

MIT Open Access Articles

Sikuli: Using GUI screenshots for search and automation

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: using GUI screenshots for search and automation. In Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09). ACM, New York, NY, USA, 183-192.

As Published: <http://dx.doi.org/10.1145/1622176.1622213>

Publisher: Association for Computing Machinery (ACM)

Persistent URL: <http://hdl.handle.net/1721.1/72686>

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0



Sikuli: Using GUI Screenshots for Search and Automation

Tom Yeh Tsung-Hsiang Chang Robert C. Miller
EECS MIT & CSAIL
Cambridge, MA, USA 02139
{tomyeh,vgod,rcm}@csail.mit.edu

ABSTRACT




We present Sikuli, a visual approach to search and automation of graphical user interfaces using screenshots. Sikuli allows users to take a screenshot of a GUI element (such as a toolbar button, icon, or dialog box) and query a help system using the screenshot instead of the element's name. Sikuli also provides a visual scripting API for automating GUI interactions, using screenshot patterns to direct mouse and keyboard events. We report a web-based user study showing that searching by screenshot is easy to learn and faster to specify than keywords. We also demonstrate several automation tasks suitable for visual scripting, such as map navigation and bus tracking, and show how visual scripting can improve interactive help systems previously proposed in the literature.


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

General terms: Design, Human Factors, Languages

Keywords: online help, image search, automation

INTRODUCTION

In human-to-human communication, asking for information about tangible objects can be naturally accomplished by making direct visual references to them. For example, to ask a tour guide to explain more about a painting, we would say “tell me more about this” while pointing to . Giving verbal commands involving tangible objects can also be naturally accomplished by making similar visual references. For example, to instruct a mover to put a lamp on top of a nightstand, we would say “put this over there” while pointing to  and  respectively.

Likewise, in human-to-computer communication, finding information or issuing commands involving GUI elements can be accomplished naturally by making direct visual reference to them. For example, asking the computer to “find information about this” while pointing to , we would like the computer to tell us about the Lasso tool for Photoshop and hopefully even give us links to web pages explaining this tool in detail. Asking the computer to “move

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

UIST'09, October 4–7, 2009, Victoria, British Columbia, Canada.
Copyright 2009 ACM 978-1-60558-745-5/09/10...\$10.00.

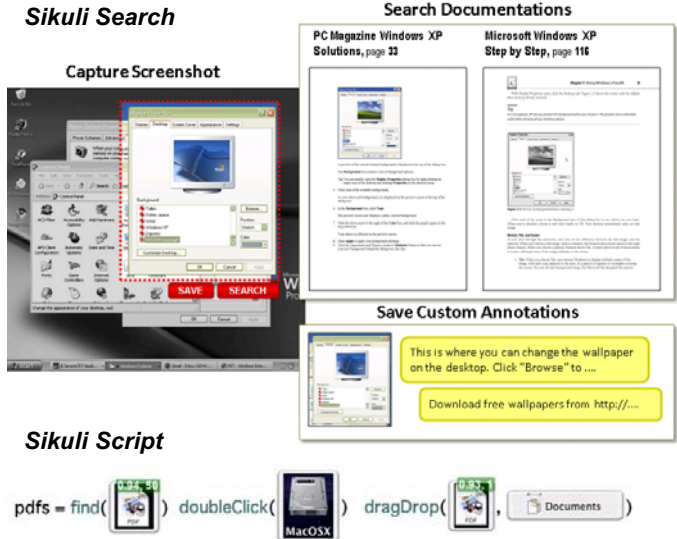




Figure 1: *Sikuli Search* allows users to search documentation and save custom annotations for a GUI element using its screenshot (captured by stretching a rectangle around it). *Sikuli Script* allows users to automate GUI interactions also using screenshots.

all these over there” while pointing to  and  respectively, means we would like the computer to move all the Word documents to the recycle bin.

However, some interfaces do not interact with us visually and force us to rely on non-visual alternatives. One example is *search*. With the explosion of information on the web, search engines are increasingly useful as a *first* resort for help with a GUI application, because the web may have fresher, more accurate, more abundant information than the application's built-in help. Searching the web currently requires coming up with the right keywords to describe an application's GUI elements, which can be challenging.

Another example is *automation*. Scripts or macros that control GUI elements either refer to an element by name, which may be unfamiliar or even unavailable to the user, or by screen location, which may change.

This paper presents Sikuli¹, a visual approach to searching and automating GUI elements (Figure 1). Sikuli allows users or programmers to make direct *visual* reference to GUI elements. To search a documentation database about a GUI element, a user can draw a rectangle around it and take

¹In Huichol Indian language, *Sikuli* means “God's Eye”, symbolic of the power of seeing and understanding things unknown.

Page



1. Surrounding Text

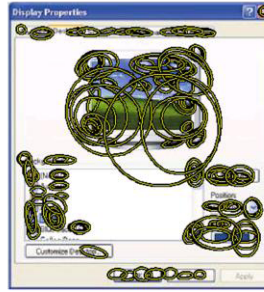
A preview of the current desktop background is displayed at the top of the dialog box.

The Background box contains a list of background options.

Tip You can quickly open the Display Properties dialog box by right-clicking an empty area of the desktop and clicking Properties on the shortcut menu.

