HAL
open science

Dwell-and-Spring: Undo for Direct Manipulation

Caroline Appert, Olivier Chapuis, Emmanuel Pietriga

To cite this version:

Caroline Appert, Olivier Chapuis, Emmanuel Pietriga. Dwell-and-Spring: Undo for Direct Manipulation. Proceedings of the 30th international conference on Human factors in computing systems, May 2012, Austin, United States. pp.1957–1966, 10.1145/2207676.2208339 . hal-00663638v2

HAL Id: hal-00663638

https://hal.science/hal-00663638v2

Submitted on 13 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dwell-and-Spring: Undo for Direct Manipulation

Caroline Appert[1,2,3]
appert@lri.fr
[1]Univ Paris-Sud (LRI)
F-91405 Orsay, France

Olivier Chapuis[1,2,3]
chapuis@lri.fr
[2]CNRS (LRI)
F-91405 Orsay, France

Emmanuel Pietriga[3,1,2]
emmanuel.pietriga@inria.fr
[3]INRIA
F-91405 Orsay, France

ABSTRACT

In graphical user interfaces, direct manipulation consists in incremental actions that should be reversible. Typical examples include manipulating geometrical shapes in a vector graphics editor, navigating a document using a scrollbar, or moving and resizing windows on the desktop. As in many such cases, there will not be any mechanism to undo them, requiring users to manually revert to the previous state using a similar sequence of direct manipulation actions. The associated motor and cognitive costs can be high. We argue that proper and consistent mechanisms to support undo in this context are lacking, and present Dwell-and-Spring, an interaction technique that uses the metaphor of springs to enable users to undo direct manipulations. A spring widget pops up whenever the user dwells during a press-drag-release interaction, giving her the opportunity to either cancel the current manipulation or undo the last one. The technique is generic and can easily be implemented on top of existing applications to complement the traditional undo command. Empirical evaluation shows that users quickly adopt it as soon as they discover it.

Author Keywords

Direct manipulation; canceling; undo; dwell; spring.

ACM Classification Keywords

H.5.2 [Information Interfaces and Presentation]: User Interfaces - Graphical user interfaces;

INTRODUCTION

Direct manipulation principles stipulate that any action should be reversible. Often implemented as press-drag-release sequences, direct manipulation actions can be reverted either manually, i.e., by performing the reciprocal press-drag-release action sequence, or via an *undo* command invoked through a menu item or keyboard shortcut. In some situations, it is also possible to *cancel* an on-going press-drag-release sequence (where release has not occurred yet) by pressing a key such as `Escape`, or by releasing the mouse button while the cursor is over an area of the display where performing the release action does not make sense or is forbidden.

Undo techniques all have drawbacks. Using the keyboard or a menu item breaks the direct manipulation paradigm. The technique that consists in manually reverting the action may have a potentially high cost. From a motor perspective, the cost of repairing the action is as high as that of the manipulation that caused the error. But the cognitive cost can also be high, as the user may have to go into a great deal of effort to revert to the original state of the system/view. Examples include precisely restoring a window to its original location and size, or navigating to a distant page in a long document using a scrollbar and trying to get back to the original paragraph, as when checking a reference in a bibliography.

The *cancel* operation is rarely supported. Only a few anecdotal direct manipulation techniques exist. For instance, in some Microsoft Windows applications, it is possible to cancel navigation, initiated with a scrollbar, by moving the cursor orthogonally to the scrollbar's gutter before releasing the mouse button. However, this technique is specific to one-dimensional navigation and does not offer any kind of feedback or feedforward. Another example is implemented on Mac OS X, where the user can cancel a drag-and-drop by dropping the icon in the top menu bar. This technique also provides very limited feedback or feedforward. Moreover it might leave the cursor far from the original area of interest.

