Creating Contextual Help for GUIs Using Screenshots

Tom Yeh, Tsung-Hsiang Chang[§], Bo Xie[†], Greg Walsh[†], Ivan Watkins[†], Krist Wongsuphasawat, Man Huang[†], Larry S. Davis, and Ben Bederson

Department of Computer Science College of Information Studies[†]
University of Maryland, College Park
College Park, MD
{tomyeh,boxie,gwalsh,iwatkins,kristw,manhuang,lsd,bederson}@umd.edu

MIT CSAIL[§]
Cambridge, MA
vgod@mit.edu

ABSTRACT

Contextual help is effective for learning how to use GUIs by showing instructions and highlights on the actual interface rather than in a separate viewer. However, end-users and third-party tech support typically cannot create contextual help to assist other users because it requires programming skill and source code access. We present a creation tool for contextual help that allows users to apply common computer skills—taking screenshots and writing simple scripts. We perform pixel analysis on screenshots to make this tool applicable to a wide range of applications and platforms without source code access. We evaluated the tool's usability with three groups of participants: developers, instructors, and tech support. We further validated the applicability of our tool with 60 real tasks supported by the tech support of a university campus.

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

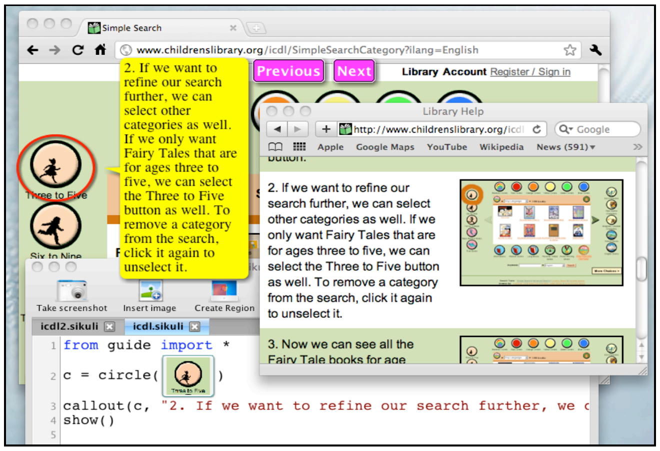
General terms: Human Factors; Design

Keywords: Help; Contextual Help; Pixel Analysis

INTRODUCTION

Contextual help has been shown effective for learning graphical user interfaces [3,4,13,16,17,18,25]. Unlike traditional help based on screenshots or screencasts, contextual help allows users to receive help in the actual interface they are interacting with, rather than in another help interface such as in a web browser or a video player. For example, with contextual help, users can see a dropdown box highlighted in the live interface and an instruction such as "select the WEP key" displayed next to it. Users can immediately practice the step as instructed. In contrast, with traditional help, users may see a captured screenshot of that dropdown box on a webpage. In order to practice the step, users need to switch to the live interface and try to locate the dropdown box that matches the screenshot. Also, they need to remember the instruction they read on the webpage regarding which option to select. This process illustrates

UIST'11, October 16–19, 2011, Santa Barbara, CA, USA.



2. If we want to refine our search further, we can select other categories as well. If we only want Fairy Tales that are for ages three to five, we can select the Three to Five button as well. To remove a category from the search, click it again to unselect it.

2. If we want to refine our search further, we can select other categories as well. If we only want Fairy Tales that are for ages three to five, we can select the Three to Five button as well. To remove a category from the search, click it again to unselect it.

3. Now we can see all the Fairy Tale books for age

```
from guide import *

c = circle(  )

callout(c, "2. If we want to refine our search further, we c
show()
```

Figure 1. Creating contextual help for a GUI. The window in the far back is the interface of the International Children's Digital Library. A separate window in the front displays existing help content for this interface. Our tool enables help designers to create help content that can be presented contextually in the actual interface (i.e., yellow callout and red circle) by writing a simple script and taking screenshots, as shown in the editor below. (www.childrenslibrary.org)

traditional help's two usability problems—split attention and delayed practice [21] that could be avoided simply by presenting help contextually.

However, for help designers, it is difficult to create GUI help for other users that can be presented contextually. For instance, to add a contextual highlight to a button, one may need to identify the source code responsible for that button and modify the code to draw highlight on the button [16,17]. Yet, many GUIs are proprietary. Third-party help designers are prevented from making contextual help to support these GUIs. Alternatively, help designers can use the accessibility API to determine that button's screen location and write an external program to draw highlight directly at that location on the screen [3,4]. Yet, for many GUIs, such API is either unavailable or unsupported [15]. Moreover, even if both the source code and the accessibility API are available, using them still requires high-level programming skills, a requirement that alienates many users from creating contextual help to assist other users.

In this work, we aimed to democratize contextual help. Specifically, we developed a tool to empower help designers in the general public (e.g., advanced users, computer instructors, third-party tech support) to create contextual help. Rather than requiring advanced programming skills, our tool allows a help designer to apply basic computer skills including taking screenshots and writing simple markup similar to HTML or Wiki. At design time, a help designer can select an interface component by capturing the component's screenshot (e.g., 🏠) and then specify the type of contextual help to add to that component using a simple visual markup (e.g., highlight(🏠)). When the help is presented to a user, the pixels on the user's screen are visually searched to locate the component that matches the screenshot. Once the component is found, contextual help can be drawn near the component. This visual search is performed using the Sikuli library [32], an open-source library for search and automation based on pixel analysis. The generalizability of screen pixels across all applications and platforms, a property already demonstrated by various applications [1,2,7,15,19,20,24,33], allows our method to be applicable to any GUI regardless whether the source code or accessibility APIs are available.

RELATED WORK

Contextual Help

Most GUI help is presented to users outside the GUI in a separate context such as in a web browser for seeing annotated screenshots or in a video player for watching screencasts. This out-of-context poses major usability problems such as split-attention and the lack of support of practice [21]. Contextual help solves these two problems [13] and allows hands-on practice and exploration at the same time, which has been shown to be key ingredients of successful learning of an interface [30].