5. Click each of the available backgrounds.

2. Visual Features



3. OCR of Embedded Text

```
Bla 3 T gpg Q5'E] Y
Themes Dã,-$ki0P Screen Saver Pppearance
Settings
.N.- ä€ç
QQ`-
2,2*% ~
R
-0
Background:
(None) | g_ Q Qowseu.
lh Ascent l-
Mumn Qosition:
```

Figure 2: Screenshots can be indexed by surrounding text, visual features, and embedded text (via OCR).

a screenshot as a query. Similarly, to automate interactions with a GUI element, a programmer can insert the element's screenshot directly into a script statement and specify what keyboard or mouse actions to invoke when this element is seen on the screen. Compared to the non-visual alternatives, taking screenshots is an intuitive way to specify a variety of GUI elements. Also, screenshots are universally accessible for all applications on all GUI platforms, since it is always possible to take a screenshot of a GUI element.

We make the following contributions in this paper:

- *Sikuli Search*, a system that enables users to search a large collection of online documentation about GUI elements using screenshots;
- an empirical demonstration of the system's ability to retrieve relevant information about a wide variety of dialog boxes, plus a user study showing that screenshots are faster than keywords for formulating queries about GUI elements;
- *Sikuli Script*, a scripting system that enables programmers to use screenshots of GUI elements to control them programmatically. The system incorporates a full-featured scripting language (Python) and an editor interface specifically designed for writing screenshot-based automation scripts;
- two examples of how screenshot-based interactive techniques can improve other innovative interactive help systems (Stencils [8] and Graphstract [7]).

This paper is divided into two parts. First we describe and evaluate Sikuli Search. Then we describe Sikuli Script and present several example scripts. Finally we review related work, discuss limitations of our approach, and conclude.

SCREENSHOTS FOR SEARCH

This section presents Sikuli Search, a system for searching GUI documentation by screenshots. We describe motivation, system architecture, prototype implementation, the user study, and performance evaluation.

Motivation

The development of our screenshot search system is motivated by the lack of an efficient and intuitive mechanism to search for documentation about a GUI element, such as a toolbar button, icon, dialog box, or error message. The abil-

ity to search for documentation about an arbitrary GUI element is crucial when users have trouble interacting with the element and the application's built-in help features are inadequate. Users may want to search not only the official documentation, but also computer books, blogs, forums, or online tutorials to find more help about the element.

Current approaches require users to enter keywords for the GUI elements in order to find information about them, but suitable keywords may not be immediately obvious.

Instead, we propose to use a screenshot of the element as a query. Given their graphical nature, GUI elements can be most directly represented by screenshots. In addition, screenshots are accessible across all applications and platforms by all users, in contrast to other mechanisms, like tooltips and help hotkeys (F1), that may or may not be implemented by the application.

System Architecture

Our screenshot search system, Sikuli Search, consists of three components: a screenshot search engine, a user interface for querying the search engine, and a user interface for adding screenshots with custom annotations to the index.

Screenshot Search Engine

Our prototype system indexes screenshots extracted from a wide variety of resources such as online tutorials, official documentation, and computer books. The system represents each screenshot using three different types of features (Figure 2). First, we use the text surrounding it in the source document, which is a typical approach taken by current keyword-based image search engines.

Second, we use visual features. Recent advances in computer vision have demonstrated the effectiveness of representing an image as a set of *visual words* [18]. A visual word is a vector of values computed to describe the visual properties of a small patch in an image. Patches are typically sampled from salient image locations such as corners that can be reliably detected in despite of variations in scale, translation, brightness, and rotation. We use the SIFT feature descriptor [11] to compute visual words from salient elliptical patches (Figure 2.3) detected by the MSER detector [12].

Screenshot images represented as visual words can be indexed and searched efficiently using an inverted index that contains an entry for each distinct visual word. To index an image, we extract visual words and for each word add the image ID to the corresponding entry. To query with another image, we also extract visual words and for each word retrieve from the corresponding entry the IDs of the images previously indexed under this word. Then, we find the IDs retrieved the most number of times and return the corresponding images as the top matches.

Third, since GUI elements often contain text, we can index their screenshots based on embedded text extracted by optical character recognition (OCR). To improve robustness to OCR errors, instead of using raw strings extracted by OCR, we compute 3-grams from the characters in these strings. For example, the word *system* might be incorrectly recognized as *system*. But when represented as a set of 3-grams over characters, these two terms are $\{sys, yst, ste, tem\}$ and $\{sys, yst, ste, ten\}$ respectively, which results in a 75% match, rather than a complete mismatch. We consider only letters, numbers and common punctuation, which together define a space of 50,000 unique 3-grams. We treat each unique 3-gram as a visual word and include it in the same index structure used for visual features.

User Interface for Searching Screenshots

Sikuli Search allows a user to select a region of interest on the screen, submit the image in the region as a query to the search engine, and browse the search results. To specify the region of interest, a user presses a hot-key to switch to Sikuli Search mode and begins to drag out a rubber-band rectangle around it (Figure 1). Users do not need to fit the rectangle perfectly around a GUI element since our screenshot representation scheme allows inexact match. After the rectangle is drawn, a search button appears next to it, which submits the image in the rectangle as a query to the search engine and opens a web browser to display the results.

User Interface for Annotating Screenshots

We have also explored using screenshots as hooks for annotation. Annotation systems are common on the web (e.g. WebNotes² and Shiftspace³), where URLs and HTML page structure provide robust attachment points, but similar systems for the desktop have previously required application support (e.g. Stencils [8]). Using screenshots as queries, we can provide general-purpose GUI element annotation for the desktop, which may be useful for both personal and community contexts. For example, consider a dialog box for opening up a remote desktop connection. A user may want to attach a personal note listing the IP addresses of the remote machines accessible by the user, whereas a community expert may want to create a tutorial document and link the document to this dialog box.