A source of possible confusion for *undo* is the lack of consistency across scenarios for a given platform. For instance, a press-drag-release interaction on an icon allows the user to both move a file to another folder on the desktop, and to change the graphical position of that icon on the desktop. However, the user can only undo the first case (usually with `Ctrl-Z`), even though these actions look very similar. Indeed, an undo command is typically available when the action to revert is at the *functional* level of the application: moving an icon to a new folder, resizing a graphical object in a drawing application, or moving a text selection in a word processor. But it is generally not available when the action only changes the state of the *view*: moving an icon on the desktop, scrolling a document, or resizing a window.

This functional level vs. view level distinction is application-dependent and is only understood by few users. Many users think, for the most part, in terms of interactions that manipulate graphical objects, but proper and consistent mechanisms to support undo and cancel at the interaction level are lacking. We present Dwell-and-Spring, a technique that uses the metaphor of springs to reify press-drag-release interactions. If the user hesitates (*dwells*) while she is manipulating a graphical object, a spring widget pops up, giving her the opportunity to either cancel the current interaction or undo the last one on that particular object. The Dwell-and-Spring technique provides feedforward and feedback through its graphi-

C. Appert, O. Chapuis, and E. Pietriga. Dwell-and-Spring: Undo for Direct Manipulation. In CHI '12: Proceedings of the SIGCHI Conference on Human Factors and Computing Systems, 1957-1966, ACM, 2012.
http://doi.acm.org/10.1145/2207676.2208339
© ACM, 2012. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in *CHI'12*, May 5–10, 2012, Austin, Texas, USA.

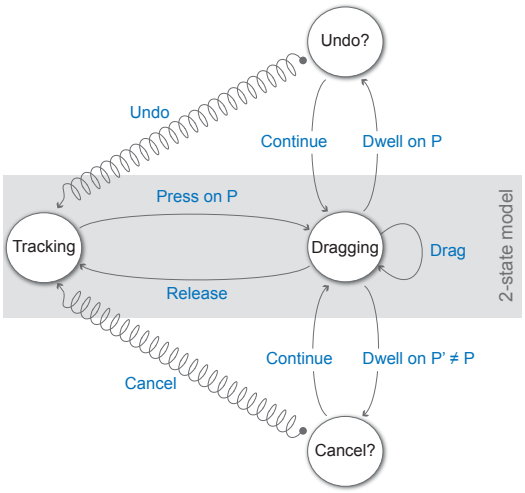


Figure 1. The 2-state model enriched with Cancel **and** Undo**. The figure represents the model for an indirect input device such as a mouse. In the case of a direct input device such as a finger, state** Tracking **would be state** Out of range**, and the transitions to, and from, state** Dragging **would be** Touch/Untouch **instead of** Press/Release**.**

cal representation, which makes the widget itself amenable to direct manipulation. The technique is generic and can easily be implemented on top of existing applications to complement the traditional undo command.

We present Dwell-and-Spring in the next section. We then report on a laboratory experiment designed to understand its potential in terms of discovery, learning and use. The fourth section describes several high-fidelity prototypes and generalizations of the basic technique. We conclude with a review of related work and directions for further research.

DWELL-AND-SPRING

The view of a system is defined by the positions of the graphical objects it contains (including the cursor) and the set of selected objects. As mentioned earlier, this view is mostly manipulated via press-drag-release interactions: drag and drop to move a graphical object, rubber-band selection, etc. These interactions all rely on Buxton's 2-state model [8], that supports press-drag-release interactions in the context of indirect input (e.g., mouse) or touch-drag-untouch interactions in the context of direct input (e.g., finger). The strength of this ubiquitous model is its simplicity. It is very easy to discover and learn, and is now deeply integrated in most users' mental model of graphical user interfaces.

However, the examples introduced earlier clearly demonstrate that we need ways to easily revert back to previous states of the view. For now, the cost of repair is high, as it basically consists in manually performing the reverse operation, requiring significant cognitive effort to figure out what was the previous state of the view and how to revert to it.

