Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure

Morgan Dixon and James Fogarty
Computer Science & Engineering
DUB Group, University of Washington
{mdixon, jfogarty}@cs.washington.edu

ABSTRACT
Current chasms between applications implemented with different user interface toolkits make it difficult to implement and explore potentially important interaction techniques in new and existing applications, limiting the progress and impact of human-computer interaction research. We examine an approach based in the single most common characteristic of all graphical user interface toolkits, that they ultimately paint pixels to a display. We present Prefab, a system for implementing advanced behaviors through the reverse engineering of the pixels in graphical interfaces. Informed by how user interface toolkits paint interfaces, Prefab features a separation of the modeling of widget layout from the recognition of widget appearance. We validate Prefab in implementations of three applications: target-aware pointing techniques, Phosphor transitions, and Side Views parameter spectrums. Working only from pixels, we demonstrate a single implementation of these enhancements in complex existing applications created in different user interface toolkits running on different windowing systems.

Author Keywords
Prefab, user interface toolkits, pixel-based reverse engineering.

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

General Terms
Human Factors

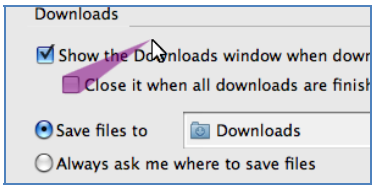
INTRODUCTION AND MOTIVATION
Nearly every modern graphical interface is implemented using some form of user interface toolkit. These toolkits provide libraries of widgets and associated frameworks that reduce the time, effort, and amount of code required to implement an interface. This provides obvious advantages to developers, but also benefits the users of applications created with these toolkits. For example, the consistent look and feel of applications created with a toolkit allows people to better transfer skills between those applications. Easier interface development also enables the iterative exploration of a greater variety of potential designs, which is critical to successful iterative design processes. User interface toolkits have enabled many successes in the past forty years of human-computer interaction research and practice [12].

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.

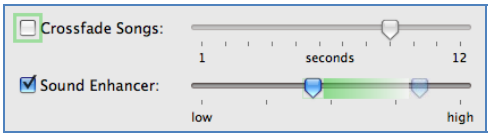
Unfortunately, the current state of user interface toolkits also creates significant challenges for research and practice. When working with an existing toolkit, it is generally difficult or impossible to modify the core behavior of that toolkit's widgets [6]. A researcher who wants to study a new interaction technique in the context of rich and realistic applications, or a practitioner who wants to adopt a technique from the literature, is generally faced with the prospect of re-implementing huge portions of a toolkit or an application. Most instead choose to demonstrate techniques only in toy applications or to develop applications based in simple combinations of standard widgets [6]. This problem is magnified by the fact people typically use a wide variety of applications built with several different toolkits. Each is implemented differently, and so it is difficult to consistently add new functionality. For example, target-aware pointing techniques have long been known to have many potential advantages [3, 24], but the diversity of implementations of existing applications and toolkits continues to limit their broader exploration and adoption. The difficulty of implementing new interaction techniques in new and existing applications and toolkits is limiting the progress and impact of human-computer interaction research.

This paper explores an approach based on the single largest commonality of this variety of applications and toolkits: they all ultimately produce pixels on a display. If it were possible to interpret the structure of these pixels, a variety of advanced behaviors might be implemented independent of individual applications and toolkits. Our *Prefab* system examines Pixel-based Reverse Engineering For Advanced Behaviors, and builds upon four fundamental insights:

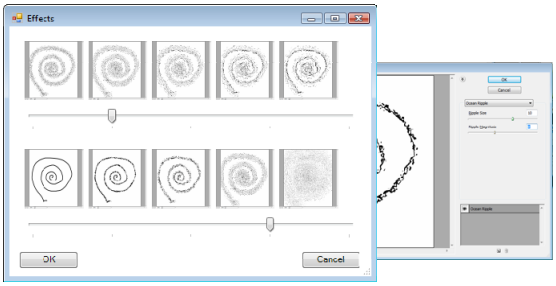
- The graphical desktop is not a physical scene. Computer vision algorithms developed to address such problems as perspective, distortion, shadows, and occlusion may be overkill or even inappropriate for this problem.
- Because the pixels constituting a particular widget are defined procedurally, those pixels are typically identical across invocations of an application on the same or different computers. If a system can be taught the definition of a particular widget, it will rarely change.
- Because consistency within an application is important to the usability of that application, widgets of the same



Grossman and Balakrishnan's Bubble Cursor is an important target-aware pointing technique [3], but is difficult to deploy because existing toolkits do not support it. These are screenshots of a Bubble Cursor implemented using Prefab, highlighting the nearest target in a Firefox settings dialog on Macintosh OS X and in a YouTube movie player.



Baudisch *et al.*'s Phosphor uses afterglows to illustrate interface changes [1]. This screenshot of our Prefab implementation of Phosphor shows a recently unchecked checkbox and a recently manipulated slider in an iTunes settings dialog on an Apple Macintosh.



Terry and Mynatt show that Side Views parameter spectrums can support effective exploration of multi-parameter spaces [22]. We use Prefab to create a parameter spectrum for an Adobe Photoshop filter running on Microsoft Windows Vista. We populate the spectrum by using Prefab to automatically interpret the interface of Photoshop's filter dialog.

Figure 1: Prefab reverse engineers raw pixels to recover the structure of graphical user interfaces. Because Prefab is agnostic of an application's underlying implementation, it enables advanced behaviors across a wide variety of applications and toolkits. All of these demonstrations of Prefab enhancements are discussed in greater detail later in this paper and in our associated video.

type are typically illustrated using similar pixels. It is likely not necessary to individually define each widget in every application, but may instead be possible to learn definitions of entire families of widgets (e.g., all Microsoft Windows Vista Steel buttons, all Java Metal checkboxes, all Apple Cocoa scrollbars).

- Because consistency across applications is important to the usability and learnability of applications, different toolkits provide similar sets of widgets that share many commonalities (e.g., buttons, checkboxes, scrollbars). Modeling these commonalities, based in part on how such toolkits are implemented, is likely to be important to pixel-based reverse engineering of interface structure.

