# Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure

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

**ABSTRACT**
The rigidity and fragmentation of GUI toolkits are fundamentally limiting the progress and impact of interaction research. Pixel-based methods offer unique potential for addressing these challenges independent of the implementation of any particular interface or toolkit. This work builds upon Prefab, which enables the modification of existing interfaces. We present new methods for hierarchical models of complex widgets, real-time interpretation of interface content, and real-time interpretation of content and hierarchy throughout an entire interface. We validate our new methods through implementations of four applications: stencil-based tutorials, ephemeral adaptation, interface translation, and end-user interface customization. We demonstrate these enhancements in complex existing applications created from different user interface toolkits running on different operating systems.

**Author Keywords**
Prefab, pixel-based reverse engineering, content, hierarchy.

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

**General Terms**
Human Factors

## INTRODUCTION AND MOTIVATION

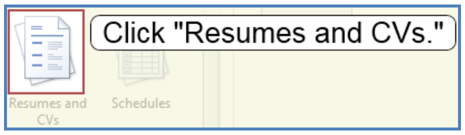
Nearly every modern graphical user interface (GUI) is implemented using some form of GUI toolkit. Toolkits provide libraries of widgets and associated frameworks that reduce the time, effort, and amount of code needed to implement an interface. Although these toolkits have enabled many successes of the past forty years of human-computer interaction research and practice [19], the current state of toolkits has become stifling [8, 12, 21].

Specifically, researchers and practitioners are limited by the *rigidity* and *fragmentation* of existing toolkits. Rigidity makes it difficult or impossible for an application developer to modify a toolkit's core behaviors. Similarly, application rigidity generally precludes modification and customization of existing interfaces (except in limited ways envisioned and supported by an application's original developer). Fragmentation results from the fact that people generally use many different applications built with a variety of toolkits. Each is implemented differently, so it is difficult to consistently add new functionality. Researchers are often limited to demonstrating new ideas in small testbeds, and practitioners often find it difficult to adopt and deploy ideas from the literature. These challenges limit the progress and impact of interaction research [8, 12, 21].

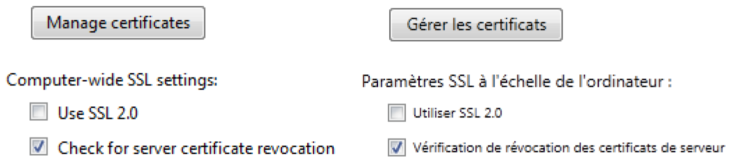
Because all GUIs ultimately consist of *pixels*, researchers have proposed methods for enhancing existing interfaces based only upon pixel-level interpretation. Pixel methods were initially proposed to support research in interface agents and programming by example [23, 25, 26, 32, 33]. More recent research examines broader opportunities for pixel-based methods: ScreenCrayons supports annotation of documents and visual information in any application [22], Sikuli applies computer vision to interface scripting and testing [3, 31], Hurst *et al.* use pixel-based methods to improve the accessibility API's target detection [15], and Prefab enables real-time modification of existing interfaces based on pixel-level interpretation [4].

The capabilities of these and other pixel-based systems are inherently defined and limited by a system's ability to meaningfully interpret raw interface pixels. This paper advances state-of-the-art systems by presenting the first pixel-based methods for real-time interpretation of interface content and hierarchy. Specifically, we build upon Prefab's pixel-based models of widget layout and appearance [4]. We first introduce the use of hierarchy to characterize complex widgets. We then introduce content regions and show how they enable efficient recovery of widget content. Finally, we show how these insights can be combined to recover a hierarchical interpretation of an entire interface. We validate our novel methods in a set of applications that demonstrate new capabilities enabled by interpretation of content and hierarchy, and we discuss future research opportunities suggested by this work.

Figure 1 illustrates several applications enabled by our new pixel-based methods. A pixel-based implementation of Kelleher and Pausch's *Stencils-based tutorials* [16] uses interface hierarchy to robustly reference specific widgets (i.e., differentiating among identical widgets by their position in the hierarchy). Our implementation of Findlater *et al.*'s *Ephemeral Adaptation* [5] is independent of interface implementation and leverages our models of content regions to create the necessary gradual onset animations. A



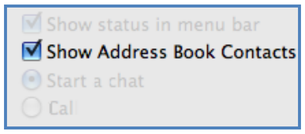
Kelleher and Pausch present the use of stencils to provide tutorials directly within the context of an interface [16]. This is a screenshot of our pixel-based implementation of a tutorial for downloading résumé templates in Microsoft Office 2010.


Manage certificates
Gérer les certificats
Computer-wide SSL settings:
Paramètres SSL à l'échelle de l'ordinateur :

☐ Use SSL 2.0
☑ Check for server certificate revocation

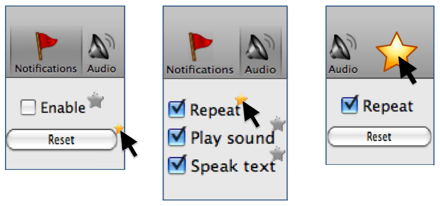
☐ Utiliser SSL 2.0
☑ Vérification de révocation des certificats de serveur

We present a GUI translator that works solely from an interface's pixels. Our translator identifies, interprets, and translates textual content while maintaining the same look and feel as the original application. Here is a portion of a Google Chrome Options dialog running in Microsoft Windows 7, translated from English into French.



☑ Show status in menu bar
☑ Show Address Book Contacts
○ Start a chat
○ Cal

Findlater *et al.*'s Ephemeral Adaptation improves targeting performance by reducing visual search while maintaining spatial consistency [5]. Likely targets appear as normal, but unlikely targets are initially missing and slowly fade in. This is a screenshot from our implementation of the technique in the context of a Skype settings dialog box running on Mac OS X.