Sikuli Search’s annotation interface allows a user to save screenshots with custom annotations that can be looked up

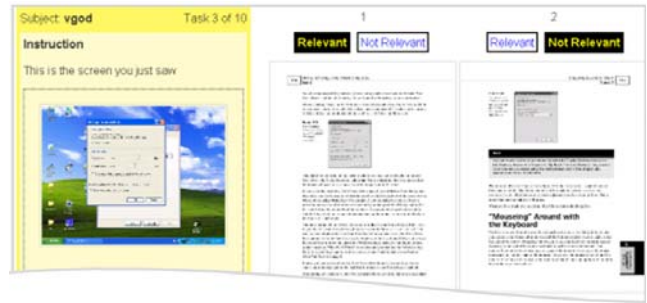


Figure 3: User study task, presenting a desktop image containing a dialog box (left) from which to formulate a query, and search results (right) to judge for relevance to the dialog box.

using screenshots. To save a screenshot of a GUI element, the user draws a rectangle around it to capture its screenshot to save in the visual index. The user then enters the annotation to be linked to the screenshot. Optionally, the user can mark a specific part of the GUI element (e.g., a button in a dialog box) to which the annotation is directed.

Prototype Implementation

The Sikuli Search prototype has a database of 102 popular computer books covering various operating systems (e.g., Windows XP, MacOS) and applications (e.g., Photoshop, Office), all represented in PDF⁴. This database contains more than 50k screenshots. The three-feature indexing scheme is written in C++ to index these screenshots, using SIFT [11] to extract visual features, Tesseract⁵ for OCR, and Ferret⁶ for indexing the text surrounding the screenshots. All other server-side functionality, such as accepting queries and formatting search results, is implemented in Ruby on Rails⁷ with a SQL database. On the client side, the interfaces for searching and annotating screenshots are implemented in Java.

User Study

We have argued that a screenshot search system can simplify query formulation without sacrificing the quality of the results. To support these claims, we carried out a user study to test two hypotheses: (1) screenshot queries are *faster* to specify than keyword queries, and (2) results of screenshot and keyword search have roughly the same relevance as judged by users. We also used a questionnaire to shed light on users’ subjective experiences of both search methods.

Method

The study was a within-subject design and took place online. Subjects were recruited from Craigslist and compensated with \$10 gift certificates. Each subject was asked to perform two sets of five search tasks (1 practice + 4 actual tasks). Each set of tasks corresponds to one of the two conditions (i.e., image or keyword) that are randomly ordered. The details of a task are as follows. First, the

² <http://www.webnotes.com/>
³ <http://shiftspace.org/>

⁴ <http://www.pdfchm.com/>
⁵ <http://code.google.com/p/tesseract-ocr/>
⁶ <http://ferret.davebalmmain.com/>
⁷ <http://rubyonrails.org/>

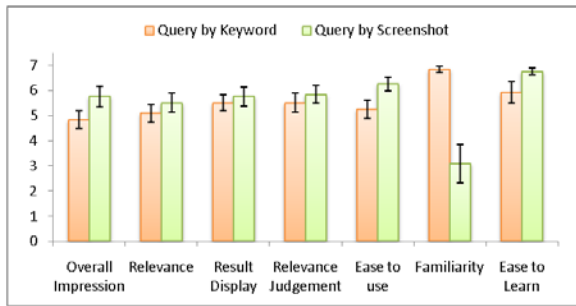


Figure 4: Mean and standard error of the subjective ratings of the two query methods.

subject was presented an image of the whole desktop with an arbitrarily-positioned dialog box window. Each dialog box was randomly drawn without replacement from the same pool of 10 dialog boxes. This pool was created by randomly choosing from those in our database known to have relevant matches. Next, the subject was told to specify queries by entering keywords or by selecting a screen region depending on which condition. The elapsed time between the first input event (keypress or mouse press) and the submit action was recorded as the *query formulation time*. Finally, the top 5 matches were shown and the subject was asked to examine each match and to indicate whether it seemed relevant or irrelevant (Figure 3).

After completing all the tasks, the subject was directed to an online questionnaire to rate subjective experiences with the two methods on a 7-point Likert scale (7: most positive). The questions were adapted from the evaluation of the Assieme search interface [6] and are listed below:

1. What is your overall impression of the system?
2. How relevant are the results?
3. Does the presentation provide good overview of the results?
4. Does the presentation help you judge the relevance?
5. Does the input method make it easy to specify your query?
6. Is the input method familiar?
7. Is the input method easy to learn?

Results

Twelve subjects, six males and six females, from diverse backgrounds (e.g., student, waiter, retiree, financial consultant) and age range (21 to 66, mean = 33.6, sd = 12.7), participated in our study and filled out the questionnaire. All but one were native English speakers.

The findings supported both hypotheses. The average query formulation time was less than half as long for screenshots (4.02 sec, s.e.=1.07) as for keyword queries (8.58 sec, s.e.=.78), which is a statistically significant difference $t(11) = 3.87, p = 0.003$. The number of results rated relevant (out of 5) averaged 2.62 (s.e.=.26) for screenshot queries and 2.87 (s.e.=.26) for keyword queries, which was not significant $t(11) = .76, p = .46$.

The responses to the questionnaire for subjective rating of the two query methods are summarized in Figure 4. The most dramatic difference was familiarity (Q6) $t(11) = 4.33, p < .001$. Most subjects found keyword queries more famil-

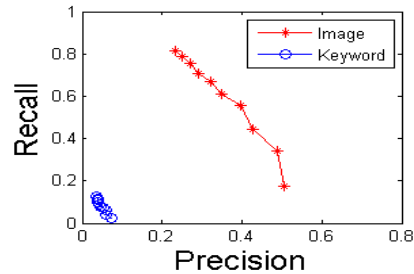


Figure 5: Retrieval performance of search prototype.

iar than screenshot queries. There was a trend that subjects reported screenshot queries as easier to use (Q5) and to learn (Q7) compared to keyword queries $p < .1$.

