

acm Transactions on Information Systems

KURT F. WENDT LIBRARY
COLLEGE

NOV 18 1992

UW-MADISON, WI 53706

Special Issue on User Interface Software and Technology

- 317 Editorial: User Interface Software and Technology
by Jock Mackinlay and Jim Rhyne

PRACTICE and EXPERIENCE

- 320 Lessons Learned from SUIT, the Simple User Interface Toolkit
by Randy Pausch, Matthew Conway, and Robert Deline

Research Contributions

- 345 A High-Level and Flexible Framework for Implementing Multiuser
User Interfaces
by Prasun Dewan and Rajiv Choudhary
- 381 EmbeddedButtons: Supporting Buttons in Documents
by Eric A. Bier
- 408 A General Framework for Bidirectional Translation between
Abstract and Pictorial Data
by Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and
Akinori Yonezawa
- 438 A Model for Input and Output of Multilingual Text in a Windowing
Environment
by Yutaka Kataoka, Masato Morisaki, Hiroshi Kuribayashi, and
Hiroyoshi Ohara



acm Transactions on Information Systems

Association for
Computing Machinery
1515 Broadway
New York, NY 10036
(212) 869-7440

Editor-in-Chief

Robert B. Allen

Bellcore/2A-367/445 South St./Morristown, NJ 07962-1910 USA/
+1-201-829-4315/rba@bellcore.com

Associate Editors

John M. Carroll

User Interface Institute/IBM Watson Research Center/Yorktown Heights, NY
10598 USA/+1-914-784-7733/carroll@watson.ibm.com

**Stavros
Christodoulakis**

Technical University of Crete/Dept. of Electronics & Computer Eng./P.O. Box
133/73 100 Chania, Crete, Greece/+30-821-64803/stavros@ced.tuc.gr

W. Bruce Croft

Computer Science Department/University of Massachusetts/Amherst, MA
01003 USA/+1-413-545-0463/croft@cs.umass.edu

Steven Feiner

Department of Computer Science/Columbia University/500 W. 120 Street/
New York, NY 10027 USA/+1-212-939-7083/feiner@cs.columbia.edu

Edward A. Fox

Department of Computer Science/Virginia Polytechnic Institute and State
University/Blacksburg, VA 24061 USA/+1-703-231-5113/
fox@vtopus.cs.vt.edu

Simon Gibbs

Centre Universitaire d'Informatique/Université of Genève/24, rue du Général-
Dufour/CH-1211 Genève 4 Switzerland/+41 22 705.7663/
gibbs@cui.unige.ch

Rob Kling

Department of Information and Computer Science/University of California at
Irvine/Irvine, CA 92717 USA/+1-714-856-5955/kling@ics.uci.edu

Wendy E. Mackay

Rank Xerox EuroPARC/61 Regent St./Cambridge CB2 1AB UK/
+44.223.341.502/mackay@europarc.xerox.com

Tom Malone

MIT/Center for Coordination Science/E53-333/Cambridge, MA 02139 USA/
+1-617-253-6843/malone@eagle.mit.edu

Brad A. Myers

School of Computer Science/Carnegie Mellon University/Pittsburgh, PA
15213-3890 USA/+1-412-268-5150/bam@cs.cmu.edu

John Mylopoulos

Department of Computer Science/University of Toronto/Toronto, Ontario
M5S 1A4 Canada/+1-416-978-5180/jm@ai.toronto.edu

Dan R. Olsen Jr.

Computer Science Department/Brigham Young University/Provo, UT 84602
USA/+1-801-378-2225/olsen@bunsen.cs.byu.edu

Peter Polson

Institute of Cognitive Science/University of Colorado at Boulder/Boulder, CO
80309-0345 USA/+1-303-492-5622/ppolson@clpr.colorado.edu

Norbert A. Streitz

GMD ISPI/Department of Integrated Publications and Information Systems
Institute/Dolivostrasse 15/D-6100 Darmstadt, Germany/+49-6151-869-919/
streitz@ darmstadt.gmd.dbp.de

C. J. van Rijsbergen

Computing Science Department/Glasgow University/Glasgow G12 8QQ
UK/+44-41.330.4463/keith@dcs.glasgow.ac.uk

Polle Zellweger

Xerox PARC/3333 Coyote Hill Rd./Palo Alto, CA 94304 USA/+1-415-812-
4426/zellweger.parc@xerox.com

Headquarters Quarterlies Staff

Mark Mandelbaum

Director of Publications

Carol A. Meyer

Associate Director of Publications

Roma Simon

Managing Editor, ACM Quarterlies

ACM Transactions on Information Systems (ISSN 1046-8188) is published quarterly by the Association for Computing Machinery, Inc., 1515 Broadway, New York, NY 10036. Second class postage paid at New York, NY 10001, and at additional mailing offices. Postmaster: Send address changes to Managing Editor, Transactions on Information Systems, ACM, 1515 Broadway, New York, NY 10036.

For subscription information, see inside back cover.

Copyright © 1992 by the Association for Computing Machinery, Inc. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage and credit to the source is given. Abstracting with credit is permitted. For other copying of articles that carry a code at the bottom of the first page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 27 Congress Street, Salem, MA 01970. For permission to republish write to the Director of Publications, Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

For submissions information, see inside back cover.

EmbeddedButtons: Supporting Buttons in Documents

ERIC A. BIER

Xerox Palo Alto Research Center

EmbeddedButtons is a library of routines and a runtime kernel that support the integration of buttons into document media, including text and graphics. Existing document editors can be modified to participate in this open architecture with the addition of a few simple routines. Unlike many button systems that insert special button objects into document media, this system supports turning existing document objects *into buttons*. As a consequence, buttons inherit all of the attributes of normal document objects, and the appearance of buttons can be edited using operations already familiar to users. Facilities are provided for linking buttons to application windows so that documents can serve as application control panels. Hence, user interface designers can lay out control panels using familiar document editors rather than special-purpose tools. Three classes of buttons have been implemented, including buttons that pop up a menu and buttons that store and display the value of a variable. New button classes, editors, and applications can be added at run time. Two editors, one for text and one for graphics, currently participate in the architecture.

Categories and Subject Descriptors: D.2.m [Software Engineering]: Miscellaneous—*rapid prototyping*; H.1.2 [Models and Principles]: User/Machine Systems—*human factors*; I.3.6 [Computer Graphics]: Methodology and Techniques—*interaction techniques*; I.7.1 [Text Processing]: Text Editing

General Terms: Human Factors

Additional Key Words and Phrases: Active documents, buttons, user interface layout

1. INTRODUCTION

Recent software applications allow buttons to be inserted into the text, graphics, and other media of electronic documents, where a button is a visible object that can be pressed, for instance, by clicking with a mouse, pointing with a touchscreen, or selecting with keyboard arrow keys. Because this integration of buttons and documents provides significant advantages to both users of and creators of software applications, we expect that buttons will be common in future electronic documents. While some existing editors already provide for buttons, there is a need for a general facility to make buttons

Author's address: Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304, email: bier.parc@Xerox.com.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 1046-8188/92/1000-0381 \$01.50

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992, Pages 381-407.

available in more editors and different media. In addition, buttons are often only partly integrated into current documents, making it necessary to navigate and edit buttons using different operations than are used to navigate and edit other parts of the document. To address these problems, we have designed and implemented `EmbeddedButtons`, a library of routines and a runtime kernel, which supports the addition of button behaviors to a potentially large variety of document editors.

This work was originally aimed at supporting the layout of user interface control panels by application designers in the Cedar programming environment [23]. As Cedar includes both a powerful text editor, *Tioga* [23, 24], and a powerful graphics editor, *Gargoyle* [21], we wanted to be able to lay out control panels using these existing tools, rather than build new tools from scratch. We reasoned that such a layout tool would require little learning, because our application designers were already familiar with *Tioga* and *Gargoyle*, and would require little coding, since we would be reusing the code in our editors instead of creating new code for laying out text and graphics.

We made two key design decisions: First, we decided to provide a library of routines that could be called by editors of a variety of different document media to support button integration in these editors. These routines provide the bulk of the functionality needed for buttons, so that only small changes are needed to the editors themselves. Second, we decided that rather than allowing special button objects to be inserted into document media, we would allow the media itself to be *turned into* buttons, by attaching button information to them. This decision was essential to reducing the amount of code we needed to change, in each editor, to integrate it into the system. This approach also supports new applications not supported when buttons are less integrated into documents.

