RAPPORT DE RECHERCHE

L R I

UIMARKS : QUICK GRAPHICAL INTERACTION WITH SPECIFIC TARGETS

ROUSSEL N / CHAPUIS O

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

06/2009

Rapport de Recherche N° 1520

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

UIMarks: Quick Graphical Interaction with Specific Targets

Nicolas Roussel[1,2] and *Olivier Chapuis*[1,2]
[1]LRI - Univ. Paris-Sud & CNRS, [2]INRIA
Orsay, France
{roussel, chapuis}@lri.fr

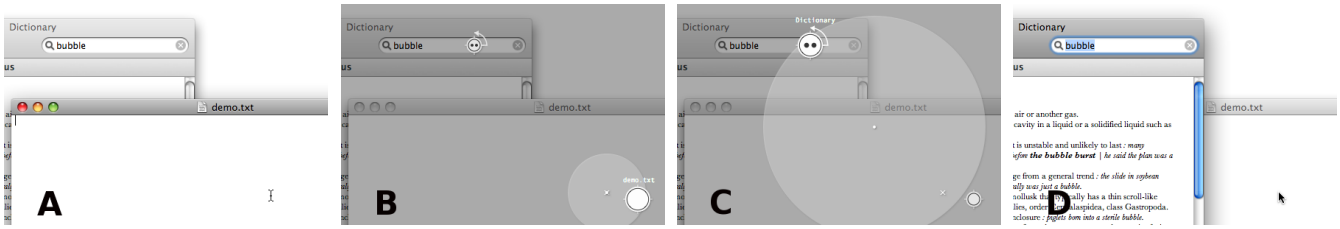

Figure 1: Entering the UIMarks mode (A → B), selecting the top mark (C), activating this mark, which double-clicks and sends the cursor back to the entering point (C → D).

ABSTRACT

In this note we present UIMarks, a novel system that lets users specify on-screen targets and associated actions by means of graphical marks. UIMarks supplements traditional pointing by providing users with an alternative mode in which they can quickly create, select, configure, activate and delete marks. Actions associated to these marks can range from basic point-and-click interactions to the execution of complex action sequences. UIMarks has been implemented on two different platforms, Metisse and OS X.

ACM Classification: H.5.2 [Information interfaces and presentation]: User interfaces - Graphical user interfaces.

General terms: Design, Human Factors.

Keywords: Pointing, macros, window management.

INTRODUCTION

Pointing facilitation techniques aim at improving the acquisition of on-screen targets with a pointing device. Pointing being arguably one of the most fundamental tasks in HCI [2], research on these techniques is usually motivated by the idea that small improvements in speed or accuracy may result in large efficiency gains. Yet very few of the techniques proposed by HCI researchers are actually used in existing systems. As explained by Wobbrock et al. [15], one reason for this is probably that most of these techniques are *target-aware*: they require some knowledge about the size and position of the targets and sometimes the ability to modify them.

Numerous target-aware pointing techniques have indeed been proposed that try to beat Fitts' law by reducing target distance, increasing target width or both [2]. *Drag-and-pop* [3], for example, temporarily brings potential targets closer to the cursor. *Ninja cursors* [12] are multiple cursors attached to a single device to reduce the average distance to targets. *Expanding targets* dynamically change their size to provide users with a larger area to interact with at their focus of attention [13]. The *bubble cursor* [9] dynamically resizes its activation area so that exactly one target is selected at any time. *Object pointing* [10] allows the cursor to skip accross empty spaces, jumping from one target to another, while *semantic pointing* [4] dynamically adjusts the control-display gain according to targets. Other techniques take the dynamic characteristics of pointing gestures into account to predict the target [1] or adapt the cursor's activation area [7].

Target-aware techniques tend to work best on sparse layouts: in dense layouts, occlusion and false activation problems can quickly obviate the potential benefits [2]. Deciding which of the on-screen objects should really be considered as potential targets is a complex problem. Basic interaction techniques like rubber-band selection also require pointing and clicking on blank parts of the interface, which is inherently impossible with some techniques. Lastly, some of them are also known to be visually distracting either because of the growing and shrinking of objects, or because of the discontinuities introduced by cursor warping.

