An Extensible Interactor Toolkit for Enhancing Information Awareness

D. Scott McCrickard
Graphics, Visualization, and Usability Center
and College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
Email: mccricks@cc.gatech.edu URL: http://www.cc.gatech.edu/~mccricks/

Abstract

The typical computer application follows a set pattern: it is executed by a user, it provides responses to commands issued by the user, and it is terminated by the user when it is no longer needed. However, the advent of networked computers and the World-Wide Web has changed the way many applications work. Agents and similar programs constantly access online information databases and identify items that may be of interest to their users. The problem lies in communicating the collected information. Most existing widgets cannot take full advantage of the constant information flux generated by these programs. They work best as part of a full-screen display, they do not react well to changes in information, or they cannot attract the attention of users when new information arrives. This paper explores these problems and describes a number of interactors that may help alleviate them.

Keywords: Information agents, visualization, information awareness

1 Introduction

The typical computer application follows a set pattern: it is executed by a user, it provides responses to commands issued by the user, and it is terminated by the user when it is no longer needed. However, a new class of program, software agents, operate in a different manner. An agent is a system that performs tasks in an autonomous manner in order to satisfy the needs of its users. By *autonomous* we mean that the agent can operate for long periods of time without direct user intervention. Agents often do the tedious, repetitive, and time-consuming tasks that humans do not want to do. The dynamic nature of the World Wide Web drives the need for software agents, which can access online information and identify items that may be of interest to their users.

One problem that arises in designing agents is in communicating the collected information to the user. Most existing interactors (also known as widgets) and interface-building guidelines were designed for typical computer applications. The constantly changing nature of agents seems to call for a new widget set and new interface guidelines. Since agents run constantly, users will want to utilize their computer desktop for other applications most of the time. This suggests that agent interfaces should not take up much space. Information is constantly generated by agents, and the display and interface should display this information in a hands-off manner so the user can stay aware of changes with minimal interruption of other tasks. This paper explores these problems and describes a number of widgets that may help alleviate them.

The goal of our work is to create a toolkit containing widgets that can increase the information awareness of their users and a set of guidelines for using the widgets. The *information awareness* generated by a widget refers to its ability to provide information without requiring explicit actions from a user and without distracting users from their normal daily tasks. One example of programs that use such a widget are the email "biff" programs, which show a mailbox flag that points up whenever new mail arrives. This graphical box provides important information in a small space without requiring user interaction. At any time, users can glance at the box and determine the status of their email folder.

As is often the case, we can turn to the entertainment industry to get a glimpse of ways to communicate this flood of information. The Headline News television network (as well as others) provide a bar at the bottom of the screen that scrolls or rotates between breaking information such as stock quotes, news headlines, and sports scores. During ball games, Fox and other major networks provide a small rectangular box in the corner of the screen that contains the score, current period, time remaining, and other relevant information about the game being seen. These techniques can allow the user to quickly and easily obtain up-to-date information about topics of interest while still paying attention to the main television attraction.

Similar methods for providing information have started to find their way onto the computer desktop. ESPN's home page (espn.com) contains a constantly-updated Java widget that fades between scores of ball games. Users can place it anywhere on their desktop and stay aware of the game scores while still doing their work. PointCast (pointcast.com) makes a screen saver that displays news items of interest and has a scrolling line of stock quotes and sports scores at the bottom of the screen. During times when a computer is not being used, it can provide its users with information.

Our contribution will be to put these new interaction techniques into the hands of pro-

grammers via a widget toolkit. Our toolkit will facilitate the design of agent interfaces by allowing programmers to incorporate useful display and interaction techniques into their own programs. We plan to implement the widgets in Tcl/Tk, the language developed by John Ousterhout at UC-Berkeley and Sun Labs. Tcl/Tk is a platform-independent scripting language used to design interfaces that "glue" together existing programs, making them easier to use. The Tk part of Tcl/Tk is a graphical toolkit that contains commands for creating buttons, scrollbars, graphical canvases, and other widgets. Our widget toolkit will be implemented in Tcl/Tk, thus providing the same platform independence as Tcl/Tk itself. Each widget will conform to the Tcl/Tk standards for widget creation, querying, and modification established by Jeffrey Hobbs of CADIX International (see www.cs.uoregon.edu/research/tcl/script/widget/). In so doing we allow programmers familiar with the Tcl/Tk toolkit to use our toolkit with ease.

2 Widget descriptions

The widgets that we plan to provide in this toolkit include a navigation bar, a fading widget, a tickering widget, and several others. The remainder of this section describes each of these widgets, discusses their current implementation status, and outlines the future plans for them.

