

Power Tools for Copying and Moving: Useful Stuff for your Desktop

Guillaume Faure^{1,2} Olivier Chapuis^{1,2} Nicolas Roussel^{1,2}
{gfaure | chapuis | rousset}@lri.fr

¹LRI - Univ. Paris-Sud & CNRS
Orsay, France

²INRIA
Orsay, France

ABSTRACT

Copy and move operations have long been supported by interactive desktops through various means. But the growing number of on-screen objects makes these means harder to use. In this note, we present new tools and techniques to enhance the existing ones: a selection, copy and drag history manager; two techniques to expose the user's desk and leaf through stacks of overlapping windows; and a technique that integrates the previous two with conventional drag-and-drop.

ACM Classification Keywords

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

Author Keywords

Copy-and-paste, Cut-and-paste, Drag-and-drop.

INTRODUCTION

The ability to copy and move data from one window to another plays an essential part in the creativity and productivity of interactive desktop users, supporting the creation of new data from various sources. Over the years, *copy-and-paste* and *cut-and-paste* emerged as standard paradigms for these operations both following the same four-steps procedure: the user first selects one or more object(s), activates the `copy` or `cut` command, specifies the destination and then activates the `paste` command. Various techniques support this *copy-or-move* action sequence, such as keyboard shortcuts, popup menus and drag-and-drop [4].

Keyboard shortcuts and popup menus dissociate the four steps of the sequence, allowing to intertwine them with other actions. This supports complex navigation tasks between the source and target locations and last-minute arrangements before pasting data. A typical problem, though, is that users often get distracted by intermediary actions and forget their initial plan. Some actions might also interfere with the pending copy or move operation. In the X Window system for example, a middle button press pastes the current selection.

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

CHI 2009, April 3 - 9, 2009, Boston, MA, USA.

Copyright 2009 ACM 978-1-60558-246-7/07/0004...\$5.00.

Although very efficient, this makes it far too easy to lose data to be copied on the way to the destination [4].

Drag-and-drop combines the last three actions of *copy-or-move* in a single, continuous, direct manipulation technique. It provides a clear visual feedback of the operation reinforced by the tension of the finger holding down the mouse button. This tension reminds the user that (s)he is in a temporary state and makes syntax and mode errors virtually impossible. But it also has some drawbacks. First, dragging is known to be slower and more error prone than pointing. Additionally, dropping errors can be difficult to correct. The necessity to hold down a mouse button seriously limits the interaction techniques usable for auxiliary tasks during the drag. Lastly, the semantics of the drop action is often unclear to the user (copy or move?), as it depends on the source and target objects.

Copy-or-move techniques were designed at a time where users had relatively few windows on screen. But today's screens are often literally filled with windows themselves filled with objects to copy and destinations to move them to. This often imposes complex navigation tasks between or inside windows and increases the risk of dropping errors. It also increases the number of auxiliary tasks users might want to perform and thus the risk they might forget their initial plan or lose selected or copied data. Existing techniques quickly show their limitations in this context. Conventional drag-and-drop is difficult to use between overlapping windows, for example, and dragging icons from or to the *desk*¹ is usually problematic.

Clipboard managers have been proposed to allow users to retrieve previously copied objects in arbitrary order, but interviews suggest these tools remain rarely used [4]. New window management techniques accessible through keyboard shortcuts and time-based interactions have been added to some systems to facilitate navigation during drag-and-drop operations, e.g. *Exposé* and the *spring-loaded folders* on OS X. But little work has been done by the HCI community on this problem. Two notable exceptions are the *Fold n' Drop* [5] and *Boomerang* [9] techniques, the former making it possible to leaf through windows during the drag-and-drop operation and the latter providing ways to suspend it and resume it later.

¹we will use this term to refer to the working area in the background of the screen that contains icons

In this note, we present several new tools and techniques designed to facilitate *copy-or-move* interactions in modern desktop environments, namely:

- a history tool that provides quick and integrated access to data previously selected, copied or dragged;
- *desk pop*, a technique that brings the user’s desk in the foreground while keeping other windows visible;
- *stack leafing*, a technique to leaf through stacks of non-overlapping windows;
- a gesture-based trailing widget that integrates *desk pop* and *stack leafing* with conventional drag-and-drop

These tools and techniques have been implemented using Metisse [3] and tested with real applications. This imposed some constraints that would have been easy to avoid in simple demonstration prototypes. This was probably the biggest challenge we faced, one that is often underestimated in this type of work. We believe this approach is extremely useful in the sense that it provides some ecological validity to our design.