The most prevalent form of contextual help is the tooltip [9]. Several efforts aimed to improve upon tooltips, such as Side Views that allow users to contextually preview the effect of a command [29], ToolClips that present to users short video clips and other rich help content next to toolbar buttons contextually [13]. In HCI research, contextual help has been applied to initial guidance to new features in the context of a map interface [16], to stencil-based tutorials for young school children in the context of Alice [17], to programming tasks in the context of an Eclipse editor [3], and to practical tasks such as booking flights in the context of a web browser [4]. In the commercial world, contextual help can also be spotted in interfaces that evolve quickly. For example, Google Docs periodically introduces new features and greets returning users with text bubbles to draw their attention to these features. However, in each instance cited above, content creation requires either advanced programming skill or privileged access to the source code, requirements that rule out a large population of help designers such as expert users, computer instructors, and third-party tech support.

Creating GUI Help

The majority of GUI help online today is created manually. Help designers use software such as Camtasia to make screencasts or MWSnap to take screenshots and annotate them with highlights and instructions. Guidelines have been provided for creating effective screencasts [23]. But meeting these guidelines requires significant manual effort. To reduce human effort, automatic methods have been proposed that aim to create tutorials based on knowledge of a UI derived from a UI model [27], UI specification [22] or a UI event log [5]. However, automatic methods have not been reliable enough to be used in practice.

A more practical approach to help creation that seeks to balance convenience and reliability has been the semi-automatic approach based on the notion of Programming by Demonstration (PBD) [3, 11,14]. To create help for a task, help designers simply perform that task. The interaction involved is captured and a descriptive summary of this interaction is automatically generated. Then, designers can manually review this summary and make necessary changes before publishing it as help. Taking this approach, Graphstrack [14] records designers' click actions and captures regional screenshots around each click location to generate minimalist help showing only images of relevant interface targets rather than the entire interface. The tool developed by Grabler et al. seeks to create succinct step-by-step help for a photo manipulation task while an expert demonstrates the task [11]. However, content generated by these two creation tools is intended to be viewed in a traditional manner in a context separate from the actual interface. DocWizard is most closely related to our work in that it also aims to simply the creation process of contextual help [3]. It allows help designers to record an interaction sequence that can be played back in the real interface, drawing a red circle on the relevant interface target in each step of the sequence. However, DocWizard is limited to the Eclipse editor or other interfaces based on the same platform (i.e., Java SWT).

Generalizability of Pixels

Pixels offer a possibility to make interaction techniques generally applicable to a wide range of interfaces [1] by providing interfaces with a general attachment point for these techniques [19]. Early effort exploring this possibility includes Triggers [24], IBOTS [33], and Segman [2]. Recent effort has seen successful applications of pixels to provide general solutions to many practical problems such as note taking [20], GUI automation [32], click target identification [15], and advanced behaviors [7]. In the current work, we seek to leverage the universality of pixels to make the creation of contextual help generally accessible. This idea has been previously suggested [32,8], but it has not been fully explored and only limited proof-of-concept examples have been presented.

CONTEXTUAL HELP USING SCREENSHOTS

We present a tool for creating and presenting contextual help for GUIs using screenshots. At creation time, this tool provides help designers with a set of visual scripting commands to specify which GUI components to provide help for and what help content to present. At presentation time,

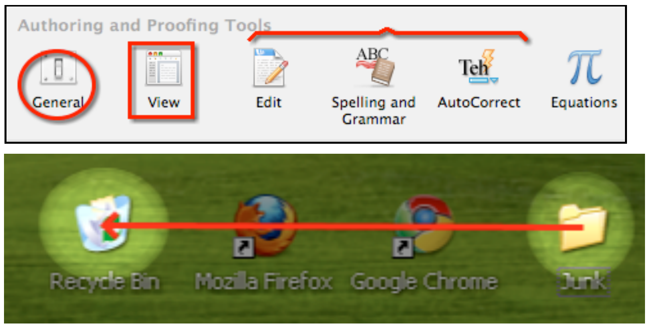


Figure 2. Graphics objects (top: circle, rectangle, bracket, bottom: spotlight, arrow) can be contextually rendered on GUIs to draw users' attention.

it uses computer vision to visually track GUI components on users' screen and display help near these components, allowing users to receive help in context.

Scripted Slideshow Metaphor

We surveyed existing GUI help based on screenshots or screencasts (e.g., eHow) and tools for creating them (e.g., MWSnap, Camtasia). We observed four important content characteristics of such help:

1. **Script:** Help tends to follow a particular script.
2. **Length:** Some help was as short as a single step while others were a dozen steps or more.
3. **Visuals:** Help can include a wide combination of text, screen capture, annotated screens and video.
4. **Pacing:** Some help was completely automated along a strict timeline while others may require user action to continue at each step.

Informed by this observation, we chose *scripted slideshow* as the metaphor to conceptualize contextual help. This metaphor adequately captures the characteristics above while remaining simple to promote learnability. First, a script can have steps that map naturally to slides in a slideshow. Second, a slideshow can be of arbitrary length. Third, each slide can include multiple graphics and/or text objects. Fourth, a slideshow can be advanced automatically or by user actions.

Visual Commands

Our tool provides help designers with a rich set of visual commands for creating a scripted slideshow. The Sikuli IDE [32] provides a convenient way for editing these visual commands. These commands include graphics, text, positioning, steps, animated effects, pacing, and support for ambiguity, each of which is described below:

Graphics: To draw a user's attention to an interface target, help designers can write a simple command based on an image of the target. For example, `circle( )` paints a circle around the icon in the actual interface. We support five types of graphical highlights: *circle*, *rectangle*, *bracket*, *arrow*, and *spotlight*, as shown in Figure 2.

Text: After drawing the user's attention to a target, it is also important to explain to the user what the target does. This is typically achieved by adding text objects near the target.

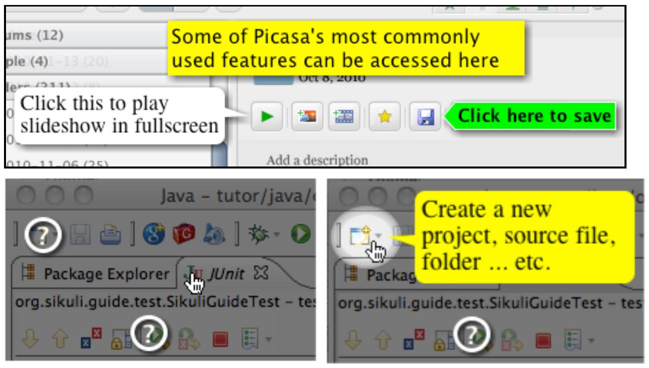

Some of Picasa's most commonly used features can be accessed here
Click this to play slideshow in fullscreen
Click here to save
Create a new project, source file, folder ... etc.