☐ Enable

☑ Repeat
☑ Play sound
☑ Speak text

☑ Repeat

Our methods enable end-user customization of everyday interfaces. We implemented a technique that allows end-users to aggregate commonly used widgets of a tab control into a "favorites" tab. This is shown here in the context of the Skype settings dialog. Clicking on stars (left) adds the corresponding widgets to the "favorites" tab (right).

**Figure 1: We introduce new pixel-based methods to reverse engineer interface content and hierarchy in real time. These methods enable a new range of advanced behaviors that are agnostic of an application's underlying implementation. All of these enhancements are implemented in the Prefab system and are discussed in greater detail later in this paper and in our associated video.**

*translantion* enhancement highlights our real-time identification of interface content by extracting text, translating it, and re-painting it in the interface's original look and feel. Finally, adding *customization* to an existing interface illustrates support for managing occlusion and rendering new content using our pixel-based models.

The contributions of this work to pixel-based methods are:

- Methods for hierarchical models of complex widgets. These improve model implementation, example-based prototype creation, and runtime widget detection.
- Methods for modeling widget content regions. These enable efficient runtime recovery and interpretation of widget content.
- Methods for example-based parameterization of widget content regions. This is challenging because examples include content that should not be part of a prototype.
- Methods for recovering the content and hierarchy of an entire interface. This enables DOM-like interpretations of interfaces independent of their implementation.
- Validation of our methods in a set of novel pixel-based applications. These demonstrate our core methods and also illustrate opportunities to leverage our methods in addressing other challenges, such as robustly referencing widgets, re-rendering interfaces from pixel-based models, and managing interface occlusion.

## RELATED WORK

This paper's pixel-based interpretation of interface content and hierarchy is implemented within the Prefab system [4]. To provide a context for our current contributions, we first discuss several relevant aspects of prior research and then review key components of the Prefab system.

### Interpreting and Customizing Existing Interfaces

Our work is informed by GUI toolkit research [19]. For example, Hudson and Smith propose toolkit support for separating interface style from content, drawing an analogy to painting the same text with different fonts [13]. Hudson and Tanaka develop toolkit methods for painting stylized widgets, including an eight-part border defined by fixed corners and variable edges [14]. Our reverse engineering strategy turns such methods for painting interfaces on their heads: an eight-part model can be used to characterize many widget borders regardless of whether they were painted using Hudson and Tanaka's method. This paper is based on similar insights regarding tree-based interface layout and common approaches to painting variable content within widgets. We thus leverage knowledge of toolkit methods but remain independent of any particular toolkit.

Extensive research has examined interface customization. Most is limited to the web, where the Document Object Model (DOM) provides a structured representation of an interface. ChickenFoot [2] and CoScripter [18] are classic examples, and systems like Highlight extend these ideas to task-centric re-authoring for mobile devices [20]. Systems like Clip, Connect, Clone [6], d.mix [10], and Vegemite [17] demonstrate promising end-user mash-up methods.

For non-web interfaces, structure analogous to the DOM is provided the accessibility API. Sturzlinger *et al.'s* research on User Interface Facades uses the accessibility API to enable runtime interface modification [27]. Compared to pixel-based methods, accessibility APIs are advantageous because they can access information that is not visible, such as items contained in a closed drop-down menu. Direct access to an interface's underlying data also removes the possibility for recognition errors. On the other hand, accessibility APIs require additional implementation effort to correctly expose hooks into an interface's underlying state. Toolkits generally attempt to provide default implementations, but many widgets are missing from the accessibility API because of application failures to implement necessary hooks. Hurst *et al.* found roughly 25% of widgets are completely missing from the accessibility API's view on many common applications [15]. The severity of this problem is magnified by the fact that it can be corrected only by an application's original developer (or somebody else with the application source). Pixel methods have the critical advantage that they do not require cooperation from the original application. The application exposes its pixels as normal, and anybody can then use those pixels as a basis for additional functionality. For example, Hurst *et al.* show a hybrid technique that uses pixel-based methods to augment the accessibility API to improve target boundary detection [15]. Our current work focuses on pixel-based methods, presenting advances toward a DOM-like representation obtained entirely from pixels without cooperation of an underlying application. Deeper exploration of additional hybrid techniques is an important opportunity for future work.

The most relevant prior work is therefore that examining pixel-based interface interpretation. Classic work by Zettlemoyer *et al.* examined widget identification in IBOTS and VisMap [32, 33] for interface agents and programming by example [23, 25]. Their methods require code-based descriptions of individual widgets, and Zettlemoyer *et al.* report 40% of VisMap code is specific to particular Microsoft Windows widgets. Our methods use examples to describe widget appearance, making it possible to scale to address the fragmentation of current interfaces. Perhaps more importantly, continued development of Zettlemoyer *et al.*'s methods in Segman found their performance insufficient for real-time demands of interactive applications [26]. In an interactive context, Olsen *et al.*'s ScreenCrayons leverages the universality of pixels to associate ink annotations with images of interfaces, but does not interpret those images [22]. Tan *et al.*'s WinCuts allows subdivision of windows with a copy-paste metaphor, but does not attempt to interpret or modify content [29]. Yeh *et al.*'s Sikuli uses template matching and voting based on invariant local features to identify occurrences of a target in an image of an interface (requiring 200msec to find all occurrences of a single target) [3, 31]. None of these existing pixel-based methods are capable of real-time interpretation of content and hierarchy, and so none are capable of the demonstrations presented in this paper.

## Prefab

Prefab modifies the apparent behavior of existing interfaces using pixel-based interpretation with input and output redirection. Figure 2 illustrates a basic mechanism, where (1) a *source* window bitmap is captured, (2) the source image is interpreted, (3) the 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 to the source, which then (5) generates new output that is captured and used to update the target. Prefab is the first system to combine pixel-based methods with input and output redirection, allowing it to modify an interface independent of that interface's implementation.

