Let PBE systems use the visual properties of on-screen interactive elements to bridge the user's view of an application and its underlying programmable functionality.

VISUAL GENERALIZATION IN Programming BY Example

In programming by example (PBE), also known as programming by demonstration, the system records actions performed by users in the interface and produces a generalized program that can be used later in analogous examples. A key issue is how to describe the actions and objects selected by the user, so the system can determine what kind of generalizations are possible.

When a user selects a graphical object on the screen, for example, most PBE systems describe the object in terms of the properties of the underlying application data. If the user selects a link on a Web page, the PBE system might represent that selection based on the link's HTML properties.

Here, we explore a different, and radical, approach—using the visual properties of the interaction elements themselves, including size, shape, color, and appearance—to describe user intentions. Only recently has the speed of image processing made feasible PBE systems' real-time analysis of screen images. We have not yet realized the goal of a PBE system that uses "visual generalization" but feel this approach is important enough to warrant describing and promoting the idea publicly. (Visual generalization means the inference of general patterns in user behavior based on the visual properties and relationships of user interface objects.)

Visual information can supplement the information available from other sources, suggesting new kinds of generalizations not possible from application data alone. In addition, these generalizations can map more closely to user intentions, especially beginning users, who rely on the same visual information when making selections. Moreover, visual generalization can sometimes remove one of the main stumbling blocks—reliance on application

Robert St. Amant, Henry Lieberman, Richard Potter, and Luke Zettlemoyer

programming interfaces (APIs)—preventing PBE from being used with conventional applications. When necessary, PBE systems can work exclusively from the visual appearance of applications—without explicit cooperation from their APIs.

If You See It, You Should Be Able to Program It

Every PBE system has what Dan Halbert, developer of SmallStar, one of the earliest PBE systems, calls the "data description problem," or how to figure out what users mean when they select objects on a screen [1]. How an object is described could produce very different effects the next time a procedure is recorded and generalized by the system. When interacting with a PBE system, a user selecting an icon for a file foo.bar in a desktop file system could mean any one of several things: that specific file alone and no other; any file whose name is foo.bar; any icon that happens to be at the location that was clicked; or something else.

PBE systems can work **exclusively** from the visual appearance of applications—without **explicit** cooperation from the application's API.

Most systems deal with this issue by mapping the user's selection to the application's data model or to its files, to email messages, to circles and boxes in a drawing, or to something else. They then permit generalizations on the data's properties, such as file names and message senders. But the user's intuitive description of an object might sometimes depend on the actual visual properties of the screen elements themselves—regardless of whether these properties are explicitly represented in the application's command set. We endorse the idea of using these visual properties to permit PBE systems to perform visual generalization.

As an example of why visual generalization may prove useful, especially in PBE applications, suppose we want to write a program to save all the links on a Web page that have not been clicked and viewed by the user at some point in time (see Figure 1). If the Netscape browser happens to include the operation "Move to the next unfollowed link" as a menu option or in its API, system developers or even everyday users might be able to automate the activity through macro recorders, such as Quickeys. Unfortunately, Netscape does not include this operation; it also lacks a "Move to the next link" operation. Even if we had access to the Web page's HTML source, we still wouldn't know which links the user had followed. Identification of previously followed links is an example of a general problem for PBE systems in interfacing to almost any application. Interactive applications make it easy for users to carry out procedures and do not expect to be treated as subroutines by external systems.

This Netscape example shows the conceptual gap between a user's view of an application and its underlying programmable functionality. Bridging this gap can be a difficult problem in the design of PBE systems in which representation of user actions might completely fail to match the user's intentions.

Perhaps we are looking at this problem from the wrong perspective. From that of the user, the functionality of an interactive application is defined by its user interface, developed to cover specific tasks, communicate through appropriate abstractions, and accommodate the user's cognitive, perceptual, and physical abilities. A PBE system might be improved significantly if it could work in the same medium as its user, processing the visual environment with all its associated information. One of our goals is to emphasize this key insight.