Figure 3. Text objects (top, from left to right: callout, text, flag, bottom: hotspot) can be contextually rendered on GUIs to provide helpful information.

We support three types of simple text objects: *text*, *callout*, and *flag* and one type of mouse-over text object: *hotspot*. Designers can add a new text object to a step by writing a simple function call that takes two arguments. The first argument specifies the target this text object is associated with and the second argument specifies the content to be displayed. Examples of these text objects can be seen in Figure 3.

Positioning: By default, graphics and text objects are automatically placed in the best position relative to the target. For instance, a *callout* is normally displayed to the left of a target but will be positioned to the right if there is not screen space to the left. Also, a *bracket* is placed according to the target's aspect ratio; it is placed above if the target is landscape and to the left if it is portrait. Alternatively, designers can position graphics and text objects themselves using a set of optional positioning parameters. Using positioning parameters designers can place objects in an empty area of the interface without occluding existing interface components. These parameters include the *side* parameter that specifies which side of a target to place contents (e.g., top, left, bottom, right), the *alignment* parameter that specifies how objects should be vertically or horizontally aligned with a target, and the *offset, margin,* and *spacing* parameters to further fine-tune the positions.

Step: Help designers can group graphics and text objects into steps using two methods. The simpler method is to express each step as a sequence of function calls for adding content objects followed by an explicit call to *show()* to mark the end of the step. Below is an example of a two-step contextual help expressed in this way:

Each call to *show()* displays the content objects added since the previous call to *show()*. It also clears the content before advancing to the next step.

```
callout(  , "Click this to save")
callout(  , "Click this to print")
show()
callout(  , "Click this to create a new document")
show()
```

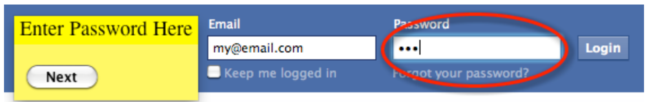
Another method is to represent steps as individual functions. At the end of the script, these functions can be passed as an array to a single call to *show()* to present these steps contextually in the users' interface. Below is the same contextual help as above re-written using this method:

```
def step1():
    callout( , "Click this to save")
    callout( , "Click this to print")
def step2():
    callout( , "Click this to create a new document")
show([step1, step2])
```

The former method offers simplicity, whereas the latter offers more power such as the ability to reorder and reuse content. These two methods can also be mixed to allow even greater flexibility.

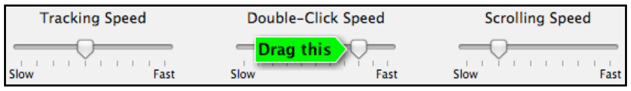
Animated Effects: Appropriate uses of animation can enhance contextual help [28] as they help grab users' attention. We enable help designers to add common animated effects such as fly-in, fade, blink, and circling, by supplying an optional animation parameter. For example, `rectangle( , animation = "flyin")` allows rectangle to enter the scene from the edge of the screen and *fly* to the target, and `flag( , "Here", animation = "circling")` makes the flag object moving in a circular motion next to the target.

Pacing: Help designers specify how the presentation of contextual help should be paced in two ways similar to the presentation of a slideshow. The typical way is to give the control to users. Users can decide when to advance to the next step by clicking the Next button. This behavior can be achieved by adding a dialog box to the current step. For example, `dialog("Enter Password Here")` displays a dialog box like below. This pacing option is preferable for problem-oriented and training-oriented contextual help, since users may spend an arbitrary amount of time to perform the operations related to a step.



Another way to control pacing is to specify a timeout period explicitly (e.g., *show(3)*). This method is more suitable for demoing new features when fewer actions are required.

Ambiguity: Ambiguity arises when multiple instances of the same or similar-looking targets are visible on the screen (e.g., multiple sliders). Help designers can disambiguate using other distinctive objects nearby (e.g., the slider under the label "Double-Click Speed") using Sikuli Script's built-in spatial operators. For example, a *flag* can be placed next to a specific slider as shown in the figure below by `flag(find(Double-Click Speed).below().find( ),"Drag this")`.



Evaluation of Prototype

We built a prototype and publicly released it in early 2011 as an extension to Sikuli. Based on an earlier survey on traditional GUI help, we identified three distinct scenarios GUI help is often administered:

- **New Features:** Introduces new capabilities to users to aid in future discovery.
- **One-time Solution:** Helps user solve specific problems (e.g., installing a printer). The problem often needs to be solved only once, which implies retention is less important.
- **Training:** Teaches users how to do important and recurring tasks. Retention is important and the training is sometimes scenario based, involving fictional data that can differ significantly from that of the user's real task.

Participants

Informed by the scenarios above, we recruited three distinct groups of help designers to participate in the evaluation of the prototype.

The first group is developers of novel GUI applications. It is crucial for this group to introduce new features to would-be users. We worked with selected members in Sikuli's user community and the researchers of the LifeFlow project [31] as representative users in this group.

The second group is tech support for large institutions. A large body of help content created by this group is to help users with one-time solutions. Participants representing this group were the members of the Office of Information Technology at the University of Maryland. We conducted an hour-long focus group with seven manager-level members overseeing matters related to user support.

The third group is computer instructors for novice computer users. Help created by this group tends to be training oriented. The representative participants we recruited for this group were three researchers who study how older adults learn and use computer technology. They regularly create computer tutorials based on screenshots and screencasts for older adults to learn how to find health information on the Web. Initially, we met with these researchers and demonstrated to them how to write simple scripts to create contextual versions of the same tutorials they previously created. After understanding the basic premise of our contextual help framework, these researchers worked with us to design and run two participatory design sessions (March 4 and 11) with older adults to further understand the framework's potential and limitations.

