of runtime enhancements that can apply to a multitude of widgets. Specifically, we show how runtime enhancement can be framed in the Model-View-Controller pattern, and we introduce *linkers* for specifying how a particular runtime enhancement relates to a particular abstract state model.

- Methods for styling runtime enhancements to preserve consistency with the design of an existing interface. Specifically, we introduce *mappers* that use pixel-level elements of an interface's appearance to style runtime enhancements to match that interface.
- An implementation of Sliding Widgets as a runtime pixel-based enhancement that can be applied to existing interfaces. This validates our new pixel-based methods and also unearths implications for the design of Sliding Widgets and future pixel-based runtime enhancements.

RELATED WORK

Surface-Level Runtime Modification. Runtime modification has broad applications in accelerating innovation and facilitating adoption. In prior work, Edwards et al. [13] and Olsen et al. [28] modify interfaces by replacing the toolkit drawing object and intercepting commands (e.g., draw_line, draw_string). They update interfaces with new functionality, such as search and bookmark widgets. More recent examples include application mash-ups [14,18,36], re-authoring desktop applications for mobile interfaces [25,26], and automating repetitious interactions [5,23].

Traditional approaches to runtime modification are based in accessibility APIs [36] or injecting into an interface's underlying toolkit [12,13,28]. Accessibility APIs expose interface state, but are frequently incomplete because application developers fail to implement the API. For example, Hurst et al. found 25% of widgets are completely missing from the accessibility API [21]. Injection techniques attempt to gain full access to an interface by inserting custom logic via the toolkit or other runtime system. However, injection must be carefully crafted for each interface and underlying toolkit, making it difficult to apply to general-purpose whole-desktop enhancements like the Sliding Widgets we pursue here. Perhaps most importantly, both methods expose widget models, not necessarily their on-screen view (e.g., the pixel-level appearance of a slider thumb is intentionally encapsulated). But understanding pixel-level appearance is a requirement for many potential enhancements. For example, we use the pixel-level appearance of existing widgets to style Sliding Widgets to match the interface.

Pixel-Based Methods. Pixel-based methods do not require cooperation from an application's original developers and also overcome the fragmentation of interfaces and toolkits. Applications expose pixels as normal, and their raw pixels are then used as a foundation for additional functionality. Researchers have explored a variety of low-level abstractions to enable a range of pixel-based runtime modifications. Classic work by Zettlemoyer et al. [44,45] examined widget identification in IBOTS and VisMap for interface agents and programming by example [32,34]. St. Amant et al. [35] developed Segman for cognitive modeling applications. Olsen et al.'s [29] ScreenCrayons link interactive ink annotations to arbitrary screen elements. Tan et al.'s [37] WinCuts interactively subdivides windows via a copy-paste metaphor. Yeh et al.'s Sikuli [42] uses template matching and voting based on invariant local features to identify targets in interface scripting and testing applications. Dixon et al.'s Prefab [9,10,11] demonstrates real-time modification using input and output redirection together with pixel-based reverse engineering of interface content and structure. They also demonstrate rudimentary pixel-based strategies for rendering interface overlays [11]. Our current work is informed by this prior research, contributing to pixel-based methods and demonstrating deep implementation of Sliding Widgets.

The strengths and limitations of pixel-based methods versus application introspection motivate hybrid strategies that combine the two approaches. For example, Chang et al. [6] explored several synergies in PAX, including use of Sikuli to obtain paths to elements in the accessibility API, pixel-level analyses to locate screen-rendered text, and the use of Sikuli to find elements in portions of the screen where the accessibility API's representation is incomplete. We focus on core pixel-based methods required for state and style, and future work could extend these to hybrid approaches if needed for more complex or niche interfaces.

Our work presents the first general-purpose pixel-based methods for modeling widgets in multiple states. Prior work explores basic solutions for monitoring interface changes, but these methods are tailored to the goals of a specific enhancement. The most sophisticated example is Banovic et al.'s Waken, a system for reverse engineering and augmenting video tutorials [2]. Their system observes changes in an interface via lightweight frame differencing, and uses these observations to identify widgets. In contrast, our abstract state models are capable of capturing any transition in a widget's appearance, and they can also invoke transitions in a widget's state. These capabilities enable advanced runtime enhancements, moving beyond the identification of widget occurrences in a frame.

This paper combines the development of new pixel-based methods with an examination of an interaction technique in the context of real-world interfaces. This combination parallels the work of Dixon al. in their implementation of a general-purpose target-aware pointing enhancement [8]. They validate novel pixel-based methods in the context of an implementation of Grossman and Balakrishnan's Bubble Cursor, an area cursor that dynamically expands to always capture the nearest target [16]. Like our Sliding Widgets implementation, their cursor functions across the desktop, providing an opportunity to examine its behavior in real-world interfaces. In contrast, the Bubble Cursor only requires the context of a single frame to interpret targets, and it does not require deep knowledge of an element's style to render its overlay. Our Sliding Widgets require this complex modeling, and we therefore advance the state-of-the-art in pixel-based methods with new support for modeling widgets in multiple states and for styling new widgets to be consistent with existing interfaces.