We observed several subjects improved speed in making screenshot queries over several tasks, which may suggest that while they were initially unfamiliar with this method, they were able to learn it rather quickly.

Performance Evaluation

We evaluated the technical performance of the Sikuli Search prototype, which employs the three-feature indexing scheme (surrounding text, embedded text, and visual features), and compared it to that of a baseline system using only traditional keyword search over surrounding text. The evaluation used a set of 500 dialog box screenshots from a tutorial website for Windows XP⁸ (which was not part of the corpus used to create the database). For the keyword search baseline, we manually generated search terms for each dialog box using words in the title bar, heading, tab, and/or the first sentence in the instruction, removing stop words, and capping the number of search terms to 10.

We measured coverage, recall, and precision. Coverage measures the likelihood that our database contains a relevant document for an arbitrary dialog box. Since the exact measurement of coverage is difficult given the size of our database, we examined the top 10 matches of both methods and obtained an estimate of 70.5% (i.e., 361/500 dialogs had at least one relevant document). To estimate precision and recall, we obtained a ground-truth sample by taking the union of all the correct matches given by both methods for the queries under coverage (since recall is undefined for queries outside the coverage).

Figure 5 shows the precision/recall curves of the two methods. As can be seen, the screenshot method achieved the best results. We speculate that the keyword baseline performed poorly because it only relies on the text surrounding a screenshot that might not necessarily correlate with the text actually embedded in the screenshot. The surrounding text often provides additional information rather than repeating what is already visible in the screenshot. However, users often choose keywords based on the visible text in the dialog box and these keywords are less likely to retrieve documents with the screenshots of the right dialog box.

⁸ <http://www.leeindy.com/>

SCREENSHOTS FOR AUTOMATION

This section presents Sikuli Script, a visual approach to UI automation by screenshots. We describe motivation, algorithms for matching screenshot patterns, our visual scripting API, an editor for composing visual scripts, and several example scripts.

Motivation

The development of our visual scripting API for UI automation is motivated by the desire to address the limitations of current automation approaches. Current approaches tend to require support from application developers (e.g., AppleScript and Windows Scripting, which require applications to provide APIs) or accessible text labels for GUI elements (e.g. DocWizards [2], Chickenfoot [3], and Co-Scripter [10]). Some macro recorders (e.g. Jitbit⁹ and QuicKeys¹⁰) achieve cross-application and cross-platform operability by capturing and replaying low-level mouse and keyboard events on a GUI element based on its absolute position on the desktop or relative position to the corner of its containing window. However, these positions may become invalid if the window is moved or if the elements in the window are rearranged due to resizing.

Therefore, we propose to use screenshots of GUI elements directly in an automation script to programmatically control the elements with low-level keyboard and mouse input. Since screenshots are universally accessible across different applications and platforms, this approach is not limited to a specific application. Furthermore, the GUI element a programmer wishes to control can be dynamically located on the screen by its visual appearance, which eliminates the movement problem suffered by existing approaches.

Finding GUI Patterns on the Screen

At the core of our visual automation approach is an efficient and reliable method for finding a target pattern on the screen. We adopt a hybrid method that uses template-matching for finding small patterns and invariant feature voting for finding large patterns (Figure 6).

If the target pattern is small, like an icon or button, template matching based on normalized cross-validation [4] can be done efficiently and produce accurate results. Template matching can also be applied at multiple scales to find resized versions of the target pattern (to handle possible changes in screen resolution) or at grayscale to find patterns texturally similar but with different color palettes (to handle custom color themes).

However, if the target pattern is large, like a window or dialog box, template-matching might become too slow for interactive applications, especially if we allow variations in scale. In this case, we can consider an algorithm based on invariant local features such as SIFT [11] that have been used to solve various computer vision problems successfully over the past few years. The particular algorithm we have adapted for our purpose [13] was originally used for



Figure 6: Examples of finding small patterns of varying sizes (a) and colors (b) by template-matching, and large patterns of varying sizes and orientations (c) by invariant feature voting.

detecting cars and pedestrians in a street scene. This algorithm learns from a set of invariant local features extracted from a training pattern (a screenshot of a GUI element) and derives an object model that is invariant to scale and rotation. Encoded in the object model is the location of its center relative to each local feature. To detect this object model in a test image (screen), we first extract invariant features from the image. For each feature, we can look up the corresponding feature in the object model and infer where the location of the object center is if this feature actually constitutes a part of the object. If a cluster of features consistently point at the same object center, it is likely these features actually form an object. Such clusters can be identified efficiently by voting on a grid, where each feature casts a vote on a grid location closest to the inferred object center. We identify the grid locations with the most votes and obtain a set of hypotheses. Each hypothesis can be verified for its geometric layout by checking whether a transformation can be computed between this hypothesis and the training pattern based on the set of feature correspondences between the two. The result is a set of matches and their positions, scales, and orientations, relative to the target pattern. Note that while rotational invariance may be unnecessary in traditional 2D desktop GUIs, it can potentially benefit next-generation GUIs such as tabletop GUIs where elements are oriented according to users' viewing angles.

Visual Scripting API


Sikuli Script is our visual scripting API for GUI automation. The goal of this API is to give an existing full-featured scripting language a set of image-based interactive capabilities. While in this paper we describe the API in Python syntax, adapting it to other scripting languages such as Ruby and JavaScript should be straightforward.

The Sikuli Script API has several components. The `find()` function takes a target pattern and returns screen regions matching the pattern. The `Pattern` and `Region` classes represent the target pattern and matching screen regions, respectively. A set of action commands invoke mouse and keyboard actions on screen regions. Finally, the visual dictionary data type stores key-values pairs using images as keys. We describe these components in more detail below.