Findings and Discussion

All participants were impressed by the prototype and were able to create contextual help using our tool to benefit the respective user population they serve. However, they also identified several limitations discussed below:

Lack of robust startup. Before contextual help can begin, the target GUI is assumed to be visible so that the relevant component in the first step can be found on the screen and highlighted. For example, to present contextual help on

Facebook's login page, the page must be visible in order to highlight the relevant input fields. However, our participants reported that this assumption is often violated; users may launch a contextual help script before switching to the target GUI. Since the target GUI is not visible, the script simply fails immediately. This finding motivated us to develop features to improve the robustness of contextual help during startup.

Lack of flexible pacing options. Our participants found the two pacing options provided by the prototype too limiting. The time-based pacing option is impractical when it is not possible to predict how much time the user might need for the current step. The dialog-based pacing option is laborious, since it requires users to always make an extra click on the Next button to advance. These limitations motivated us to develop additional pacing options that are more flexible.

Lack of conditional help. A strict slideshow model follows a predetermined sequence of steps. While this is acceptable for many simple procedural tasks, our participants found it restrictive in certain scenarios where subsequent steps may be conditioned on what users have done and/or what the GUI looks like in the current step. The need to support these scenarios motivated us to develop features to allow contextual help to react to user actions and GUI states.

Lack of area-based selection. Our participants in the developer group informed us that when the purpose of contextual help is to give a feature overview of a GUI, it is often necessary to point at a general area in the GUI rather than individual components, for example, to introduce the toolbar area and the content area. In traditional screenshot help, an area is often marked and explained by a rectangle drawn around it and some text placed in the middle. In screencasts, a commonly used (but not necessarily effective) technique is to move the mouse cursor in a circular motion around an area while talking about it. However, it is often not possible to select an area by a single screenshot, as the size and the location the area can vary widely. For example, the toolbar area often changes as the size of the container window changes. This finding motivated us to develop an area selection method based on multiple visual landmarks.

Lack of cursor support. Our participants in the computer instructor group reported that in addition to receiving the cognitive knowledge of which interface target to interact with and how, some older adult learners would also like to receive assistance to perform the motor movement required to interact with the target. While it is possible to automate this interaction, doing so would deny these learners the chance of hands-on practice, which is crucial for learning [30]. This observation motivated us to implement a number of cursor enhancement techniques help designers can incorporate in contextual help to simplify motor movement.

Lack of tracking. In pixel-based GUI automation, a target's screen location needs to be found only once. As soon as the location is found, interaction commands such as *click* is delivered immediately to that location to simulate the effect of a human user clicking on that target. Similarly, in contextual help, drawing commands such as *circle()* can be applied to the target's location immediately after the location is determined. While the effect of an interaction command (e.g., click) ends as soon as that interaction is carried out, the effect of a drawing command needs to last for as long as users are still viewing the current step. Users may operate the GUI in a way that causes the target to move (e.g., scrolling) or disappear (e.g., switching to another window). However, the prototype did not re-compute the target's location; contextual help remained visible in the old location even though it was no longer relevant to any target nearby. Our participants found this behavior confusing. This observation motivated us to implement a visual tracker to continuously monitor the visual states of targets.

Lack of a WYSIWYG editor. While our participants in the tech support group and GUI developer group found the syntax for writing contextual help scripts easy to learn, our participants in the computer instructor group still preferred a visual editor supporting WYSIWYG. They were more familiar with GUI tools such as PowerPoint and Camtasia. This finding motivated us to develop a visual editor to further simplify the creation of contextual help.

Lack of an effective deployment strategy. To deploy a contextual help script, help designers can saved the script as a Sikuli executable (i.e., a runnable jar) and publish it on a website. Users can download this executable and run it on their own machines to receive contextual help. Our participants in the tech support group informed us that a more desirable deployment strategy would be to publish contextual help scripts alongside the traditional help already hosted on the official support website. The reason is that this website is the default place these users have learned to visit and look for help and the place they would trust. This finding motivated us to consider how contextual help can be seamlessly integrated with existing traditional help for more effective deployment.

ADVANCED FEATURES

We further present eight advanced features developed as a direct response to the limitations identified by our participants during the evaluation of the prototype.

Switching to the Target GUI

We provide two methods for help designers to assist users to switch to the target GUI when contextual help begins.

First, help designers can automate the steps to switch to the target GUI by composing a simple Sikuli automation script, for example, opening the System Preferences window by `click(); click(System Preferences...)`.

Second, help designers can explicitly instruct users how to switch to the target GUI when the GUI is not already visible. This instruction can also be provided in the form of contextual help. For example, many configuration tasks on Mac share a common starting point at the System Preferences window. A help designer can create a common help script for switching to this window. When needed, any other contextual help that begins from this window can invoke this script to help users start in the right place.

Pacing by Visual Triggers and User Clicks

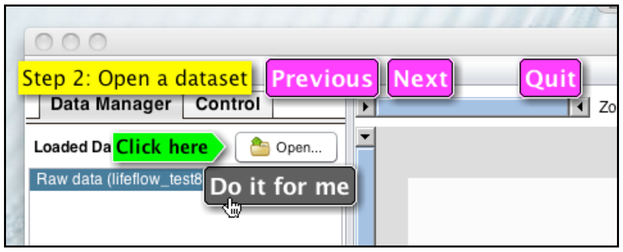
We developed two additional pacing options based on visual triggers and user clicks. To pace by visual triggers, help designers can give a list of visual patterns expected to be visible only in the next step. For example, the statement next(Empty Trash) means the Empty Trash button will be seen next and can be interpreted as a signal that the current step is over. To pace by user clicks, help designers can write a simple statement like clickable(Empty Trash). This statement places a mouse click detector on top the button. When the user clicks on that button, help automatically proceeds to the next step. We implemented this detector in pure Java using an almost transparent window on top of the button to intercept clicks. Once a click is detected, the window is hidden and the click is immediately delivered to the interface underneath the window.