In short, graphical desktops are not physical scenes, but are instead made of pre-fabricated units combined according to very particular rules. Prefab uses raw pixels to reverse engineer interface structure by identifying these pre-fabricated units and then modeling their relations.

Figure 1 presents several examples of advanced behaviors implemented using Prefab. These examples are embedded in a variety of applications implemented using different toolkits running on different platforms. All are implemented entirely based upon reverse engineering pixels, without knowledge of the underlying toolkit or implementation.

The next section provides a brief description of how Prefab's interpretations can be used to implement advanced behaviors like those in Figure 1. We then provide an introduction to each of Prefab's major components and describe their relationships. Next, we introduce two Prefab models that we use as examples throughout this paper. In the main portion of our technical content, we first present how Prefab uses a library of prototypes to reverse engineer a graphical interface. We then present our current support for creating the necessary library. We next validate Prefab in three example applications and then discuss related work. Finally, we discuss Prefab, its current limitations, and opportunities for future work.

The specific contributions of this work are:

- An architecture for pixel-based reverse engineering of interface structure. Informed by how user interface toolkits paint interfaces, our architecture features a separation of the modeling of widget layout from the recognition of widget appearance.
- Initial methods for effectively implementing necessary components of this architecture and discussion of opportunities to improve and extend these methods.
- Initial methods supporting the creation of Prefab prototype libraries, including a branch-and-bound method for fitting prototype parameters according to positive and negative examples of occurrences.
- A tool that enables a wide variety of applications using pixel-based reverse engineering of interface structure.

INPUT AND OUTPUT REDIRECTION

The examples in this paper are based on modifying the apparent behavior of interfaces using input and output redirection. A basic mechanism is illustrated in Figure 2, wherein (1) a bitmap of an existing *source* window is captured, (2) the contents of the source are interpreted, (3) a modified interface is presented in a *target* window (with the source potentially hidden using virtual desktop methods), (4) input in the target is mapped back the source, which then (5) generates new output that is captured and used to update the target. We discuss prior redirection research in a

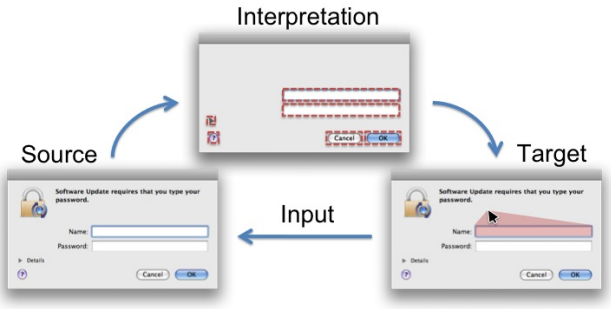


Figure 2: Redirection mechanisms allow modification of the apparent behavior of interfaces through modification of their input and output. Prefab provides a unique approach to rapid pixel-based interpretation of interfaces.

later section [11, 20, 21], but our research is the first to combine redirection with pixel-based interpretation of interfaces. The types of modifications that are possible with these mechanisms depend upon the completeness of the available interpretation, so this paper focuses on Prefab's methods for identifying widgets many times per second. Our later discussion provides more insight into our example applications, current limitations, and applications that could be enabled by further improvements to Prefab.

PREFAB COMPONENT OVERVIEW

The major components of Prefab's architecture are: *models*, *prototypes*, *parts* (including *features* and *regions*), *constraints*, and *transitions*. This section briefly introduces each component so that future sections can provide a detailed discussion of their usage and relationships.

A *model* consists of a set of abstract *parts* and a set of concrete *constraints* regarding those *parts*. A typical *model*, for example, might include several *constraints* requiring that particular *parts* are adjacent. The *parts* of a *model* are abstract, and so a *model* does not describe any particular widget or set of widgets. Instead, a *model* describes a pattern for composing a set of *parts* to create a widget.

Parts can be either *features* or *regions*. A *feature* stores an exact patch of pixels (exact colors in a spatial arrangement of an exact size). For reasons that are obvious when we discuss how Prefab reverse engineers a particular interface, every *model* includes at least one *feature*. A *region* stores a procedural definition of a method for generating a set of pixels in an area of variable size (e.g., painting a repeating pattern, painting a gradient). Because the same *parts* can be arranged in many different ways, they alone do not describe any particular widget or set of widgets.

A *prototype* parameterizes a *model* with concrete *parts*, characterizing both the appearance of a set of *parts* and applicable *constraints* upon the relationships of those parts. A *prototype* therefore describes the appearance of a particular widget or set of widgets (e.g., the Mozilla Firefox Home toolbar button, all Microsoft Windows Vista Steel buttons). Prefab is implemented as a library of *prototypes*, together with methods for effectively applying those prototypes to identify *occurrences* of widgets.

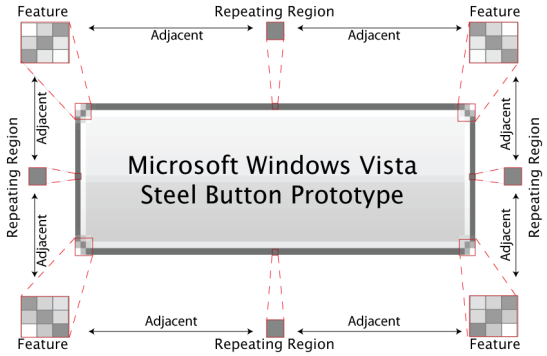


Figure 3: This Prefab prototype for Microsoft Windows Vista Steel buttons is an example of an eight-part model. Four features define the corners, each edge is defined by a region, and constraints require the parts form a rectangle. This prototype recognizes all Microsoft Windows Vista Steel buttons, independent of their interior content.