1.1 Supporting Control Panels

In order to support the use of documents as control panels, `EmbeddedButtons` includes routines that link a set of buttons in a document to a particular application window or other application data structure. Once linked, these buttons can perform operations on this application and store and display application state. For example, Figure 1 portrays *DrawShapes*, a toy drawing tool. The upper rectangular region is a *text document*, acting as the control panel. The bottom region is a drawing area; clicking with a mouse in this region draws circles or boxes. *DrawShapes* calls `EmbeddedButtons` to link the control panel to the drawing area. The user can then use the control panel buttons to erase the picture, change the stroke width or shape that is used for object creation, and choose whether a circle or box will be drawn next.

1.2 Novel Applications

While originally intended as a user interface layout tool, `EmbeddedButtons` is a general facility for adding buttons to documents. We have added features to support a broad set of applications. While some of these applications are also supported by many document button systems, others are possible or straightforward only because we allow document objects to become buttons.

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

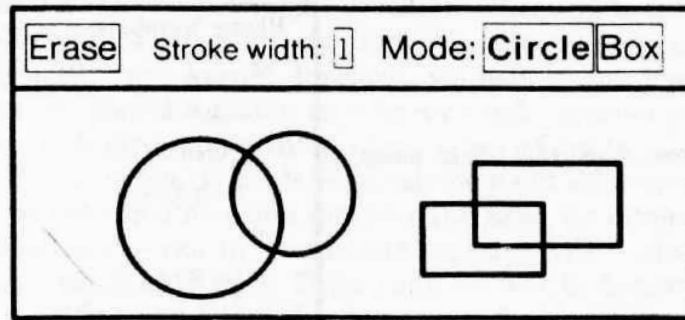


Fig. 1. The DrawShapes two-paned window.

One popular application is a simple file browser, based on a command shell. The command shell in Cedar is itself a Tioga document. Cedar commands take advantage of this fact by displaying some of their results in bold or italic for emphasis. Like these bold and italic characters, our buttons are normal text characters annotated with additional properties. Thus, programs that emit multifont text into a command shell can just as easily emit buttons. Taking advantage of this fact, we reimplemented the Cedar file listing program LS, which is much like the UNIXTM "ls" command, to emit a button next to each file name that opens the file when pressed.

As another application, textual search-and-replace and graphical search-and-replace [14] are used to make user interfaces quickly from existing documents. For instance, Figure 2 shows an on-line address book that has been turned into an active document by searching for strings that look like phone numbers and turning them into buttons that dial that number.

One unusual set of applications uses buttons to modify the behavior of a document editor. For example, say we wish to play with different colorings of the circles shown in Figure 3. If we had a graphics editor without a buttons capability, we would probably use editor commands to select each circle in turn and apply a color to it, taking several mouse clicks per color change. On the other hand, if the editor uses EmbeddedButtons, we could select all of the circles at once and turn them *en masse* into on/off buttons that are, say, black when on and white when off. Now, we need only one mouse click per color change. When we are done experimenting with colors, we can restore the normal (nonbutton) behavior of the circles, leaving a recolored picture that can once again be edited with standard drawing tool commands.

One final application of document objects that are buttons is to support unusual button shapes. The hit-testing methods used by each editor are used to determine whether or not a button is being pressed. For instance, each line in Figure 4 is a button in an illustration in the Gargoyle graphics editor. As part of the snap-dragging drawing technique, Gargoyle provides a gravity function that computes the nearest object that is within a small distance of the cursor [3]. This capability allows the user to press narrow buttons like those of Figure 4, which are used to provide the user with a selection of dash patterns and stroke widths.

Fig. 2. An address book that dials phone numbers.

Phone Numbers	
A	
Aardvark's Ark	765-3456
Lurch Adams	231-6755
Fred Astaire	345-8767
B	
Big Bird	234-8878
Yogi Bear	444-7777

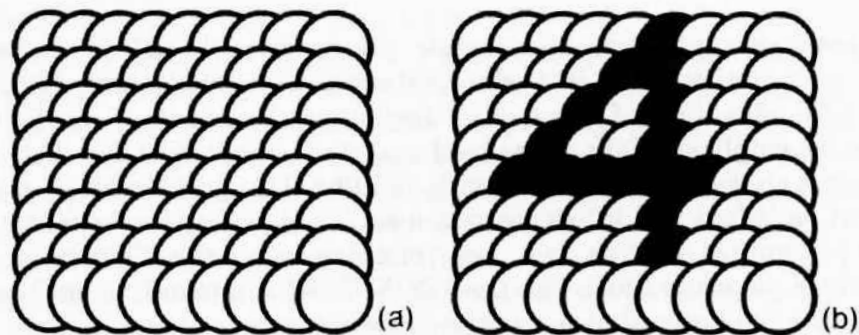


Fig. 3. (a) A pattern of circles; (b) the recolored pattern.



Fig. 4. Pressing narrow buttons using gravity. These buttons transfer a dash pattern or stroke width to the selected object on the right.

1.3 The EmbeddedButtons Prototype

EmbeddedButtons is implemented in Cedar, a programming language and environment [23] that runs on top of the SunOS UNIX-compatible operating system on Sun Microsystems SPARCstations and other computers. We have modified the Tioga text editor [23, 24], and the Gargoyle graphics editor [21] to participate in the EmbeddedButtons system.

Like other button systems, EmbeddedButtons provides a scripting language to represent the behavior of a button. Users can edit strings in our scripting language, called Poppy, at runtime to modify a button's behavior.

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

Our system uses Cedar's lightweight processes (threads) to allow our editors to directly call the EmbeddedButtons library routines and to share the state in the EmbeddedButtons kernel; however, we believe the basic form of the library and kernel would function in a heavyweight process environment as well. We have initially implemented three types of buttons: pop-up, multi-state, and radio, all of which are familiar elements of existing control panels. This set can be extended as users discover the need for other button types.

EmbeddedButtons works in documents together with other active document systems, including Active Tioga [24], in which document activity is triggered by scrolling or editing the document; Scripted Documents [28], in which activity is triggered at a document site when the system arrives at that site via a scripted tour; and CaminoReal [1], in which mathematical equations embedded in a document are reevaluated in response to document redisplay. This paper builds on two previous conference papers [4, 5].

1.4 Outline of the Paper

To fully describe the EmbeddedButtons system, we explain the interactions between users, button designers (who may also be users), the EmbeddedButtons kernel, document editors, and applications that use documents as control panels. Section 2 describes the buttons as seen by the user who presses them. Section 3 describes the tools used by button designers to turn objects into buttons and describes our scripting language, Poppy. Section 4 introduces the EmbeddedButtons kernel, our library of editor-independent routines that responds to button presses by coordinating the activities of editors, button classes, and applications. Section 5 describes how editors are modified to support buttons. Section 6 describes how application programs link themselves to a document to use it as a control panel. The final sections of this paper discuss EmbeddedButtons as a whole. Section 7 discusses some of our experiences with EmbeddedButtons including applications, problems, performance, and security. Section 8 describes related work, Section 9 presents our conclusions, and Section 10 presents our plans for future work.

2. USING BUTTONS

This section describes buttons as seen by a user. First, we describe how button activity can be turned on and off in a document. Then, we describe the three classes of buttons that we provide: pop-up, multistate, and radio.

2.1 Turning Buttons On and Off

Users need to be able to both edit a document containing buttons and press those buttons. Both Tioga and Gargoyle provide a way to turn button activity on and off within an entire document (using an on/off button in each control panel). When button activity is on, clicking on a button performs the button action. When button activity is off, buttons can be selected and edited like other objects.

In fact, expert button users also learn to edit buttons while button activity is turned on. Both Tioga and Gargoyle allow the user to hold down the mouse

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

button and move the mouse during a selection operation; as the mouse moves, different objects are selected. Thus, it is possible to select an active button by beginning a selection operation with the mouse cursor off of the button and then "sliding onto" the button. Once the button is selected, it can be edited with any operations that apply to selected objects.

2.2 Three Button Classes

We have found three button classes to be useful to support our applications: *pop-up*, *multistate*, and *radio*. This section presents the appearance and use of each of these button classes from the user's point of view.

2.2.1 Pop-up Buttons. When the user presses a mouse button over a document object that has been made into a pop-up button and holds it down for at least 400 milliseconds, a pop-up menu appears, with one of its entries centered on the cursor. When the user makes a selection, the pop-up button sends a command to the application indicated by the button. For instance, in Figure 5, the user has clicked on the text string "Alphabet." The pop-up menu offers to change the alphabet of the text string that is currently selected in some other document (not shown). The rectangle below the menu displays documentation for the currently selected menu entry. The pictures on the left edge of the menu indicate which of the three mouse buttons can be used as an accelerator for future button presses. If the user clicks the "Alphabet" button and releases it before 400 milliseconds has passed, no pop-up menu appears; the action is performed immediately. This pop-up menu facility has been previously described [21].