Supporting Conditional Contextual Help

To address the lack of conditional help, we augmented the scripted slideshow model with the ability to present help conditionally. There are two scenarios when conditional contextual help naturally arises. The first scenario is when users need to make a choice in a particular step and the choice may take the users on a different path in subsequent steps. For example, at some point during printer installation it may be necessary to choose whether the printer is shared or not. Subsequent help depends on this choice. If users need to click on one of two buttons to make the choice, help designers can place a *clickable* object on each button. This clickable object can detect users' click on the button. Each clickable object is associated with a unique ID. The ID is returned by *show()* and can be used in a conditional statement to decide which help path to take next. The script below gives an example:

```
clickable( ,"KEYBOARD")
clickable( ,"MOUSE")
if (show() == "KEYBOARD"):
    help_keyboard()
else:
    help_mouse()
```

Alternatively, if the actual GUI does not offer the choices explicitly, help designers can place *virtual* buttons on the GUI and assign unique ID's to them to offer conditional contextual help, e.g., button("Do it"). The example shown next combines all of these mechanisms giving the choices to return to the previous step, advance to the next step, quit, or let the computer perform the step automatically.

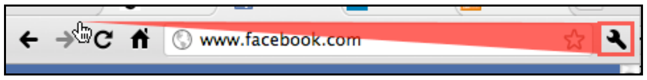


The second scenario conditional contextual help is useful is when help designers want to account for the different states the system may be in at run time and choose a help path accordingly. For example, a printer installer GUI may display different messages depending on whether or not the printer's driver can be found. Designers can use Sikuli Script's *exists()* function to check whether certain indicative visual pattern exists and choose the appropriate next step, using a conditional statement like below.

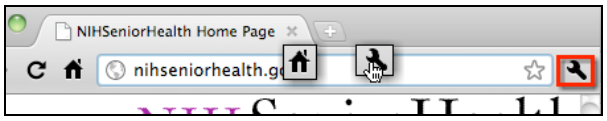
```
if exists( Printer not found. Unable to locate a printer with this address. Please check the IP address and try again. ):
    help_find_driver()
else:
    help_install_driver()
```

Assisting Target Selection

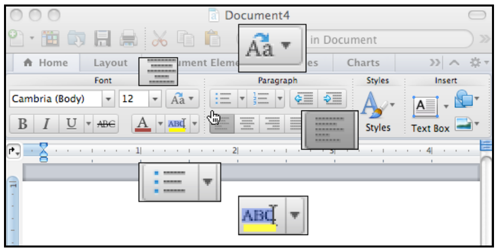
To address the lack of cursor support, we implemented three cursor enhancement techniques inspired by the state-of-the-art [10, 12] that can be invoked by help designers with simple visual commands. First, the *beam* cursor paints a lighted path from the current cursor location to the target and accelerates the movement toward the target if the user begins to move the cursor in that direction. This technique is suitable for problem-oriented help where each step tends to involve a single target. For example, beam(🔧) produces the following effect at presentation time:



Second, the *magnet* cursor acts like a magnet that attracts targets to the cursor's vicinity as *virtual* targets. Users can interact with these virtual targets as if interacting with the real targets. Since these virtual targets are closer, less effort is needed to operate the cursor. When users move the cursor over a virtual target, the corresponding real target is highlighted, allowing the users to learn where the target actually is. This cursor technique is especially useful when relevant targets are scattered across the GUI. For example, magnet([🔧,🏠]) achieves the effect shown below. It brings the two targets from opposite sides of the GUI to where the mouse cursor is; users do not need to move the cursor far across the screen to choose.



Third, the *clickcross* cursor [10] spreads a group of targets along a large orbit. This technique is useful when the targets are closely situated and making precise selection among them is hard for certain users without fine-motor control due to aging or disabilities. By spreading the targets apart, this technique has been shown to significantly reduce the physical demand on fine-motor control. For example, clickcross(, [ ]) spreads a group of targets (2nd argument) when the user clicks on the target area (1st argument), as shown next:



The theoretical capability of pixels to implement cursor enhancement techniques on a wide range of interfaces has been previously demonstrated by Prefab [7]. In this work, we further contribute by enabling designers to deploy these techniques in practice. Also, rather than system-wide application to every interface widget as was done in Prefab, we allow designers to apply these techniques judicially only to the parts of an interface where users need help the most.

Tracking Targets

When changes in a target's location or visibility are detected, the contextual help associated with that target must be repositioned or hidden/shown accordingly, maintaining the relevance of the help content, as exemplified in Figure 4. To detect location changes efficiently, we devised heuristics informed by several observations regarding how targets typically change their location:

1. In practice, users tend *not* to move interface targets when they try to follow contextual help. This implies that the tracker should always look at the target's last location to see if it remains before looking elsewhere to find where it might have moved.
2. When users did move interface targets, one common reason is scrolling such as scrolling a web interface. This implies that when a target is no longer seen in its last location, the first logical place to look for the target again is along the vertical strip and the horizontal strip extending from the target's last location.
3. Another common reason interface targets might move is that the user drags their container window to a different location. This implies that the next logical place to look should be around the same relative location from the target's last location as the relative location of the cursor from where it was dragged.
4. Interface targets in the same interface tend to move in the same way. This implies that after finding the first target's new location relative to its previous location, the search for the other targets should begin from the same relative locations from their previous locations.

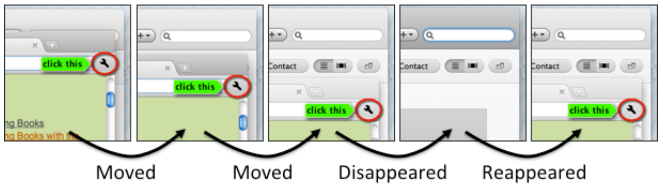


Figure 4. Tracking targets and updating contextual help.

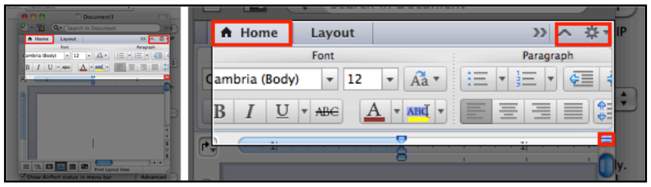
5. Lastly, if none of these heuristics worked, the tracker will fall back to full-screen search to relocate a moved interface target.