The separation of layout (described by a *model*) from appearance (when *parts* are specified to create a *prototype*) is critical to Prefab and is informed by important aspects of how toolkits paint widgets. For example, to support widgets with content of varying sizes, toolkits often paint a border of the required size (delimiting space dedicated to a widget) and then paint the content of the widget within that border (e.g., centering a label or an icon within a button, displaying the current value of an editable text field). Similarly, a slider widget paints a trough and a thumb, regardless of the underlying toolkit. Prefab separates such general insight about how widgets are drawn from details of the appearance of individual widgets (e.g., the stroke used to paint a border, the shape of a slider's thumb). For example, a *model* of eight *parts* creating a rectangular border is capable of describing a wide variety of widgets.

Prefab's final major component is a *transition*. Although many potential applications of Prefab can be based on reverse engineering a single frame or state of an interface, many others require interpreting the contents of an interface across multiple frames or states. A Prefab *transition* is defined as a pair of *prototypes* and a set of *constraints* that specify when the *transition* is allowable. When an *occurrence* of the first *prototype* is observed, Prefab begins tracking the *transition*. If an *occurrence* of the second *prototype* is observed that is compatible with the specified *constraints*, the *transition* is fired.

TWO EXAMPLE MODELS

The next two sections discuss the use and creation of a library of Prefab prototypes, consistently using two example models: a *one-part model* and an *eight-part model*. Prefab is an extensible system, and these were selected to provide insight into extension by illustrating the opposite extremes of complexity in our current models.

The *one-part model* is our simplest, consisting of only a single feature. A prototype specifies a single exact patch of pixels, and Prefab identifies occurrences whenever it observes those exact pixels. This model might appear to be a strawman, but our experiences suggest that it can be quite

effective at supporting widgets for which there is not yet a more specialized model (with the obvious limitation that it does not generalize across families of widgets).

The *eight-part model* is illustrated in Figure 3 with its parts parameterized by the appearance of the Microsoft Windows Vista Steel button. Features define the corners, a region defines each edge, and constraints require the parts form a rectangle. This prototype's edges are a single repeating pixel, but other region types are possible (e.g., a repeating sequence, a repeating multi-row pattern, a gradient).

REVERSE ENGINEERING AN INTERFACE

Our goal is to identify all occurrences of widgets from our prototype library in an image of an interface. To support real-time interactive enhancements of interfaces, we want to do this many times per second. Because the graphical desktop is not a physical scene, and because we need to identify many widgets many times per second, techniques developed in computer vision are a poor fit for our problem. For example, Yeh *et al.*'s recently developed Sikuli system uses a combination of template matching and voting based on invariant local features, and it requires a reported 200msec to identify all occurrences of a *single* target [25]. Because we need to more quickly identify all occurrences of *many* widgets, we develop an approach tailored to the recognition of widgets in images of graphical interfaces.

Prefab first conducts a single pass over an image to identify all occurrences of features from the prototype library. Based on the detected features, models generate hypotheses regarding potential occurrences. Actual occurrences are detected by filtering these according to the constraints of the relevant model, including checking the validity of pixels in any regions. After identifying occurrences of prototypes in the current image, Prefab determines whether any relevant transitions have occurred and updates its set of transitions that are potentially in progress. For the sake of clarity, this section presents each step in its simplest form, using our one-part and eight-part models as examples. In our later discussion, we note several aspects of the process presented here that can be optimized for performance.

Locating Features

When a library of prototypes is created, Prefab chooses a non-transparent hotspot within the patch of pixels defining each feature in the library. Prefab constructs a decision tree for determining whether a pixel in an image of an interface is the hotspot of any feature in the tree, as in Figure 4. Each internal node specifies an offset relative to the hotspot, each edge corresponds to the color at that offset, and each leaf corresponds to a feature. Traversing the tree to a leaf tests every pixel in a feature (e.g., the leftmost path in Figure 4 tests the dark grey pixel, then the blue pixel, then the light grey, and finally the yellow pixel). If an internal node lacks an edge corresponding to the color at the specified offset, then traversal ends and the pixel to which the tree is currently being applied is not the hotspot of any feature. This decision tree is stored in the library and evaluated against images of interfaces to locate features at runtime.

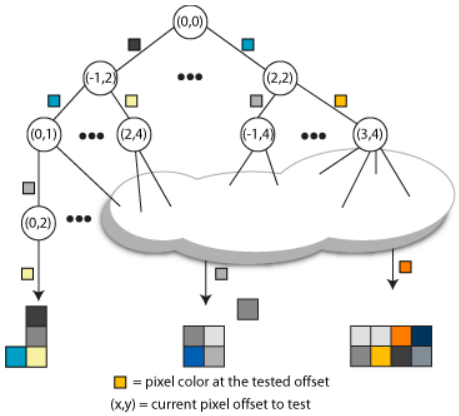


Figure 4: Prefab constructs a decision tree that tests whether a pixel is the hotspot of a feature from the prototype library. It uses this tree to scan an image of an interface, detecting all features in a single pass.

The hotspot for each feature and the offset at each internal node can be chosen arbitrarily, but simple heuristics improve performance. Our implementation currently first tests the non-transparent hotspot (a transparent pixel within a feature indicates the color of that pixel is irrelevant to the feature), then chooses the offset for each internal node that maximizes information gain. When choosing a hotspot for each feature in the prototype library, our implementation chooses a pixel of a color that is least common among all features in the prototype library (using the distribution of colors in the features as a proxy for the distribution of colors in interfaces). These heuristics combine to minimize the tree depth and the length of typical partial traversals.

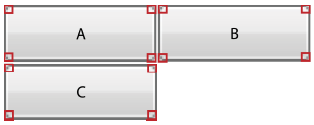
Generating Hypotheses

After identifying all feature occurrences, each Prefab model generates hypotheses of potential prototype occurrences. Importantly, the overwhelming majority of prototypes in the library have already been removed from consideration. Every model contains at least one feature, and Prefab has identified all occurrences of all features, so any prototype that includes features which have not been detected cannot appear in the current image.

Hypotheses for one-part prototypes are trivial to generate. Because the one-part model consists of a single feature, a single hypothesis for a one-part prototype is generated at each location in the image where that feature occurs.