Navigation bar

Many computer applications require a user to find and select items from a list using a rectangular widget called a listbox. However, many lists are too long to be displayed in their entirety in a listbox, a problem that is magnified with the reduced space of agent interfaces. To change the visible portion of the list, most applications provide a scrollbar, but a scrollbar provides little information about the contents of the list. To address this problem, we are developing a *navigation bar* widget that maintains the familiar scrollbar paradigm while increasing the amount of information provided to the user. By building on the existing mental model for a scrollbar, users should have a short learning curve for the new widget.

The navigation bar communicates information about the list contents by using the space inside the scrollbar to represent each list entry with a graphical line, where properties of the list entries are reflected in properties of the graphical lines. For example, the length of the list entry could be represented by the length of the line, and the position of the entry in the list could correspond to the position of the line in the display area. Categories or classifications of the entries could be shown using the color, size, or offset of the line.

While several systems have used this type of graphical representation to communicate information [2, 3, 5, 6], none have provided the flexibility, power, and availability needed by programmers to include them in their own interfaces. We will address this need with a Tcl/Tk implementation of the navigation bar. Our implementation will have many of the same components as a scrollbar (arrows, a thumb, a trough) that will function in a similar manner, but the trough will contain colored horizontal lines that represent the items in the list. The length and relative position of each line will correspond to the length and relative position of its corresponding list entry. The colors correspond to groupings of list items. For example, a listing of files might be colored by file type (red could indicate html files, blue

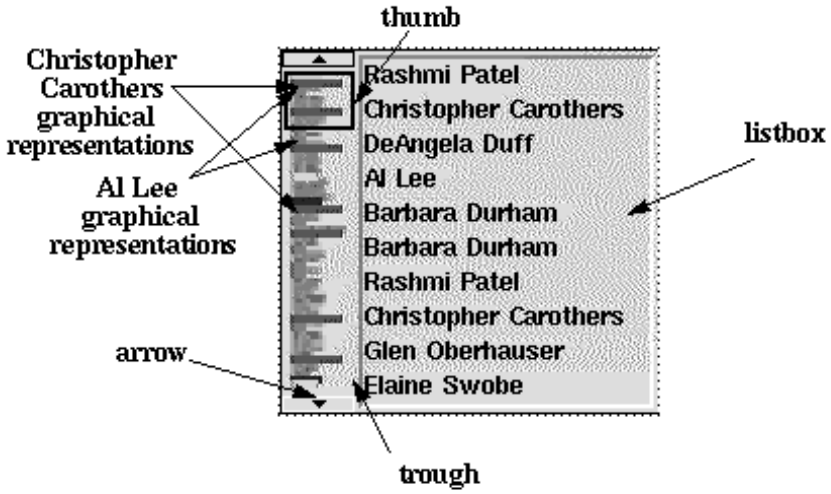

Figure 1: Prototype of a listbox and navigation bar using the information mural technique. The listbox contains names, much like you would find in an email or Usenet news reader, and each list item is represented in the navigation bar with a horizontal line. While most entry representations are grey, certain ones, such as those for Christopher Carothers and Al Lee, are highlighted with color, making it easy to find their location in the list. The thumb surrounds the representations that are shown in the listbox. As with a scrollbar, the user can click on the arrows or in the trough to adjust the visible portion of the list.

code files, and so on) or a listing of email messages could be colored by user-selected sender name or by frequency of subject. Repeated color patterns can then reveal related items. The height and layout of the bars will depend on the programmer-configurable navigation bar type, which includes information murals, pile views, zoom views, and fisheye views.

Information murals are compressed graphical displays of information that have been used in a wide range of applications, including visualizations of software execution, numerical data, and general information [7]. Information murals represent a large information space (in this case a textual list) with a smaller graphical space by mapping the elements of the large space into the smaller one. In the case of a list, the characters in the list map to the pixels in the navigation bar to create a scaled-down picture of the entire list. To understand how the information mural is created, think of a navigation bar that is M pixels wide and N pixels high as an MxN grid that overlaps the entire list. A pixel in the grid is shaded with intensity corresponding to the number of characters that touch it: the more characters that map to a pixel, the darker it becomes. If a word is associated with some color as described previously, the pixels corresponding to the characters in the word take on that color.