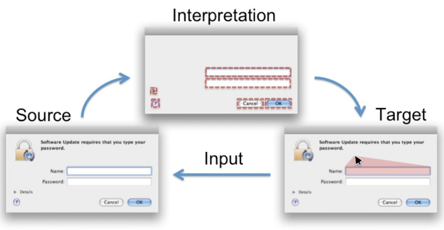


**Figure 2: Prefab combines pixel-based interpretation with input and output redirection, allowing it to modify an interface independent of that interface's implementation. This requires real-time pixel-based interpretation.**

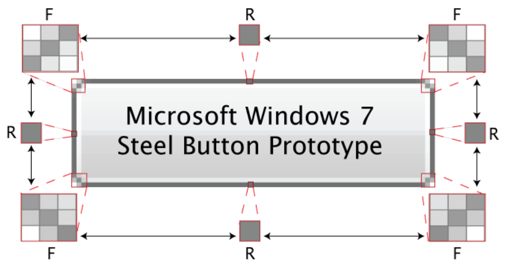


**Figure 3: This Prefab prototype for Microsoft Windows 7 Steel buttons is an example of an eight-part model. Four features define the corners, each edge is defined by a region, and constraints require the parts form a rectangle. This prototype recognizes all Microsoft Windows 7 Steel buttons, independent of their interior content.**

This requires interpreting images of an interface many times per second, which Prefab accomplishes using four major types of components: *models*, *prototypes*, *parts* (including *features* and *regions*), and *constraints*.

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

*Parts* can be either *features* or *regions*. A *feature* stores an exact patch of pixels (exact colors in an exact arrangement). A *region* stores a procedural description of an area (e.g., painting a gradient or a repeating pattern). Because the same *parts* can be arranged in multiple ways, they alone do not describe any particular widget or set of widgets.

*Prototypes* parameterize a *model* with concrete *parts*, thus characterizing both the arrangement and the appearance of those *parts*. A *prototype* therefore describes a particular widget or set of widgets (e.g., the Mozilla Firefox Home toolbar button, all Microsoft Windows 7 Steel buttons).

Figure 3 presents an example eight-part *prototype* that parameterizes an abstract *model* of an eight-part border with *parts* corresponding to a Microsoft Windows 7 Steel button. It recognizes all Microsoft Windows 7 Steel buttons, independent of their interior content. Prototypes that assign different pixels to its parts can recognize different styles of buttons or different widgets that paint a border. Prefab is implemented as a library of *prototypes* with methods for applying those *prototypes* to identify widgets. Specifically, Prefab locates all *features* in a pass over an image and then tests the *regions* and *constraints* of potential *prototypes*.

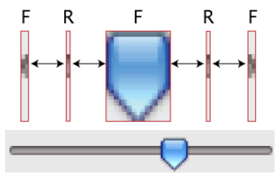
Creating a *prototype* by manually specifying the *parts* of a *model* is possible, but tedious and error-prone. Prefab helps by fitting *prototypes* from examples. We have found that the most appropriate *prototype* for a widget is typically the one which requires the *fewest pixels* to explain its appearance. The intuition behind this approach is similar to the minimum description length principle, a formalization of Occam's Razor in machine learning [24]. Prefab uses a branch-and-bound search to determine what assignment of parts to a model best explains an example. For example, the prototype in Figure 3 is learned from a single example of a Microsoft Windows 7 Steel button. This ability to quickly create new prototypes is important to Prefab's potential for scaling to address GUI toolkit fragmentation.

## INTERPRETING CONTENT AND HIERARCHY

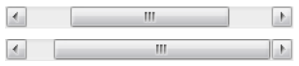
This section addresses key challenges facing Prefab and other pixel-based systems. We first address the complexity of widgets with multiple components by introducing *hierarchical models.* We then address the difficulty of modeling unpredictable content by leveraging knowledge of containment in *content regions*. Finally, we combine and extend these ideas to enable efficient interpretation of the *content and hierarchy of an entire interface*.

### Hierarchical Models

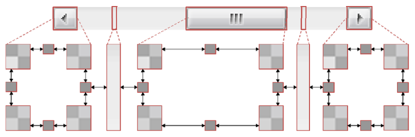
The eight-part model in Figure 3 was the most complex model explored in Prefab's initial development. Although Prefab's original methods can characterize many widgets, significant challenges arise when considering more complex models needed to represent widgets consisting of multiple components. Figure 4 illustrates this by comparing the modeling of a *slider* with that of a *scrollbar*.



4a: Prefab's original methods support a five-part model of sliders.

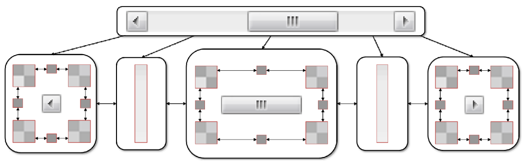


4b: Scrollbars vary their thumb size, so cannot be modeled with five parts.



4c: This model accounts for the presence of scroll buttons and the varying size of the scroll thumb, but is too complex for practical use.

4d: Other types of scrollbars arrange buttons differently, which would require still greater complexity in the above model.



4e: Hierarchical models simplify the implementation, example-based parameterization, and recognition of complex widgets.

**Figure 4: This paper introduces hierarchical models of widget layout, improving Prefab's support for complex widgets defined by hierarchies of simpler components.**

Figure 4a shows a five-part model of a slider, consisting of a feature for the thumb, features for the trough endpoints, and regions for the variable-length trough. This model effectively characterizes many sliders. Although a scrollbar might seem to have a similar layout, Figure 4b shows that scrollbars vary the size of their thumb to illustrate what portion of the scrollable area is currently within view. A single feature is therefore insufficient for characterizing the scrollbar thumb, so we replace the feature with eight parts describing a thumb of varying size. If we want our model to represent the buttons at either end of the scrollbar, we also need to replace the end features with eight parts. Figure 4c shows the resulting model, which characterizes the components of these scrollbars but has become too complex for practical use. It is difficult to correctly implement, and its many parameters create a high branching factor that makes it computationally expensive to fit prototypes from examples. The effort to implement and optimize complex models can be justified when they characterize a wide variety of widgets, but this model still does not characterize some common scrollbars. For example, Figure 4d shows a scrollbar from Mac OS X's Cocoa toolkit painted with both scroll buttons together.