A naïve but sufficient general method is for a model to enumerate all mappings between a prototype's features and occurrences of those features in an image. For example, a model that contains k features, each of which occurs f times in a particular image, could generate f^k unique hypotheses.

Prefab models can apply constraints to generate a much smaller set of hypotheses. For our eight-part model, its four features are constrained to a rectangle. The model therefore generates hypotheses by starting from each occurrence of a prototype's top-left feature, checking to the right for the top-right feature, then down for the bottom-right feature, left for the bottom-left feature, and finally confirming the



Three occurrences of the same eight-part prototype, with their total of 12 features identified.

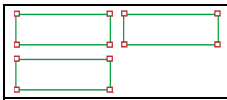
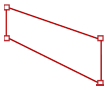
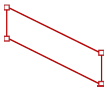
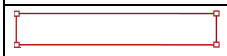
	The correct hypotheses are all detected.
	This combination never generates a hypothesis. Its upper-right feature (bottom-right of the "A" occurrence) is not the prototype's required type.
	This combination includes all four required features, but fails the model's rectangle constraint and never generates a hypothesis.
	This combination includes correct features in a valid layout, but the gap between buttons is inconsistent with the repeating edge region.

Figure 5: Based upon the features detected in an image of an interface, Prefab generates a set of prototype hypotheses and then identifies occurrences by testing those hypotheses against the pixels in the image of the interface.

bottom-left feature is directly below the top-left feature. The number of arrangements considered and the number of hypotheses generated are both far less than the naïve f^k. We have found it relatively easy to implement such methods for efficiently generating small sets of high-quality hypotheses, but more importantly we note that they are implemented in the model and therefore shared by many prototypes. Even if it were relatively difficult to implement an efficient method for generating good hypotheses in a particular model, we believe this effort would be justified if the model could then be applied to a large and wide variety of prototypes.

Detecting Prototype Occurrences

After generating a set of hypotheses, actual occurrences are detected by filtering hypotheses according to the constraints of the relevant model, including checking the validity of pixels in any regions. Hypotheses that pass these filters are reported to applications as prototype occurrences.

Prefab's models can use arbitrary code for constraints before or after region validation. In our current models, we have found the combination of constraints on feature arrangement during hypothesis generation and validation of region pixels to be sufficient and effective. Prefab's support of arbitrarily logic for filtering hypotheses may prove useful in future extensions with additional models.

During region validation, models identify sets of pixels to be validated and delegate validation to the region object. In our eight-part model, for example, a hypothesis defines the locations of the four corner features. Based on these, the model defines four sets of pixels corresponding to the top, right, bottom, and left edges. It then delegates validation of each set of pixels to the region objects in the prototype being tested. For the Microsoft Windows Vista Steel button prototype from Figure 3, the region validates the presence of the single repeating pixel along the edge. Different types of regions (e.g., a repeating sequence, a repeating multi-row pattern, a gradient) take their own approaches to validating pixels delegated to them by the model. If the region rejects the pixels, the model rejects the current hypothesis.

Monitoring Transitions

Prefab applications can be based entirely upon notifications of prototype occurrences, but we have found that many potential applications require observation of a transition from one prototype to another. We therefore added explicit support for observing such transitions with Prefab. Each transition is defined by a prototype whose occurrence initiates monitoring of the potential transition, a prototype whose occurrence indicates it may be appropriate to trigger the transition notification, and a set of constraints (including both support for arbitrary logic and pre-packaged versions of commonly used constraints, such as timeouts).

Prefab maintains a set of transitions that are potentially in progress. After determining which prototypes occur in the current frame (and reporting them to applications), Prefab checks for transitions that are potentially in progress and could be triggered by an occurrence in the current frame. These are given the option of triggering, subject to their constraints. All transitions in the set are then given the option to expire (separating triggering from expiration allows transitions that trigger and then expire, expire without triggering, or trigger multiple times before expiring). Finally, the current occurrences are examined for prototypes that could initiate a transition and those transitions are added to the set potentially in progress.

SUPPORTING PREFAB PROTOTYPE CREATION

There are a range of approaches to developing libraries of prototypes required for Prefab applications. A researcher evaluating an interaction technique in a set of existing applications might create a prototype library for the widgets used in those applications. A practitioner or hobbyist who wants to use a Prefab enhancement with their favorite application might create the necessary library and share it with other users of the application. Communities might create and maintain large shared libraries, perhaps using wiki functionality like that developed for web mashups with d.Mix [4]. Explicit specification might be enhanced with automated learning of prototypes through passive observation of people using everyday interfaces. We are ultimately interested in all of these possibilities, but this initial work focuses on a common first requirement: supporting effective creation of Prefab prototypes.

It is possible to manually specify the parameters of a model corresponding to a particular widget or family of widgets, but such a process is tedious and is also likely to introduce errors. We therefore develop methods for the lightweight interactive specification of Prefab prototypes based on positive and negative examples (examples of widgets that a prototype should or should not identify). Our goal is not necessarily complete automation of example-based prototype creation, as we do not believe this is necessary for supporting the effective creation of prototype libraries. Instead, we believe it is sufficient to reduce the required effort to a small number of clicks. For example, a person

might click on widget in an image of an interface and then choose from a small sorted list of prototypes (e.g., choosing the prototype that indicates their click was on a slider).

We address this goal using a branch-and-bound search [17]. This section discusses lightweight example extraction and our posing of the parameter search problem.

Example Extraction

Most example widgets can be quickly extracted from an image of an interface with one or two clicks. Widgets are designed to be easily visible against their background, and so they typically include well-defined edges. A person using our prototype authoring tool captures one or more images of an interface and then provides one or more clicks in the interior of a widget. Allowing multiple clicks accounts for the case where a widget contains multiple apparently disjoint pieces. Our tool then identifies a set of increasingly large rectangles that contain these clicks, that do not cross pixels where the gradient exceeds a threshold (the zero threshold defining rectangles where all pixels are the same color), and where each larger rectangle contains an additional set of connected pixels where the gradient does exceed the threshold. For a particular threshold, all such rectangles can be found in a simple greedy search that is at worst linear in the number of pixels in the image. A simple extension allows finding all unique rectangles for all thresholds in much less than the product of thresholds by pixels. We find all such rectangles, prioritize the smallest, and present a sorted list to the person using the authoring tool. For all prototypes we have created, this generates the correct example extraction within the top handful of results. We note, however, that tools for arbitrary pixel selection could be provided if a situation were found where these heuristics fail to identify the correct segmentation.