Our goal is to integrate in the 2-state model a way of navigating in the view's history without sacrificing the model's simplicity. Figure 1 depicts the model we propose. The original 2-state model is kept as is. But other states can be reached from the Dragging state through Dwell events. If dwelling occurs very close to the location of the Press event, the model

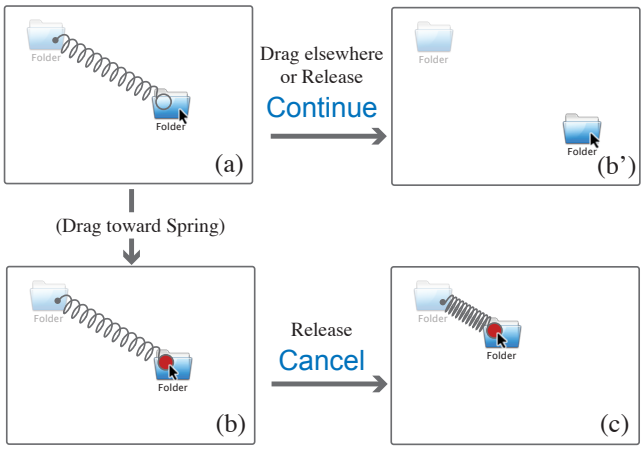


Figure 2. Cancel scenario: the user dwells while dragging an icon (a). A spring pops up. She either (b) catches the spring's handle and releases the mouse button to cancel the current drag and drop ; the spring then smoothly shrinks (c), bringing the cursor and the icon back to their original locations. Or she (b') continues dragging in any direction but that of the spring's handle. The model then gets back in the Dragging **state.**

offers users a way to easily perform the opposite of the last move action that affected the object at that location at the time of press (Undo case). If dwelling occurs after the user has started moving the cursor, the model offers a way of easily canceling that move (Cancel case). In this context, using *dwell* as a trigger is a natural choice: dwell is a temporal event that does not interfere with the input trajectory, and does not make the usual interactions captured by the original model more complicated. It is also known as an event that usually occurs in case of hesitation [24].

Beyond the model, important design choices have to be made as to how users may express *Cancel* (resp. *Undo*) and *Continue* high-level events to trigger the transitions from the Cancel? (resp. Undo?) state. These events should be:

- easy to discover and learn (C_1),
- fast to invoke (C_2), and
- not prone to accidental trigger, and easy to repair (C_3).

To address criterion C_1, we use springs as the fundamental metaphor. We find the idea of a device that returns to its former shape when released to be an appropriate depiction of actions enabled by Dwell-and-Spring. We draw a spring between the current cursor location and the location the cursor will have if the user activates a cancel (resp. undo) action. The spring is a link between two points, that can be stretched under tension, but that will recover its initial length if the tension is released. Figure 2 illustrates how we use this metaphor to express the high-level events we need. So as not to interfere with the 2-state model, the spring's handle is slightly offset from the current cursor location (10 pixels by default). The user can thus easily continue his current interaction, implicitly triggering the *Continue* event (Figure 2-(b')). This offset is combined with a temporal delay (dwell), so that Dwell-and-Spring will not interfere with most existing operations.

Figure 2-(b-c) shows how users can easily cancel an on-going manipulation (C_2), by simply catching the spring's handle and releasing the mouse button to activate the spring. This

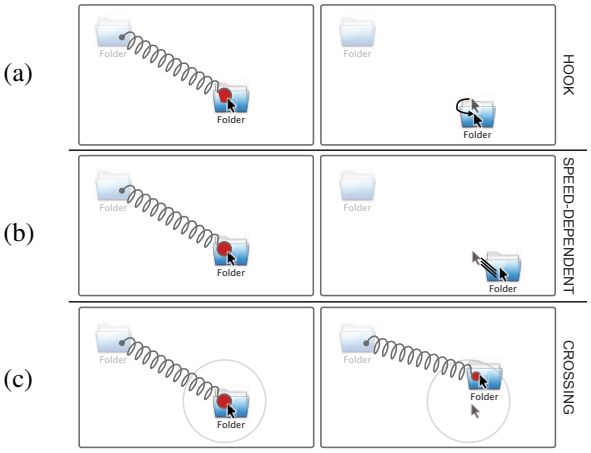


Figure 3. Three alternatives to discard a spring triggered accidentally. (a) Getting out of the handle through an aperture. (b) Moving fast enough to get out. (c) Crossing the boundary of a predefined activation area around the handle (translucent gray circle).

triggers an animation where the spring shrinks as if it were compressed indefinitely, taking the cursor and any object it may carry along, back to the spring's origin.