To detect visibility changes, we also rely on heuristics informed by how targets typically disappear and re-appear:

1. Common reasons a target may disappear are that the user minimizes the window or switches to another window. After a while, the user may restore the window or bring the window to the front, making that target visible again in its last location. This implies that once a target is no longer found and relevant content hidden, it is necessary to check the target's last location periodically so that the content can be redisplayed as soon as the target reappears there.
2. In some less common scenarios, a temporarily occluded target may reappear in a completely different location from its last location. The only way to detect reappearance of this kind is to conduct full-screen search. Since such scenarios are less common, we can perform full-screen search at lower frequency.

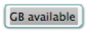
Defining Resizable Areas

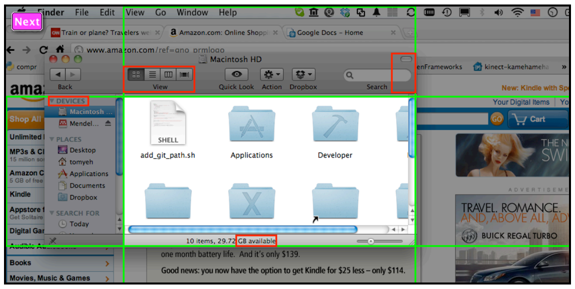
To address the lack of area-based selection, we implemented a new visual command *area*() to allow designers to describe an area by a set of salient landmarks. For example, spotlight(area([ ])) defines a rectangular area containing three visual landmarks and paints a spotlight on it, like below:



These landmarks will be found and tracked on the user's screen at run time to dynamically update the area.

Another way to specify an area is to describe the landmarks that form the vertical and horizontal bounds. Moreover, an area can also be derived from other areas through set operations such as union and intersection. For example, the script below defines the area shown on the next page,

```
h = area([ , ], range = "horizontal")
v = area([, ], range = "vertical")
spotlight(intersection(h,v))
```



Note that using this method, it is possible to describe an area using landmarks outside the area.

An alternative pixel-based approach would be to analyze the structure of an interface and derive a structural path (e.g., XPath) from the root window to the desired area [8]. But this approach cannot be applied to arbitrary areas that do not correspond to any container widget.

Supporting WYSIWYG

To further lower the skill requirement, we developed a visual editor to complement the script editor. To promote learnability, we borrowed familiar design elements from popular presentation software such as PowerPoint and Keynote. As a result, the editor has a layout that consists of the main editing area, an overview panel, and a toolbar, as shown in Figure 5.

Using the visual editor to create contextual help for a particular interface, a designer first captures the screenshot of the *entire* interface and imports it into the editor. The screenshot is placed on a slide in the editing area. The designer then can choose from the toolbar a type of content object (e.g., callout), select a location on the slide to insert the object (e.g., 20 pixels to the left of the OK button), and edit the properties of the object (e.g., set the text to Click Here). The above process is similar to annotating a screenshot using typical presentation software. The only difference is that the designer needs to explicitly mark the target (e.g., the OK button) and link relevant content objects to the target. This marking is achieved by placing a special *anchor* object on the target. An anchor object is a rectangle that can be resized by the designer to fit the target. At presentation time, pixels within this rectangle will be used to form a template to search the screen for the target's actual location. Once the target is found, content objects linked to the target will be displayed on the screen in the same relative locations to the target as indicated in the slide.

As in typical presentation software, our visual editor allows the designer to preview the content while they are developing it. In the preview mode, the editor is hidden temporarily so that the screenshot in the editing area would not be mistaken for the real interface. The designer can then bring the real interface to view, see whether content objects are rendered correctly with respect to the targets, and return to the editor to continue editing. Once the content is ready, it can be converted to equivalent script commands and exported to an executable to be distributed to users.

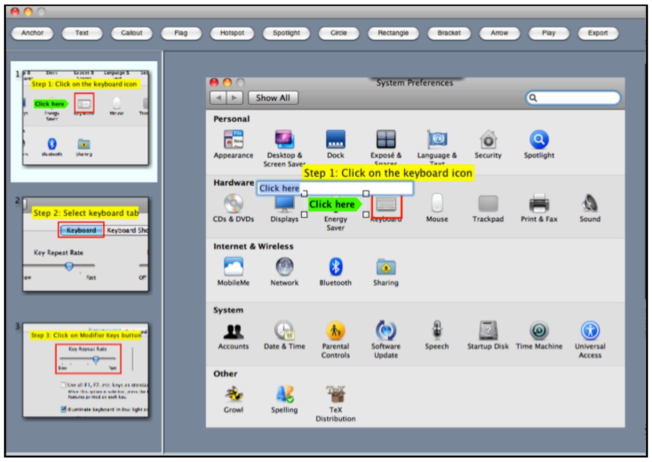

Anchor
Text
Callout
Flag
Hotspot
Spotlight
Circle
Rectangle
Bracket
Arrow
Play
Export
Step 1: Click on the keyboard icon
Click here
Step 2: Select keyboard tab
Step 3: Click on Modifier Keys button

Figure 5. Visual editor for creating contextual help.

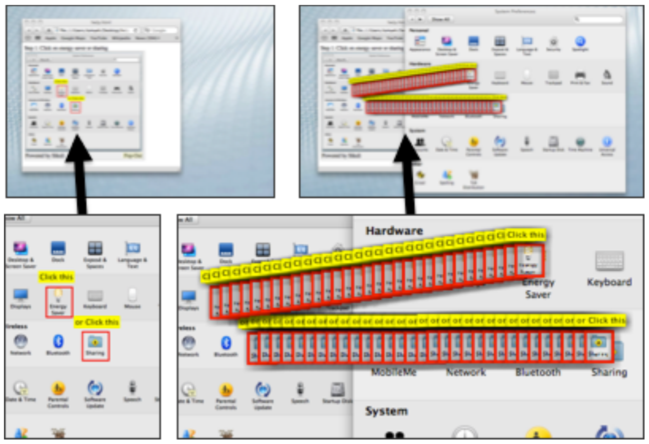


Figure 6. *Popout & Connect* for visualizing the mapping from static annotations on a screenshot to contextual help in a live interface.

Integrating with Existing Screenshot Help

To integrate contextual help more seamlessly with traditional screenshot-based help, we introduced the *Popout & Connect* feature to provide a clear transition from static screenshots in a web browser to contextual help on a live interface. When users click on an interface screenshot with some components annotated, they will see these components *popping out* from the screenshot along with the annotations. Once users switch to the live interface, these popped out virtual components then fly across the screen and land on top of the corresponding components on the live interface, carrying with them the annotations. This animated effect helps users visualize the *connection* between the static screenshot and the live interface. For example, in Figure 6, a screenshot in a web browser contains two annotated components. They popped out and flew across to the live interface on the right side of the screen.