A few target-agnostic techniques have also been proposed that focus on user actions instead of possible targets. The *angle mouse*, for example, adjusts the control-display gain based on angular deviation [15]. *MAGIC* uses eye tracking to coarsely define an area of interest in which the cursor is automatically warped [16]. *Rake cursor* uses the gaze position to select a cursor from a grid of several [5]. It has also been suggested to associate a small magnetic force to each past click or drag to make targets easier to select without requiring prior knowledge of them [11]. All these techniques have some advantages over the target-aware ones, but have the disadvantage of considering all pixels equal, at least initially, which leads to limited performance improvements.

LRI Technical Repport Number 1520
JUNE 2009

LABORATOIRE DE RECHERCHE EN INFORMATIQUE
UMR 8623, CNRS Université de Paris Sud
Bât. 490, 91405 ORSAY Cedex France

Target-aware techniques aim at providing quick access to all possible targets at a given time. The work presented here was initially motivated by the idea of supporting quick access to only a few targets mainly, if not exclusively, specified by the user. In this perspective, our work can be seen as a generalization of the approach successfully applied by Tsandilas & schraefel in the particular case of menus [14]. However, we are not only interested in facilitating pointing, but also in the operations that usually follow. We are particularly interested in repeated interaction sequences that imply going back and forth between two or more locations. Since few target-aware pointing techniques have been implemented in real systems, we also wanted our work to be easily applicable to standard graphical environments, without necessarily requiring the modification of applications.

The rest of this note describes UIMarks[1], a system designed to facilitate the selection and activation of a target chosen from a limited and specific set. We start by defining the concept behind that system. We then describe the interactions between the user and the system, the two versions that we implemented, and conclude with directions for future work.

CONCEPT

UIMarks was not designed to replace traditional pointing, but rather to supplement it by providing users with an alternative mode they can deliberately activate based on their specific needs. UIMarks supports the creation, configuration and use of *user interface marks*, graphical objects that explicitly locate on-screen targets and provide quick ways to interact with them. Typical use consists of entering the mode, selecting a mark and leaving the mode, which activates the mark and can in turn trigger some actions.

The marks can be seen as an equivalent of bookmarklets for graphical interfaces. One might create a mark over the *Get mail* button of a mailer that clicks it and send the pointer back to where it was. One might put a mark that double-clicks on a desktop icon. Or one might leave a more complex mark on a menu or palette that would provide a way of getting back after some user interaction, facilitating tasks requiring back and forth pointing and interaction.

Before describing user interactions in more detail, we will define more precisely the UIMarks concept by describing the mark attributes, the actions that can be attached to them and how they can be graphically represented.

Mark attributes

A mark is a uniquely identifiable object associated to an on-screen (x, y) position and further characterized by three attributes: its creator, target and lifetime

Creator – Marks can be created by the user, in anticipation of future use. They can be created by UIMarks itself in an implicit way, in response to user actions, or as a consequence of the explicit activation of another mark. Lastly, marks can also be created by UIMarks on behalf of other applications. An application might request the creation of a temporary mark on the `Ok` button of a dialog box, for example.

[1] pronounced ['wi ˌmärk], like UIST

Target – Marks can be either attached to the screen, or to a particular graphical object. In the latter case, activating the mark will raise the enclosing window and give it the keyboard focus. User actions on that window such as `move`, `resize`, `iconify` or `close` will also impact on the mark.

Lifetime – Marks can be permanent or temporary, lasting for only a limited time or number of activations.

Possible values for mark attributes can be summarized as:

$$\begin{aligned} creator &\in \{user, uimarks, otherapp\} \\ target &\in \{screen, window(id)\} \\ lifetime &\in \{permanent, temporary\} \end{aligned}$$

Actions

Each mark has an associated action that will be triggered every time the mark is activated. This action will be further referred to as the *primary action*. Incidental actions can also take place immediately before and after the primary action. We will refer to those as *preceding* and *following* actions. When executed, all three actions have access to the mark attributes as well as the location at which the user entered the UIMarks mode, i.e. the *entering point*.