Improving Touch Interaction. A fundamental challenge in touch-based interaction is addressing the fat finger problem, the ambiguity caused when a finger touches multiple targets. Our implementation of Moscovich et al.'s Sliding Widgets is exemplary of a large body of techniques that also explore the problem (e.g., see [24] for an enumeration of many techniques). Few of these techniques were specifically designed for hybrid touch and mouse devices, but many have explored touch support for legacy interfaces. These include techniques designed to overcome the occlusion of the target and selection point, such as [3,38,40,41]. Other techniques address the ambiguity caused by reducing a touch's contact area down to a single selection point, such as Pointing by Zooming and Rubbing [1,4,30]. Although these techniques offer great potential, many of them remain difficult to evaluate and deploy in the context of real interfaces, due to the difficulty of modifying existing interfaces at runtime. Thus our new pixel-based methods offer the opportunity to better understand and deploy techniques like these, helping researchers bridge the gap from the lab to the field.

IDENTIFYING AND INTERPRETING ELEMENTS

We interpret interfaces using Prefab's pixel-based methods for reverse engineering interface structure [9,10,11]. Specifically, we use Prefab Layers and Prefab Annotations to identify an interface hierarchy and then recover Sliding Widget metadata for nodes in that hierarchy.

Prefab Layers structure pixel interpretation as a series of tree transformations. The root corresponds to the processed image, and each identified interface element is added as a node in the tree. Each transformation might add new elements, tag elements with metadata, or remove elements. Transformation code is organized into layers, which can be grouped into chains. An interface is then interpreted by executing a layer chain: the raw image is passed into the first layer, then the output of each layer is passed into the next. Developers reuse and compose existing functionality by concatenating layer chains.

Prefab Annotations support robust annotation of interface elements with metadata that has been inferred, provided by a developer, or collected from end-users of pixel-based enhancements. More concretely, an annotation stores metadata to be applied to a node in a recovered hierarchy. For example, Figure 2's green node has been annotated with metadata indicating it should be replaced with a Sliding Button. Collections of annotations are stored in libraries that layers use to inform their interpretation logic.

Figure 3 shows the resulting layers implemented for our enhancement along with their accompanying annotation libraries. The black layers and libraries are Prefab's existing methods for identifying interface hierarchy. We implement the layers shown in red, and populate the red annotation libraries with data specific to Sliding Widgets. The next two subsections first present Prefab's existing layers and annotations and then describe our layers and annotations.

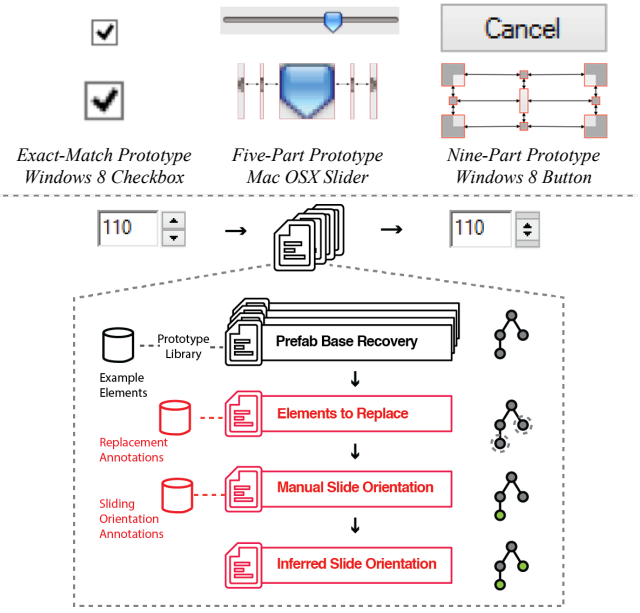


Figure 3: Prefab uses examples of interface elements to generalize prototypes of the appearance of families of widgets. Building from a library of these prototypes, we interpret which elements to replace with Sliding Widgets as well as their sliding orientation. Our novel methods are shown in red, while Prefab's existing methods are in black.

Identifying Interface Hierarchy with Prefab

Prefab's identification layers use annotation libraries containing images of example elements (e.g., an image annotated to identify a region containing a button). The layers then extract a library of *prototypes* generalized from these annotations (e.g., generalizing the example button to describe all buttons of the same type). A prototype describes an arrangement of pixels, and the layers use two high-level strategies to identify elements based on prototypes: (1) exactly matching prototype pixels against an image, or (2) modeling prototype background and then differencing pixels in an image to identify foreground interface elements. Prototype parts can be features (defined as exact patches of pixels) or regions (defined as methods for painting areas of variable size, such as gradients or repeating patterns). The top of Figure 3 illustrates three prototypes selected to show a range of complexity.