If a pop-up button can perform an action that is potentially dangerous, such as loading a file into an editor, the designer can specify that this button should be *guarded*. The user must click a guarded button twice to invoke its action. The two clicks must be at least 100 milliseconds apart (to prevent accidental double clicks) and at most 5 seconds apart; after 5 seconds the button returns to its initial guarded state. While guarded multistate and radio buttons would also be useful, they are not part of our current prototype.

2.2.2 Multistate Buttons. When pressed, a multistate button changes state (represented internally by a variable that belongs to the button) and updates its appearance to display its new state. For example, a textual button may change its font, the relative color of its text and background, or its text contents, as shown in Figure 6. In fact, any document editor operation can be used to achieve highlighting.

2.2.3 Radio Buttons. A radio button is a member of a group of buttons, each of which can be on or off and only one of which may be on at a time. They are equivalent to a constrained set of multistate buttons. As with multistate buttons, any document editor operation can be used to display the state change of a radio button. The radio buttons in a group need not be positioned near each other.

Figure 7 shows a group of three radio buttons, each shaped like an arrow. Clicking on one arrow makes it black and turns all of the other arrows white.

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

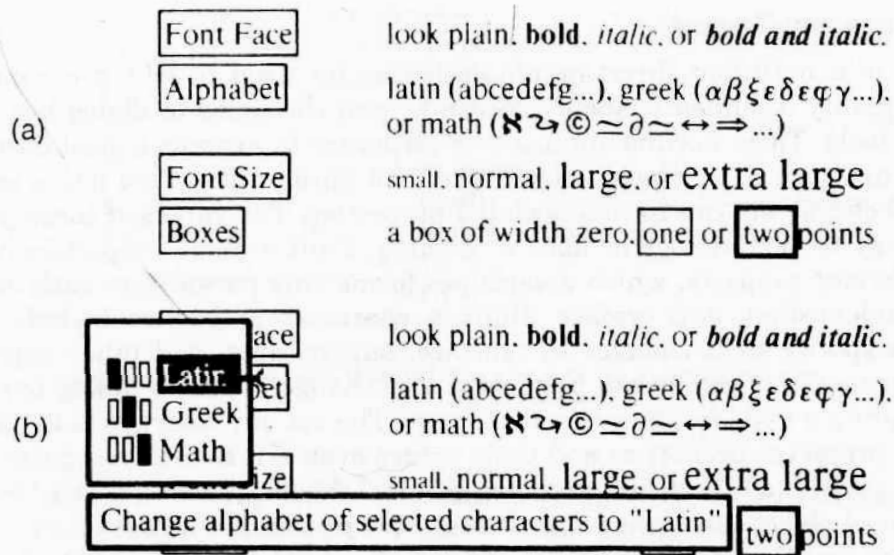


Fig. 5. (a) A pop-up button that performs text editor macros; (b) the menu and documentation string pop up over the "Alphabet" button.

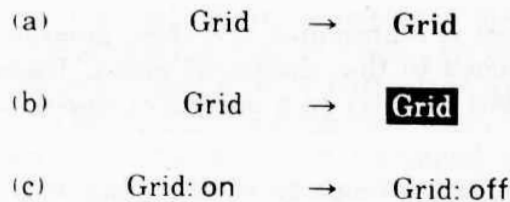


Fig. 6. Button highlighting. (a) Font change; (b) video reverse; (c) text change.

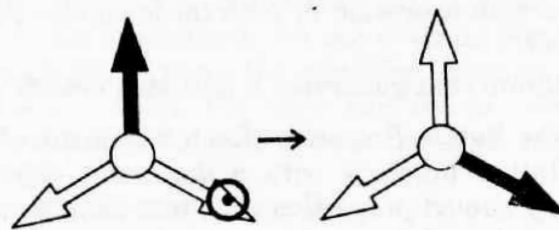


Fig. 7. Graphical radio buttons.

3. MAKING BUTTONS

Using our system, button designers can make standard types of buttons quickly using interactive tools. To make new kinds of buttons, designers can write button programs in the Poppy button language. This section describes: some properties of Tioga and Gargoyle that are relevant to designers, several interactive tools that make buttons, and the Poppy scripting language.

3.1 Tioga and Gargoyle

Tioga is a multifont direct-manipulation editor used to edit program text, high-quality documents, shell commands, and the fields of dialog boxes and other tools. Tioga documents use tree structure to express logical document structure such as sections, subsections, and paragraphs. Tree nodes or individual characters can have associated properties. The values of these properties may be text strings or data structures. Built-in node properties include the *Format* property, which determines formatting parameters such as leading, indentation, and typeface. Built-in character properties include *looks*, which specify local changes to typeface, superscripts, and other aspects of appearance. Properties can be viewed and changed using a dialog box or by expanding a node in place in a document. The set of properties is extensible; Tioga preserves properties and their values even if it doesn't recognize them.

Gargoyle is a structured graphics editor. Like Tioga objects, any Gargoyle graphical object can have a set of associated named properties.

Tioga and Gargoyle can both be driven by interpreted languages consisting of a sequence of editing commands and their arguments. By including these strings in buttons, we can take advantage of the full power of these editors when we display button feedback.

3.2 Button Creation Tools

The button behavior of a document object is represented in a short program, called a *Button property*, that is attached to that document object. Users create new buttons (i.e., attach a Button property to a document object) in four ways. They can

- (1) copy an existing button and edit its Button property or contents;
- (2) select the objects that they want to turn into buttons and click on a button-making button (called a *buttonizer*);
- (3) use a global search operation to turn those objects that match a pattern into buttons; or
- (4) execute a program that generates a buttoned document.

3.2.1 Editing the Button Property. Each document editor provides a way to associate a Button property with a document object. Since Tioga can associate arbitrary named properties with text strings, we implemented the Button property on top of this mechanism. Tioga also provides a command that expands a node (e.g., a paragraph) in place in a document to reveal the node's properties and allow the user to edit them. For example, Figure 8a shows a short Tioga node with the text *Here* in italics and button, which is a button, in a box. Figure 8b shows Tioga's expanded form of this node. For brevity, the actual value of the Button property is not shown in Figure 8. The user can change the Button property (and other Tioga properties) by editing this expanded textual form and then causing Tioga to reconstruct the revised node.

As part of the EmbeddedButtons project, we added a property mechanism, like Tioga's, to Gargoyle. Because some Gargoyle objects are chains of line

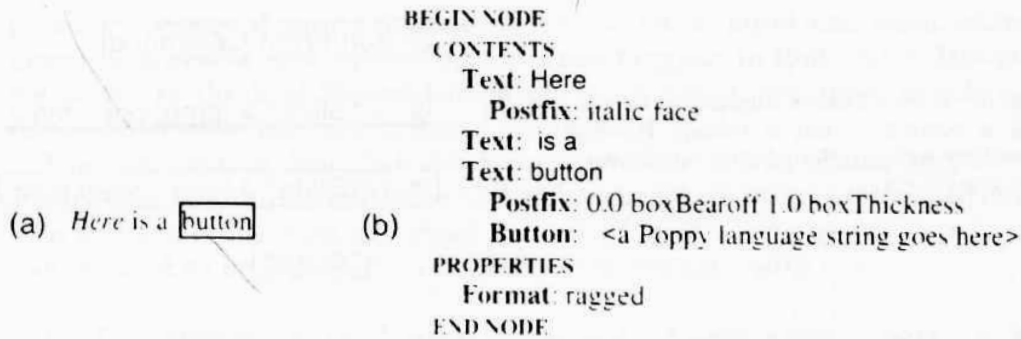


Fig. 8. (a) A Tioga node that includes a button; (b) its expanded form.

segments, it would be possible to associate properties with individual segments. However, we guessed that users would rarely want individual segments to have different button behaviors, so we allow only entire shapes, such as circles, boxes, and polygons to be buttons. Gargoyle users edit the Button property in a dialog box and then apply the revised property to one or more selected shapes.

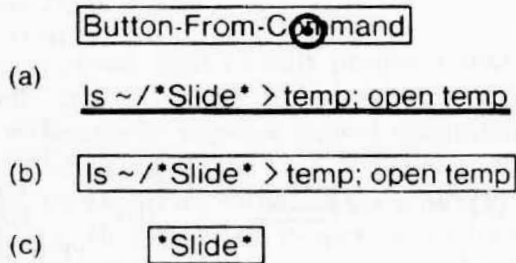
3.2.2 Buttonizer Buttons. Ready-to-use buttonizer buttons are included in the EmbeddedButtons reference manual. These buttons can be pressed *in situ* in the manual, or users can copy them into their own documents and press the copies. One buttonizer, Button-From-Command, turns a selected text string into a button that sends that string to a command shell. In Figure 9a, the user selects a command string and clicks on the Button-From-Command button. Clicking on the resulting button (Figure 9b) will execute this command. The command is executed in a fresh shell whose working directory is initially set to root and whose output and error streams are directed to a typescript window called the System Script. Most buttons use full path names for files or explicitly set the working directory.