Primary action – It can be as simple as `go there`, `click` or `double-click`. It can consist in a user-defined combination of such basic interactions. But it can also be arbitrarily complex, although it should remain describable to the user in a simple way, either textual (e.g. *switch to the virtual desktop on the right*) or graphical. The window associated to the mark, if any, will be raised and given the keyboard focus before this action is executed.

Preceding action [optional] – The only preceding action that can take place is the creation of a new mark (presumably temporary) at the entering point. The characteristics of this mark, however, can be fully specified. Note that this action must be executed first because of the side effects the primary one might have on the initial context (e.g. it might change the window stacking order).

Following action [optional] – Any action taking place at the entering point, i.e. `come back [and do xxx]`.

The execution steps triggered by the activation of a mark can be summarized as follows:

1. [create a mark configured like this at the entering point]
2. [raise the window and give it the keyboard focus]
3. execute the primary action
4. [come back [and do xxx]]

Graphical representation

When in UIMarks mode, each visible mark is represented on-screen by a disk shown at its location. The external and internal borders of the disk and its fill color are used to visualize the basic attributes of the mark (Figure 2). The primary action it triggers is represented by drawings inside the disk while incidental actions are represented on the outside (Figure 2). The preceding creation of a new mark is denoted by an arrow that starts on the left side of the disk and points forward. The new mark can be represented with extensive

details at the end of this arrow. More compact representations are also possible if it uses standard attribute values and causes no incidental action. Following actions are denoted using an annotated arrow that starts on the right side of the disk and points backward.

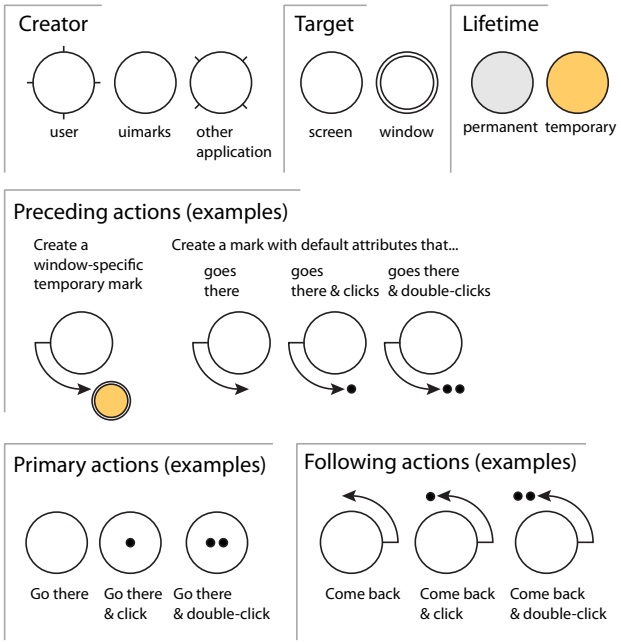

Figure 2: Visual representation of attributes and actions

To facilitate its selection and activation in certain situations, a mark can be offset from the underlying on-screen target. In this case, a small unselectable spot is left to indicate the target, connected to the mark by a line segment. Figure 3 shows a sample composition of the elements we described: an offseted user-created, window-specific, permanent mark that clicks and sends the cursor back to the entering point.

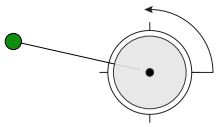

Figure 3: UIMark example

INTERACTION

In this section, we describe the interactions used to create, select, configure, activate and delete marks. We focus our attention on the principles that guided the design of these interactions rather than implementation details, some of which will be described in the next section.

All UIMarks interactions take place in a specific mode. Outside this mode, the marks are not visible. In this mode, a semi-transparent overlay is displayed on top of all windows, which preserves the user's context but helps to visually differentiate the marks (Figure 1, B and C). Switching between views, i.e. entering and leaving the mode, should be quick and easy. The mode can be associated, for example, to continuous pressure on a specific key (e.g. `Windows` or, `Fn`), button, or pedal. Leaving the mode activates the selected mark, if any, by default. Two specific operations allow the user to leave the mode without activating any mark: one that moves the system pointer back to where the user entered the mode, and the other that simply leaves it where it is. These operations can be triggered by pressing specific keys (e.g. `Escape` and `Enter`) or buttons.