The *pile metaphor* introduces onto the computer desktop the concept of piling items (in this case, encodings) on top of each other. Just as paper documents can be piled on a desk and identified by their appearance, items in a pile view can overlap and be recognized by their color, size, or shape. The pile metaphor was introduced in a desktop document layout system [8]. Piles created by the user had a disheveled appearance with parts of the document icon sticking out slightly so the user could find a document even if it were in the middle of the pile. The pile bar will employ the pile metaphor in its arrangement of graphical bars. Each entry will have a fixed height that is large enough to be seen and clicked. To show all of the representations in the limited space that is available, the bars must be piled

on top of each other, meaning that some bars may be partially or completely obscured. An additional protocol is needed to define the stacking order, for example, the order in which the items were inserted may correspond to the stacking order of their graphical bars.

A *zoom view* provides an intermediate step between the small number of items in a listbox and the large number in an entire list. The zoom technique is incorporated in many user interfaces and in fact is central to the Pad++ software system [1]. In a navigation bar, the graphical bars would be of uniform size and initially would be large enough to be identified and selected with ease. Users will be able to shrink the size of the bars (zoom out) to get an overview, or enlarge them (zoom in) to see more detail. Since the bars may not all fit in the available space, when the view is panned, the navigation bar will adjust accordingly to show the visible list region and the surrounding items. Although not all of the bars can be seen simultaneously, those that are visible are unobscured by other bars (unlike with the pile metaphor), making it easier to find items within the visible region.

A fisheye camera lens is a wide angle lens that enlarges items of interest at the center and shrinks items of less interest along the periphery. George Furnas employed this idea in his *fisheye view* algorithm, which enlarged important portions of a graph and shrunk the less important portions [5]. Others have used this fisheye technique in image magnification, Web browsers, and text readers. In a navigation bar, items of interest could be enlarged and less interesting items shrunk using a similar algorithm. The interesting areas will have larger representations than the less interesting areas, thus making them easier to see and easier to select. The concept of "interesting" will be defined using a protocol similar to the one used to stack representations in the pile view.

To ensure the extensibility of the navigation bar widget, programmers will be able to introduce their own layout algorithms. These algorithms will have access to data about the list and will control the size and layout of the bars in the navigation bar.

Fade

Rapid changes in the appearance of a widget often attracts the user's attention to the widget. While this is advantageous in many situations, it could be detrimental in an agent interface where the users' attention is primarily focused on other tasks. Instead, we need a widget that can change continuously to match the large and dynamic information space but will change gradually to avoid interrupting the everyday tasks of the user.

In our toolkit, we will provide a *fade* widget, which will display several blocks of text and graphics within the same space by gradually fading between them. The gradual change will be less distracting than a sudden switch, yet will allow multiple information blocks to be displayed in a single area. The speed with which the fade occurs can vary depending on the nature of the application: if the widget is used in a primary (non-agent) application interface, a quick fade might be used, while a secondary agent application might call for a slower, less noticeable fade. See Figure 2 for an example of how the fade widget will work.

The programmer will be able to control the time it takes to fade between blocks of information as well as many standard options like colors, sizes, and fonts. To achieve the fading effect, the foreground colors gradually will be blended with the background colors until the foreground can no longer be seen. Different commands can be associated with mouse actions and key presses to allow the user to access the appearance of the widget. The location of the text and graphics will be specified using points of the compass (*nw* for

Figure 2: Five snapshots in the operation of a fade widget that displays the scores of baseball games. The first frame shows an initial block of text. The next two frames show how the text fades away into the background, and the final two frames show how the new text will appear in the same place. The time it takes to fade between text messages can be controlled by the programmer.

northwest corner, *n* for north, and so on), plus *c* to center the block.

Thus far, we have implemented the text-only non-color widget seen in Figure 2 to show that our concept is feasible. We plan to expand the widget to include full color text and graphics, a Tk-style programmer interface, and programmer-specified key and mouse bindings. We then plan to incorporate the fade widget into a number of applications to demonstrate its usefulness.

Ticker

The fade widget provided one way to show multiple information blocks in a limited space without distracting the user. However, the information must be formatted into blocks that will fit in the available space. In addition, there are no natural actions that the user could perform for controlling the presentation of information. For example, if you as a user saw a fade widget, how would you cause it to back up to the previous block, or pause, or slow down?

The *ticker* widget addresses many of these problems. It provides a ticker-tape-style display that scrolls or "tickers" the information across the screen. As with the fade widget, a slow tickering can be less distracting than a sudden switch, but the ticker widget has the added advantage that a stream of text could be any length since it will ticker across the screen in a readable way.

In addition, there seems to be a natural reaction to grab and pull the ticker widget to make it slow down, stop, back up, or move forward. We will include a hand cursor that will appear when the cursor enters the ticker widget. When users click within the widget, they can manipulate the information to move in the desired direction and speed. By providing this functionality, the user has more control over the way in which the information is displayed. The programmer no longer needs to worry about providing functionality that will please all users.