Because the Button property on the resulting command button contains a copy of the entire command, the user can delete some of the button's characters to produce a smaller button with the same function, as shown in Figure 9c.

Users can also create their own buttonizers, by copying and editing existing buttonizers. Internally buttonizers are like normal buttons except they have two Poppy strings—the Button property and the *ButtonLiteral* property. The Button property describes the behavior of the buttonizer, namely to send a message to a button-making program when the buttonizer is pressed. The button-making program copies the ButtonLiteral property of the buttonizer to be the Button property of the new button. Thus, by editing the ButtonLiteral property of a buttonizer, the user can determine the type of button that the buttonizer creates.

3.2.3 Using Global Search. Tioga allows the user to search for character string patterns and apply Tioga operations on each matching string. This

Fig. 9. Button-From-Command. (a) Turning a command into a button; (b) the resulting button; (c) a smaller button with the same function.



capability can be used to make many buttons at once. For instance, in the address book of Figure 2, each character string that matches the pattern of a phone number has been turned into a button that dials that number. Likewise, graphical search and replace [14] could be used in an illustration to selectively turn shapes into buttons.

3.2.4 Generating Active Documents By Program. The Cedar programming environment includes several programs that generate structured text. These programs can be enhanced to generate buttons as well. As described in Section 1.2, we have modified the “ls” file-listing program, to add a button next to each file name that starts a text editor on that file when pressed.

3.3 The Button Language

The content of each Button property is specified in our button language, Poppy. Poppy is not a single language, but a set of syntactic conventions that allow button control constructs and the interpreted languages of editors and command handlers to be combined in a single string. Poppy achieves several goals: it constructs command strings to be sent to editors and applications in response to button clicks, it represents the values of variables that can be queried and set, and it determines the order in which to send command strings to editors and applications when the button is pressed or when the button’s value is set. Command strings are computed each time a button is pressed, so they may change with time.

One weakness of Poppy is that its strings are often editor-dependent. Because Poppy relies on the input languages currently provided by each editor, designers cannot, in general, take a Poppy string from a button in one editor and apply it to a button in another. The problem would be reduced if our editors relied on a common input language syntax, such as Ousterhout’s *tcl* [19] and if they provided the same names for related operations, e.g., the “ChangeColor” command might perform the same function in all editors. In this case, Poppy strings would be less editor-dependent and more readable. In its current form, Poppy provides the user with great flexibility—any editor operation can be used to perform button feedback—at the cost of a heterogeneous syntax.

Another potential problem is that Poppy’s flexibility can be misused. Different users can adopt different conventions for how buttons should be displayed and what feedback should be presented when they are pressed. To

prevent a chaos of user interface styles, we could build into each editor a description of how each type of button should appear in that editor. However, we prefer to think of EmbeddedButtons as a foundation upon which more constrained user interface systems can be built.

The next section describes the keyword/value and LISP-like syntax that Poppy uses to group together its nested languages and to represent variables. The subsequent section describes the set of built-in Poppy commands that can be used to produce button behavior that changes with time.

3.3.1 A Framework for Nested Languages. Poppy strings consist of keyword/value pairs. Each keyword is followed by a colon. Each value is either a single word or a string surrounded by matching parentheses, for literals, or matching angle brackets, for interpreted commands. Except for the keyword "Class", all keywords are optional.

As an example, the Button property in Figure 10 describes a pop-up button in a Tioga document. The "Class" keyword/value pair identifies this button's class as PopUpButton. The "CommandHandler" pair names the application that will carry out this button's action. The "Menu" pair describes the three-entry menu this button displays when pressed (see Figure 10b); the first menu entry is labelled "Tioga." When the cursor passes over this entry, the string "Open Tioga manual" is displayed below the menu as documentation. If this entry is chosen, the command "open TiogaDoc" is sent to this button's command handler, `CommandTool`. The "IO" and "Gargoyle" menu entries work similarly. The "Feedback" pair describes how the appearance of the button or the cursor will change in response to various state changes; in this example, the cursor takes on a bullseye pattern when the cursor passes over the button.

In Figure 10a, different typefaces identify sublanguages. Plain-faced characters are part of the Poppy language proper. Italicized strings are interpreted by the PopUpButton class. Bold strings are interpreted by our command shell, the `CommandTool`.

As another example, Figure 11 describes a multistate button. This button has a local variable, *On*, that can be either TRUE or FALSE. Applications can use this button's name, `GridOption`, to get or set *On*. If this button had no name, *On*'s value would be private. This button has two feedback rules. When *On* becomes TRUE, the button sends the string "(BeginButton Bold ApplyLook EndButton)" to the Tioga editor. This string specifies that the button should become bold-faced. When *On* becomes FALSE, the bold looks are removed.

The type of each Poppy variable must be declared. Legal types include boolean, integer, floating point, enumerated type, or string. Poppy uses its knowledge of these types to determine, if possible, what the next higher or next lower value of a variable should be when neither the button nor the button's command handler specify the next value explicitly (e.g., when the user wants to test a button that is not yet linked to its intended application).

Programs write Poppy strings as well as read them. For instance, if the value of *On* in Figure 11's button changes from TRUE to FALSE and its

```

Class: PopUpButton
CommandHandler: CommandTool
Menu: (
  ( "open TiogaDoc" "Tioga" "Open Tioga manual")
  ( "open IODoc" "IO" "Open IO manual")
  ( "open GargoyleDoc" "Gargoyle"
    "Open Gargoyle manual" ) )
Feedback: ((MouseMoved <SetCursor bullseye>))

```

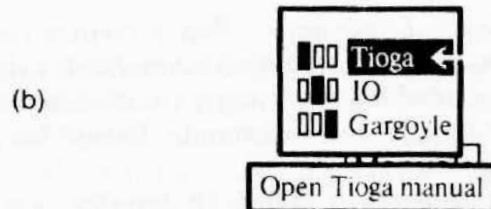


Fig. 10. A pop-up button. (a) Its Button property; (b) the menu it pops up, and a documentation string that appears below the menu.

```

Class: MultiStateButton
Variables: (On: BOOLEAN = TRUE)
Name: GridOption
Feedback: (
  ((On TRUE) (BeginButton Bold ApplyLook EndButton))
  ((On FALSE) (BeginButton ClearLooks EndButton)) )

```

(a)

Grid ↔ Grid

(b)

Fig. 11. A multistate button. (a) Its Button property; (b) clicking on the button changes its typeface.

document is saved to disk, then its Button property is updated to reflect the new value. The simplicity of Poppy makes it easy to generate this new Button property string.

While we use Poppy for most of our buttons, other languages may be used as well. In our current buttons, the Button property begins with the string "Poppy1" which identifies Poppy version 1 as the language used in this button. Other values of this initial string can be used to introduce other languages.

3.3.2 Poppy Commands. Substrings of a Button property can be constant or they can be computed when the button is pressed. Computed operations are invoked by putting the command name and arguments in angle brackets (e.g., <Command argument1 argument2>). These commands can be nested to any depth.

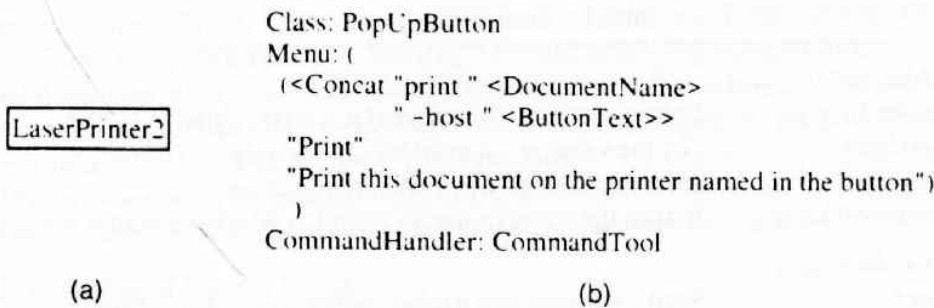


Fig. 12. (a) A button that computes its command string; (b) its button property.

Figure 12 shows a button that prints the document in which it is located. The `Concat` command concatenates any number of string arguments into a single string. `DocumentName` returns the name of the document in which this button is located. `ButtonText` returns the text of the button itself, which in this case is "LaserPrinter2." By editing the text of the button, the user can determine which printer the document will be sent to. Note that the string "<ButtonText>" could be replaced by the name of a particular printer to produce a print button that sends a document to that printer, independent of the text of the button.