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 standard checked Windows 8 checkboxes.

A more complex *slider* prototype uses multiple parts to account for the variable length of the slider and the variable thumb position. Five parts characterize 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 illustrated example and identifies all standard Mac OS X sliders.

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 8 default buttons and any text or icons painted over their gradient background. It was generalized from the illustrated example button. Nine-part prototypes are first identified by matching their four corners and four edges. Prefab then uses the interior content region to identify elements painted over the background. Discussion of content regions, including how Prefab generalizes a background from examples that include foreground elements, is available in [11].

Interpreting Replacement and Sliding Widget Direction

We have described how Prefab identifies a hierarchy of elements, but any hierarchy is by itself insufficient for most applications. In our case, Sliding Widgets requires metadata indicating (1) which elements should be replaced and (2) the direction in which a replacement Sliding Widget should slide. To obtain these, we combine automated interpretation of Prefab's recovered hierarchy with social annotations, wherein people interactively correct erroneous behavior. Interfaces are procedurally generated, so their pixel-level appearance rarely changes. Familiar interfaces will therefore be thoroughly annotated and "just work". But our combination of automatic interpretation and social annotations allows end-users or designers to interactively correct newly-released or niche interfaces.

For manual annotation, any node can be tagged with a type of Sliding Widget to replace it (e.g., *sliding button*, *sliding dropdown menu*). Nodes can be explicitly tagged *do not replace*, or can be implicitly unavailable for replacement due to replacement of an ancestor. For sliding direction, elements can be tagged *left*, *right*, *up*, *down*, or a diagonal direction (e.g., *down right*). Finally, for multi-function Sliding Widgets such as the spinner replacement in Figure 3, we allow *horizontal* and *vertical* tags. We store tags using support provided by Prefab Annotations, and tag nodes using two layers. Specifically, for each annotation we store a path descriptor based on properties of an element, its location in the hierarchy, and its ancestors. At runtime the layer retrieves annotations by matching against each node's path. Thus, the first layer tags the set of nodes annotated with replacement metadata, and the second specifies the direction for those nodes.

Social annotation can be sufficient in a broad deployment that leverages social mechanisms, but can be expedited by even minimal automation. We currently include a layer that automatically tags nodes with a sliding direction. This layer iterates over elements marked for replacement and assigns directions in a rotating clockwise manner. We ignore elements in dense layouts, and leave widgets in sparse layouts with a default *left* or *horizontal* direction. Finally, the layer ignores any element with a direction assigned by the preceding exact-match layer, thus allowing end-users or designers to manually override the default. Our later section on Examining Sliding Widgets presents examples of interfaces where it is desirable to override sliding direction due to unexpected usability issues in real-world interfaces.

MODELING AND LINKING WIDGET STATES

The previous section describes how Prefab interprets elements within a static frame. Because our methods modify interfaces at runtime, defining the behavior of Sliding Widgets requires monitoring an interface many frames per second and then programmatically redirecting input to manipulate that interface. For example, when a person activates a Sliding Widget by moving its thumb, the system must send a click to the underlying original element. And if that button or any other widget then becomes disabled in response to the interaction, the Sliding Widgets should mirror this change in their behavior and appearance.

The solution to these challenges comes from the insight that a change in appearance of an interface element corresponds to an underlying change in widget state. We therefore model widget dynamics using finite state machines. Although a widget may not be explicitly implemented with an internal state machine, the event-driven nature of interfaces means such a state machine is generally implicit (i.e., widgets exist in some state and change that state in response to interaction events). State machines offer an explicit model of the dynamics of interface elements, which we use to provide *getters* and *setters* for monitoring and manipulating widgets. Specifically, getters allow enhancements to subscribe to events fired when there is a traversal across one or more edges (e.g., signifying rollovers, clicks, drags). Setters can prepackage logic that sends input to an element, transitioning it to a new state.

The critical challenge in modeling state machines is that there are both: (1) a wide variety of existing widgets to be translated into Sliding Widgets, and (2) several types of Sliding Widgets, each requiring a different mapping. A naïve approach would therefore create a combinatorial explosion translating each widget to each Sliding Widget. We instead decouple parts of the translation using an approach similar to a Model-View-Controller pattern. A *state model* is the model, a Sliding Widget or other enhancement the view, and a *linker* the controller that synchronizes state models of the original interface with the representations used in a pixel-based enhancement. Figure 4 illustrates this decomposition.