The solution to these modeling challenges comes from the insight that complex widgets are typically defined by a hierarchy of simpler widgets. We introduce hierarchical models of widget layout, as illustrated with a scrollbar model in Figure 4e. This extends Prefab's original notion of delegating regions to procedural code (e.g., a gradient or a repeating pattern) by allowing delegation to another model. Portions of a hierarchy can be re-used in implementing multiple models. A model can account for portions of the hierarchy appearing in different arrangements or being optionally absent. The hierarchy can also be used when fitting prototypes from examples, with annotations of simpler components (e.g., the scrollbar thumb) constraining a search of the overall hierarchy. At runtime, hierarchical prototypes are identified by locating simpler components and then testing constraints and regions in the hierarchy.

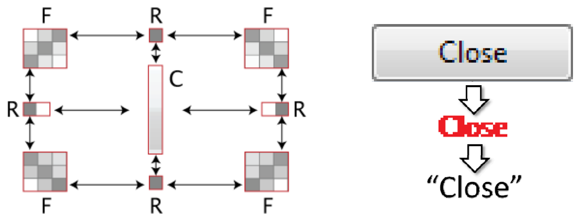


**Figure 5: This prototype for Microsoft Windows 7 Steel buttons is an example of a nine-part model. Our nine-part model is identical to the eight-part model in Figure 3 except for the addition of an interior content region. This prototype's content region has been parameterized as a single repeating column. At runtime, content within a button is obtained by differencing the repeating column against the pixels inside the button's content area.**

## Content Regions

Prefab's pixel-based methods are built on the insight that the pixels of a widget are procedurally defined. This is critical to Prefab's real-time performance, as it allows exact feature matching as a basis for prototype detection. But some interface content varies dramatically and cannot easily be identified through exact matching. For example, modern toolkits often employ sub-pixel rendering and anti-aliasing techniques in text rendering. This improves readability, but also modifies text's pixel-level appearance in unpredictable ways. The same characters can be rendered as many different combinations of pixels. Prefab's original methods therefore could not address the recovery of widget content (e.g., Figure 3's prototype can identify Microsoft Windows 7 Steel buttons, but cannot recover their label).

We address this challenge by building upon several insights. First, toolkits construct interfaces as trees. Second, content appears at the leaves of a tree (i.e., labels and icons do not contain other widgets). Every piece of content is therefore contained within a parent. Third, the parents of these leaf nodes paint simple backgrounds (often a single color, sometimes a simple gradient). This is critical to interface usability, as a person must be able to easily see the content painted over that background. Instead of directly modeling unpredictable content, we introduce *content regions* that model the much simpler background of a parent and efficiently identify content using runtime differencing.

Figure 5 illustrates a nine-part model of a border with an interior content region (i.e., it is identical to the eight-part model from Figure 3 except for the addition of the interior content region). The model's constraints require that the content region describe every pixel not accounted for by the corner features or the edge regions. In this case, a prototype of the Microsoft Windows 7 Steel Button parameterizes the content region with a single repeating column. At runtime, content is obtained by differencing the repeating column against pixels inside the button's content area. Figure 5 illustrates this differencing with red pixels in an example button. A character recognition algorithm is then applied to recover the text "Close". Because such character

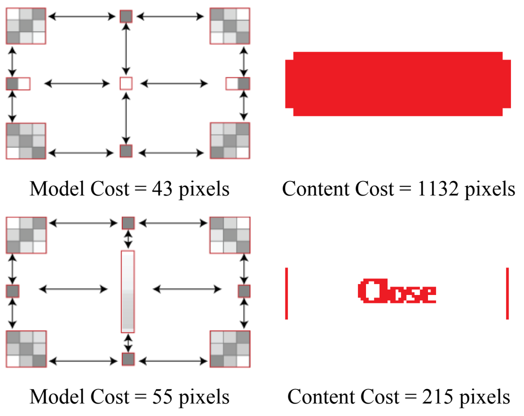


**Figure 6: These are both valid nine-part prototypes for a single example of a Microsoft Windows Vista Steel button. They incur total costs of 1175 and 270 pixels. Prefab thus prefers the nine-part prototype shown in Figure 5, which costs only 246 pixels and is also more general. Note that Figure 5's prototype also identifies the correct content.**

recognition is relatively expensive, it is important to note that our content region method identifies a small portion of an interface to which more expensive methods are applied. Its computation can also be cached, as there is no need to re-execute an interpretation of identical content pixels.

### *Parameterizing Content Regions by Example*

Recall that Prefab supports the use of *examples* to create prototypes. Parts are assigned by a search minimizing the number of pixels needed to describe those examples in a manner consistent with the model. Like other regions, content regions are modeled as procedural methods for pixel generation (e.g., painting a single color, repeating a pattern, painting a gradient). Prefab's original methods cannot be applied to content regions because each example contains unpredictable content. A simple part cannot characterize this content, and so the search fails to fit a good prototype that generalizes from the example.

We address this problem by defining the cost of a potential prototype as the sum of two components: *model cost* and *content cost*. As before, the model cost is the number of pixels used to define the parts of a prototype. The content cost is the number of pixels in an example that do not match the prototype specified by a content region. The intuition behind this approach is that minimizing the sum requires the search to both *describe* the background and *identify* the foreground. Because we lack a meaningful method for generating that unpredictable foreground, we pay full cost for the pixels it occupies. Note that this is a generalization of Prefab's original method, as content cost is always zero in models without content regions.

As an example, Figure 5's prototype sets the width and height of each corner feature to three, top and bottom edge depths to one, and left and right edge depths to two. The content region is a repeating column of pixels. With these settings (selected by the branch-and-bound search), the prototype has a model cost of 57 (9 for each corner, 6 for the edges, and 15 for the content region). The text results in a content cost of 189 (the red pixels shown on the right side of Figure 5), yielding a total cost of 246 pixels.