What does visual generalization buy us? Let's imagine a PBE system incorporating techniques to process a visual interactive environment, extracting information that is potentially relevant to the user's intentions. The system would gain at least the following performance benefits:

Integration into existing environments. Historically, most PBE systems have been built on top of isolated research systems, rather than on commercial applications. Some have been promising but haven't been adopted because of integration difficulties. Visual PBE systems, independent of source code and API constraints, could potentially reach an unlimited user audience.

Consistency. Independence from an application's source code or API gives a PBE system added flexibility. Similar applications often have a similar appearance and behavior; for example, users can switch between Web browsers with little difficulty. A visual PBE system could take advantage of functional and visual consistency to operate

Figure 1. Can we write a program that saves all the links that haven't been followed?

across similar applications with little or no modification.

New sources of information. Most important, some kinds of visual information may be difficult or impossible to obtain through other means. Moreover, this information is generally related closely to the user's understanding of a particular application.

These benefits are all available to PBE system developers but apply equally to PBE system users. In the Netscape example, a visual PBE system would be able to run on top of the existing browser, without requiring the use of a substitute research system. Because the standard Netscape browser employs the convention of displaying the followed links in red and the unfollowed links in blue, a user might specify the "Save the next unfollowed link" action in visual terms as "Move to the next line of blue text," then invoke the "Save link as" operation. This specification exploits a new source of visual information—the color of the link. Finally, the general consistency among browsers should allow the same PBE system to work with both Netscape and Microsoft Internet Explorer, a much trickier proposition for API-based systems.

However, providing a visual processing ability raises some novel challenges for PBE systems.

Image processing. How can these systems extract visual information at the image-processing level? Such processing must happen in an interactive system, interleaved with user actions and observations of the system, thus raising significant efficiency issues. Information extraction is an issue of the basic technical feasibility of the visual approach to PBE. Our experience with VisMap, a software agent system under development since 1998 by the authors St. Amant and Zettlemoyer, found that real-time analysis of the screen is possible on today's high-end machines.

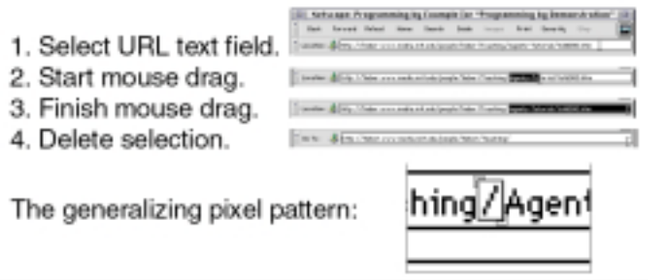
Information management. How can a system process low-level visual data to infer high-level information relevant to user intentions? For example, a visual object under a mouse pointer might be represented as a rectangle, a generic window region, or a window region specialized for some purpose, such as text entry or freehand drawing. A text box with a number in it might be an element of a fill-in form, a table in a text document, or a cell in a spreadsheet. Concern over what is being represented is also important for generalization from low-level events to the abstractions they implement. Is the user simply clicking on a rectangle or performing a confirmation action?

Brittleness. How can a system deal gracefully with visual variations beyond the scope of a solution? In the Netscape example of collecting unfollowed links, users may change the colors Netscape uses to distinguish followed from unfollowed links, thereby (perhaps) rendering obsolete a previously recorded procedure. A link may extend over more than a single line of text, so the mapping between lines and links is not exact. Similar blue text might appear in a GIF image and be captured inadvertently by the procedure. Moreover, if the program visually parses the screen, links that do not appear (because they are below the current scrolling position) are not included. Out of sight, out of mind. Though the problem of lost or misplaced links might be cured by programming a loop scrolled through the page, in the same way a user would scroll through the page, we can put most of these problems in a novel light by observing that they can be difficult for even a human to solve. Almost everyone has been fooled now and then by advertising graphics camouflaged as legitimate interface objects; without more information (such as might be provided by an API call), a visual PBE system cannot hope to do better.