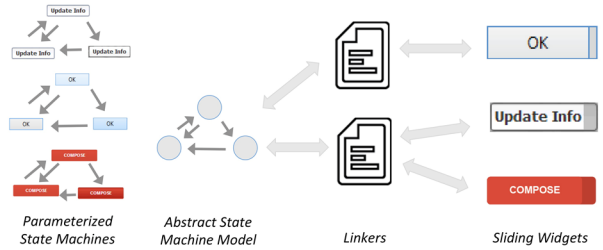
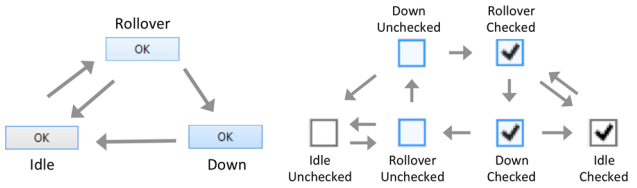


Figure 4: Abstract State Models and Linkers decouple the logic of translating widgets to Sliding Widgets, thus avoiding the combinatorial explosion of direct translation.



Button State Machine
Windows 8 Default Button

Checkbox State Machine
Windows 8 Default Checkbox

Figure 5: Abstract State Models are parameterized with prototypes describing the appearance of a widget. These two state models are parameterized with prototypes generated from Windows 8 Default Buttons and Checkboxes.

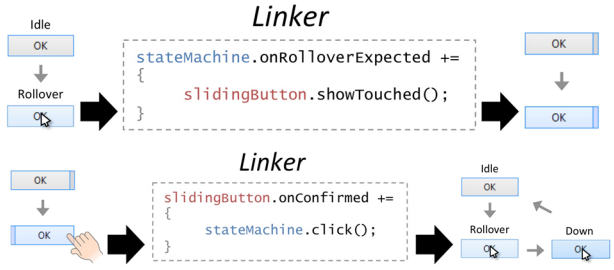


Figure 6: Linkers synchronize state models and Sliding Widgets. These two snippets are from a linker that synchronizes Sliding Buttons with mouse-based buttons.

Abstract State Models

An abstract state model is defined as a combination of states and transitions describing the behavior of a class of widgets. This state model is then parameterized with prototypes describing the appearance of a particular widget in each of the states. For example, Figure 5 presents two state models parameterized to represent behavior for the Windows 8 default button and checkbox. The same models could be parameterized with different prototypes to represent different buttons or checkboxes.

Abstract state models build on Prefab's rudimentary support for observing a transition from one prototype to another [10]. Each transition is defined by a prototype that initiates the transition, a prototype that triggers the transition, and a set of constraints. At runtime, Prefab checks for transitions that are in progress and could be triggered subject to their constraints. After given the option to trigger, these transitions are then given the option to expire (i.e., removed from the set in progress). Finally, the current frame is examined for prototypes that initiate new transitions, which are added to the set in progress.

An abstract state model composes multiple transitions, each defining an edge between two states. States are then parameterized by providing the specific prototypes used in each transition. There are several potential approaches to parameterizing a state model, including interactive authoring tools or more automated methods that passively observe interaction with interfaces, choose among possible state models, and populate each state. We currently use an authoring tool, leaving integration of more advanced methods for future work. Importantly, our methods separate modeling of state machines from recognition of widget appearance, allowing development of abstract models that can be used across a variety of concrete widgets.

At runtime our system maintains a set of parameterized state models that are potentially in progress. This set is populated using the active transitions monitored by Prefab. The triggering of a transition invokes a traversal across the corresponding edge in a state model. When a transition expires, its state model is given the option to expire, denoting a widget is no longer present in the interface. Abstract state models include setters for transitioning between states via manipulation of the interface. Specifically, the setter logic executes input events necessary for traversing edges in a state model. For example, the state model in Figure 5 packages a "click" function that sends mouse down and up events to the source application. This code is defined in the abstract state model, requiring only a single definition for any widgets that share this behavior.

Linkers

A state model defines the dynamics of an element, but those dynamics also need to be related to a Sliding Widget. We therefore provide *linkers*. Specifically, a linker (1) listens for edge traversals in a state machine and updates the Sliding Widget accordingly, and (2) listens for interaction events fired in a Sliding Widget and updates the state model. A single linker can be implemented for an abstract state model paired with a class of Sliding Widgets. For example, Figure 6 illustrates a linker that connects a button abstract state model to a Sliding Button, highlighting the flow of events passed between them.

Upon observing an interaction with a Sliding Widget, a linker can induce arbitrarily complex state machine manipulations. For example, our linker in Figure 6 executes a range of actions depending on the observed Sliding Button interaction. The simplest case is when a user touches down on a Sliding Button and the linker induces a mouse rollover, traversing a single edge in the state machine. In contrast, confirming activation of a Sliding Button induces multiple edge traversals. The linker sends a click to the underlying button, which transitions it first to the state machine's pressed state and then back to the rollover state. Importantly, the separation of a linker from the Sliding Widget allows designers to create Sliding Widgets without a need to implement complex input redirection logic.