The commands that we have implemented for our applications serve six different purposes: accessing button fields, altering flow of control, manipulating text strings, manipulating file names, computing arithmetic operations, and controlling the cursor. The specific commands we have implemented are summarized in Table I. The set of Poppy commands can be extended by writing new routines and registering their names with the EmbeddedButtons kernel, described next.

4. THE EMBEDDEDBUTTONS KERNEL

Most of the code supporting button behavior is part of a core of routines called the EmbeddedButtons kernel. As shown in Figure 13, this kernel communicates bidirectionally with editors and command-handling applications. The kernel also relies on a body of code called ButtonClasses, which implements the different behaviors of pop-up, multistate, and radio buttons.

Editors, button classes, and command handlers establish their communication channel with the EmbeddedButtons by registering themselves with the kernel at runtime. Section 4.1 describes this process of registration. Once these communication channels are established, the whole buttons system can respond to button presses, as described in Section 4.2.

4.1 Dynamic Registration

The EmbeddedButtons architecture allows button classes and command handlers to join the system dynamically. When a button class or command handler begins to run, it calls a kernel registration routine to associate its name with a routine or set of routines that the kernel can call later. For

Table I. Basic Poppy Commands

Button access:	
<i>ButtonText</i>	Return the text of a button, if it is a text object
<i>GetValue</i>	Get the value of a named button variable
<i>SetValue</i>	Alter the value of a named button variable
<i>DocumentName</i>	Return the name of the document in which the button is located.
Control constructs:	
<i>Send</i>	Send a message to a named application, instead of this button's CommandHandler
<i>Select</i>	A case statement, e.g., <Select <GetValue> TRUE 5 FALSE 7.3> returns 5 if this button's main variable is TRUE
<i>Do</i>	Evaluates its arguments left to right, returning nothing.
<i>Sequence</i>	Like Do but, returns the value of its last argument.
String manipulation:	
<i>Concat</i>	Concatenate all arguments into a single string.
File name:	
<i>DirectoryPart</i>	Given /dir/subdir1/file.text, returns /dir/subdir1/
<i>ShortName</i>	Given /dir/subdir1/file.text, returns file.text
<i>BaseName</i>	Given /dir/subdir1/file.text, returns file
Arithmetic operators:	
<i>+ - * /</i>	Standard arithmetic functions
<i>ABS, MAX, and MIN</i>	Absolute value, and maximum and minimum values
Cursor control:	
<i>SetCursor</i>	Set the cursor to a named pattern

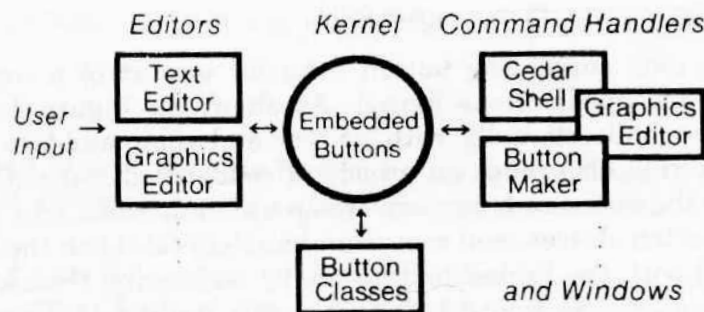


Fig. 13. The EmbeddedButtons kernel acts as a go-between.

example, the pop-up button class implementation calls a kernel routine to associate the name "PopUpButton" with the routines that are specific to pop-up buttons. Likewise, the Cedar shell command handler registers the name "CommandTool" with its command handling routine. The last-registered implementation of a button class or application supercedes all

previous ones. Thus, button software and button applications can be tested conveniently by dynamically loading revised software components.

Editors can also join the system dynamically. To allow the kernel to call editors, a pointer to editor-specific routines is included as part of each active document; Section 5 describes these routines.

When they wish to initiate communication with the kernel, editors, button classes, and command handlers directly call kernel routines.

4.2 Handling a Button Press

While different button classes have distinct responses to button presses, most button presses are handled in five steps.

- (1) The editor notifies the kernel of the button press, passing the Button property to the kernel. The kernel parses this property, identifies the class of the button and forwards the event to that class.
- (2) The button class decides which initial feedback actions should be performed and sends a message, via the kernel, asking the button's editor to perform them.
- (3) The button class determines what command string to send (e.g., depending on which menu entry of a pop-up button is selected) and forwards it, via the kernel, to the command handler.
- (4) If the button has local variables, the command handler may query or modify these variables (see Section 6). If there is no command handler, the button class increments or decrements the variable (depending on which mouse button was used) to the next legal value of its declared type.
- (5) The button class sends a message, via the kernel, asking the button's editor to perform any final feedback actions.

5. INTEGRATING EDITORS WITH EMBEDDEDBUTTONS

While most EmbeddedButtons functionality is implemented in the kernel, editors must be modified in four ways to participate in the architecture. They must: provide a mode that determines if button activity is on or off in a document, route mouse events to the kernel when the cursor is over a button, implement five routines for the kernel to call, and call the kernel when documents containing buttons are read from or written to a file.

5.1 Routing Button Events to the Kernel

As described in Section 2.1, we have added a mode to our editors that determines whether button activity is on or off in a given document. When an editor is not active, mouse motion and mouse button events performed over buttons are treated as normal editor operations; otherwise, such events are delivered to EmbeddedButtons. Each editor can use its own method for determining that a button has been pressed or entered by the mouse cursor. In Tioga, for instance, button events occur when the cursor is over a character that has a Button property. In Gargoyle, a button event occurs when the cursor is over or within a specified distance from a shape that has a Button property.

5.2 Five New Methods

Each class of editor must implement the five simple methods declared in the Cedar language in Table II. For readers not familiar with Cedar, parameter names have initial lower-case, followed by a colon and then their type names in initial upper-case. Values returned by a procedure are declared following the word RETURNS. Procedures are typed and may be passed as parameters.

MapButtons enumerates all of the objects in a document that have a Button property, calling a procedure on each one; this routine is used, for instance, when the kernel needs to find all buttons of a given name. Because finding buttons only requires identifying the Button property on document objects, the routine is easy to implement.

GetProperty returns the Button property or another property of the document object. The returned value is usually a string. SetProperty can modify the Button property or other properties of the buttonized document object. If *edited* is TRUE, the editor will mark the document as changed, and, at the end of a session, the user will be asked to save the changes or confirm losing them. SetProperty is used by buttonizers to create new buttons.

Feedback takes a string *do*, which is usually the value of the "Feedback:" field of a button, and performs the requested document edits.

GetText returns the characters that make up a button, if the button is a text string, or the empty string otherwise. It is used, for instance, to implement the ButtonText Poppy command, described in Section 3.3.

5.3 Saving and Restoring Buttons

Because parsing the Button properties of all of the buttons in a document can take seconds or more, we do not parse all buttons every time a document is loaded from disk. Instead, when a document is read from disk, its editor calls the kernel to convert each Button property to a runtime data structure, the *button record*, that includes the original Button property string, storage for a parsed version of the string (which will be generated when needed), and a flag describing whether or not the Button property has been modified during the session. Allocating this record takes little time because no parsing is done.

The Button property of a button is parsed when that button is first pressed, or when GetValue or SetValue is first called on its name. As parsing the Button property of a single button takes under 100 milliseconds on our system (see Section 7.3), response remains interactive. In addition, all buttons are partially parsed when GetValue or SetValue is performed, to determine their names. When a button is fully parsed, the kernel creates a symbol table that maps each Poppy keyword to its parsed value. This symbol table is stored in the button record for use the next time the button is pressed. If a button's variables change value, its symbol table is updated, and its modified flag is set.

When the document is stored to disk, the kernel is called again to turn this data structure back into a text string. If the Button property has not changed, the kernel writes out the original string. Otherwise, it generates a new string from the modified symbol table.

Table II. The Protocol for Editors

MapButtons: PROCEDURE [d: ActiveDoc, p: PROCEDURE [b: Button]]:

GetProperty: PROCEDURE [name: String, b: Button, d: ActiveDoc]
 RETURNS [prop: POINTER];

SetProperty: PROCEDURE [name: String, b: Button, d: ActiveDoc,
 prop: POINTER, edited: BOOLEAN];

Feedback: PROCEDURE [b: Button, d: ActiveDoc, do: String];

GetText: PROCEDURE [b: Button, d: ActiveDoc] RETURNS [text: String];

6. USING DOCUMENTS AS CONTROL PANELS

EmbeddedButtons supports the use of documents as control panels. This section describes how document control panels are linked to a particular instance of an application, and how applications can get and set application state values that are being stored and displayed by a document. Finally, the Gargoyle control panel is presented as an example.