⁹ <http://www.jitbit.com/macrorecorder.aspx>

¹⁰ <http://www.startly.com/products/qkx.html>

Find

The `find()` function locates a particular GUI element to interact with. It takes a visual pattern that specifies the element's appearance, searches the whole screen or part of the screen, and returns regions matching this pattern or *false* if no such region can be found. For example, `find()` returns regions containing a Word document icon.


Pattern


The Pattern class is an abstraction for visual patterns. A pattern object can be created from an image or a string of text. When created from an image, the computer vision algorithm described earlier is used to find matching screen regions. When created from a string, OCR is used to find screen regions matching the text of the string.

An *image*-based pattern object has four methods for tuning how general or specific the desired matches must be:



- **exact()**: Require matches to be identical to the given search pattern pixel-by-pixel.
- **similar(float similarity)**: Allow matches that are somewhat different from the given pattern. A similarity threshold between 0 and 1 specifies how similar the matching regions must be (1.0 = exact).
- **anyColor()**: Allow matches with different colors than the given pattern.
- **anySize()**: Allow matches of a different size than the given pattern.


Each method produces a new pattern, so they can be chained together. For example,

```
Pattern().similar(0.8).anyColor().anySize()
```



matches screen regions that are 80% similar to  of any size and of any color composition. Note that these pattern methods can impact the computational cost of the search; the more general the pattern, the longer it takes to find it.

Region





The Region class provides an abstraction for the screen region(s) returned by the `find()` function matching a given visual pattern. Its attributes are *x* and *y* coordinates, height, width, and similarity score. Typically, a Region object represents the top match, for example, `r = find()` finds the region most similar to  and assigns it to the variable *r*. When used in conjunction with an iterative statement, a Region object represents an array of matches. For example,

`for r in find()` iterates through an array of matching regions and the programmer can specify what operations to perform on each region represented by *r*.

Another use of a Region object is to constrain the search to a particular region instead of the entire screen. For exam-





ple, `find().find()` constrains the search space of the second `find()` for the ok button to only the region occupied by the dialog box returned by the first `find()`.

To support other types of constrained search, our visual scripting API provides a versatile set of constraint operators: **left**, **right**, **above**, **below**, **nearby**, **inside**, **outside** in 2D screen space and **after**, **before** in reading order (e.g., top-down, left-right for Western reading order). These operators can be used in combination to express a rich set of search semantics, for example,











```
find()  
.inside()  
.right()  
.find()
```

Action

The action commands specify what keyword and/or mouse events to be issued to the center of a region found by `find()`. The set of commands currently supported in our API are:

- **click(Region)**, **doubleClick(Region)**: These two commands issue mouse-click events to the center of a target region. For example, `click()` performs a single click on the first close button found on the screen. Modifier keys such as Ctrl and Command can be passed as a second argument.
- **dragDrop(Region target, Region destination)**: This command drags the element in the center of a target region and drops it in the center of a destination region. For example, `dragDrop(, )` drags a word icon and drops it in the recycle bin.
- **type(Region target, String text)**: This command enters a given text in a target region by sending keystrokes to its center. For example, `type(, "UIST")` types the "UIST" in the Google search box.

Visual Dictionary

A visual dictionary is a data type for storing key-value pairs using images as keys. It provides Sikuli Script with a convenient programming interface to access Sikuli Search's core functionality. Using a visual dictionary, a user can easily automate the tasks of saving and retrieving data based on images. The syntax of the visual dictionary is modeled after that of the built-in Python dictionary. For example, `d = VisualDict({: "word", : "powerpoint"})` creates a visual dictionary associating two application names with their icon images. Then, `d[]` retrieves the string *powerpoint* and `d[]` = "excel" stores the string *excel* under . Because  is not a key, `d[]` returns *false* and `d[]` raises a *KeyError* exception. Using the pattern modifiers described earlier, it is possible to explicitly control how strict or fuzzy the matching criterion should be. For instance, `d[Pattern().exact()]` requires pixel perfect matching, whereas `d[Pattern().anySize()]` retrieves an array of values associated with different sizes of the same image.

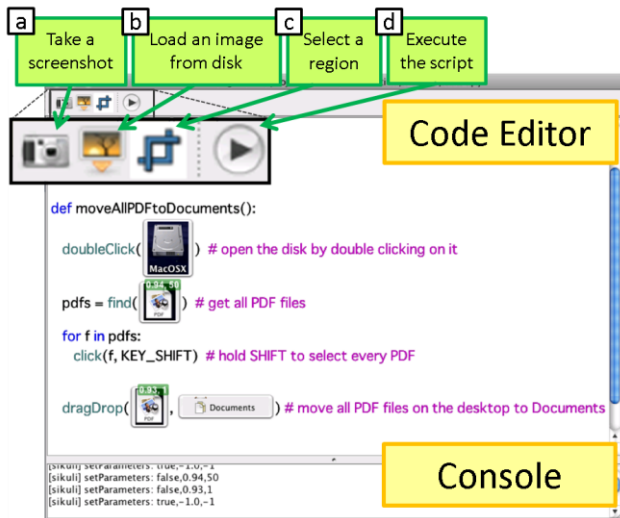



Figure 7: Editor for writing Sikuli scripts in Python.