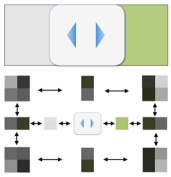
Although a state model can be manipulated by sending input events to the source interface, its state does not change until the underlying element's appearance changes. The length of this delay depends on time it takes to process input, and could be problematic for Sliding Widgets or other enhancements that want to provide instant visual feedback after an interaction. To address this problem, each active state model maintains pointers to two states: the *current* state and the *expected* state. The expected state reflects the state to which the machine is expected to traverse, but has not yet visualized. When a state model is

manipulated via a setter, it first sets its expected state and then sends input to the underlying widget. A state model also fires an event when its expected state changes (e.g., `onRolloverExpected` in Figure 6), allowing a linker to provide instant updates to the Sliding Widget.

MAPPING WIDGET APPEARANCE

We have presented methods for modeling existing widgets, but there is also a challenge in styling the appearance of the runtime enhancement. Because our Sliding Widgets overlay existing interfaces, it is important that they maintain a style that is consistent with the underlying interface.

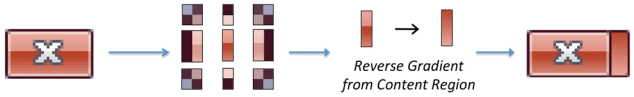
We address this problem with *mappers*, which automatically style Sliding Widgets consistent with their underlying original interface. In our solution, we first separate Sliding Widget style from content as in Hudson and Smith [19]. We also use customizable parts as in Hudson and Tanaka [20]. We then use mappers to translate Prefab prototype parts to the Sliding Widget parts. Prototypes model the pixel-level appearance of existing elements, and so we extend their usage from identification to also informing the rendering of new widgets.



Decomposition of a Sliding Widget into customizable parts is straightforward. Parts render fixed bitmaps or repeating patterns, such as simple repeating colors or more complex gradients. The pixels used in these parts are parameterized to define a specific appearance. Here a Sliding Button is decomposed into eleven parts, each parameterized with a simple appearance. Four bitmaps define the corners of the slider, four patterns the edges, two patterns the left and right troughs behind the thumb, and a single bitmap for the thumb itself. We use similar but more complex models to render other Sliding Widgets, such as the Sliding Spinners and Sliding Dropdown Menus in Figure 1.

Designers can leverage these decompositions to manually style Sliding Widgets, but our mappers expedite this process with methods for styling Sliding Widgets consistent with their underlying original interface. Mappers can be designed for reuse among many prototypes or for a specific prototype. They accept source prototypes and output Sliding Widgets with their parts parameterized based on those prototypes. The following examples illustrate three mappers currently in our framework: basic one-part mappers, nine-part mappers, and multi-prototype mappers.

One-part mappers translate one-part prototypes into Sliding Widgets. An example we have implemented uses two colors extracted from the prototype to define chrome in a Sliding Button. Specifically, we render the corners and edges of the Sliding Widget using the darkest color in the prototype. We then render the trough by extracting the lightest color in the prototype and creating a gradient from the darkest color to the lightest color. The above example shows this one-part mapper applied to a save button.



Nine-part mappers obtain more complex renderings using Prefab's nine-part prototypes. This is an example nine-part mapper that translates the appearance of a Windows 8 close button. It maps the corners and edges of the button prototype to the Sliding Widget thumb. It then obtains a recessed look by painting the trough using a reverse of the gradient obtained from the prototype content region.

The previous two examples use parts from a single prototype to parameterize the appearance of a sliding widget. It is also possible to perform more complex mappings using multiple prototypes. Here we show an example multi-prototype mapper where we map from prototypes that define the two arrows of a spinner. The content regions of those prototypes provide the up and down arrows, which are mapped into the thumb of the slider. Our associated video shows additional Sliding Widgets created using multi-prototype mappers, including a Sliding Checkbox created from two one-part prototypes that describe the appearance of a Windows 8 checkbox in its checked and unchecked states.

EXAMINING SLIDING WIDGETS IN THE WILD

Sliding Widgets and other techniques addressing the fat finger problem are often designed, discussed, and evaluated using fields of abstract widgets or other limited testbeds. Because our implementation provides the ability to deploy Sliding Widgets, we sought to determine what insights we could gain from examining them in real-world interfaces. Our findings identify three challenges: (1) the challenge of supporting both area and point cursors within the same interface, (2) limitations of sliding in complex interfaces, and (3) limitations of replacing individual widgets.

Challenges with Supporting Area and Point Cursors

A major challenge is defining Sliding Widgets that work within the same interface as mouse-based widgets. Consider the spinner shown in the bottom-center of Figure 7, which illustrates a common scenario where a spinner controls the contents of an adjacent textbox. The spinner can be replaced with a Sliding Widget, but it then becomes difficult to manipulate the textbox directly. This is because Sliding Widgets leverage the entire contact area of touch input, but the touch here also overlaps the text field for which area cursor input is undefined. Sliding Widgets can resolve this ambiguity by alternating their orientation for disambiguation among other adjacent Sliding Widgets, but this strategy does not resolve the scenario where a touch area overlaps both a Sliding Widget and an element designed only for point-cursor interaction.