While a person is creating a prototype, our authoring tool applies that prototype to images currently loaded in the tool and displays any detected occurrences. This is helpful for seeing if a prototype is effective, and also eases specification of negative examples. Negative examples are used to correct over-generalization, where the prototype fit from a set of examples results in false detections. We have so far only encountered one situation requiring a negative example. The custom widgets in the YouTube movie player in Figure 1 are painted such that they share vertical edges (the pixel defining the right edge of one widget also defines the left edge of the adjacent widget). This led Prefab to over-generalize by interpreting each group of k adjacent occurrences as an occurrence. Marking any one of these as a negative example corrects the prototype (prompting Prefab to base the prototype on the combination of the single-pixel edge and the adjacent interior pixel color).

Model Parameters and Prototype Definitions

Each model exposes a set of parameters, possible values, and constraints on allowable combinations. For example, our eight-part model exposes ordinals for the width and height of each corner feature and for the depth of each edge region. It also constrains the corners and edges to jointly specify non-overlapping adjacent areas (i.e., a border).

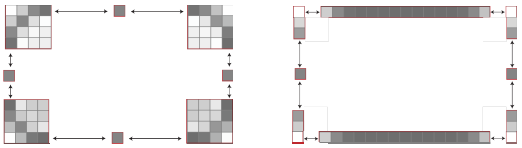


Figure 6: These are both valid prototypes for a single example Microsoft Windows Vista Steel button, but require 68 and 44 pixels to define. Prefab thus prefers the prototype shown in Figure 2, which requires only 40 pixels. Note Figure 2's prototype also generalizes better.

Importantly, it is *not* a model developer's responsibility to determine optimal parameters (i.e., to optimally divide pixels from an example into parts). A model specifies allowable combinations, and the branch-and-bound search algorithm then chooses the optimal combination. Models have two responsibilities: (1) given a complete parameter assignment, create a prototype and assign it a cost, and (2) given a partial parameter assignment, compute a lower bound on the cost of prototypes that can result. Prefab poses both as a matter of assigning example pixels to model parts.

Creating Prototypes and Assigning Cost

We have found that the most appropriate prototype is typically the one which requires the *fewest pixels* to explain the appearance of positive examples. The intuition behind this is similar to the minimum description length principle, a formalization of Occam's Razor in machine learning [16]. Prefab creates a prototype by minimizing the number of pixels needed to describe each part, then computes the cost of the prototype as the total number of pixels in its parts.

For example, the prototype in Figure 3 sets the width and height of each corner feature to three and edge depth to one. At these settings (selected by the branch-and-bound search), our eight-part model first assigns the nine pixels in each corner to their corresponding features. It then fits a region to the remaining pixels along each edge. The repeating region type can characterize each of the edges with a single pixel, and so is selected by the model. The prototype's total cost is therefore 40 pixels.

In contrast, Figure 6 shows two other prototypes that characterize the same example. The left has larger corner features (for which it must pay additional pixels), but gains no benefit from them (it pays the same number of pixels to explain its edges as the prototype from Figure 3). The right prototype has smaller corners, but the resulting top and bottom edges cannot be described by a single repeating pixel. Because these require 68 and 44 pixels, they are rejected in favor of Figure 3's prototype. Figure 3's prototype is also more robust (the right prototype above describes only buttons of exactly the same width shown).

Each model takes its own approach to assigning pixels to parts based on a set of assigned values for its parameters. For example, our five-part model of a slider includes a feature for each end of the trough, a feature for the thumb,

and two regions for the trough on either side of the thumb. It first assigns pixels to the end features (based on provided parameters), then scans the remaining pixels. At each step, it either attributes a column of pixels to the left region or skips a fixed number of pixels (a thumb width parameter) and attributes them to the right. Assignments are generally straightforward, as any challenging aspects of assignment should be exposed as a parameter and delegated to the search (e.g., the optimal size of edges or the slider thumb).

Multiple positive examples add the requirement a prototype describe all of them. Negative examples override the cost optimization: a prototype with a lower cost than the current best-known prototype is checked against negative examples before being promoted to the best (there is no reason to check the negative examples otherwise). If a prototype matches a negative example, it effectively has infinite cost.

Computing Lower Bounds on Partial Assignments

Prefab uses a branch-and-bound search, so models must compute lower bounds on partial assignments. For example, if the current best prototype for a particular eight-part model requires 40 pixels and the first several parameters of a partial assignment configure the top corners such that they consume 40 pixels (perhaps making them each 4x5), then no possible assignment of the other parameters can lead to an improvement over the current best prototype.

Models implement lower bound estimates according to how they assign pixels to parts. If enough parameters have been assigned to compute the actual pixels required by one or more parts, the actual cost of those parts can be used. An insight that allows re-use of previous computations is that the cost of pixels within a particular part is at least as much as any subarea of those pixels. If a model has previously evaluated the true cost of a part, that cost can be used as a lower bound in any assignment that grows the bounds of that part. Lower bound correctness is required for the branch-and-bound search, so models estimate a single pixel for parts for which they cannot provide a tighter bound.

VALIDATION IN APPLICATIONS

Prefab demonstrates a new approach to enhancements independent of the toolkits used to implement interfaces. Because this is a fundamentally new capability, there is no reasonable comparison to other approaches for obtaining the same effect. We instead validate, demonstrate, and provide insight into Prefab by implementing and discussing a set of applications. We select these three applications with the goal of illustrating a range of complexity.