Requiring users to actually release the mouse button to confirm spring activation minimizes chances that the latter will be triggered accidentally, as it introduces an intermediate state that gives users an opportunity to discard the spring, after the spring has been caught, but before it actually gets activated. During that period, the spring widget behaves according to the metaphor (C_1): it gets stretched or compressed to give the impression that the cursor is pulling or pushing it by direct manipulation of its handle.

To discard the spring in case of accidental trigger (C_3), we tested three methods that all come down to getting the cursor out of the spring's handle, as illustrated in Figure 3:

- HOOK method: the spring's handle features an aperture through which the cursor can get out;
- SPEED-DEPENDENT method: the cursor can get out of the spring's handle if moved fast enough;
- CROSSING method: crossing the boundary of a predefined activation area around the spring's handle. The handle shrinks as the cursor gets further away from the activation point, providing feedback that the "link" is getting weaker.

These techniques can be fine-tuned to make the spring more or less easy to discard: the hook's aperture can be adjusted, and so can the threshold speed and activation area's radius. In our implementation, the default values are aperture = $\pi/3$, threshold speed = 1 pixel.ms^{-1} and area radius = 100 pixels. Informal tests showed that the CROSSING design was the easiest to manipulate. The HOOK design leads to accidental discarding when slightly overshooting the handle, and the SPEED-DEPENDENT one needs a fast gesture that can interfere with the intended trajectory. We thus used the CROSSING design for the evaluation described in the next section. Daily use of the technique in a window manager and experiment pilots helped us adjust dwell time to 1000ms for Cancel and to 500ms for Undo.

Implementation

Implementations of Dwell-and-Spring involve two main objects: an overlay to the interface, and a data structure keeping a history of objects that have moved. The overlay listens to events and displays the spring in response to dwell events when the mouse button is pressed. It directly re-dispatches all events to the underlying graphical component, except for release events: those are dispatched after a series of drag events, while the spring gets animated as described above, so as to simulate the sequence of events that would be received if the user had been performing the cancel/undo manipulation by herself. The data structure maintains a history of actions per object. This way, when the user dwells on a given press location, the spring offers the action that is the exact reciprocal of the last interaction this graphical object has been involved in. This enables an object-centric, sometimes called *regional*, undo. For instance, the user can resize a window to make an icon visible ($Interaction_1$), move the icon ($Interaction_2$) and then dwell on the window corner used for resizing to undo $Interaction_1$ without having to undo $Interaction_2$.

Based on this method, we developed (i) an implementation of Dwell-and-Spring for a window manager [9] (in C++); and (ii) a Java library[1] using SwingStates [3], that enables Dwell-and-Spring in any Java Swing application with a single line of code. See the Applications section for more detail.

EXPERIMENT

We conducted an experiment to capture what users typically do in situations where they want to cancel or undo a press-drag-release direct manipulation. We also wanted to evaluate whether Dwell-and-Spring is a viable alternative or not. The experiment lasted around 45 minutes and contained two parts: an interactive questionnaire to gather data about how users currently undo various representative direct manipulation actions, followed by a formal experiment to evaluate how easy it is to discover and understand Dwell-and-Spring, and how often they would actually use it once discovered.

Participants & Apparatus

Twelve unpaid volunteers (10 male, 2 female), aged 24 to 36 year-old (average 29.1, median 28.5), all daily users of personal computers, participated in this experiment. 7 used Mac OS X, 4 Microsoft Windows, and 1 an X-Window system.

Each session started with a short paper questionnaire asking participants about their familiarity with, and use of, undo operations. Nine participants said that they use the undo operation very often, two often, and one sometimes. All but one participants reported using keyboard shortcuts often (e.g., `Ctrl/Cmd-Z`). Only one said that she mainly uses a toolbar button, with five participants sometimes using such a button. One participant also mentioned using an elaborate menu to navigate in the command history of an image editor (namely Adobe Photoshop).