Script Editor

We developed an editor to help users write visual scripts (Figure 7). To take a screenshot of a GUI element to add to a script, a user can click on the camera button (a) in the toolbar to enter the screen capture mode. The editor hides itself automatically to reveal the desktop underneath and the user can draw a rectangle around an element to capture its screenshot. The captured image can be embedded in any statement and displayed as an inline image. The editor also provides code completion. When the user types a command, the editor automatically displays the corresponding command template to remind the user what arguments to supply. For example, when the user types `find`, the editor will expand the command into `find()`. The user can click on the camera button to capture a screenshot to be the argument for this `find()` statement. Alternatively, the user can load an existing image file from disk (b), or type the filename or URL of an image, and the editor automatically loads it and displays it as a thumbnail. The editor also allows the user to specify an arbitrary region of screen to confine the search to that region (c). Finally, the user can press the execute button (d) and the editor will be hidden and the script will be executed.

The editor can also preview how a pattern matches the current desktop (Figure 8) under different parameters such as similarity threshold (a) and maximum number of matches (b), so that these can be tuned to include only the desired regions. Match scores are mapped to the hue and the alpha value of the highlight, so that regions with higher score are redder and more visible.

Implementation and Performance

We implemented the core pattern matching algorithm in C++ using OpenCV, an open-source computer vision library. The full API was implemented in Java using the Java Robot class to execute keyboard and mouse actions. Based on the Java API, we built a Python library to offer high-

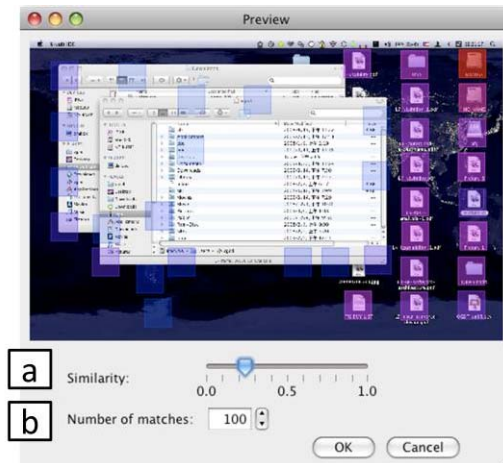


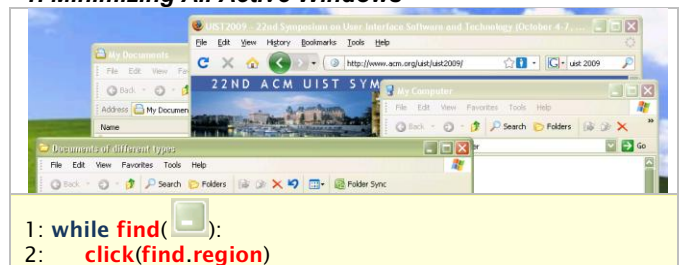
Figure 8: The user can adjust the similarity threshold and preview the results. Here, the threshold (0.25) is too low, resulting in many false positives.

level scripting syntax described above. Libraries for other high-level scripting languages can also be built based on this API. We built the editor using Java Swing. To visualize the search patterns embedded in a script, we implemented a custom `EditorKit` to convert from pure-text to rich-text view. Finally, we used `Jython` to execute the Python scripts written by programmers. All components of the system are highly portable and have been tested on Windows and Mac OS X. We benchmarked the speed on a 3.2 GHz Windows PC. A typical call to `find()` for a 100x100 target on a 1600x1200 screen takes less than 200 msec, which is reasonable for many interactive applications. Further speed gains might be obtained by moving functionality to the GPU if needed in the future.

Sikuli Script Examples

We present six example scripts to demonstrate the basic features of Sikuli Script. For convenience in Python programming, we introduce two variables; `find.region` and `find.regions`, that respectively cache the top region and all the regions returned by the last call to `find`. While each script can be executed alone, it can also be integrated into a larger Python script that contains calls to other Python libraries and/or more complex logic statements.

1. Minimizing All Active Windows







This script minimizes all active windows by calling `find` repeatedly in a `while` loop (1) and calling `click` on each minimize button found (2), until no more can be found.

2. Deleting Documents of Multiple Types



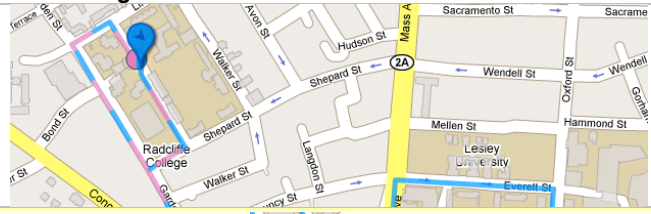
```

1: def recycleAll(x):
2:   for region in find(x).anySize().regions:
3:     dragDrop(region, )
4: patterns = [ , ,  ]
5: for x in patterns:
6:   recycleAll(x)



```

This script deletes all visible Office files (Words, Excel, PowerPoint) by moving them to the recycle bin. First, it defines a function `recycleAll()` to find all icons matching the pattern of a given file type and move them to the recycle bin (1-3). Since icons may appear in various sizes depending on the view setting, `anySize` is used to find icons of other sizes (2). A `for` loop iterates through all matching regions and calls `dragDrop` to move each match to the recycle bin (3). Next, an array is created to hold the patterns of the three Office file types (4) and `recycleAll()` is called on each pattern (5-6) to delete the files. This example demonstrates Sikuli Script's ability to define reusable functions, treat visual patterns as variables, perform fuzzy matching (`anySize`), and interact with built-in types (array).

3. Tracking Bus Movement



```

1: street_corner = find()
2: while not street_corner.inside().find().similar(0.7):
3:   sleep(60)
4: popup("The bus is arriving!")