All of our applications run on Microsoft Windows Vista and are implemented in Microsoft's C#, using redirection mechanisms as discussed with Figure 2. In order to demonstrate Prefab enhancements running on interfaces in Macintosh OS X, we connect via remote desktop software. Prefab thus continues to run on the Microsoft Windows Vista machine, adding its enhancements based entirely on the pixels delivered through the remote desktop connection.

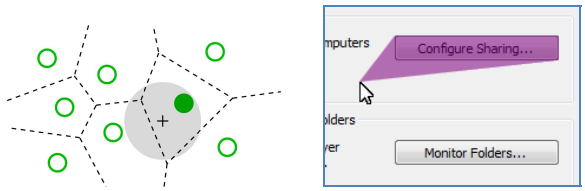


Figure 7: Grossman and Balakrishnan's Bubble Cursor expands to ensure that the nearest target is selected [3]. Prefab can enable such target-aware pointing techniques as general enhancements across a variety of applications independent of their underlying toolkit implementation.

Target Aware Pointing Techniques

As noted in our introduction, target-aware pointing techniques, such as Grossman and Balakrishnan's Bubble Cursor [3] and Worden *et al.*'s Sticky Icons [24], have long been known to have many potential advantages, but the diversity of existing applications and toolkits continues to limit their broader exploration and adoption. Identification of targets across existing applications and toolkits is sufficiently difficult that researchers have instead explored other approaches to approximating the desired behavior, such as target-agnostic pointing enhancements [23] and the use of click location histories as proxies for targets [10].

Figure 7 shows a Prefab Bubble Cursor implementation in a dialog from Windows Media Player on Microsoft Windows Vista. Figure 1 shows the same Bubble Cursor in a Mozilla Firefox dialog on Macintosh OS X (using the remote desktop method discussed earlier) and in a YouTube movie player on Microsoft Windows Vista. Our associated video shows a similar implementation of Sticky Icons. To the best of our knowledge, Prefab is the first approach to implementing target-aware pointing techniques as general enhancements across a variety of existing applications independent of their underlying toolkit implementation.

These target-aware pointing techniques are straightforward applications of Prefab. We extracted examples and built prototypes to identify Microsoft Windows Vista and Macintosh OS X buttons, checkboxes, textfields, and other standard widgets. We similarly created several prototypes of more specific widgets, such as the custom slider in the YouTube movie player (based on the same models as the standard widgets). These target-aware pointing techniques require only the location and size of current targets, so did not use it was not necessary to implement any transitions. The Sticky Icons enhancement simply adjusts the mouse gain, while the Bubble Cursor annotates the target window and manipulates how click events are mapped to the source.

Phosphor

Baudisch *et al.* developed Phosphor, showing the use of afterglows to explain user interface transitions [1]. One potential benefit Baudisch *et al.* identify is in remote collaboration, where the afterglow makes it easier to determine what changes a remote collaborator has made in an interface. Despite the promise of Phosphor, it has been difficult to evaluate its effectiveness in realistic use or to deploy it in collaborative meeting software.

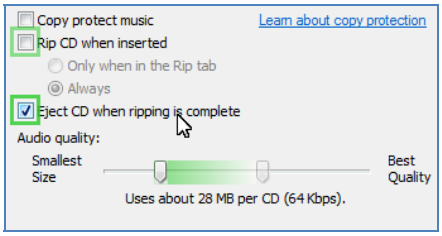


Figure 8: We have used Prefab to implement Baudisch *et al.*'s Phosphor [1] based entirely on an interface's pixels.

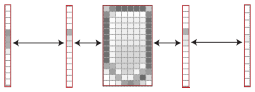


Figure 9: We use the image of the thumb from the slider's prototype to paint the ghosted thumb in the afterglow.

Figure 8 shows our Prefab implementation of Phosphor, with afterglow effects showing recent manipulation of two checkboxes and a slider in Windows Media Player on Microsoft Windows Vista. Figure 1 shows the same code creating Phosphor effects for Macintosh OS X widgets in Apple's iTunes (using the remote desktop method discussed earlier). Note the afterglow applied to the sliders in these figures includes a ghosted thumb, painted using the actual thumb for those sliders. Also note these afterglows require near instantaneous recognition of widget manipulation. Because Prefab is based entirely on pixels, Phosphor could be included in collaborative meeting software independent of the toolkits used to implement shared applications.

Building upon the same prototypes from our target-aware pointing demonstrations, our implementation of Phosphor uses Prefab transitions to observe widget state changes. These expire and are reset whenever a widget is observed in the same state as the previous frame. A timeout on the transition allows a widget a reasonable time interval to animate into a new state, and we use output redirection to superimpose our afterglow. The ghost of the slider's actual thumb is painted using the thumb feature from the Prefab prototype that detects the slider, as illustrated in Figure 9.

Side Views Parameter Spectrums

Our previous applications show local widget enhancement, but Prefab can also be used as part of larger enhancements. To demonstrate this, we implemented Terry and Mynatt's Side Views parameter spectrums [22]. The goal of parameter spectrums is to better support open-ended tasks by simultaneously previewing a range of options for values of multiple parameters. Terry and Mynatt showed parameter spectrums for image editing filters in the GNU Image Manipulation Program (the GIMP), chosen because its open-source implementation allowed necessary modifications. Despite the promise of this technique, the closed-source nature of Adobe's Photoshop prevents researchers and practitioners from studying or deploying parameter spectrums in this more widely-used package. This problem is typical of situations where researchers and practitioners find themselves unable to modify, personalize, or interoperate with existing closed-source interfaces.

Figure 1 shows a parameter spectrum for an image filter in Adobe Photoshop on Microsoft Windows Vista. Our associated video also shows parameter spectrums for the GIMP implemented without modifying its source. In both cases, we use Prefab to interpret the filter dialogs and use synthetic mouse events to manipulate sliders within the filter dialogs to obtain images to populate the spectrums. An eight-part prototype identifies the preview area in each dialog, from which we extract the image for a particular parameter configuration. An interesting difference between the two is how our implementation determines when to capture the preview image for a particular parameter configuration. Photoshop provides a small progress bar at the bottom-left of the filter dialog, and we use Prefab to observe the completion of this progress bar before grabbing the preview. The GIMP does not provide such a progress bar, and so we monitor the contents of the preview area and capture an image after observing a stable change. In both cases, we use a timeout to recover from the situation where Prefab does not observe the expected transition.