Just the Pixels, Ma'am (Low-level Visual Generalization)

The author Potter's work on pixel-based data access

Figure 2. Steps in a mouse macro to move a browser up one directory. And using Triggers to select a pixel pattern that generalizes the macro.



pioneered the approach of treating a screen image as the source for generating descriptions for generalization. The Triggers PBE system he helped develop at the University of Maryland performs exact pattern matching on screen pixels to infer information that is otherwise unavailable to an external system [7]. A "trigger" is a condition-action pair. For example, triggers are defined for such tasks as surrounding a text field with a rounded rectangle in a drawing program, shortening lines so they intersect an arbitrary shape and converting text to a bold typeface. The user defines a trigger by stepping through a sequence of actions in an application, adding annotations to be used later by Triggers when appropriate. Once a set of triggers is defined, users can activate them (iteratively and exhaustively) to carry out their actions.

Several strategies can be used to process visual pixel information so it can be used to generalize computer programs (see Figure 2). The visual processing strategy used by the Triggers system computes locations of exact patterns within the screen image. For example, suppose a user records a mouse macro that modifies a URL in order to display the next higher directory in a Web browser. Running the macro can automate this process, but only for one specific URL, because the mouse locations are recorded with fixed coordinates. However, this macro can be generalized by using pixel pattern matching on the screen image. The pattern the system should use is what users would look for if they were doing the task manually—in this case, the pixel pattern of a slash character. Finding the next to last occurrence of this pattern gives a location from which the macro can begin the macro's mouse drag, thus generalizing the macro so it works with most URLs.

Although this macro program affects targets, such as characters, strings, URLs, and Web pages, the program's internal data is only low-level pixel patterns and screen coordinates. It is how this low-level data is used within the rich graphical user interface (GUI) context that gives higher-level meaning to the low-level data. The fact that a low-level program can map so directly to much higher-level meaning reveals how conveniently a GUI's visual information is organized for productive work [7].

The most valuable advantage of this visual processing strategy is that the low-level data and operators of the programming system can map to many high-level meanings, even those not originally envisioned by the programming system's developer. The disadvantage is that high-level internal processing of the information is difficult, since an outside context is required for most interpretation.

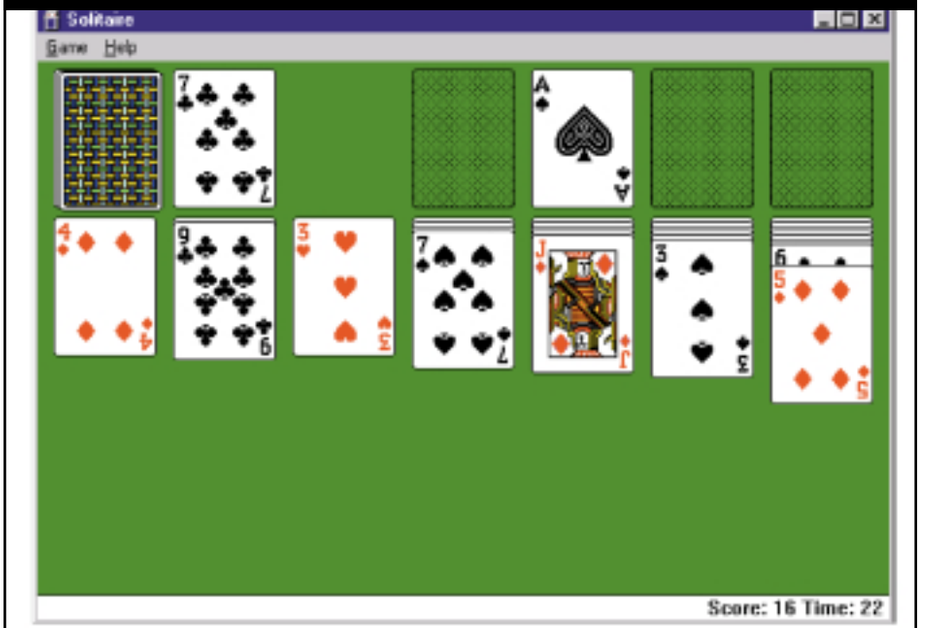
How can a system process low-level visual data to infer high-level information relevant to a user's intention?