6.1 Linking Documents to Applications and Windows

When a button is pressed, it sends an event to the command handler specified in the `CommandHandler` field of the `Button` property. This command handler may be a registered system-wide application, or it may be a particular application window or data structure.

Some applications register their names with EmbeddedButtons when they are loaded. These applications are available, system-wide, to any button that names them. For instance, many of our example buttons use `CommandHandler: CommandTool` to send commands to the Cedar shell. Other registered applications include `MessageWindow` and `Typescript`, which print out the button's command on a message line or typescript, respectively, for the user to read. The `Tioga` and `Gargoyle` command handlers interpret the button command as a text editor macro or graphics editor macro respectively.

Other applications only wish to be visible to a specific set of buttons. An application can explicitly link buttons to one of its windows or data structures by calling a kernel routine, `Link`. Each participating button names the kind of window or data structure it is to be linked to in its "`CommandHandler:`" field. For instance, for an application with two subwindows, named "`LeftPane`" and "`RightPane`", some buttons might name "`LeftPane`" as their `CommandHandler` and others "`RightPane`". Our example application would call `Link` twice, once to link the "`LeftPane`" buttons to the left pane, and once to link the "`RightPane`" buttons to the right pane.

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

The Link routine does not affect the buttons. Instead, it modifies a table in the kernel to record that all buttons in a particular document that name a particular CommandHandler should send their events to the specified application. One advantage of using a table to represent the link is that a new button added to a document after Link has been called will send events to the linked application if that button names the application as its CommandHandler; the application need not explicitly link the new button. This allows users to experiment with new control panel designs for an application without reloading that application after each change.

6.2 Getting and Setting Button Values

When documents are used as control panels, applications can use multistate and radio buttons to store and display application state. To read or write the value of a button variable, the application calls the kernel, providing the button name, variable name, and document. The application does not need to know where the button is placed in the document or what the class of the button is (e.g., multistate versus radio button). For instance, in the drawing tool of Figure 1, the stroke width button, , and the shape mode buttons,  and , store and display application state. The drawing tool can query and modify these values by calling the procedures:

```
GetValue: PROCEDURE [buttonName: String, variableName: String, d: ActiveDoc]
    RETURNS [value: String];
SetValue: PROCEDURE [buttonName: String, variableName: String, d: ActiveDoc,
    value: String];
```

GetValue searches through a document for a button with the given name. If it finds one, it looks in that button for a variable of the given name, and returns its value as a text string, if it exists. Routines are available to convert strings into standard types such as booleans and integers. SetValue searches through a document to find *all buttons* with the given name. For each such button, it sets the named variable to have the specified value. It is not possible to get or set the value of unnamed buttons.

When the user clicks on a multistate or radio button, it does not change state immediately, but informs the application of the state change request, so the change can be serialized with other application events; this synchronization is necessary because the lightweight processes used to distribute mouse events may be different from the lightweight process used by the application to execute its commands. For instance, in Figure 1, if the user selects Circle mode, specifies several circles, selects Box mode, and specifies a box, the application should not allow the shape mode to change from Circle to Box until all of the circles have been drawn, even if the application is delayed in processing the drawing events.

Since applications may call GetValue many times, its performance is important. EmbeddedButtons maintains a hash table of button values that have been recently accessed, keyed by document and button name. The first time a value is accessed, EmbeddedButtons searches the document for its

value and stores that value in the hash table. If `GetValue` is called on this value again, it is found in the hash table. `SetValue` updates both the hash table and all of the buttons. Thus, while `SetValue` runs in time linear in the number of buttons, subsequent calls to `GetValue` run in constant time. This design works well because values are generally queried much more frequently than they are changed.

When several buttons have the same name, `EmbeddedButtons` updates all such buttons in parallel. This mechanism can be used to create the effect of a single button that appears at several different document locations. For instance, a multistate button that displays the current value of a mode can be copied to several locations in a document, allowing a user to see the mode from all of those document locations.

As a complete example of linking, Figure 14 shows the Button property of the `DrawShapes` stroke width button. To use this button, `DrawShapes` links the name “`DrawShapes`” to its drawing area. Then, when a user clicks the left mouse button, this button sends the command “`Thicker`” to the `DrawShapes` application; the middle mouse button results in the command “`Thinner`.” The application responds to a “`Thicker`” command by calling `GetValue`["`Stroke-Width`", "`Width`", `d`] to get the current value of variable `Width`, adds 1 to it, and calls `SetValue` to set the new value. As feedback, when `Width` changes value, the button takes the new value as a text string, returned by `GetValue`["`Width`"], and displays this number as the new button text (e.g., the button [1] becomes [2]).

6.3 An Example: The Gargoyle Control Panel

We have produced two document control panels for the Gargoyle graphics editor itself. Our first control panel, shown in Figure 15a is a Tioga document. Our second control panel, Figure 15b, is a Gargoyle picture. A control panel can be linked to a particular Gargoyle window or can be dynamically linked to whichever Gargoyle window the user last clicked the mouse on. In fact, a picture control panel can be linked to the window occupied by itself and can edit itself (e.g., change its own colors, rotate itself, etc.).

7. DISCUSSION

`EmbeddedButtons` became available for use in September of 1989. Since then, it has been used in a number of applications, some of which are described in Section 7.1. We have encountered several problems with using documents as user interfaces, most of them due to assumptions built into our document editors about how documents will be read and used. These are described in Section 7.2. Some performance measurements of `EmbeddedButtons` are described in Section 7.3. Finally, Section 7.4 describes security issues.

7.1 Applications

Applications to date include active online documentation, file browsers, programming tools, and several control panels. For instance, buttons have been used in documentation files to create windows that display related documents

```

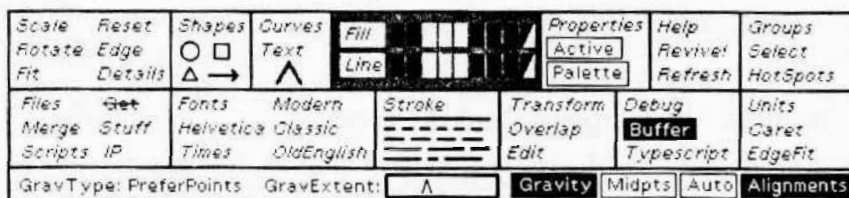
Class: MultiStateButton
Menu: (
  ("Thicker" "One Thicker" "Add 1 to width")
  ("Thinner" "One Thinner" "Subtract 1 from width"))
CommandHandler: DrawShapes
Name: StrokeWidth
Variables: (Width: INT = 1)
Feedback: (
  ((Width) (BeginButton <GetValue "Width"> ReplaceText EndButton)))

```

Fig. 14. Button property for the StrokeWidth button.

Scale Rotate Fit Reset Edge Prev CenterFit Details Refresh **Active** Props
 Clear Restore Get Store Save Merge Stuff IP Script Help Reviver Typescript
 Caret Transform Overlap Curves Shapes Text Fonts Groups EdgeFit Debug **Palette**
 Stroke Color Fill Edit Select HotSpots Units **ScreenStyle** SpecifiedFonts **Buffer**
GravType PreferPoints GravExtent: 25.0 **Gravity** Midpts **Auto** **Alignments**
 Fill colors: ■ ■ ■ ■ ■ Line colors: ■ ■ ■ ■ ■

(a)



(b)

Fig. 15. (a) A control panel for the Gargoyles graphics editor that is a Tioga document; (b) a second control panel that is itself a Gargoyles picture.

and to start application programs that are described in that file. One online help document created by an administrative assistant includes buttons that open forms and graphics templates, start up document editing tools, perform text editing macros, and congratulate the reader on completing the task. The author of this paper makes frequent use of a document whose buttons open commonly used files and forms for creating common types of electronic mail messages and office documents.

Buttons that encapsulate a particular textual search-and-replace operation have been used for programming tasks, including updating a set of program files to use a new name for a procedure or variable.

The new Gargoyles control panel, shown in Figure 15, is our most ambitious control panel to date. It is a delight to be able to edit this control panel as graphics, and several of us have made our own versions of it to satisfy our personal aesthetics and to try out different groupings of functions.

7.2 Problems Encountered

This section describes four problems that we encountered using documents as control panels.

Document editors do not necessarily provide adequate facilities for reformatting control panels when windows are resized. For instance, because Gargoyle keeps the center of an illustration in the center of its window when the window is resized, the buttons on the left edge of a control panel that is a Gargoyle picture will disappear off the edge of a window that becomes small; we plan to modify Gargoyle to allow a choice of options for window resizing.