The programmer will be able to control the speed with which the ticker widget moves and the direction (left-right or top-bottom) in which the tickering occurs, plus the usual color, size, and font options. Thus far, we have implemented a text-only widget that tickers left-right at a programmer-defined speed. We plan to expand the widget to include graphical icons, user manipulation, and a Tk-style programmer interface. We then plan to incorporate the ticker widget into a number of applications to demonstrate its usefulness.

Other widgets

We have preliminary plans for several other types of widgets.

One widget (or class of widgets) is *hidden widgets*. A hidden widget overlaps a standard widget, but remains out of sight until triggered by a user action. Then the widget becomes visible and acts just like a normal widget. Consider for example adding buttons to the fade widget that will allow the user to move forward and backward to other information blocks. It would be wasteful to add button widgets for this functionality – most of the time they would waste space by sitting unused. Instead, hidden buttons could lie on top of the fade widget and become visible only when the cursor enters the fade widget, at which time they would become visible and functional. Users could click on the buttons to browse through the information blocks at their own pace. When the cursor exits the widget, the hidden buttons would disappear again. Hidden widgets could prove useful when implemented for buttons, scrollbars (or navigation bars), or menus.

Significant work has been done in the College of Computing and elsewhere on the use of *anthropomorphism* in computer-human communication [4]. It stems in part from the belief that humans can naturally and easily recognize emotions and meaning in human faces. Consider incorporating the anthropomorphism into a widget where a face communicates a general feeling for the state of information: if an important event occurred, the face could look anxious, if a bad event occurred, the face could look upset, and so on. By utilizing the brow, eyebrows, eyes, nose, cheeks, lips, and chin, many different emotions could be communicated. Of course, this type of widget would create the problem of mapping information changes into emotional states, but for some information resources that might not be too difficult.

Much of the content of the World-Wide Web (perhaps the largest and most volatile information resource ever) is graphical, yet few widgets exist that manipulate graphical images in useful ways. Consider an agent that finds graphical images of various sizes that then need to be displayed in a small space. Based on the size and shape of the original and on the size and shape of the destination area, a widget could automatically crop (cut off parts of the image) and shrink (reduce the size of the image) the original in a way to best capture the meaning of the image. The widget should be able to put images into categories such as portraits, landscapes, banners, imagemaps, icons, and so on based on the size, shape, colors, and other easily attainable image characteristics. Different reduction schemes would be used with each category. For example, to reduce a portrait, it would be advisable to crop the lower portion and keep the upper portion (that contains the person's face). On the other hand, to reduce an icon might not require any cropping and only minimal shrinking.

3 Conclusions and future work

This paper has outlined a set of widgets that will facilitate the design of agent interfaces by allowing programmers to incorporate useful display and interaction techniques into their own programs. Many of the techniques are in use in existing interfaces, but there are no guidelines or set programming principles for how they should be used. The widgets described in this paper can communicate constantly changing information using different media types. By providing a programming definition in an established interface design language for these widgets, we expect that designers will be able to incorporate them into their interfaces quickly and easily. The agent interface domain provides new interface challenges for application designers. This widget toolkit can help designers overcome the challenges.

References

[1] Benjamin B. Bederson and James D. Hollan. Advances in the pad++ zoomable graphics widget. In *Proceedings of the USENIX Tcl/Tk '95 Workshop*, pages 206–207, 1995.

[2] Richard Chimera. Value bars: An information visualization and navigation tool for multimedia listings. In *Proceedings of the ACM Human Factors in Computing Systems Conference (CHI '92)*, pages 293–294, Monterey, CA, 1992.

[3] Stephen G. Eick. Data visualization sliders. In *Proceedings of the 7th Annual Symposium on User Interface Software and Technology (UIST '94)*, pages 119–120, Marina del Rey, CA, November 1994.

[4] I. Essa and A. Pentland. Coding, analysis, interpretation and recognition of facial expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7), July 1997.

[5] George W. Furnas. Generalized fisheye views. In *Proceedings of the ACM Human Factors in Computing Systems Conference (CHI '86)*, pages 16–23, Boston, MA, 1986.

[6] William C. Hill and James D. Hollan. History-enriched digital objects: Prototypes and policy issues. *The Information Society*, 10(2), April 1994.

[7] Dean Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 43–50, Atlanta, GA, November 1995.

[8] Daniel E. Rose, Richard Mander, Tim Oren, Dulce B. Ponceleon, Gitta Salomon, and Yin Yin Wong. Content awareness in a file system interface: Implementing the 'pile' metaphor for organizing information. In *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR '93)*, pages 260–269, Pittsburgh, PA, 1993.