Another system that performs data access at the pixel level is AutoMouse, developed by Kakuya Yamamoto, a researcher in Kyoto University in Japan; it searches the screen for rectangular pixel patterns and click anywhere within a pattern [9]. Copies of the patterns can then be arranged on a document and connected to form simple visual programs. Each pattern can be associated with different mouse and keyboard actions.

What You See Is What You Record (High-level Visual Generalization)

Our VisMap, which is in some ways a conceptual successor to Triggers, is a programmable set of sensors, effectors, and skeleton controllers for visual interaction with off-the-shelf applications [10]. Sensor modules take pixel-level input from the display, run the data through image-processing algorithms, and build a structured representation of visible interface objects. Effector modules generate mouse and keyboard gestures to manipulate these objects.

Figure 3. VisSolitaire source data and visual processing results.



Visual Processing Results

Stock:	Tableau:	Foundation:
(:BACK)	(0 :BEHIND) (4 :DIAMONDS)	(:BLANK)
(7 :CLUBS)	(1 :BEHIND) (9 :CLUBS)	(:ACE :SPADES)
	(0 :BEHIND) (3 :HEARTS)	(:BLANK)
	(3 :BEHIND) (7 :SPADES)	(:BLANK)
	(4 :BEHIND) (:JACK :DIAMONDS)	
	(5 :BEHIND) (3 :SPADES)	

VisMap is designed as a programmable user model—an artificial user through which developers can explore the characteristics of a particular user interface.

VisMap is not, by itself, a PBE system but demonstrates that visual generalization is practical in any interface. Our current research seeks to apply its approach in a full PBE system. VisMap translates the pixel information to data types that have more meaning outside the GUI context. For example, building on VisMap in 1999, we developed VisSolitaire, a simple visual application that plays Microsoft Windows Solitaire (see Figure 3). VisMap translates the pixel information to data types representing the state of a generic game of Solitaire. This state provides input to an AI planning system that plays a reasonably adept game of solitaire, from the starting deal to a win or loss. It does not use an API or otherwise get any cooperation from Microsoft Solitaire.

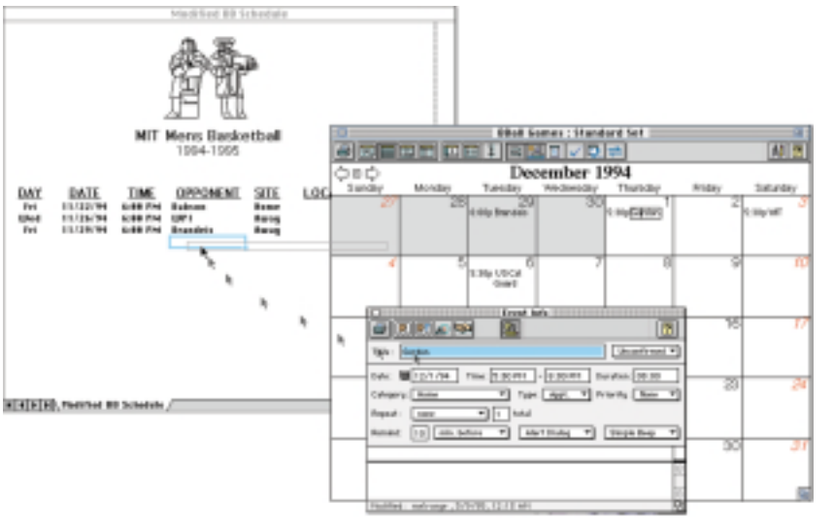
VisSolitaire's control cycle alternates between screen parsing and generalized action. VisSolitaire processes the screen image to identify cards and their positions. When the cards are located by the application's image-processing algorithms, a visual grammar characterizes them based on relative location and visual properties. In this way, the system identifies the stacks of cards that form the stock, tableau, and foundation, while classifying each card based on visual identification of its suit and rank.