In contrast, Figure 6 shows two other prototypes the search might consider for the same example. The first has the same corners and edges but attempts to fit a single color to the background of the content region. This improves its model cost to 43, but the poor match results in a content cost of 1142 and a total cost of 1175 pixels. The second example has the correct content region with the corner and edge configuration from Figure 3 (which was fit to an eight-part model that does not consider content). Specifically, notice its left and right edges are 1 pixel wide. This results in a model cost of 55 (9 for each corner, 4 for the edges, and 15 for the content region), but the content cost is increased to 215 by two 13-pixel columns at either end of the content region. Its total cost is 270 pixels. These and other prototypes are rejected, with the search ultimately selecting the configuration from Figure 5 as the best fit.

Note that the content region in Figure 5 has actually resulted in a better characterization of the prototype's other parts. Without a content region, there is no reason for Prefab to determine that the left and right edges of this example are two pixels wide (and so it finds the prototype form Figure 3). The inclusion of a content region has in this case lead Prefab to produce a prototype that describes *every pixel* in the example. Our validating applications present implications of this more complete interpretation.

### Interpreting Content and Hierarchy in an Entire Interface

The intuition behind our methods for individual widgets can be extended to support pixel-based interpretation of content and hierarchy in an entire interface. Instead of considering content only in terms of text within a button, the necessary insight is that every widget is content relative to its parent. Our challenge is to recover the content and hierarchy of the entire interface while retaining Prefab's performance. We implement this in four steps, as illustrated in Figure 7.

We first apply Prefab's library of prototypes to locate widgets. This uses feature-based detection to identify a set of widget occurrences. We then organize the detected occurrences into a tree. The root is the image itself (typically a top-level window in Prefab's current input and output framework). The tree is constructed using constraints provided by each occurrence's model. These typically enforce spatial containment within a content region of the occurrence. The primary exception is for widgets that float above an interface (e.g., tooltips, popup menus, drop-down boxes). Tagging prototypes of these widgets allows our tree construction algorithm to link them directly to the root.

This organizes occurrences that were detected using our feature-based methods, but we still need to apply our differencing method to locate unpredictable content from content regions. It is important this differencing respect the

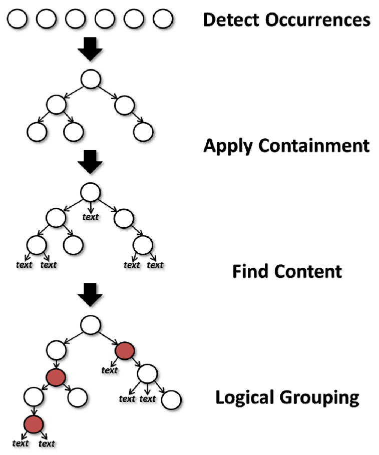


**Figure 7: We interpret interface content and hierarchy by detecting widget occurrences, applying containment to construct a tree, finding content within widgets, and logically grouping nodes within the tree.**

existing hierarchy (i.e., nested widgets must consider the proper background and must not generate spurious content in areas owned by children). Our current implementation uses a post-order traversal. We generate a composite background image when traversing down the tree, then test and mark pixels when traversing back up the tree. Widgets only test pixels within their content regions that were not marked by children. Identified content is interpreted and added as a child of the widget that detected it.

The resulting tree includes all detected widgets arranged by their containment. Additional organization can be added by considering that siblings in this *visual* tree may suggest an additional component in a *logical* tree. For example, several pieces of text might be grouped together and then related to an adjacent checkbox. Prior research has developed methods for semantic grouping of widgets [7]. Given our focus on pixel-based detection of the visual tree, we perform logical grouping using a set of heuristics.

## VALIDATION THROUGH APPLICATIONS

This paper presents new methods for real-time pixel-based interpretation of widget content and hierarchy. Because this is a new capability, there is no reasonable comparison to other approaches for obtaining the same effect. We instead validate and provide insight into our work through a set of demonstrations. We select these with the goal of illustrating a range of complexity in applying our methods.

All of our applications are implemented in Microsoft's C# running on Microsoft Windows 7 and using redirection provided by Prefab. We use remote desktop software to demonstrate enhancements running on Mac OS X interfaces. Prefab thus continues to run on the Microsoft Windows 7 machine, adding its enhancements based entirely on the pixels delivered through the remote desktop connection. We apply enhancements to a variety of well-known applications to highlight that our methods are independent of the underlying implementation.

## Stencils-Based Tutorials

Kelleher and Pausch's *Stencils-based tutorials* provide help directly within applications using translucent stencils with holes to direct a person's attention to the correct interface component [16]. Such an enhancement is difficult to broadly deploy because of the rigidity and fragmentation of existing applications and toolkits. It is beyond the capabilities of previous pixel-based systems because authoring such a tutorial requires support for referencing *specific* interface elements. For example, there may be several buttons of identical appearance within an interface, but only one of them is the appropriate next action.

Figure 1 and our associated video show our Prefab implementation of Stencils-based tutorials. The tutorial instructs a person on how to download résumé design templates in Microsoft Word 2010. Our video highlights the real-time responsiveness enabled by our new methods.

Stencils-based tutorials are a straightforward application of widget hierarchy. Knowledge of the full hierarchy allows us to reference widgets using simple path descriptors on the tree. We implemented this demonstration by building prototypes to identify the majority of widgets in Microsoft Office 2010. For example, we used nine-part models to characterize many of the containers and buttons. We also used one-part models to identify less structured content (e.g., the icons used to represent different types of templates). These one-part models are typically easy to construct (i.e., a model of the background of their parent makes it trivial to segment the one-part example). We converted Prefab's hierarchical interpretations into an XML format, allowing the use of XPath descriptors to reference widgets within the hierarchy. Tutorials are thus authored as a list of XPath descriptors paired with textual instructions for each step. Additional capabilities could be developed, and we have not yet explored the best approach to an authoring tool, but this demonstration highlights our use of the pixel-based hierarchy to reference specific widgets.