```

This script tracks bus movement in the context of a GPS-based bus tracking application. Suppose a user wishes to be notified when a bus is just around the corner so that the user can head out and catch the bus. First, the script identifies the region corresponding to the street corner (1). Then, it enters a `while` loop and tries to `find` the bus marker `inside` the region every 60 seconds (2-3). Notice that about 30% of the marker is occupied by the background that may change as the marker moves. Thus, the `similar` pattern modifier is used to look for a target 70% similar to the given pattern. Once such target is found, a popup will be shown to notify the user the bus is arriving (4). This example demonstrates Sikuli Script's potential to help with everyday tasks.

4. Navigating a Map



```

1: while find() and
   not find.regions.nearby().find("Houston"):
2:   target = find.region
3:   dragDrop(target, [target.x - 100, target.y])


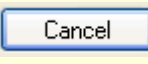

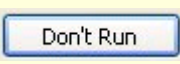
```

This script automatically navigates east to Houston following Interstate 10 on the map (by dragging the map to the left). A `while` loop repeatedly looks for the Interstate 10 symbol and checks if a string `Houston` appears nearby (1). Each time the string is not found, the position 100 pixels to the left of the Interstate 10 symbol is calculated and the map is dragged to that position (3), which in effect moves the map to the east. This movement continues until the Interstate 10 can no longer be found or Houston is reached.

5. Responding to Message Boxes Automatically



```


1: d = VisualDict({  :  })
...
100: d[  ] = 
101: import win32gui
102: while true:
103:   w = win32gui.getActiveWindow()
104:   img = getScreenshot(w)
105:   if img in d:
106:     button = d[img]
107:     click(Region(w).inside().find(button))

```

This script generates automatic responses to a predefined set of message boxes. A screenshot of each message box is stored in a visual dictionary `d` as a key and the image of the button to automatically press is stored as a value. A large number of message boxes and desired responses are defined in this way (1-100). Suppose the `win32gui` library is imported (101) to provide the function `getActiveWindow()`, which is called periodically (102) to obtain the handle to the active window (103). Then, we take a screenshot by calling `getScreenshot()` (104) and check if it is a key of `d` (105). If so, this window must be one of the message boxes specified earlier. To generate an automatic response, the relevant button image is extracted from `d` (106) and the region `inside` the active window matching the button image is found and clicked (107). This example shows Sikuli Script can interact with any Python library to accomplish tasks neither can do it alone. Also, using a `VisualDict`, it is possible to handle a large number of patterns efficiently.

6. Monitoring a Baby



```
1: while find() .similar(0.7):
2:   sleep(60)
3: popup("Check the baby!")
```

This script demonstrates how visual scripting can go beyond the realm of desktop to interact with the physical world. The purpose of this script is to monitor for baby rollover through a webcam that streams video to the screen. A special green marker is pasted on the baby's forehead. By periodically checking if the marker is present (1- 2), the script can detect baby rollover when the marker is absent and issue notification (3).

INTEGRATION WITH OTHER SYSTEMS

The visual approach to search and automation can potentially impact a broad range of interactive systems. In this section we present two such systems that can be enhanced by our image-based interactive techniques.

Creating Stencils-based Tutorials

The Stencils system described in [8] is a tutorial presentation technique that seeks to direct the user's attention to correct interface components by displaying translucent colored stencils with holes. This technique has been shown to enable users to complete tutorials faster. However, to adopt this technique, an application must implement certain system-specific functionality (i.e., a Java interface), which requires changing the application's source code.

Our screenshot annotation tool can potentially broaden the applicability of Stencils to other applications by linking tutorials to their screenshots without any modification to the source code. For example, Figure 9 shows a tutorial we created following the Stencils design guideline that instructs users how to add new contacts for Skype, based on an actual tutorial in the official documentation.

Automating Minimal Graphical Help

The Graphstrack system described in [7] is a graphical help presentation technique based on abstracted screenshots. Instead of showing the screenshot of the whole interactive window to illustrate a step in a sequential task, Graphstrack automatically extracts and displays the screenshot of only the region around the relevant UI element in the window. Graphstrack can record and replay user interactions across applications based on low-level input. However, as mentioned in [7], the fidelity of the replay depends on the absolute locations of the GUI elements and may break if the windows containing those elements are moved.

Sikuli Script can help overcome this drawback since it identifies GUI elements by appearances rather than absolute position. Figure 10 shows the first 3 of a 12-step Graphstrack tutorial in [7] illustrating how to move an im-



Figure 9: A Stencils-based tutorial [8] generated by the Sikuli search and annotation tool.



Figure 10: Converting a Graphstrack [7] (a) to a Sikuli script (b).

age from Photoshop to Word. Converting this tutorial to a Sikuli script is straightforward since the screenshots of relevant GUI elements are already in the tutorial and can be used directly as search patterns for Action commands.

RELATED WORK

Help systems: Graphical illustrations such as screenshots are helpful for teaching users about a software application. Technical writers have long used specialized tools such as RoboHelp to produce documentation with captured screen images of applications. Harrison [5] found that users learned more quickly following tutorials illustrated by screenshots than reading textual-only tutorials. However, Knabe [9] found that users often find it difficult to locate the interface components pictured in the tutorials. Several attempts have been made to address this difficulty. Bergman *et al* [2] proposed the *follow-me* documentation wizard that steps a user through a script representation of a procedure by highlighting portions of the text as well application UI elements. These help systems are image-oriented and can be enhanced by searching screen patterns.

Creating graphically illustrated documentation can be time consuming and expensive. Several attempts have been made to automate this task. For example, Sukaviriya & Foley [19] proposed a framework called Cartoonists for generating animated help as a sequence of user actions needed to perform the task by demonstrating these actions first. Moriyon *et al* [14] proposed a technique to generate hypertext help messages from a user interface design model. Furthermore, Pangoli & Paternó [15] demonstrated how to generate task-oriented help from user interface specification. However, these generators assume the knowledge of the user interface design model, which requires expert designers. In contrast, our screenshot annotation tool requires no such expertise and can potentially allow crowd sourcing.