The selection of an existing mark should also be as fast as possible. To achieve this, the UIMarks mode uses an adaptation of the bubble cursor [9]. The selected mark is scaled up by a factor (e.g. 5/3) to differentiate it. A small cross locates the entering point, and the location of the system pointer is indicated by a small dot (Figure 1, B and C). Additionally, we increase the system pointer's acceleration to reduce clutching for long distances. This change does not seem to cause any problem, presumably because of the important effective size of the marks with respect to the bubble cursor (Casiez et al. found little impact of high gain levels on performance except for very small targets and constant gain [6]). The default acceleration is automatically restored when leaving the UIMarks mode. It can also be temporarily restored within the mode by pressing a specific key (e.g. `Ctrl`), to support precise (re)positioning of the marks. Note that in addition to pointing-based selection, users can also circulate between marks using the `Tab` key.

A button click with the system pointer on an empty space creates a new mark with default attribute values that can be affected by the button used or keyboard modifiers. The newly created mark is automatically selected. Whenever a mark is selected, specific interaction techniques make it possible to reconfigure its target, lifetime and associated actions, the latter being chosen from a predefined set or incrementally specified. The selected mark can be deleted by pressing the `Backspace` key. Marks can be moved by a simple drag-and-drop interaction and offseted the same way using a keyboard modifier or alternative button.

We have also experimented with the use of the touchpad of a laptop combined with an external mouse. The touchpad was initially envisioned as a specific UIMarks device allowing implicit bimodal interaction, with the mouse being used for standard pointing. Operating the accelerated bubble cursor with the non dominant hand proved to be quite difficult, however. The touchpad and mouse are thus now configured to be used in the UIMarks mode in a bimanual rather than bimodal way: the accelerated mouse for fast selection, and the unaccelerated touchpad for precise positioning and configuration.

We acknowledge the fact that configuring a mark can be time consuming. However, this should not happen too often and the power of the marks resides in their use: their fast selection and activation, which can trigger operations that go beyond simple pointing.

IMPLEMENTATION

Implementing UIMarks notably requires the ability to observe, alter and generate input events, to display a semi-transparent overlay, and to determine target windows and track them. We have implemented the proposed techniques

on two different platforms: OS X and the experimental X Window system Metisse [8].

Because Metisse provides full control over its input and output mechanisms, implementing UIMarks on it was pretty straightforward. Minor details of this implementation include the creation of holes in windows that cover marked windows to show marks in context, the possibility to use various actions related to virtual desktops, and simple mouse-based interactions to configure the marks basic attributes and incrementally specify the associated actions.

The OS X implementation uses the Quartz 2D and Event Services APIs. It uses an overlay window that covers the whole display space independently of the number of physical displays. It controls pointer acceleration through the HID System Manager and uses CoreFoundation's distributed notification center to receive mark-related commands from external applications written in any language. Two lines of Python code are all it takes to send it a command to create, configure or delete a mark, for example. Simple keyboard-based configuration techniques are available to users in this version that only allow them to cycle between pre-defined combinations of primary and incidental actions (e.g. "*go there, click & come back*", "*create a temporary mark & go there*").

The window management services offered by Apple's public APIs are quite limited. A special connection to the window server is notably required to manage other applications' windows, which is exclusively maintained by the Dock. We resorted to using the `SetFrontProcess` function and the Accessibility API to raise windows, to associate marks to some of them and to track them. This causes an unwanted flash when we attach a mark to a particular window, as we need to temporarily hide the UIMarks window to query the Accessibility API for the target one. It also causes occasional unwanted modifications of the window stacking order.