Given a contextual help script and a static screenshot on a webpage, the *Popout & Connect* feature is implemented by first running the script only on that screenshot to locate the annotated elements so that we know where to render the *popout* effect. Since the screenshot may have been resized,

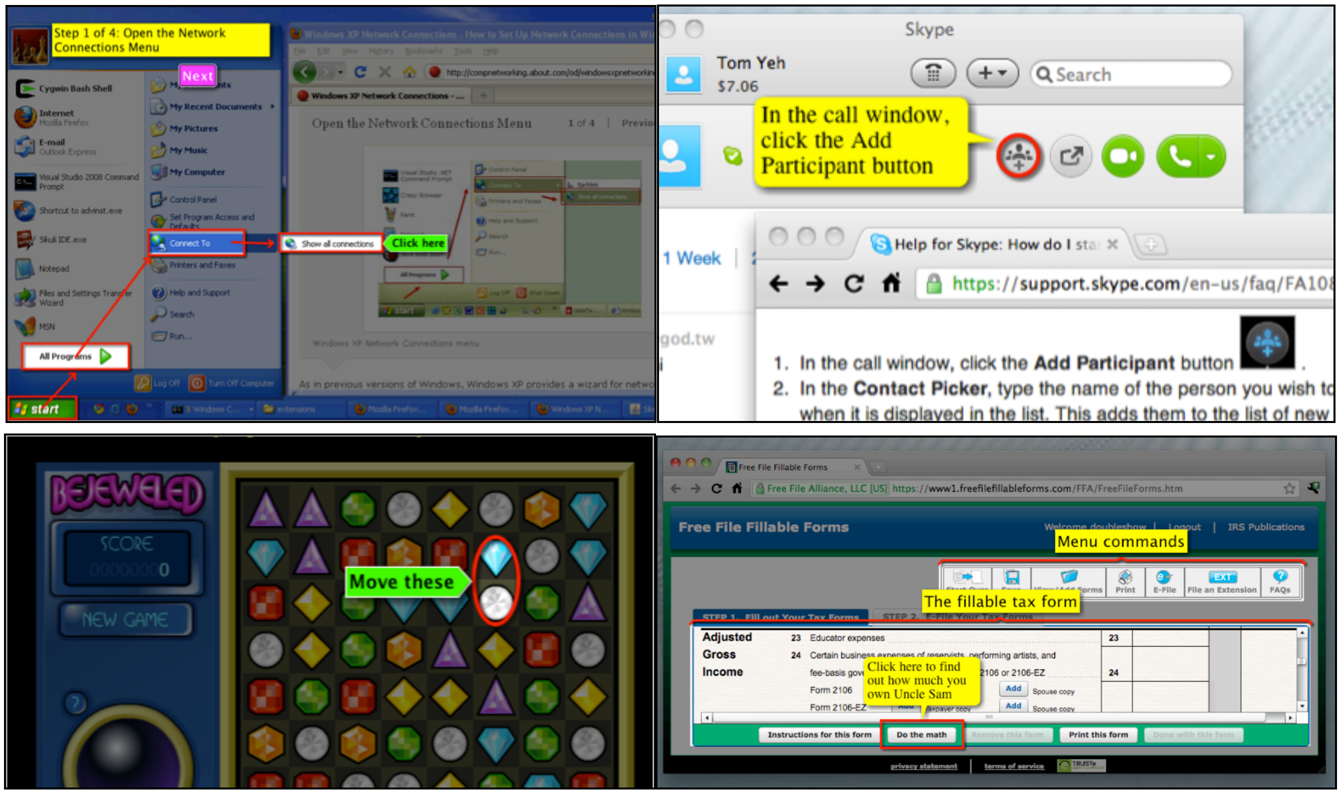

Step 1 of 4: Open the Network Connections Menu
Next
Click here
In the call window, click the Add Participant button
Move these
Menu commands
The fillable tax form
Click here to find out how much you own Uncle Sam

Figure 7. Examples of contextual help created by our tool. These examples demonstrate our tool's wide applicability to many applications (Skype, Flash), platforms (Windows, Mac, Web browser), and tasks (setup, chat, game, filing tax).

pattern matching needs to be performed across multiple scales in order to find the right scale. Afterward, the same script is run on the whole screen on regular intervals until the live interface becomes visible. Once the locations of relevant components are known on the live interface, it is possible to render the animation of these components flying across the screen to achieve the *connect* effect.

VALIDATION THROUGH APPLICATIONS

Our primary goal for this research is to enable a large population of help designers to generate contextual help for a broad range of interfaces and interactive tasks. As validation, Figure 7 shows a set of four representative examples of real-world applications of our tool.

To more systematically understand how broadly our technique can work, we attempted to create contextual help for a real life set of static help currently offered by University of Maryland's help desk website. We picked what we were told by our participants was one of the most visited topics: configuring Exchange Calendars. We sampled 60 of the 110 topics currently available related to this task on the site. There are a total of 376 steps in this sample. The average length of each topic is 6.26 step-long. We found 173 of 376 steps (46%) were already illustrated by screenshots. Out of these, 167 (96.5%) can be converted to contextual help using our tool. The problems we did find were due to three reasons:

1. The content is specific to time and date, for example, a dropdown menu prepopulated with dates corresponding to the day before and after.
2. The content is specific to the user, for example, a profile screen showing personal information.
3. The content is specific to a particular example, such as the name of a fictional meeting entered in a text box.

IMPLEMENTATION

The ability to locate widgets based on screenshots is provided by the Sikuli Library, an open-source cross-platform library for automating graphical user interfaces based on screenshots [32]. Rendering of contextual help on an interface is achieved by painting on a full screen transparent window that always stays on top. This transparent window is implemented based on Java Swing's JWindow class. The algorithm for tracking the movement of a target and repositioning relevant help content dynamically is implemented in Java. The WISIWYG editor is implemented in Java.