## Ephemeral Adaptation

Findlater *et al.* developed Ephemeral Adaptation, an adaptive technique that improves performance by reducing visual search time and maintaining spatial consistency [5]. Ephemeral Adaptation helps draw visual attention to likely targets in an interface. Specifically, likely targets appear as normal within an interface, but unlikely targets are initially missing and then slowly fade in. Despite the promise of this technique, it has been difficult to evaluate in realistic use or to widely deploy in everyday software. A pixel-based implementation is beyond prior systems for two reasons. As before, it requires the ability to reference specific widgets (e.g., to monitor how frequently they are clicked in order to model which are likely targets). In addition, this application requires the ability to remove unlikely targets from the interface and then render their gradual onset.

Figure 8 and our associated video show our implementation of Ephemeral Adaptation using Prefab within a Skype settings dialog box running in Mac OS X. Upon moving between tabs, likely targets in each tab are initially visible. Unlikely targets then fade in over time. This enhancement uses a nine-part prototype of the tab pane and various prototypes for each of the interior widgets. We use our XPath descriptors to tally the frequency of interaction with each widget and use a simple model of likely targets (the most commonly-used widgets in each tab).

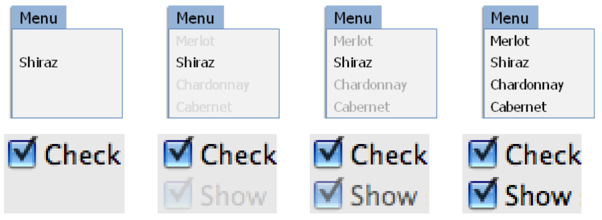


**Figure 8: Findlater *et al.*'s Ephemeral Adaptation technique uses the gradual onset of unlikely targets to facilitate easier targetting of likely targets [5]. This image shows Findlater *et al.*'s original implementation of a menu testbed together with our pixel-based implementation within a Skype dialog running on Mac OS X.**

We render the gradual onset animation using the content region from the tab's nine-part model. Specifically, we render tab background (i.e., the pixel-level appearance of its content region) as an overlay at the location of each unlikely widget. We then gradually fade this overlay from opaque to transparent. This creates the illusion that the widget is gradually fading into view. Note this technique requires identifying all of the content throughout the interface in order to appropriately animate its onset, a capability not supported by prior pixel-based methods.

## Language Translation

In addition to pixel-based *identification* of interface content, our methods can help enable real-time *interpretation* of interface content. To demonstrate this, we implemented a pixel-based enhancement that automatically translates the language of an interface and then presents the translated content in the same look and feel as the original interface. Because of the rigidity and fragmentation of current tools, interfaces usually must be translated by their original developer (or somebody else with the application source). Our methods allow anybody to translate an interface and could thus form a basis for community-driven translation (similar to advances in social accessibility [28]). To the best of our knowledge, ours is the first method for real-time translation of interfaces independent of their underlying implementations. Although translation is not the same as complete localization, it is an important step.

Figure 1 and our associated video show our translation enhancement applied to a Google Chrome Options dialog running on Microsoft Windows 7. The left image illustrates a portion of the original dialog in English. The right image shows that same portion of the dialog with the text translated into French. Our associated video also includes a Spanish translation of the same dialog. We implemented

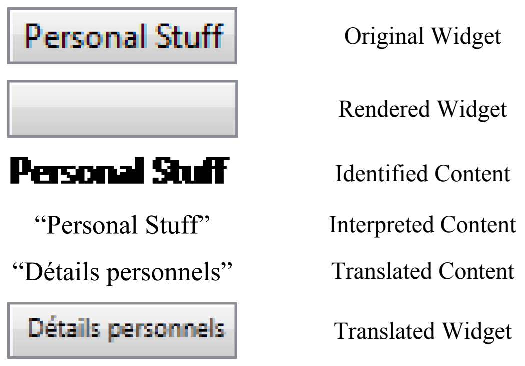


**Figure 9: We use our knowledge of interface content and hierarchy to implement a translation enhancement that preserves the look and feel of the original application. A translated widget is rendered by compositing the translated text with its prototype's content region.**

this by interpreting textual content identified by our methods, translating that text, and then rendering the new text in the original interface. Our associated video shows this enhancement running in real-time. This requires identification and interpretation of content occur quickly enough to handle the appearance and movement of content within a scroll pane.

There are several potential approaches to interpreting screen-rendered text [30]. Our current implementation uses an ad hoc template matching method, leaving integration of more advanced methods as an opportunity for future work. Importantly, our methods separate the identification of text from interpretation of that text. The interpretation of a region of pixels can thus be cached to eliminate potentially expensive re-interpretation of those same pixels (e.g., using a hash of the pixels). In our video, each piece of text is interpreted only the first time it appears. We then translate it using a machine translation service. As the text moves within the scroll pane, our content detection recovers the same pixels and retrieves the text from cache.

To maintain the application look and feel, we paint the translation into the original interface. This is implemented by using each widget's content region to render an overlay masking its content (i.e., its English text). The translated text is then rendered within the bounds of the original content region. For example, Figure 9 shows a button before its translated text is rendered. Because English text is typically shorter than translated text, we adjust the font size of the translated text to fit in the available region. Our next demonstration explores a more sophisticated modification of the interface to accommodate new content.