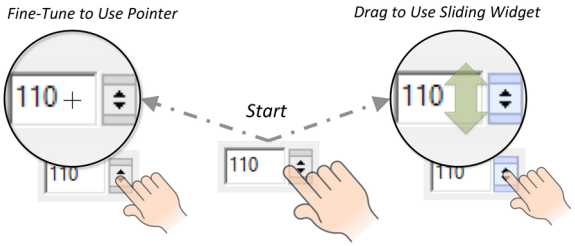
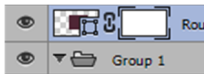


Figure 7: We modify Vogel and Baudisch's Shift [38], employing pointing when the user fine-tunes and contact area-based touches when the user drags.

Sliding Widgets were unusable without the ability to disambiguate between pointing and sliding, so we developed a strategy for gracefully degrading from a contact area to a standard point cursor. More concretely, when a person touches a point-cursor element, we pass the standard touch point to the underlying interface (typically computed as the contact area centroid). However, when a person touches a Sliding Widget, our overlay will send the entire contact area to the widget. In dense layouts, we employ an enhanced implementation of Vogel and Baudisch's Shift to enable fine cursor adjustment [38]. After a touch event, Shift creates a callout showing a copy of the occluded screen area and places it in a non-occluded location. Shift was designed for point-cursor interaction, so we extend it to support area cursor input. Our goal is to make it possible to swiftly activate a Sliding Widget without eliminating precise control of a pointer. Thus, the core challenge in this task is to separate fine-tuning input from the dragging needed to activate a Sliding Widget.

Figure 7 storyboards our solution for disambiguating elements in dense layouts. When a user dwells over an element, we present a callout showing the occluded screen area. We then distinguish fine-tuning from dragging using a threshold on input velocity. Specifically, slow movements below the threshold are categorized as fine-tuning, and fast movements are treated as drags. Fine-tuning moves the position of a point cursor over mouse-based targets, and for any Sliding Widgets under the point cursor it sends touch down events. Alternatively, dragging only manipulates Sliding Widgets. The drag moves a Sliding Widget's thumb in the direction of the drag, allowing activation while hiding that input from any other point-cursor targets.

Limitations of Sliding in Complex Interfaces



Sliding Widgets use the metaphor of real-world sliding manipulations, but it is unclear how this applies to certain elements. One example is in hierarchical widgets, where an interactive element contains an interactive child element. In this screenshot of a panel in Adobe Photoshop, each row is clickable (to select a layer) along with its inner buttons (to toggle layer visibility). Unfortunately, this has no clear Sliding Widget analog. Sliding Widgets could potentially be nested in a larger Sliding Button, but this would be awkward when sliding either the outer or inner widgets.

We also found limitations of the sliding metaphor in the context of groups of related elements. Moscovich et al. demonstrate the promising feature of Sliding Widgets where a list of elements is selected in a single stroke. Our associated video also demonstrates an example of this interaction in the context of a Skype dialog. However, in most real interfaces where multiple elements are frequently toggled together, developers provide a "select all" toggle. In other cases, elements tend to be unrelated or infrequently toggled, leaving single stroke interactions undesirable.

We believe this problem is largely due to conflicts between Sliding Widget behavior and the intentional design of interfaces for point cursors. More concretely, it may be possible to redesign an entire interface to exploit the advantages of single-stroke sliding interactions, but only at the obvious cost of needing to overhaul much of the design. Dixon et al. raised a similar tension between target-aware pointing and existing pointing-based interfaces [8], so perhaps this problem surfaces a larger conflict present in designing surface-level modifications to existing designs.

Limitations of Individual Widget Replacement

Our implementation mostly replaces individual elements with individual Sliding Widgets. However, this approach can introduce conflicts when multiple widgets are related.

We partially anticipated this in our implementation of a Sliding Spinner. A naïve implementation would replace each button in the spinner with a Sliding Button and alternate their direction to help with disambiguation. However, this can conflict with a person's expectations of the interface if the chosen sliding direction is different than the semantic direction of the spinner. For example, the spinner in Figure 1 adjusts the numerical value in its textbox, and sliding horizontally may seem less appropriate than sliding vertically. The problem is magnified if the arrows from the original spinner are mapped into the appearance of the Sliding Widget. We addressed this problem with the Sliding Spinner that understand the relation between the two buttons and behaves appropriately.

While our strategy for grouping multiple related widgets works for spinners, the strategy is much more difficult to implement for larger collections of related widgets. For example, consider this enhanced Windows 8 Calculator interface. It includes a typical grid of buttons, but replacing each individual button yields an interface that is jarring and difficult to use. The calculator's original layout is relatively dense, so it seems useful to alternate sliding directions. But then a person has to carefully inspect each button to understand how to slide it. Instead, the entire panel of buttons should be replaced with a Sliding Widgets optimized for these types of grid layouts.