LIMITATIONS AND FUTURE WORK

By taking the pixel approach to generalizability, our tool inevitably suffers from the same limitations faced by any pixel-based technique such as scale changes and theme variations. In our particular case, the severity of these limitations varies depending on the user population. According to our participants in the tech support group, many institutional users they support share a standard issued computing environment and thus less susceptible to these limitations. On the other hand, older adult computer users often enlarge the font sizes for better readability, making scale invariance an important issue. A possible solution worth pursing in the future is to include multiple versions of the same pattern at different scales.

ACKNOWLEDGEMENT

This research is supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20071[1]. We thank the HCIL, Raimund Hocke, Chip Denman, Linda Rossi, and our older adult participants for their ideas, feedback and support.

REFERENCES

1. St. Amant, R., Lieberman, H., Potter, R., and Zettlemoyer, L. Programming by example: visual generalization in programming by example. *Communications of the ACM 43*, 3 (2000), 107-114.
2. St. Amant, R., Rey, M., Riedl, M.O., Ritter, F.E., and Reifers, A. Image Processing in Cognitive Models with SegMan Image processing in SegMan. In *Proc. of HCI*, 2001.
3. Bergman, L., Castelli, V., Lau, T., and Oblinger, D. DocWizards: A System for Authoring Follow-me Documentation Wizards. In *Proc. of UIST,* 2005.
4. Bigham, J.P., Lau, T., and Nichols, J. TrailBlazer: Enabling Blind Users to Blaze Trails Through the Web. In *Proc. of IUI,* 2009.
5. Chakravarthi, Y.A., Lutteroth, C., and Weber, G. AIMHelp: Generating Help for GUI Applications Automatically. In *Proc. of CHINZ,* 2009.
6. Chapuis, O. and Roussel, N. UIMarks: Quick Graphical Interaction with Specific Targets. In *Proc. of UIST,* 2010.
7. Dixon, M. and Fogarty, J. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proc. of CHI,* 2010.
8. Dixon, M., Leventhal, D. and Fogarty, J. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. In *Proc. of CHI 2011*.
9. Farkas, D.K. The role of balloon help. *ACM SIGDOC Asterisk Journal of Computer Documentation 17*, 2 (1993), 3-19.
10. Findlater, L., Jansen, A., Shinohara, K., et al. Enhanced Area Cursors: Reducing Fine Pointing Demands for People with Motor Impairments. In *Proc. of UIST,* 2010.
11. Grabler, F., Agrawala, M., Li, W., Dontcheva, M., and Igarashi, T. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics 28*, 3 (2009), 1.
12. Grossman, T. and Balakrishnan, R. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. In *Proc. of CHI,* 2005.
13. Grossman, T. and Fitzmaurice, G. ToolClips: An Investigation of Contextual Video Assistance for Functionality Understanding. In *Proc. of CHI,* 2010.
14. Huang, J. and B. Twidale, M.B. Graphstract: Minimal Graphical Help for Computers. In *Proc. of UIST,* 2007.
15. Hurst, A., Hudson, S.E., and Mankoff, J. Automatically identifying targets users interact with during real world tasks. In *Proc. of IUI,* 2010.
16. Kang, H. and Plaisant, C. New approaches to help users get started with visual interfaces: multi-layered interfaces and integrated initial guidance. In *Proc. of the 2003 annual national conference on Digital government research*, (2003).
17. Kelleher, C. and Pausch, R. Stencils-Based Tutorials: Design and Evaluation. In *Proc. of CHI,* 2005.
18. Lau, T., Bergman, L., Castelli, V., and Oblinger, D. Sheepdog: Learning Procedures for Technical Support. In *Proc. of IUI,* 2004.
19. Olsen, D.R., Hudson, S.E., Verratti, T., Heiner, J.M., and Phelps, M. Implementing interface attachments based on surface representations. In *Proc. of CHI,* 1999.
20. Olsen, D.R., Taufer, T., and Fails, J.A. ScreenCrayons: Annotating Anything. In *Proc. of UIST,* 2004.
21. Palaigeorgiou, G. and Despotakis, T. Known and Unknown Weaknesses in Software Animated Demonstrations (Screencasts): A Study in Self-Paced Learning Settings. *Journal of Information Technology Education 9*, (2010).
22. Pangoli, S. and Paternó, F. *Automatic generation of task-oriented help*. ACM Press, 1995.
23. Plaisant, C. and Shneiderman, B. Show Me! Guidelines for Producing Recorded Demonstrations. In *Proc. of VL/HCC,* 2005.
24. Potter, R.L.-S. Pixel data access: interprocess communication in the user interface for end-user programming and graphical macros. (1999).
25. Prabaker, M., Bergman, L., and Castelli, V. *An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and utilizing documentation*. ACM Press, New York, New York, USA, 2006.
26. Riedl, M.O. and St. Amant, R. Toward automated exploration of interactive systems. *Proceedings of the 7th international conference on Intelligent user interfaces - IUI '02*, ACM Press (2002), 135.
27. Sukaviriya, P. and Foley, J.D. *Coupling a UI framework with automatic generation of context-sensitive animated help*. ACM Press, New York, New York, USA, 1990.
28. Sukaviriya, P. *Dynamic construction of animated help from application context*. ACM Press.
29. Terry, M. and Mynatt, E.D. Side Views: Persistent, On-Demand Previews for Open-Ended Tasks. In *Proc. of UIST,* 2002.
30. Wiedenbeck, S. and Zila, P.L. Hands-on practice in learning to use software: a comparison of exercise, exploration, and combined formats. *ACM Transactions on Computer-Human Interaction 4*, 2 (1997), 169-196.
31. Wongsuphasawat, K., Guerra Gómez, J., Plaisant, C., Wang, T., Taieb-Maimon, M., Shneiderman, B. LifeFlow: Visualizing an Overview of Event Sequences. In *Proc. of CHI 2011*.
32. Yeh, T., Chang, T.-H., and Miller, R.C. Sikuli: Using GUI Screenshots for Search and Automation. In *Proc. of UIST, 2009*.
33. Zettlemoyer, L.S., St. Amant, R., and Dulberg, M.S. IBOTS: Agent Control Through the User Interface. In *Proc. of IUI,* 1999.

[1] The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.