RELATED WORK

Hudson and Smith propose toolkit support for separating interface style from content, drawing an analogy to painting the same text with different fonts [7]. Hudson and Tanaka build upon this, developing methods for painting highly stylized widgets [8]. Hudson and Tanaka's methods include an eight-part border based in painting fixed corners and variable edges, analogous to the eight-part model discussed in this paper. Prefab turns these approaches to painting widgets on their heads, separating recognition of interface content (described by Prefab models) from style (described by Prefab prototypes). Prefab is simultaneously informed by the workings of user interface toolkits and independent of a toolkit's implementation: our eight-part model can characterize many widgets regardless of whether they were painted using Hudson and Tanaka's eight-part method.

Edwards *et al.* [2] and Olsen *et al.* [13] develop approaches to modifying and enhancing interfaces based on replacing an application's drawing object and intercepting application commands (e.g., draw_line, draw_string). They require minimal modification to applications (limiting relevance to closed-source interfaces), but more importantly require implementation particular to each toolkit. Our pixel-based approach requires no modification to existing applications and is independent of underlying toolkit implementation.

Hutchings and Stasko's mudibo uses graphical interface input and output redirection to present windows in multiple locations, allowing a person to choose to interact with the window in the desired location [11]. Tan *et al.*'s WinCuts allows subdivision of windows using a copy-paste approach to configuring redirection [21]. Stuerzlinger *et al.* present advanced customizations in their work on User Interface Facades, many based on toolkit-specific introspection [20]. These each use redirection mechanisms similar to those from Figure 2, but are either limited by a lack of meaningful interpretation of the source or rely upon toolkit

introspection for that interpretation. Our work is unique in combining input and output redirection with pixel-based interpretation, enabling interpretation-based modifications independent of interface implementation mechanisms.

The most relevant prior work thus examines pixel-based interpretation and manipulation of graphical user interfaces. Olsen *et al.*'s ScreenCrayons builds upon the universality of pixels, analyzing images of interfaces to associate ink annotations, but does not interpret those images [14]. Classic work by Zettlemoyer *et al.* examines pixel-based identification of widgets for IBOTS and VisMap [26, 27] in the context of supporting interface agents and programming by example [15, 18]. Zettlemoyer *et al.*'s methods require code-based descriptions of the *appearance* of widgets, and Zettlemoyer *et al.* report that 40% of VisMap code is specific to the particular Microsoft Windows widgets it recognized. Continued development of these methods in Segman also found their performance insufficient for interactive applications [19]. Prefab's performance is an obvious improvement, as is our use of example-based prototype creation and the fact that none of our model code is specific to any particular toolkit. Yeh *et al.* recently developed Sikuli, which supports image-based interface search and automation [25]. Sikuli uses computer vision methods (template matching and voting based on invariant local features), and Yeh *et al.* report these require 200msec to identify all occurrences of a *single* target in an image of a desktop. In contrast, recall that Prefab must identify many prototypes many times per second. Hurst *et al.* use several pixel-based techniques to improve the accessibility API's detection of target boundaries [9]. Their specific goals lead them to leverage information unavailable in the general interface interpretation problem (e.g., image differences available only after a person clicks on a target), and they do not generalize a notion of edges and corners to consider different arrangements. Thus none of this prior work leverages Prefab's core insight of separating the modeling of widget layout from the description of widget appearance.

DISCUSSION

Prefab represents a first step towards pixel-based reverse engineering of interfaces based on models of how those interfaces are painted by applications and user interface toolkits. This section discusses several important aspects of Prefab and identifies important directions for future work.

This paper intentionally presents the simplest description of using a prototype library to reverse engineer an image, focusing on information flow as Prefab locates features, generates hypotheses, and tests those hypotheses. There are many opportunities for principled optimizations. The most important is probably that most pixels do not change between successive frames. Prefab's entire process can be implemented with incremental evaluation, using lightweight dataflow to re-compute exactly the features and prototypes that could possibly have changed as a result of differences between successive frames [5]. Prefab's entire process also supports a multi-core approach, as feature detection can be applied to multiple pixels simultaneously, multiple models can simultaneously generate hypotheses from detected features, and those hypotheses can be tested in parallel. It is also possible to vastly reduce the number of pixels that must be tested for features in each image. If every feature contains at least two adjacent pixels, for example, only half the pixels in an image need to be tested. Given these and other opportunities for principled optimization, we do not foresee performance as problematic for most applications. Our current single-threaded implementation uses a simple ad-hoc frame difference optimization, and our associated video shows multiple applications recognizing prototypes in real interfaces of existing applications with frame-to-frame computations typically well under 50msec.

Not all interfaces are created entirely from standard widgets. Prefab can still be effective in such interfaces for two reasons. First, even non-standard interfaces are procedurally generated. Once a prototype of such an interface is created, it is unlikely to change. Second, non-standard interfaces cannot be generally successful unless they *look like an interface*. Prefab may ultimately prove to be a more reliable approach to non-standard interfaces because it is based on their appearance instead of toolkit introspection mechanisms that developers often neglect to implement in non-standard components.