This problem suggests that a more global understanding of interface layout is necessary for replacing some elements. One possible strategy is to employ a mechanism similar to

Nichols et al.'s Smart Templates for replacing entire groups of controls [27]. This technique uses parameterized templates to specify when different conventions might be applied. Our approach to replacing spinner widgets can be seen as an initial example of this, but we imagine there is an opportunity to extend this idea to more sophisticated templates capturing a variety of conventions and semantics.

DISCUSSION AND CONCLUSION

Although we describe our pixel-based methods in the context of implementing Sliding Widgets, our modeling of state and our approaches to styling widgets using prototype parts can enable and inform a variety of future enhancements. For example, other enhancements can directly re-use our state models. An enhancement to support crossing-based widgets on hybrid pen and mouse devices could use our same state models, implement new linkers relating their crossing widgets to our models, and style their widgets using our techniques. Even more complex enhancements might go beyond replacing individual widgets, using our models to drive entire new interfaces automatically generated by a system like Supple [15].

We described how our state models provide getters and setters for widget state, encapsulating low-level details required for monitoring and manipulating widgets. These turned out to be additionally beneficial because they provide an API that is similar to the API of the underlying toolkit itself. For example, our checkbox state model provides a familiar `getChecked()` method, which hides details of what states and prototypes the model uses to represent the checkbox. We believe that there is a rich opportunity for future work that investigates a more general set of *semantic views* that encapsulate pixel-based methods. The goal of these views would be to provide developers of pixel-based enhancements with a higher-level toolkit that is more similar to existing interface toolkits. For example, a `getLabel()` method on a semantic view of a button would manage the details of obtaining the button's prototype, recovering its content, and processing those pixels to recover its text.

Our linkers automatically map Sliding Widget interactions into a sequence of mouse events to manipulate the underlying interface element. Current windowing systems are not designed to support this type of behavior. In particular, the overlay and the underlying enhancement must share the same input stream, which requires blocking input from one or the other during different stages of an interaction. This limitation also makes it difficult to support simultaneous manipulation of elements (e.g., Moscovich et al. suggest using an area cursor to drag two sliders at the same time). Deeper exploration of new input management frameworks is an opportunity for future work. For example, a framework might provide multiple input streams, separating live input from redirected input.

We implemented several basic linkers for synchronizing Sliding Widgets and state machines in most interfaces, but we imagine developers will extend these or create more sophisticated linkers tailored for specific applications. As one motivation, our example linker maps touches to mouse rollovers, thus making it possible to view tooltips or other rollover responses in existing interfaces. But Moscovich et al. also demonstrated more complex responses to touch events with their action-preview-on-touch behaviors [24]. In their design, touching a button displays a preview of the action it will perform if activated. Future work might therefore examine linkers that execute these previews by monitoring and manipulating multiple state models. Importantly, our Model-View-Controller pattern for separating linkers from state models and Sliding Widgets makes it possible to define custom behaviors without modifying state model or Sliding Widget code.

Our application is the first practical general-purpose implementation of Sliding Widgets. There is nearly always a fundamental gap between the knowledge that can be gained in lab studies versus the implications of this knowledge for real-world contexts, and we believe this work narrows the gap for Sliding Widgets. We are exploring the best way to deploy our application on Windows 8 hybrid devices to further bridge the gap from the lab to the field. We also envision tools accompanying the deployment where developers can gather logged usage data to provide insight into real-world challenges (e.g., testing alternative designs, visualizing problems reported by end-users of pixel-based enhancements). Our hope is to help catalyze interaction research in escaping the lab, putting it into the hands of end-users who stand to benefit from the field's rich innovation.

ACKNOWLEDGEMENTS

We thank Scott E. Hudson and Daniel S. Weld for advice, discussion, and thoughts that helped shape this research. This work was supported in part by the National Science Foundation under award IIS-1053868.

REFERENCES

1. Albinsson, P.-A. and Zhai, S. High Precision Touch Screen Interaction. *CHI 2003*, 105–112.
2. Banovic, N., Grossman, T., Matejka, J., and Fitzmaurice, G. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. *UIST 2012*, 83–92.
3. Baudisch, P. and Chu, G. Back-of-Device Interaction Allows Creating Very Small Touch Devices. *CHI 2009*, 1923–1932.
4. Benko, H., Wilson, A.D., and Baudisch, P. Precise Selection Techniques for Multi-Touch Screens. *CHI 2006*, 1263–1272.
5. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. Automation and Customization of Rendered Web Pages. *UIST 2005*, 163–172.
6. Chang, T.-H., Yeh, T., and Miller, R. Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. *UIST 2011*, 245–254.
7. Chang, T.-H., Yeh, T., and Miller, R.C. GUI Testing Using Computer Vision. *CHI 2010*, 1535–1544.