All sessions were conducted on a workstation with a 30" LCD monitor (2560×1600, 100 dpi, 1 pixel is about 0.256 mm in width) running Mac OS X. The mouse was a standard optical

[1] `http://insitu.lri.fr/das/`

mouse with 400 dpi resolution and default system acceleration. The software was implemented in Java.

Capturing Users' Habits

To gather data about how users cancel or undo a press-drag-release direct manipulation, we used an interactive questionnaire where participants actually played several scenarios leading to cognitive states where they want to either cancel an on-going interaction or undo that interaction right after they have completed it. To simulate this cognitive state, participants were instructed to move a graphical object to a target location highlighted on screen. We considered two cases:

- DRAGGING case: an instruction pops up in the middle of the press-drag-release interaction (i.e., the user has not yet released the mouse button) asking the participant to stop and to put the object back where she grabbed it (*Cancel*);

- DROPPED case: an instruction pops up as soon as the participant has dropped the object at the target location (i.e., the user has just released the mouse button) asking her to restore the object to its previous location (*Undo*).

In both cases, we considered four scenarios involving different graphical objects: a desktop icon (S_{icon}), a scrollbar knob ($S_{scrollbar}$), a window (S_{window}), and a geometrical shape in a vector graphics editor (S_{editor}). As mentioned before, the answer can be highly dependent on the context of use, as there is no unified way of doing such a cancel/undo operation across systems. This sample of scenarios was aimed at collecting answers representative of the different contexts of use. We also believe that asking participants to interactively *show us* what they would do in each scenario, as opposed to simply *tell us* in response to a verbal description, captures answers that have higher ecological validity.

We asked questions for the four scenarios, first in the DRAGGING case, and then in the DROPPED case. We decided to use a fixed order of presentation for the two cases because the DRAGGING case can always be solved the same way the corresponding DROPPED case is, i.e., the user can always decide to commit his current drag by releasing the mouse button and then undo it. Within both cases, the order of presentation of scenarios was counterbalanced using a Latin Square.

Our interactive questionnaire presented the user with a desktop environment where all existing techniques were made available, in all contexts we tested. This means that the drag-and-drop of any graphical object could be cancelled by right-clicking, dropping in the menu bar, or pressing the `Escape` key. Once committed (i.e., mouse button released), the user could undo the last action by either using the `Cmd-Z` keyboard shortcut or by selecting an Undo item in the menu bar (always displayed at the top edge of the screen). In scenario (S_{editor}), there was an additional possibility: an undo button in a toolbar, as most applications of this kind actually feature one. Our environment offered a kind of "ideal setting" by making all possible techniques available, whatever the scenario. Our goal was to let participants show us both what they *would like to do* (Q_1), and what they *usually do with their current system* (Q_2) in each situation.

DRAGGING case (would like to do, usually do)								
	S_{icon}		S_{window}		S_{editor}		$S_{scrollbar}$	
Manual	9	9	10	12	8	4	12	11
Escape Key	2	2	1	0	1	1	0	0
Menubar drop	0	0	0	0	0	0	0	1
Drop-then-Undo	1	1	1	0	3	7	0	0
DROPPED case (would like to do, usually do)								
	S_{icon}		S_{window}		S_{editor}		$S_{scrollbar}$	
Manual	9	11	8	11	3	1	10	11
Cmd-Z	3	1	4	1	8	10	2	1
Toolbar button	-	-	-	-	1	1	-	-
Menu item	0	0	0	0	0	0	0	0

Table 1. Strategies reported in the interactive questionnaire. Each cell corresponds to a strategy and contains two numbers: first the number of participants who effectively used this strategy in the questionnaire, second the number of participants who usually employ this strategy.