VisSolitaire interleaves a bottom-up pattern-recognition process with a top-down interpretation of visual patterns. Key to VisSolitaire's effectiveness is the loose coupling between these two components. The strategic, game-playing module represents its actions in general terms, such as "Move any ace that is on top of a tableau pile to an empty foundation slot." The system's visual processing component maps this command to the specific state of the Solitaire application—by invoking the command "Move the ace of spades to the second foundation slot." VisSolitaire, like a human solitaire player, relies on the layout of the cards to guide its actions, rather than relying solely on the visual representations of the cards alone.

VisMap's ability to recognize cards illustrates an application-specific visual recognition procedure that can be used in visual generalization. To make a visual-recognition approach work for PBE in general, we may have to define visual grammars that describe the meaning of particular interface elements, that is, the visual language of a particular application.

For example, if we understand that the format of a monthly calendar is a grid of boxes, with each box representing the date and lines within the boxes representing particular appointments, we can infer the properties of an appointment object in the Now-Up-to-Date calendar program from Power On Software. It is also possible that there are other properties of the appointment object, such as the duration of the

Figure 4. Tatlin infers that the user wants to copy data from a calendar to a spreadsheet.



appointment, that are not represented in the visual display, so we may not be able to infer them from the screen representation alone.

Developing application-display-format grammars is time-consuming work for even expert developers and is not for end users. However, a developer's effort for a particular application can be amortized over all the uses of the application. The model of the application can be incomplete, capturing only the aspects of the application data of current interest.

One way to use the results of this kind of processing in a PBE system is to adopt an approach like that of Tatlin [5], which infers user actions by periodically polling applications for their state and comparing successive states to determine user actions. Tatlin uses the "examinability" of the application data models in a Microsoft Excel spreadsheet and Now Up-to-Date via the Applescript interprocess communication language (see Figure 4). In the scenario in the figure, a user copies information from a calendar and pastes it into the spreadsheet. Tatlin "sees" that the data pasted into the spreadsheet is the same as the data selected in the calendar and infers the transfer operation.

If PBE system developers wanted to develop descriptions of the visual interface of the calendar and the spreadsheet, they could analyze the screen image, even without access to the underlying application data.

Other research offers further evidence of the potential of visual generalization. For example, Fred Lakin, a researcher in the Performing Graphics Co., built several programming environments around an object-oriented graphical editor called Vmacs [4]. He used a recognition procedure on the visual relationships between objects to attach semantics to sketched objects, thus implementing a kind of visual generalization. Notably, the grammars used to drive the recognition procedure were themselves represented visually in Vmacs. A kind of visual generalization was used by David Kurlander, a Microsoft researcher, to automate search-andreplace procedures [2]. But while Lakin and Kurlander were able to access the visual properties of objects in their own purpose-built graphical editors directly, our methods extract the same kind of visual properties directly from a pixel-level analysis of the screen.

Novel Generalizations (Generalizing on Grids)