8. Dixon, M., Fogarty, J., and Wobbrock, J. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. *CHI 2012*, 3167–3176.
9. Dixon, M. and Fogarty, J. Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. *In Preparation.*
10. Dixon, M. and Fogarty, J. Prefab : Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *CHI 2010*, 1525–1534.
11. Dixon, M., Leventhal, D., and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *CHI 2011*, 969–978.
12. Eagan, J.R., Beaudouin-Lafon, M., and Mackay, W.E. Cracking the Cocoa Nut: User Interface Programming at Runtime. *UIST 2011*, 225–234.
13. 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.
14. Fujima, J., Lunzer, A., Hornbæk, K., and Tanaka, Y. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. *UIST 2004*, 175.
15. Gajos, K.Z., Wobbrock, J.O., and Weld, D.S. Automatically Generating User Interfaces Adapted To Users' Motor and Vision Capabilities. *UIST 2007*, 231–240.
16. Grossman, T. and Balakrishnan, R. The Bubble Cursor : Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *CHI 2005*, 281–290.
17. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. *UIST 2010*, 143–152.
18. 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.
19. Hudson, S.E. and Smith, I. Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit. *UIST 1997*, 159–168.
20. Hudson, S.E. and Tanaka, K. Providing Visually Rich Resizable Images for User Interface Components. *UIST 2000*, 227–235.
21. Hurst, A., Hudson, S.E., and Mankoff, J. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI 2010*, 11–20.
22. Kumar, R., Talton, J.O., Ahmad, S., and Klemmer, S.R. Bricolage : Example-Based Retargeting for Web Design. *CHI 2011*, 2197–2206.
23. 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*, 943–952.
24. Moscovich, T. Contact Area Interaction with Sliding Widgets. *UIST*, (2009), 13–22.
25. Nichols, J., Hua, Z., and Barton, J. Highlight: A System for Creating and Deploying Mobile Web Applications. *UIST 2008*, 249–258.
26. Nichols, J. and Lau, T. Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites. *IUI 2008*, 149–160.
27. Nichols, J., Myers, B. a., and Litwack, K. Improving Automatic Interface Generation with Smart Templates. *IUI 2004*, 286–288.
28. 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.
29. Olsen, D.R., Taufer, T., and Fails, J.A. ScreenCrayons: Annotating Anything. *UIST 2004*, 165–174.
30. Olwal, A., Feiner, S., and Heyman, S. Rubbing and Tapping for Precise and Rapid Selection on Touch-Screen Displays. *CHI 2008*, 295–304.
31. Pongnumkul, S., Dontcheva, M., Li, W., Wang, J., Bourdev, L., Avidan, S., and Cohen, M.F. Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications. *UIST 2011*, 135–144.
32. Potter, R. Triggers: Guiding Automation with Pixels to Achieve Data Access. *A. Cypher, eds. MIT Press*.
33. Savva, M., Kong, N., Chhajta, A., Fei-fei, L., Agrawala, M., and Heer, J. ReVision : Automated Classification , Analysis and Redesign of Chart Images. *UIST 2011*, 393–402.
34. St Amant, R., Lieberman, H., and Potter, R. Visual Generalization in Programming by Example. *Communications of the ACM 43*, 3 (2000), 107–114.
35. St Amant, R., Riedl, R., Ritter, F.E., and Reifers, A. Image Processing in Cognitive Models with SegMan. *HCII 2005*.
36. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*, 309–318.
37. Tan, D.S., Meyers, B., and Czerwinski, M. WinCuts : Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *CHI 2004*, 1525–1528.
38. Vogel, D. and Baudisch, P. Shift : A Technique for Operating Pen-Based Interfaces Using Touch. *CHI 2007*, 657–666.
39. Waldner, M., Steinberger, M., Grasset, R., and Schmalstieg, D. Importance-Driven Compositing Window Management: *CHI 2011*, 959–968.
40. Weldon, J. and Shneidermanss, B. Improving the Accuracy of Touch Screens: An Experimental Evaluation of Three Strategies. *CHI 1988*, 27–32.
41. Wigdor, D., Forlines, C., Baudisch, P., Barnwell, J., and Shen, C. LucidTouch : A See-Through Mobile Device. *CHI 2007*, 269–278.
42. Yeh, T., Chang, T.-H., and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009*, 183–194.
43. Yeh, T., Chang, T.-H., Xie, B., Walsh, G., Watkins, I., Wongsuphasawat, K., Huang, M., Davis, L.S., and Bederson, B.B. Creating Contextual Help for GUIs Using Screenshots. *UIST 2011*, 145–154.
44. Zettlemoyer, L.S. and St. Amant, R. A Visual Medium for Programmatic Control of Interactive Applications. *CHI 1999*, 199–206.
45. Zettlemoyer, L.S., Amant, R.S., and Dulberg, M.S. IBOTS : Agent Control Through the User Interface. *IUI 1999*.