We have shown that a number of interesting applications are enabled by Prefab's current implementation, but Prefab also currently has several important limitations. There is currently no structured approach to interpreting the content portions of prototype occurrences (e.g., Prefab can identify the three buttons in Figure 5, but not the fact that they are labeled "A", "B", and "C"). Prefab transitions allow monitoring of simple dynamics, but are defined at such a low level that they can be tedious to configure in large enhancements. Prefab would thus benefit from an approach to modeling state machine relations among prototypes corresponding to the different states of a widget as well as improved approaches to managing other common types of related transitions (e.g., animations). Prefab also currently does not model relationships among widgets, and so does not have an understanding of the behavior of group of radio buttons or the hidden content associated with a scroll pane or a tab panel. We ultimately intend for Prefab to support a wide variety of applications beyond those explored in this paper, such as accessibility enhancements, tutorials for off-the-shelf software, context-sensitive help mechanisms, and CSCW extensions to existing applications. Whether Prefab's current limitations are problematic depends upon the intended enhancement. In the case of accessibility enhancements, for example, Prefab can enable target-aware techniques with significant implications for people with motor impairments, but the lack of content interpretation means Prefab cannot be used to implement screen reading for people for visual impairments. Finally, we note it is also non-trivial to create sophisticated models. We have stated that we believe this effort can be justified if the resulting

model applies to a large and wide variety of prototypes, but improved authoring tools also warrant serious exploration. We see all of these limitations as rich areas for future work.

CONCLUSION

Prefab contributes pixel-based reverse engineering of interfaces that separates modeling of widget layout from recognition of widget appearance, Prefab enables a practical approach to adding advanced behaviors to new and existing interfaces independent of particular user interface toolkits. In addition to its direct implications for human-computer interaction research and practice, we hope Prefab can help to break some of the critical mass and chicken-and-egg problems currently limiting user interface tools research.

ACKNOWLEDGEMENTS

We thank Jeff Bigham, Scott Hudson, Brian Meyers, Steve Seitz and Greg Smith for discussions related to this work. This work was supported in part by the UW CSE Hacherl Graduate Fellowship and by a fellowship from the Seattle Chapter of the ARCS Foundation.

REFERENCES

[1] Baudisch, P., Tan, D.S., Collomb, M., Robbins, D., Hinckley, K., Agrawala, M., Zhao, S. and Ramos, G. (2006). Phosphor: Explaining Transitions in the User Interface using Afterglow Effects. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2006), 169-178.

[2] Edwards, W.K., Hudson, S.E., Marinacci, J., Rodenstein, R., Rodriguez, T. and Smith, I. (1997). Systematic Output Modification in a 2D User Interface Toolkit. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 1997), 151-158.

[3] Grossman, T. and Balakrishnan, R. (2005). The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 2005), 281-290.

[4] Hartmann, B., Wu, L., Collins, K. and Klemmer, S.R. (2007). Programming by a Sample: Rapidly Creating Web Applications with d.Mix. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2007), 241-250.

[5] Hudson, S.E. (1991). Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems* (TOPLAS), **13**(3). 315-341.

[6] Hudson, S.E., Mankoff, J. and Smith, I. (2005). Extensible Input Handling in the subArctic Toolkit. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 2005), 381-390.

[7] Hudson, S.E. and Smith, I. (1997). Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 1997), 159-168.

[8] Hudson, S.E. and Tanaka, K. (2000). Providing Visually Rich Resizable Images for User Interface Components. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2000), 227-235.

[9] Hurst, A., Hudson, S.E. and Mankoff, J. (2010). Automatically Identifying Targets Users Interact with During Real World Tasks. *Proc. of the International Conference on Intelligent User Interfaces* (IUI 2010), To Appear.

[10] Hurst, A., Mankoff, J., Dey, A.K. and Hudson, S.E. (2007). Dirty Desktops: Using a Patina of Magnetic Mouse Dust to Make Common Interactor Targets Easier to Select. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2007), 183-186.

[11] Hutchings, D.R. and Stasko, J. (2005). mudibo: Multiple Dialog Boxes for Multiple Monitors. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems* (CHI 2005), 1471-1474.

[12] Myers, B.A., Hudson, S.E. and Pausch, R. (2000). Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* (TOCHI), **7**(1). 3-28.

[13] Olsen, D.R., Hudson, S.E., Verratti, T., Heiner, J.M. and Phelps, M. (1999). Implementing Interface Attachments Based on Surface Representations. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 1999), 191-198.

[14] Olsen, D.R., Taufer, T. and Fails, J.A. (2004). ScreenCrayons: Annotating Anything. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2004), 165-174.

[15] Potter, R. (1993). *Triggers: Guiding Automaton with Pixel to Achieve Data Access*. A. Cypher, eds. MIT Press.

[16] Rissanen, J. (1978). Modeling by Shortest Data Description. *Automatica* **14**(5). 465-471.

[17] Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

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

[19] St. Amant, R., Riedl, M.O., Ritter, F.E. and Reifers, A. (2005). Image Processing in Cognitive Models with SegMan. *Proc. of the International Conference on Human-Computer Interaction* (HCII 2005).

[20] Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. (2006). User Interface Façades: Towards Fully Adaptable User Interfaces. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2006), 309-318.

[21] Tan, D.S., Meyers, B.R. and Czerwinski, M. (2004). WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems* (CHI 2004), 1525-1528.

[22] Terry, M. and Mynatt, E.D. (2002). Side Views: Persistent, On-Demand Previews for Open-Ended Tasks. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2002), 71-80.

[23] Wobbrock, J.O., Fogarty, J., Liu, S., Kimuro, S. and Harada, S. (2009). The Angle Mouse: Target-Agnostic Dynamic Gain Adjustment Based on Angular Deviation. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 2009), 1401-1410.

[24] Worden, A., Walker, N., Bharat, K. and Hudson, S.E. (1997). Making Computers Easier for Older Adults to Use: Area Cursors and Sticky Icons. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 1997), 266-271.

[25] Yeh, T., Change, T.-H. and Miller, R.C. (2009). Sikuli: Using GUI Screenshots for Search and Automation. *Proc. of the ACM Symposium on User Interface Software and Technology* (UIST 2009), 183-192.

[26] Zettlemoyer, L.S. and St. Amant, R. (1999). A Visual Medium for Programmatic Control of Interactive Applications. *Proc. of the ACM Conference on Human Factors in Computing Systems* (CHI 1999), 199-206.

[27] Zettlemoyer, L.S., St. Amant, R. and Dulberg, M.S. (1998). IBOTS: Agent Control Through the User Interface. *Proc. of the International Conference on Intelligent User Interfaces* (IUI 1998), 31-37.