Image-based interaction: The idea of supporting interactions by analyzing the visual patterns rendered on the

screen was examined in the late 90's. Potter [16] was the first to explore this idea and referred to it as *direct pixel access* and championed its potential for supporting application-independent end-user programming. His Triggers system supported novel visual tasks such as graphical search-and-replace and simulated floating menus. While Triggers can be configured through an interactive dialog to perform some basic tasks similar to the Sikuli Script examples presented earlier (i.e., 1,3,4), it is not a full scripting language and does not support fuzzy matching. Zettlemoyer & St. Amant [20] described VisMap and VisScript. The former inferred high-level, structured representations of interface objects from their appearances and generated mouse and keyboard gestures to manipulate these objects. The later provided a basic set of scripting commands (*mouse-move*, *single-click*, *double-click* and *move-mouse-to-text*) based on the output of VisMap, but was not integrated with a full-feature scripting language. Furthermore, St. Amant *et al* [1] explored several possibilities of programming-by-example through *visual generalization* by observing user behaviors and inferring general patterns based on the visual properties and relationships of user interface objects. While these early pioneering works shed light on the potential of image-based interaction, they led to almost no follow-up work, mostly because the practicality was limited by the hardware and computer vision algorithms of the time. However, faster hardware and recent advances in vision algorithms particularly those based on invariant local features have now presented us with an opportunity to reexamine this idea and develop practical image-based interactive applications.

CONCLUSION AND FUTURE WORK

This paper presented a visual approach to search and automation and demonstrated its various benefits and capabilities. We conclude by mentioning two limitations of the approach and offering possible solutions for future work.

Theme variations: Many users prefer a personalized appearance theme with different colors, fonts, and desktop backgrounds, which may pose challenges to a screenshot search engine. Possible solutions would be to tinker with the image-matching algorithm to make it robust to theme variation or to provide a utility to temporarily switch to the default theme whenever users wish to search for screenshots. UI automation is less affected by theme variations when users write scripts to run on their own machines. However, sharing scripts across different themes may be difficult. Possible solutions would be to derive a conversion function to map patterns between themes or to require users to *normalize* the execution environment by switching to the default theme when writing sharable scripts.

Visibility constraints: Currently, Sikuli Script operates only in the visible screen space and thus is not applicable to invisible GUI elements, such as those hidden underneath other windows, in another tab, or scrolled out of view. One solution would be to automate scrolling or tab switching actions to bring the GUI elements into view to interact with it visually. Another solution would resort to platform- or

application-specific techniques to obtain the full contents of windows and scrolling panes, regardless of their visibility.

REFERENCES

1. St. Amant, R., H. Lieberman, R. Potter, and L. Zettlemoyer. Programming by example: visual generalization in programming by example. *Commun. ACM* 43(3), 107–114, 2003.
2. Bergman, L., V. Castelli, T. Lau, and D. Oblinger. DocWizards: a system for authoring follow-me documentation wizards. *Proc. UIST '05*, 191–200, 2005.
3. Bolin, M., M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. *Proc. UIST '05*, 163–172, 2005.
4. Forsyth, D. and Ponce, J., *Computer Vision: A Modern Approach*, Prentice Hall, USA, 2002.
5. Harrison, S. M. A comparison of still, animated, or nonillustrated on-line help with written or spoken instructions in a graphical user interface. *Proc. CHI '95*, 82–89, 1995.
6. Hoffmann, R., J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proc. UIST '07*, 13–22, 2007.
7. Huang, J. and M. B. Twidale. Graphstract: minimal graphical help for computers. *Proc. UIST '07*, 203–212, 2007.
8. Kelleher, C. and R. Pausch. Stencils-based tutorials: design and evaluation. *Proc. CHI '05*, 541–550, 2005.
9. Knabe, K. Apple guide: a case study in user-aided design of online help. *Proc. CHI '95*, 286–287, 1995.
10. Leshed, G., E. M. Haber, T. Matthews, and T. Lau. CoScripter: automating & sharing how-to knowledge in the enterprise. *Proc. CHI '08*, 1719–1728, 2008.
11. Lowe, D. G. Object recognition from local scale-invariant features. *Proc. International Conference on Computer Vision*, 1150–1157, 1999.
12. Matas, J., O. Chum, U. Martin, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. *Proc. British Machine Vision Conference '02*, 2002.
13. Mikolajczyk, K., B. Leibe, and B. Schiele. Multiple object class detection with a generative model. *Proc. Computer Vision and Pattern Recognition '06*, 26–36, 2006.
14. Moriyon, R., P. Szekely, and R. Neches. Automatic generation of help from interface design models. *Proc. CHI '94*, 225–231, 1994.
15. Pangoli, S. and F. Paternó. Automatic generation of task-oriented help. *Proc. UIST '95*, 181–187, 1995.
16. Potter, R. L. Pixel data access: interprocess communication in the user interface for end-user programming and graphical macros. *Ph. D. thesis*, College Park, MD, USA, 1999.
17. Prabaker, M., L. Bergman, and V. Castelli. An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and utilizing documentation. *Proc. CHI '06*, 241–250, 2006.
18. Sivic, J. and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. *Proc. International Conference on Computer Vision*, 2003.
19. Sukaviriya, P. and J. D. Foley. Coupling a UI framework with automatic generation of context-sensitive animated help. *Proc. UIST '90*, 152–166, 1990.
20. Zettlemoyer, L. S. and St. Amant, R. A visual medium for programmatic control of interactive applications. *Proc. CHI '99*, 199–206, 1999.