Real use of UIMarks on the two platforms suggests that certain global aspects of it should be left configurable by the user. This includes the default target and lifetime for newly created marks, for example. But it also includes the definition of the *temporary* lifetime. Having experimented with several definitions that combine temporal and activation limits, we see different advantages and drawbacks in all of them. The automatic creation of a temporary mark at the entering point as a substitute for preceding actions has been vividly discussed. The impact of some window management operations is also difficult to decide. Although it seems reasonable to destroy the marks associated to a window being closed, what should be done when it is iconified, for example? Hiding the associated marks might seem the more coherent thing to do, but keeping them visible provides a quick way of deiconifying the window on activation. Similarly, should marks be moved with the underlying window or offseted? All these aspects are currently configurable at run-time in our implementations.

FUTURE WORK

We have presented UIMarks, a system designed to support quick graphical interaction with a specific and limited set of targets. We are now interested in determining the conditions under which such a system provides a real benefit. Informal use suggests it might mainly provide an advantage for long distance or small targets. Lab experiments will help to assess the cost of mode switching and cursor warping depending on the distance, size and nature of the marks. UIMarks has already been tested in dual-monitor configurations. We are currently adapting the software to make it work on a high-resolution interactive wall.

REFERENCES

1. T. Asano, E. Sharlin, Y. Kitamura, K. Takashima, and F. Kishino. Predictive interaction using the delphian desktop. In *Proc. UIST '05*, 133–141. ACM, 2005.
2. R. Balakrishnan. "Beating" Fitts' law: virtual enhancements for pointing facilitation. *IJHCS*, 61(6):857–874, 2004.
3. P. Baudisch, E. Cutrell, M. Czerwinski, D. C. Robbins, P. Tandler, B. B. Bederson, and A. Zierlinger. Drag-and-pop and drag-and-pick: techniques for accessing remote screen content on touch- and pen-operated systems. In *Proc. of Interact 2003*, 57–64. IOS Press, 2003.
4. R. Blanch, Y. Guiard, and M. Beaudouin-Lafon. Semantic pointing: improving target acquisition with control-display ratio adaptation. In *Proc. of CHI '04*, 519–526. ACM, 2004.
5. R. Blanch and M. Ortega. Rake cursor: improving pointing performance with concurrent input channels. In *Proc. of CHI '09*. ACM, 2009.
6. G. Casiez, D. Vogel, R. Balakrishnan, and A. Cockburn. The impact of control-display gain on user performance in pointing tasks. *HCI*, 23(3):215–250, 2008.
7. O. Chapuis, J. Labrune, and E. Pietriga. Dynaspot: speed-dependent area cursor. In *Proc. of CHI '09*. ACM, 2009.
8. O. Chapuis and N. Roussel. Metisse is not a 3D desktop! In *Proc. of UIST'05*, 13–22. ACM, 2005.
9. T. Grossman and R. Balakrishnan. The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. In *Proc. of CHI '05*, 281–290. ACM, 2005.
10. Y. Guiard, R. Blanch, and M. Beaudouin-Lafon. Object pointing: a complement to bitmap pointing in GUIs. In *Proc. of GI '04*, 9–16. CHCCS, 2004.
11. A. Hurst, J. Mankoff, A. K. Dey, and S. E. Hudson. Dirty desktops: using a patina of magnetic mouse dust to make common interactor targets easier to select. In *Proc. UIST '07*, 183–186. ACM, 2007.
12. M. Kobayashi and T. Igarashi. Ninja cursors: using multiple cursors to assist target acquisition on large screens. In *Proc. of CHI '08*, 949–958. ACM, 2008.
13. M. J. McGuffin and R. Balakrishnan. Fitts' law and expanding targets: experimental studies and designs for user interfaces. *ACM ToCHI*, 12(4):388–422, 2005.
14. T. Tsandilas and m. c. schraefel. Bubbling menus: a selective mechanism for accessing hierarchical drop-down menus. In *Proc. CHI '07*, 1195–1204. ACM, 2007.
15. J. Wobbrock, J. Fogarty, S. Liu, S. Kimuro, and S. Harada. The angle mouse: target-agnostic dynamic gain adjustment based on angular deviation. In *Proc. of CHI '09*. ACM, 2009.
16. S. Zhai, C. Morimoto, and S. Ihde. Manual and gaze input cascaded (MAGIC) pointing. In *Proc. of CHI '99*, 246–253. ACM, 1999.