SELECTION, COPY AND DRAG HISTORY MANAGEMENT

Although the services they provide seem valuable, clipboard managers remain rarely used. We believe the main reason for this is that these tools break the user’s regular interaction flow. Pasting from a clipboard manager typically follows the following steps: 1) pop-up the history list; 2) select the desired item in the list using the mouse or keyboard; 3) paste using one of the usual techniques. The interaction that brings up the history menu is critical: it needs to be carefully designed if one doesn’t want to break the user’s workflow. Unfortunately, existing clipboard managers tend to use complex keyboard shortcuts, e.g. with two modifiers, or icons and menus placed on the borders of the screen which cause a switch of attention and may require significant mouse travel.

As already mentioned, dragging is error prone and conventional drag-and-drop does not provide any particular mechanism to recover from dropping errors. A user who just accidentally released the mouse button while dragging a file icon over a set of folders may not even know where the icon actually went. On most systems in this case, an `undo` command can put the icon back in its original place. However, what the user most probably wants is to properly complete the drag-and-drop operation, not to cancel it. What is needed is thus the ability to resume or continue a drag-and-drop operation, i.e. to easily recapture objects that have been dropped.

We have implemented a history tool that keeps track of all copy and drag operations. In order to support the copy-on-selection feature of X Window, this tool also keeps track of all selections. Selection, copy and drag histories are available as three separate scrollable menus (Figure 1). The interactions used to bring up these menus were carefully chosen to be as consistent as possible with the interaction flow of the corresponding copy-or-move mechanisms.

We used time in a way similar to Hinckley et al’s Timeout delimiter [7] to smoothly integrate new interaction techniques with existing ones. A timeout was used in our imple-

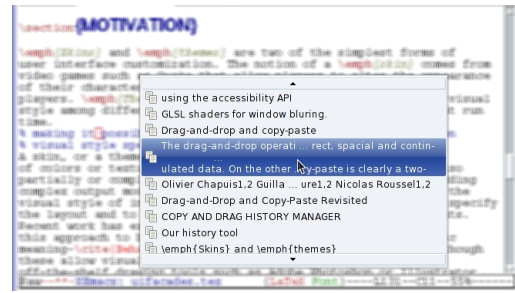


Figure 1. Sample selection history menu.

mentations to differentiate regular mouse and keyboard interaction handled by applications from special window system commands to be handled by the windowing system. The benefit of this approach is that it allows us to augment the expressive power of common interaction sequences with minimal impact on standard event handling mechanisms and in a way that preserves a clean separation between applications and the window system. The state machine shown in Figure 2 illustrates this in the case of mouse interaction.

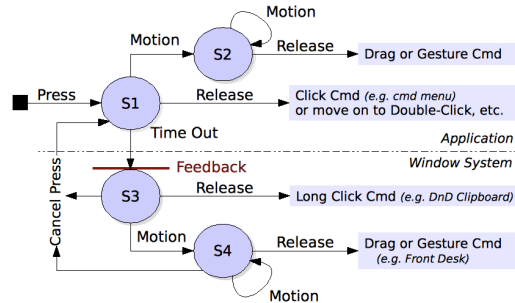


Figure 2. Timed-interactions allow a clear distinction between application level and window system interactions. Feedback is provided as the user crosses the boundary between these two levels. “Cancel Press” denotes a press on any another mouse button.

The copy history menu can be brought on screen by issuing a long `Ctrl-V`, i.e. pressing these keys for more than 250 ms, extending the well established pasting shortcut. Users can then cycle through the menu items by repeatedly pressing the `v` key while holding down the `Ctrl` key the way `Alt-Tab` cycles between windows. They can also navigate in the menu using the mouse or arrow keys. Pressing `C` or `ESC` cancels the menu, while releasing the `Ctrl` key triggers the paste command.

A long click on the middle mouse button (250 ms timeout) brings up the selection-history menu, taking advantage of the user’s familiarity with the primary-selection pasting command issued by a middle mouse button click in X Window systems. Users can navigate this menu using the mouse and paste any of its items by simply clicking on it. They can also drag an item out of it to initiate a drag-and-drop operation.

Application level commands are usually accessible through a contextual menu triggered by a right click. Our system allows the use of long right clicks or long presses along with gestures to activate window system level commands, e.g. application launching. For instance, a long right click brings up

the drag history menu. It operates like the selection-history menu except that clicking on an item starts a button-free drag-and-drop instead of pasting it. In that mode, the drop operation is triggered by a left click. An interesting aspect of our drag history mechanism is that it can be used to handle multiple interrupted drag-and-drop operations, providing a partial reification of the Boomerang technique [9]: dragging objects only a few pixels away is enough to insert them in the history list from which they can be later retrieved at a convenient time and place.