### Interface Customization

Our pixel-based interpretation of interface hierarchy also provides a framework for modeling some common forms of *occlusion* in interfaces. Occlusion is at the very core of the desktop metaphor, as it allows interfaces to limit the complexity of presented interfaces via the illusion that additional portions of the interface continue to exist even

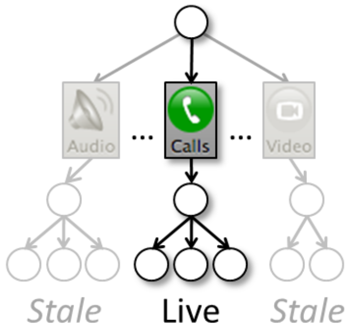


**Figure 10: We use our knowledge of hierarchy to manage widget occlusion in a tab pane. We store the most recently observed image of each widget, annotating occluded widgets to indicate those images are currently stale.**

when they are not visible. For example, tab controls use occlusion to limit attention to related subsets of complex interfaces. Existing pixel-based methods are strictly limited to interpreting visible portions of an interface. The need to observe an interface is inherent to pixel-based methods, but we can use our knowledge of interface hierarchy to help manage common forms of occlusion.

Figure 1 and our associated video present a demonstration of this in the context of interface customization. Instead of automatically adapting an interface according to widgets that are likely to be used, this example allows people to manually flag widgets as "favorites" for quick access. Figure 1 shows this applied to the same Skype dialog box from our Ephemeral Adaptation example. A small star is added to each widget in the interface. Clicking this star adds the widget to a "favorites" tab we added to the interface. Viewing that tab presents all starred widgets and allows interaction with each of them. As with all of our demonstrations, this is implemented using pixel-based interpretation with input and output redirection.

The management of occlusion is inherent to this example. We enhance the hierarchy to store the most recently observed version of each tab (using our XPath descriptors to reference each tab and its contents). We annotate these nodes in the hierarchy as *stale* to capture the fact they are occluded. Figure 10 depicts a simplified snapshot of the interpreted hierarchy with occluded nodes. If a "favorite" widget is currently occluded in the source window, it is painted using its stale version from the hierarchy. When a person moves to interact with a widget, synthetic input events are generated to bring that tab of the source window into view (i.e., to ensure that portion of the hierarchy is responsive to interaction via standard redirection mechanisms). Synthetic events could also be generated to regularly poll stale portions of the hierarchy, but this was not needed in our demonstration.

This example also demonstrates the use of our pixel-based methods to add new elements to the interface. Nine-part models of the window, the tab button container, and the tab pane are used to create a larger version of the window, insert the new “favorites” tab button, and to render the blank tab area into which “favorite” widgets are added. Figure 1 and our associated video show the added tab button and the extended window. We view this as an initial peek into opportunities to fundamentally transform the rigidity and fragmentation of existing interfaces.

## DISCUSSION AND LIMITATIONS

This paper advances state-of-the-art pixel-based systems by presenting the first methods for real-time interpretation of interface content and hierarchy. We now briefly discuss some important aspects of our pixel-based interpretation and identify some opportunities for future work.

For the sake of clarity, this paper presents the simplest description of our pixel-based methods for interpreting interface content and hierarchy. A variety of optimizations could improve performance. For example, the entire interpretation process can be implemented using lightweight incremental evaluation to compute exactly the sub-tree of the hierarchy that could possibly have changed between successive frames [11]. Many stages in the process can allow a multi-core approach (e.g., siblings can detect content simultaneously). Given these and other potential optimizations, we generally do not expect performance to be problematic in most applications. Our implementation currently computes frame differences to efficiently detect features and uses parallelization when interpreting content. We currently re-compute the entire hierarchy whenever Prefab identifies new features. Our associated video shows multiple demonstrations that interpret content and hierarchy in real interfaces of existing applications with computations between frames typically under 100msec. We believe this is sufficient for the applications we explore.

The preparation of our demonstrations highlighted another advantage of our approach to interpreting interface content and hierarchy. If our methods are applied to an interface that contains widgets that are not already in Prefab’s prototype library, the parents of those currently unknown widgets identify their pixels as content. We used this fact to quickly extract the examples used to create prototypes for our demonstrations, and we believe it could provide a basis for an improved prototype authoring tool.

Our translation demonstration currently uses an ad hoc approach to text interpretation (exact matching against a library of labeled character snippets). Off-the-shelf OCR technologies are generally ineffective because of the extreme low resolution of typical interface text. We previously noted the availability of recognition methods for screen-rendered text [30], but these are not optimized for Prefab’s scenario. A deeper investigation of robust text interpretation methods is an opportunity for future work.

We have noted the existence of prior work examining logical grouping of interface elements [7]. It is unclear whether these methods are compatible with the real-time requirements of pixel-based interpretation. The tree-based organization of interfaces and our ability to interpret visual containment provides an important advantage: we can likely consider only logical groupings of siblings. Based on the hierarchies we have encountered in our work, there are typically a small number of siblings in any given node (i.e., most interface elements are intentionally designed to contain a small number of content items). Our current implementation uses simple heuristics to perform logical grouping. For example, checkboxes are matched to their corresponding content using a threshold on the proximity of the nearest text. Future work can explore more advanced approaches to creating logical groupings by matching elements of an interface. Errors in an automated process could be also corrected by storing annotations that record the need for a specialized grouping (e.g., using our XPath descriptors to override the default behavior).

This paper focuses on core methods for interpreting content and hierarchy together with demonstrations of their value in example applications. There is a significant opportunity for future work that more thoroughly characterizes these and other pixel-based methods. Such work might examine the variety of widgets encountered in applications in the field, how well pixel-based methods can characterize those widgets, how many types of models and parts are necessary, and which of those models and parts are most broadly effective. Our pixel-based methods are the first to rival the accessibility API in terms of completeness, so comparisons between our methods and the accessibility API may be appropriate. Such a comparison should preferably go beyond simple frequency of failure to also probe the nature of failure (e.g., its impact in applications, the difficulty of correcting a failure). As in Hurst *et al.* [15], there are likely significant opportunities for hybrid approaches that combine the strengths of the accessibility API with the strengths of pixel-based methods. The contributions of this paper are a necessary step toward future characterizations of pixel-based methods, and our current validations are appropriate for this ongoing work.