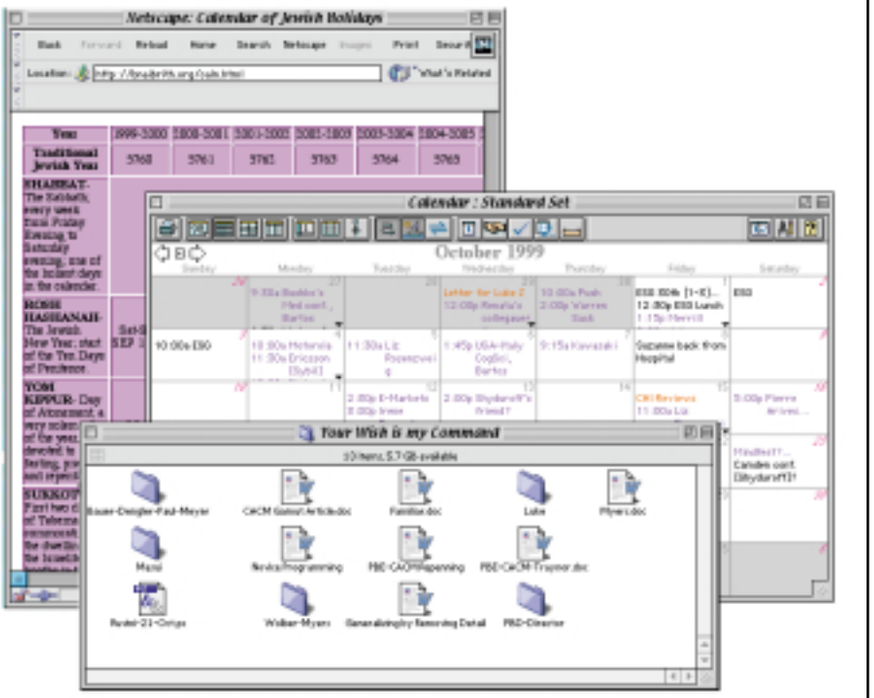
Visual generalization promises the option of having more kinds of generalizations than are possible by generalizing solely from the properties of the underlying application data. As an example of the kind of useful generalization not possible with data-based approaches, consider that it might be possible to convey the general notion of a grid, so procedures might be iterated throughout the grid's elements.

The idea of a grid can be expressed purely through visual relations; you program the system to start at one object, then move right until you find the next, and so on, until there are no more objects to the right. You then return to the object at the beginning of the row, move down one object, then start moving to the right again; you keep scanning through each row until you can't move down any more.

Once you have the "idea" of a grid, you can apply it in a variety of applications. The same program can work whether operating on daily schedules in a calendar program, icons in a folder window, or tables in Netscape (see Figure 5).

For generalizing on a grid to work, the definition of "Move to the next object to the left" and "Move to the next object down" may have to be redefined for each application. But given the ability of the PBE system to do so, the developer can make real the user's perception that all grids are basically the same, despite the artificial barriers that independently programmed

Figure 5. Examples of grids in a calendar, the finder, and Netscape.



applications impose on this form of generalization.

Conclusions

We should ask ourselves a number of questions when exploring any new programming perspective, such as the one offered by visual generalization. How can it contribute in a way other perspectives do not? Existing techniques, such as Apple Events and OLE Automation, can sometimes provide powerful perspectives from which to build programs. But adding a new perspective to a system can significantly increase user interface complexity. If there is a large overlap in the range of information being processed by more than one application, the information's new form must provide some advantage—as demonstrated by the Triggers system and by VisMap.

What new user interface challenges are raised by these new perspective? What tools address these challenges? For example, the Triggers system has to accurately specify pixel patterns and distances that are quite cryptic when viewed out of context. It addresses this challenge through its built-in Desktop Blanket, a technique Potter devised for Triggers to allow direct manipulation widgets to float above the display's screen pixels. VisMap has to infer high-level features from low-level pixel data, addressing this challenge through a two-stage translation process. The first stage works bottom up to identify low-level features. The second works top down to infer high-level features from the low-level features.

Can complete software solutions be built within the visual generalization perspective as we've described it? Such solutions may indicate the potential for an elegant special-purpose system. Working from one perspective, it has the potential to produce a simple elegant interface; Triggers, for example, shows that a small set of functions can be used to automate nontrivial tasks. More work has to be done, however, to show that a significant user group can make use of this functionality.

How can we integrate our visual generalization perspective with other perspectives? Triggers addresses this question by showing how its Desktop Blanket can be added to a conventional programming language [8]. And VisMap has a textual interface that can be integrated easily with textual programming languages that use other techniques.