A keystroke level analysis reveals that our clipboard history implementation is at least 30% faster than traditional clipboard managers like Klipper. The application-specific Office Clipboard that automatically pops up when the paste operation is available at the insertion point is 18% faster. But our implementation offers two clear advantages: 1) it is implemented at the system level, making it application agnostic; 2) all interactions with our history tool happen at the users' locus of attention, e.g. insertion point or mouse pointer, preserving their workflow.

In order to evaluate our timeout values, we analyzed middle and right button press-release sequences recorded from nine Linux users in their everyday use of a computer [2]. The analyzed logs include 32,571 middle clicks and 65,980 right clicks (among 1,473,029 clicks). We considered the 95%-quantile of the time between the button press and the action that triggers a drag (a four pixels motion) or the button release. The middle button shows very regular values across all users: a mean and a median of 222 ms and a standard deviation of 44 ms. This confirms that a timeout of 250 ms is a reasonable value. The right button, on the other hand, reveals a wider distribution: a median of 261 ms and a mean of 417 ms with a standard deviation of 310 ms. This lead us to use a default value of 500 ms and indicates that this timeout should be configurable.

DESK POP & STACK LEAFING

A major problem with conventional drag-and-drop is the difficulty or impossibility for the user to navigate to the destination while holding objects. In this section, we present two techniques that make it possible to expose partially overlapped or fully covered windows including the user's desk and a technique that integrates the previous two with conventional drag-and-drop.

Desk pop

The desk is used by many people as a temporary storage location to provide fast access to a variable number of resources. Icons can be grouped on it in specific patterns to facilitate visual searching. Yet a major problem is that it is always displayed in the background and is thus at least partially covered by windows, which most often results in some icons being inaccessible. In order to solve this problem, most systems provide a "Show Desktop" function that temporarily moves all windows away to expose the desk. But this solution is far from ideal since it imposes a radical context change.

Our *desk pop* technique was designed to provide access to icons on the desk while preserving as much as possible of the working context. The general idea is to bring the desk in the foreground and render it in a way that lets the user see both its contents and the windows that previously covered it (Figure 3).

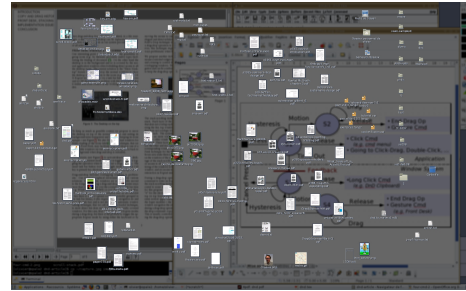


Figure 3. Desk pop: temporarily bringing the desk to the front while keeping application windows visible.

Our implementation replaces the desk's original background with a semi-transparent black one while preserving the contents original opacity. Windows displayed under the desk are blurred a little to increase icon labels readability [1] and continually remind the user that the desk now covers them. Two variants of the *desk pop* technique can be activated by long mouse gestures (Figure 2, bottom). In the first one, dragging icons automatically puts the desk back in its original place, allowing the user to drop the icons in a window. In the second one, the system enters a mode where the desk stays in the foreground until the user sends it back.

Stack leafing

Window navigation is a well known problem in overlapping-windows environments. A noteworthy solution is Apple's Exposé which tiles all opened windows or those of the active application so that they are all visible at once. But as the number of opened windows increases, the legibility of their contents decreases, making them hard to distinguish. Content-aware free-space transparency has been proposed as another solution to view and manipulate hidden content through unimportant regions of overlapping windows [8]. However, the associated navigation techniques only work for windows close to the mouse cursor.

The *stack leafing* technique we propose is based on a widget that combines generalized scrolling [10] and crossing to control the stacking order of layers of non-overlapping windows. This technique has the advantages of minimizing mouse navigation and preserving the size and position of windows while still providing access to all of their content. Layers are created by considering windows in decreasing Z order, i.e. from top to bottom, and creating a new group each time a window overlaps with the current layer. The desk is considered like any other window and is thus the bottom layer of the stack.

The *stack leafing* widget consists of a vertical series of boxes representing the various layers initially displayed under the mouse pointer. Moving the mouse over a box raises the corresponding layer, the *desk pop* technique being used in the

case of the desk. Moving the mouse pointer out of the widget makes it disappear and lets the user access the windows of the last layer exposed (Figure 4).

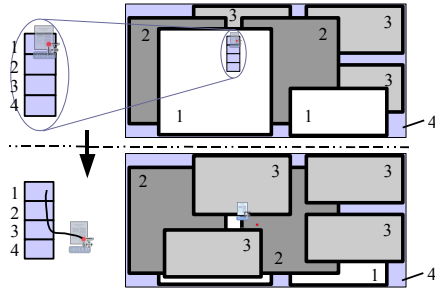


Figure 4. Stack leafing example: revealing windows of the third layer by moving down two boxes and leaving the widget.