Our interface customization demonstration dynamically re-renders a dialog box at a different size to create room for the “favorites” tab. This is possible because the nine-part prototype that detects the dialog box describes all of the pixels needed to generate it. To the best of our knowledge, we are the first to demonstrate pixel-based methods to create new widgets matching an existing interface. But not all Prefab prototypes necessarily have this property. The ability to seamlessly add new widgets to the interfaces of existing applications would dramatically extend pixel-based methods, and further examination of pixel-based methods is an additional opportunity for future work.

Our customization demonstration illustrates one approach to managing occlusion (using the hierarchy to maintain a memory of occluded components). Tab controls are perhaps the simplest case and this method may not immediately generalize to other forms of occlusion. For example, popup menus create less predictable occlusions that can span multiple nodes in a hierarchy (because they float above the hierarchy). Occlusion within a scrollpane also presents different challenges. We have shown that interface content and hierarchy provide a useful framework for reasoning about occlusion, and future work could examine more advanced methods building upon these initial insights.

## CONCLUSION

This paper advances pixel-based systems by contributing new methods for hierarchical models of complex widgets, new methods for real-time interpretation of interface content, and new methods for real-time interpretation of the content and hierarchy of an entire interface. We validated our pixel-based methods in implementations of four applications: Stencils-based tutorials, Ephemeral Adaptation, interface translation, and the addition of customization support to an existing interface. Working only from pixels, we demonstrated these enhancements in complex existing applications created in different user interface toolkits running on different operating systems.

## ACKNOWLEDGEMENTS

We thank Dan Grossman, Rick LeFaivre, Scott Saponas, and Dan Weld for discussions related to this work. This work was supported in part by a gift from Intel 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. Phosphor: Explaining Transitions in the User Interface using Afterglow Effects. *UIST 2006*. 169-178.

[2] Bolin, M., Webber, M., Rha, P., Wilson, T. and Miller, R.C. Automation and Customization of Rendered Web Pages. *UIST 2005*. 163-172.

[3] Chang, T.-H., Yeh, T. and Miller, R.C. GUI Testing User Computer Vision. *CHI 2010*. 1535-1544.

[4] Dixon, M. and Fogarty, J. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *CHI 2010*. 1525-1534.

[5] Findlater, L., Moffatt, K., McGrenere, J. and Dawson, J. Ephemeral Adaptation: The Use of Gradual Onset to Improve Menu Selection Performance. *CHI 2009*. 1655-1664.

[6] Fujima, J., Lunzer, A., Hornbæk, K. and Tanaka, Y. Clip, Connect, Clone: Combining Applications Elements to Build Custom Interfaces for Information Access. *UIST 2004*. 175-184.

[7] Gaeremynck, Y., Bergman, L.D. and Lau, T.A. MORE for Less: Model Recovery from Visual Interfaces for Multi-Device Application Design. *IUI 2003*. 69-76.

[8] Greenberg, S. and Buxton, B. Usability Evaluation Considered Harmful (Some of the Time). *CHI 2008*. 111-120.

[9] Grossman, T. and Balakrishnan, R. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *CHI 2005*. 281-290.

[10] 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.

[11] Hudson, S.E. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *TOPLAS*, **13**(3). 315-341.

[12] Hudson, S.E., Mankoff, J. and Smith, I. Extensible Input Handling in the subArctic Toolkit. *CHI 2005*. 381-390.

[13] Hudson, S.E. and Smith, I. Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit. *UIST 1997*. 159-168.

[14] Hudson, S.E. and Tanaka, K. Providing Visually Rich Resizable Images for User Interface Components. *UIST 2000*. 227-235.

[15] Hurst, A., Hudson, S.E. and Mankoff, J. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI 2010*. 11-20.

[16] Kelleher, C. and Pausch, R. Stencils-Based Tutorials: Design and Evaluation. *CHI 2005*. 541-550.

[17] Lin, J., Wong, J., Nichols, J., Cypher, A. and Lau, T.A. End-User Programming of Mashups with Vegemite. *IUI 2009*. 97-106.

[18] Little, G., Lau, T.A., Cypher, A., Lin, J., Haber, E.M. and Kandogan, E. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *CHI 2007*. 943-946.

[19] Myers, B.A., Hudson, S.E. and Pausch, R. Past, Present, and Future of User Interface Software Tools. *TOCHI*, **7**(1). 3-28.

[20] Nichols, J. and Lau, T.A. Mobilizing by Demonstration: Using Traces to Re-Author Existing Web Sites. *IUI 2008*. 149-158.

[21] Olsen, D.R. Evaluating User Interface Systems Research. *UIST 2007*. 251-258.

[22] Olsen, D.R., Taufer, T. and Fails, J.A. ScreenCrayons: Annotating Anything. *UIST 2004*. 165-174.

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

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

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

[26] St. Amant, R., Riedl, M.O., Ritter, F.E. and Reifers, A. Image Processing in Cognitive Models with SegMan. *HCII 2005*.

[27] Stuerzlinger, W., Chapuis, O., Phillips, D. and Roussel, N. User Interface Façades: Towards Fully Adaptable User Interfaces. *UIST 2006*. 309-318.

[28] Takagi, H., Kawanaka, S., Kobayashi, M., Itoh, T. and Asakawa, C. Social Accessibility: Achieving Accessibility through Collaborative Metadata Authoring. *ASSETS 2008*. 193-200.

[29] Tan, D.S., Meyers, B.R. and Czerwinski, M. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *CHI 2004*. 1525-1528.

[30] Wachenfeld, S., Klein, H.-U. and Jiang, X. Recognition of Screen-Rendered Text. *ICPR 2006*. 1086-1089.

[31] Yeh, T., Chang, T.-H. and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. *UIST 2009*. 183-192.

[32] Zettlemoyer, L.S. and St. Amant, R. A Visual Medium for Programmatic Control of Interactive Applications. *CHI 1999*. 199-206.

[33] Zettlemoyer, L.S., St. Amant, R. and Dulberg, M.S. IBOTS: Agent Control Through the User Interface. *IUI 1998*. 31-37.