Some of the checking performed by document editors to make sure that edits have been saved is not appropriate for control panels. EmbeddedButtons “edits” a document to provide feedback when buttons are pressed. However, the user is unlikely to be interested in saving these edits when he or she destroys the application window in which the control panel was displayed. The warning can be avoided either by programming the editor to ignore certain edits or by suppressing the warning for documents that are in use as control panels.

In a click-to-type window system, keyboard input is directed at the last window in which the user clicked. If the edits performed by EmbeddedButtons are interpreted as clicks, the user’s keyboard events may go to the wrong window. Currently, we work around this problem by having our editors replace the keyboard focus when the button feedback is finished.

Editors do not always update the screen at the right granularity for control panels. For instance, Gargoyle generally updates the screen after each command. However, when EmbeddedButtons asks Gargoyle to perform a set of commands as button feedback (e.g., to change a button from black on a white background to white on a black background), we only want the screen to show the final result, rather than all of the intermediate steps. We have modified Gargoyle to optionally suppress screen update during a series of commands.

7.3 Performance

The performance of buttons is quite satisfactory. In some cases, we modified our editors to improve performance. This section describes some of our performance results on a SUN Microsystems SPARCstation 2.

One expensive operation on a button is to parse the Button property and create the run-time button data structures. This occurs the first time the cursor passes over a button. For the buttons in the Gargoyle control panel of Figure 15a, we tested performance by loading a fresh copy of the control panel, moving the cursor over each button in turn, and repeating several times. This test reveals that instantiating these buttons takes an average of 41 milliseconds per button for each of its 51 pop-up buttons, and 14 milliseconds for its 9 multistate buttons. Gargoyle’s pop-up buttons have large menus. The pop-up buttons that we use daily to open commonly-used files average 15 milliseconds to instantiate. All of these times are satisfactory for interactive use.

Because we do not parse the Button property when button-containing files are loaded into a document editor, buttons cause little delay during loading.

For instance, the total delay in loading caused by all of the buttons in the Gargoyle control panel of Figure 15a is 15 milliseconds.

Likewise, when the file is saved again to disk, most buttons write their original button property. The multistate buttons write a new value if they have been pressed. For the Gargoyle control panel, if all the multistate buttons have been pressed, the total delay to convert all of the button data structures back to text strings is currently 33 milliseconds.

7.4 Security

Like many systems that integrate buttons into documents, our system makes it easy for users to share buttons, either by sharing a document on a file server or by sending a document through electronic mail. If these buttons are given access to arbitrary system functions, users could create buttons, either accidentally or intentionally, that have destructive consequences.

We have not addressed this problem in our prototype system; all of our users register the CommandTool command handler, which provides access to general shell commands. However, because our button language provides access to any number of command interpreters, we could implement a more secure system as follows: First, we could implement command interpreters that are more restrictive than a general shell and encourage button designers to use these interpreters instead of general shells where possible. Then, users concerned with security would load only the restricted interpreters. For these users, buttons calling the general shell would fail to operate, but other buttons would continue to work.

8. RELATED WORK

EmbeddedButtons cooperates with document editors and applications to provide button functionality both to people who create and read documents and to user interface designers. Hence, it builds on work in many disciplines, including document editors, user interface toolkits, hypertext systems, button packages, and open software systems.

To cooperate with EmbeddedButtons, editors must be able to store the Button property string on their objects. In addition, these editors must be able to display documents on the screen and detect button presses from a pointing device. Thus, EmbeddedButtons relies on modern interactive document editors that allow attributes to be attached to document elements. While editors originally used attributes largely to represent appearance parameters such as font and color changes, several modern editors now provide for arbitrary attributes. In addition, because such editors provide appearance parameters, EmbeddedButtons can use them as feedback when buttons are pressed. Document editors with these features include Tioga [23, 24], Star [11], Grif and **pedtnt** [9], Andrew [17], FrameMaker, Interleaf [7], and others.

EmbeddedButtons makes it possible to lay out control panels using document editors. In this arena, it builds on and competes with existing user interface layout tools, including NextStep [25], FormsVBT [2], Cardelli's user

interface editor [6], and SUIT [20]. Like these tools, EmbeddedButtons allows the user to modify a control panel and then test out the new design without recompilation delays. These tools are, in general, superior to our document editors in the time required to create new user interface objects or to switch among a set of standard ways to present a value; SUIT, for instance, can show a bounded value as a slider, a gauge, a scroll bar, or a pie chart, cycling among these presentations in a single keystroke [20]. They are also superior in that they use constraints to describe how the control panel should be reformatted if a window is resized; however, constraint-based document editors remove this difference. On the other hand, EmbeddedButtons control panels can be included in regular documents and edited using familiar document operations. Furthermore, document editors are the tool of choice to create interfaces that require integrating well-formatted text and arbitrary graphics with push-buttons and state indicators.

Documents containing our buttons can be generated by programs as well as by users. Previous systems have used computer-generated documents as user interfaces. For example, some applications of the Symbolics Presentation Manager [16] generate elaborate documents that serve as control panels. This saves programmer time, because the layout of the resulting control panels is performed by document formatting code. Other systems, like DEMO [26] allow the user to lay out and demonstrate user interfaces using a drawing editor. EmbeddedButtons differs from Presentation Manager and DEMO in that control panel documents are indistinguishable from other documents allowing them to be edited, copied, mailed, and so on.

Naturally, EmbeddedButtons builds on the many existing systems that integrate buttons with one or more document media and shares many of their advantages. For example, the Interactive Graphical Documents and Hyper-ties systems [8, 13] demonstrate that buttons embedded in text or graphics are an improvement over linear menus of buttons, because the surrounding parts of the document serve as context for the buttons. The popular HyperCard system [10] demonstrates that by learning a single user interface, users can position both information and buttons. Xerox Lisp Buttons [15], CUSP buttons [27], and Andrew buttons [17] show the utility of buttons that can be integrated in editable documents and sent through electronic mail.

EmbeddedButtons shares many of the features of the Interleaf active documents system [7], which allows objects in user-editable documents to become buttons by associating these objects with scripts in a small LISP-based language. Our work differs in that it supports the easy addition of new media editors and emphasizes the use of documents as control panels.

We also draw on work that characterizes and extends the functionality of buttons. XButtons perform different actions depending on the gesture used to press them [22]. Similarly, our pop-up buttons can perform different actions depending on how they are pressed. Garnet Interactors [18] provide a framework for describing the behavior of a variety of interactors, including buttons and menus. The finite state machine used by our buttons to determine when to start, stop, run, and abort is similar to the state machine reported in this paper by Myers.

A number of recent document systems make use of a single centralized library and runtime kernel to provide a common facility across a set of document editors. For example, editors in the PROXHY system call into a shared hypertext facility that keeps track of hypertext links and negotiates with the editor to determine how the source and destination objects should be drawn on the screen [12]. Also, as in EmbeddedButtons, editors notify the kernel when a button is pressed. EmbeddedButtons differs from PROXHY in that it provides a general-purpose button facility rather than a hypertext facility and in that no communication is needed between an editor and EmbeddedButtons during screen refresh, thus reducing communication requirements and simplifying the modifications that must be made to editors.

9. CONCLUSIONS

EmbeddedButtons combines three old ideas into a single system: (1) It provides a library of routines and a runtime kernel that can be shared by an extensible set of document editors. (2) It supports the use of document objects as buttons. (3) It allows buttons to be linked to applications so that documents can serve as control panels. EmbeddedButtons was a small effort. Our system was written in one summer by an undergraduate and later revised and extended by the author. However, we are excited about the EmbeddedButtons project because it enables a set of features that are not usually found together and makes them all available in electronic documents. In particular, features of documents become features of control panels and features of document elements become features of buttons. In addition, users, interface designers, and programmers benefit from not having to learn about documents and control panels as separate systems with different conventions.

As we try to discover new applications for the EmbeddedButtons system, we consider the rich set of features that document editors support. For instance, editors contain function to create, select, move, copy, delete, store, retrieve, search for, scroll through, change the style of, selective display, mail, and print document information. When these features are mixed with the features of buttons and control panels, an interesting new set of applications becomes possible. For instance: String search can be used to find a button in a long document. Control panels can be stored to disk and retrieved for later use. Unwanted parts of control panels can be deleted, freeing up screen space. Control panels can be included in reference manuals and sent through the mail. A customized color palette can be created by turning all shapes in an illustration into buttons that transfer their color when pressed.