Our intuition about the design of a visual generalization system for PBE leans toward a broad-based approach that applies pixel-level operators, as in Triggers, where appropriate, but also generates higher-level information inferred from the pixel data, as in VisMap. If the user knows what a particular piece of information looks like on the screen but does not know how to describe it, a low-level pixel-based approach may be the best option. If displayed information needed by a program is not provided by formal techniques and its visual appearance is complicated, a high-level pixel-based approach may be the best solution. If the program needs efficient access to an application's large data structures, the user can choose a conventional programming technique, such as OLE Automation and Apple Events, assuming the application includes the necessary support.

Other issues in visual generalization in PBE applications include the granularity of event protocols, styles of interaction with the user, and parallelism [5]. Event granularity determines the level of abstraction at which a visual system interacts with an interface. For example, should mouse movements be included

in the information being exchanged? And if not all mouse movements, which ones are important enough to include? Moreover, issues of parallelism can enter the picture when the system and the user each try to manipulate the same interface object.

The opportunities and challenges of visual generalization represent a fruitful new direction for PBE in the future. It might turn out that when it comes to graphical interfaces, beauty may indeed be only skin deep. C

REFERENCES

1. Halbert, D. Programming by demonstration in the desktop metaphor. In *Watch What I Do: Programming by Demonstration,* A. Cypher, Ed. MIT Press, Cambridge, Mass., 1993.
2. Kurlander, D. and Bier, E. Graphical search and replace. In *Proceedings of ACM SIGGRAPH'88* (Atlanta, Aug. 1–5). ACM Press, New York, 1988, 113–120.
3. Kurlander, D. and Feiner, S. A history-based macro by example system. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Monterey, Calif., Nov. 15–18). ACM Press, New York, 1992, 99–106.
4. Lakin, F. Visual grammars for visual languages. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87),* (Seattle, July 12–17). AAAI Press, Menlo Park, Calif., 1987, 683–688.
5. Lieberman, H. Integrating user interface agents with conventional applications. *Knowl.-Based Syst. J. 11,* 1 (Sept. 1998), 15–24; see also *Proceedings of the ACM Conference on Intelligent User Interfaces* (San Francisco, Jan. Jan. 6–9). ACM Press, New York, 1998, 39–46.
6. Olsen, D. Interacting in chaos. *ACM Interact. 6,* 5 (Sept.–Oct. 1999), 42–54.
7. Potter, R. Triggers: Guiding automation with pixels to achieve data access. In *Watch What I Do: Programming by Demonstration,* A. Cypher, Ed. MIT Press, Cambridge, Mass., 1993.
8. Potter, R. *Pixel Data Access: Interprocess Communication in the User Interface for End-User Programming and Graphical Macros.* Ph.D. dissertation. University of Maryland Department of Computer Science, May 1999.
9. Yamamoto, K. A programming method of using GUI as API. *Transact. Info. Proc. Soc. Japan 39* (Dec. 1998), 26–33 (in Japanese).
10. Zettlemoyer, L. and St. Amant, R. A visual medium for programmatic control of interactive applications. In *Proceedings of ACM CHI'99 Human Factors in Computing Systems* (Pittsburgh, May 15–20). ACM Press, New York, 1999, 199–206.

ROBERT ST. AMANT (stamant@csc.ncsu.edu) is an assistant professor in the Computer Science Department at North Carolina State University in Raleigh, N.C.

HENRY LIEBERMAN (lieber@media.mit.edu) is a research scientist in the Media Laboratory at the Massachusetts Institute of Technology in Cambridge, Mass.

RICHARD POTTER (potter@cs.umd.edu) is a researcher in the Japan Science and Technology Corp. in Tokyo.

LUKE ZETTLEMOYER (lszettle@eos.ncsu.edu) is an undergraduate in the Computer Science Department of North Carolina State University in Raleigh, N.C.