Integration with conventional drag-and-drop

Access to techniques designed to facilitate or control drag-and-drop interactions need to be somehow integrated with these interactions. A common solution to this problem is the use of keyboard shortcuts. As an example, the semantics of the drop operation (e.g. copy, move or link to) can usually be specified using keyboard modifiers (e.g. `Ctrl` and `Shift`). In our case, pressing `d` or `s` during a drag activates the *desk pop* or *stack leafing* techniques. The same shortcuts apply when not dragging an object, with some extra modifier keys.

Using the keyboard to perform commands during a drag-and-drop is acceptable in some situations, but more direct mouse or stylus interactions might be preferable in others. We designed a new trailing widget [6] for the latter, displayed as a small red point that follows dragged objects in an elastic way. By performing a short and fast movement in the direction of this point, users can “catch” it, which reveals a pie-menu (Figure 5). This menu uses a crossing approach to provide access to drag-and-drop related commands. Leaving the pie in the top-left direction activates the *desk pop* technique. Leaving it in the top-right direction activates the *stack leafing* technique. the bottom-left direction pops up another pie menu to specify the semantics of the drop operation and the bottom-right direction was specifically chosen to cancel the menu so as to minimize unwanted activations.



Figure 5. Catching the red dot reveals the pie menu.

We have not yet conducted a detailed comparison of drag-and-drop performance between our techniques and others. This evaluation will require a thoughtful selection of window layouts, a key but intricate factor in evaluating window navigation techniques. Nevertheless, we believe that the exploratory facet of *stack leafing* offers a notable advantage over techniques like *Fold n’ Drop* [5] that require users to know what they look for beforehand to be efficient.

IMPLEMENTATION REQUIREMENTS

Besides the classical services provided by any high level GUI toolkit, implementation of the proposed tools and techniques require the following:

- Full control over mouse and keyboard input to be able to intercept and transform events before they are sent to applications. This is needed to implement the state machine described in Figure 2 and is a basic feature of *Metisse*.
- Ability to monitor high level window system operations like copy, cut and drag. This can be done by polling the window system at regular intervals. The X Window *XFixes* extension provides a more elegant solution.
- Ability to monitor file system activity to detect file moves. This is needed in order for the history tool to keep track of the location of files that have been dragged and dropped. The Linux kernel provides such a service through the *inotify* notification system.
- Full control over the rendering of the desk to implement the *desk pop* technique. In our case, *Metisse* handles all the window management and composition tasks and the desk is simply a window like any other. We use a GLSL shader to alter its rendering to produce the visual effect shown on Figure 3. In order to leave the icons untouched, we impose that the desk uses a plain background. We obtain that color by picking a pixel not covered by any icon (using the accessibility API to obtain their bounding boxes) and pass it to the shader.

REFERENCES

1. P. Baudisch and C. Gutwin. Multiblending: displaying overlapping windows simultaneously without the drawbacks of alpha blending. In *Proc. CHI’04*, 367–374. ACM, 2004.
2. O. Chapuis, R. Blanch, and M. Beaudouin-Lafon. *Fitts’ Law in the Wild: A Field Study of Aimed Movements*. Technical report, LRI, Univ. Paris-Sud, France, 2007. <http://www.lri.fr/~chapuis/publications/RR1480.pdf>.
3. O. Chapuis and N. Roussel. *Metisse is not a 3D desktop!* In *Proc. UIST’05*, 13–22. ACM, 2005.
4. O. Chapuis and N. Roussel. Copy-and-paste between overlapping windows. In *Proc. CHI’07*, 201–210. ACM, 2007.
5. P. Dragicevic. Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows. In *Proc. UIST’04*, 193–196. ACM, 2004.
6. C. Forlines, D. Vogel, and R. Balakrishnan. Hybridpointing: fluid switching between absolute and relative pointing with a direct input device. In *Proc. UIST’06*, 211–220. ACM, 2006.
7. K. Hinckley, P. Baudisch, G. Ramos, and F. Guimbretiere. Design and analysis of delimiters for selection-action pen gesture phrases in scriboli. In *Proc. CHI’05*, 451–460. ACM, 2005.
8. E. W. Ishak and S. K. Feiner. Interacting with hidden content using content-aware free-space transparency. In *Proc. UIST’04*, 189–192. ACM, 2004.
9. M. Kobayashi and T. Igarashi. Boomerang: suspendable drag-and-drop interactions based on a throw-and-catch metaphor. In *Proc. UIST’07*, 187–190. ACM, 2007.
10. R. B. Smith and A. Taivalsaari. Generalized and stationary scrolling. In *Proc. UIST’99*, 1–9. ACM, 1999.