As we blur the distinction between documents and user interfaces, we also blur the distinction between users and interface designers. Because our control panels are always displayed in a document editor, users can learn how to manipulate interface components with little effort. For example, a user may copy part of a control panel to use in an illustration, regroup control panels elements to better support a task, or make several copies of a control panel to use for several specific tasks.

Our system is a flexible tool for user interface designers. While user interface layout tools generally provide only a restricted set of user interface

components, graphical editors and other document editors provide a rich set of operations that the designer can use both for layout and button feedback. While we have encountered mismatches between the assumptions built into document editors and the functionality needed for control panels, these problems can usually be repaired by making minor changes to the document editors. In fact, these changes often make the editors more useful for producing static documents as well as user interfaces.

EmbeddedButtons helps programmers in several ways: Software that a programmer writes to improve the way document editors lay out media can be used as well to lay out application control panels, without further effort. No compiled code need be written to generate application control panels; the control panels are created directly by editing documents. Finally, buttons can be used to test new code. For instance, a programmer can make a button to test each of the commands of a new application. Because buttons can be added to the system at run-time, new tests can often be added without recompiling code.

EmbeddedButtons is a largely successful project. Integrating our existing editors into the system required only a modest effort, it performs well on current hardware, and it is in use for several applications, including control panels, personal macro buttons, and as part of a simple file browser. Perhaps the main failing of the project is that its software is currently only available to the small group of researchers who use the Cedar programming environment. It would be interesting to see what EmbeddedButtons would be used for in a larger more diverse user community.

10. PLANS FOR FUTURE WORK

In the future, we plan to produce control panel documents for several real applications, like the Gargoyle control panels, and to collect information about how they are used. We will experiment with new ways to edit buttons, including pop-up property sheets that are themselves active documents. We will integrate buttons with new applications; for instance, incoming electronic mail messages could include buttons to telephone the message sender. We hope to integrate buttons into new media types, including raster images, solid models, and animations. Finally, we plan to investigate the issues of interchanging buttons between different document editors.

ACKNOWLEDGMENTS

I give many thanks to Aaron Goodisman, who enthusiastically programmed EmbeddedButtons as a summer project in 1989. I thank Doug Wyatt for modifications to the Tioga text editor, and Ken Pier both for modifications to the Gargoyle illustrator and for proofreading this manuscript. I thank Polle Zellweger, my project leader, for comments that lead to improvements in the format and content of this paper and to improvements in the EmbeddedButtons system. Finally, special thanks go to Jock Mackinlay and my TOIS reviewers, whose detailed and helpful comments pointed the way to a better structured and more readable paper.

ACM Transactions on Information Systems, Vol. 10, No. 4, October 1992.

REFERENCES

1. ARNON, D., BEACH, R. McISAAC K., AND WALDSPURGER, C. CaminoReal: An interactive mathematical notebook. In *Document Manipulation and Typography, Proceedings of EP'88* (Nice, France, Apr. 20-22, 1988), Cambridge University Press, pp. 1-18.
2. AVRAHAMI, G. BROOKS, K. P., AND BROWN, M. H. A two-view approach to constructing user interfaces. In *Proceedings of SIGGRAPH '89* (Boston, Mass., July 1989), *Comput. Graph.* 23, 3 (1989), 137-146.
3. BIER, E. A. AND STONE, M. Snap-dragging. In *Proceedings of SIGGRAPH '86* (Dallas, Tex., Aug. 1986), *Comput. Graph.* 20, 4 (1986), 233-240.
4. BIER, E. A. AND GOODISMAN, A. Documents as user interfaces. In *EP90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* (Gaithersburg, Md., Sept. 1990), R. Furuta, Ed., Cambridge University Press, 1990, pp. 249-262.
5. BIER, E. A. EmbeddedButtons: Documents as user interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (UIST '91, Hilton Head, S.C., Nov. 1991), pp. 45-53.
6. CARDELLI, L. Building user interfaces by direct manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Oct. 17-19, 1988), pp. 152-166.
7. ENGLISH, P. M., JACOBSON, E. S., MORRIS, R. A., MUNDY, K. B., PELLETIER, S. D., POLUCCI, T. A., AND SCARBRO, H. D. An extensible, object-oriented system for active documents. In *EP90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography* (Gaithersburg, Md., Sept. 1990), R. Furuta, Ed. Cambridge University Press, 1990, pp. 263-276.
8. FEINER, S., NAGY, S., AND VAN DAM, A. An experimental system for creating and presenting interactive graphical documents. *ACM Trans. Graph.* 1, 1 (Jan. 1982), 59-77.
9. FURUTA, R., QUINT, V., AND ANDRÉ, J. Interactively editing structured documents. *Electron. Pub.* 1, 1 (Apr. 1988), 19-44.
10. GOODMAN, D. *The Complete HyperCard Handbook*. Bantam Books, 1987.
11. JOHNSON, J., ROBERTS, T. L., VERPLANK, W. SMITH, D. C., IRBY, C. H., BEARD, M., AND MACKEY, K. The Xerox Star: A retrospective. *Computer* 22, 9 (Sept. 1989), 11-29.
12. KACMAR, C. J., AND LEGGETT, J. J. PROXY: A process-oriented extensible hypertext architecture. *ACM Trans. Inf. Syst.* 9, 4 (Oct. 1991), 399-419.
13. KOVED, L., AND SHNEIDERMAN, B. Embedded menus: Selecting items in context. *Commun. ACM.* 29, 4 (Apr. 1986), 312-318.
14. KURLANDER, D., AND BIER, E. A. Graphical search and replace. In *Proceedings of SIGGRAPH '88* (Atlanta, Ga., Aug. 1988), *Comput. Graph.* 22, 4 (1988), 113-120.
15. MACLEAN, A. CARTER, K., LÖVSTRAND, L., AND MORAN, T. User-tailorable systems: pressing the issues with buttons. In *Proceedings of CHI '90* (Seattle, Wash., Apr. 1990) *Human Factors in Computing Systems*, ACM Press, New York, 1990, pp. 175-182.
16. MCKAY, S., YORK, W., AND MCMAHON, M. A presentation manager based on application semantics. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, (UIST '89, Williamsburg, Va., Nov. 1989), pp. 141-148.
17. MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. H., AND SMITH, F. D. Andrew: A distributed personal computing environment. *Commun. ACM.* 29, 3 (Mar. 1986), 184-201.
18. MYERS, B. A. A new model for handling input. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 289-320.
19. OUSTERHOUT, J. K. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter Conference* (Jan. 1990), pp. 133-146. Also available as Report UCB/CSD 89/541, Computer Science Div. (EECS), Univ. of California, Berkeley, Nov. 1989.
20. PAUSCH, R. SUIT: The Pascal of user interface toolkits. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '91, Hilton Head, S.C., Nov. 1991), pp. 117-125.
21. PIER, K., BIER, E., AND STONE, M. An introduction to Gargoyle: An interactive illustration *ACM Transactions on Information Systems*, Vol. 10, No. 4, October 1992.

- tool. In *Document Manipulation and Typography, Proceedings of EP'88*, Cambridge University Press, 1988, pp. 223-238. Also available in Xerox PARC Tech. Rep. EDL-89-1.
22. ROBERTSON, G. G., HENDERSON, D. A., AND CARD, S. K. Buttons as first class objects on an X Desktop. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '91, Hilton Head, S.C., Nov. 1991)*, pp. 35-44.
 23. SWINEHART, D. C., ZELLWEGER, P. T., BEACH, R. J., AND HAGMANN, R. B. A structural view of the Cedar programming environment. *ACM Trans. Program. Lang. Syst.* 8, 4 (1986) 419-490. Also available as Xerox PARC Tech. Rep. CSL-86-1.
 24. TERRY, D. B., AND BAKER, D. G. Active Tioga documents: An exploration of two paradigms. *Electron. Publ.: Origination, Dissemination, Des.* 3, 2 (May 1990), 105-122.
 25. WEBSTER, B. F. *The NeXT Book*. Addison-Wesley, Reading, Mass., 1989, Ch. 7.
 26. WOLBER, D., AND FISHER, G. A demonstrational technique for developing interfaces with dynamically created objects. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '91, Hilton Head, S. C., Nov., 1991)*, pp. 221-230.
 27. XEROX CORPORATION. *VP CUSP Buttons Reference*. VP Series Reference Library, Version 2.0, Xerox Viewpoint, Xerox Corporation, 1988, Xerox Product Education, 701 South Aviation Boulevard, ESCN-215, El Segundo, CA 90245.
 28. ZELLWEGER, P. T. Scripted documents: A hypermedia path mechanism. In *Hypertext '89 Proceedings* (Pittsburgh, Penn., Nov. 1989), ACM, 1989, pp. 1-14.