Table 1 summarizes the answers we collected. It first shows a clear difference between the graphical editor scenario (S_{editor}), which is at the functional level, and the other scenarios, which are at the view level. With S_{editor}, many more participants employed another technique than manually reverting. Ten participants reported using the `Cmd-Z` keyboard shortcut once they have DROPPED the object. Seven participants usually choose to drop and then undo when they are still DRAGGING. Very few participants used the `Escape` key and no participant used the right-click technique to abort and cancel the current action. These techniques are actually inconsistent with the original 2-state model (the user has stopped her press-drag-release interaction while the left button can still be pressed). In the three other scenarios, participants mainly restored the object to its original position manually, i.e., by performing the same action in the opposite direction. This is even more pronounced in the DRAGGING case, where participants almost never used another technique.

There was almost no difference between answers to what participants *would like* to do and answers to what they *usually* do. However, we did collect a few surprising answers. For instance, one participant said that she usually used `Cmd-Z` in the DROPPED case under all the presented scenarios while her system only supports undo for the S_{editor} scenario. This indicates that some users might expect their system to be consistent over these four scenarios. Three (resp. four) participants told us that they usually use `Cmd-Z` to move back a desktop icon (resp. window) to its original position. They might have been thinking they can do this because undo works when the user drops an icon to a new folder, i.e., a command at the functional level. This reinforces our intuition that the distinction between the functional level and the view level is not always clear to users.

Some participants made interesting comments during this interactive questionnaire. In particular, four participants told us that they would like to have an undo mechanism when scrolling, such as a button or an implicit bookmarking system. This supports findings reported in [2], that describes a scrollbar that facilitates revisitation through a set of colored marks added by the system according to the number of times a document portion has been visited. As we will see later, Dwell-and-Spring is particularly well-suited to canceling on-going, or undoing just-performed, scrolling actions.

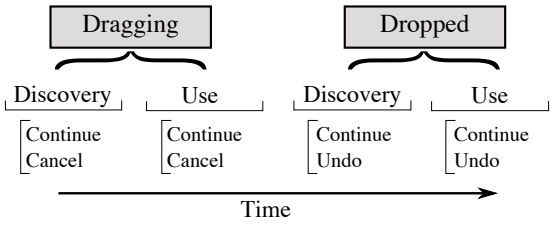


Figure 4. Design outlines of the discovery-and-use experiment.

Discovery and Use: Experiment Design

After the interactive questionnaire, participants ran an experiment whose purpose was to study the following questions:

- Is Dwell-and-Spring easy to discover?
- Will people be willing to use Dwell-and-Spring once they have discovered it?

Figure 4 outlines our experimental design. As in the questionnaire, we considered both cases DRAGGING and DROPPED. Trials were blocked by case, with the DRAGGING case always presented first. As mentioned before, we chose this fixed presentation order because the DRAGGING case can always be considered as a DROPPED case. We also expect that, in a real context of use, there should be transfer from the situations modeled by the DRAGGING case to the situations modeled by the DROPPED case. Dwelling in the middle of a movement that the user finally wants to cancel seems rather natural: consider, e.g., the scenario where the user takes a quick look at a given object in a scrollable view before coming back to the location where she was editing; or the scenario where the user temporarily moves a window to look at the graphical scene under it. We expect this case to lead to discovery of Dwell-and-Spring so that users will more easily understand they can adopt a similar approach in the DROPPED case.

To limit the length of the experiment, we only considered the desktop icon scenario (S_{icon}). The task consisted in moving the icon to a target location shown as a red rectangle. In the DRAGGING case, the participant was told before starting that she would be interrupted in the middle of her move by a pop-up message that would give her further instructions about how to finish the trial. The instruction would be either to put the icon back to its original location (*Cancel* condition) or to finish the current operation, i.e., move the icon to drop it at the intended target location (*Continue* condition). Once she had followed the new instructions, the participant had to press the Space bar to end the trial. In the DROPPED case, the participant also had to drag-and-drop a desktop icon to a target location. As soon as she had dropped the icon, she got a message asking her to either put it back where it was before she moved it, and then press the Space bar (*Undo* condition), or to just press the Space bar immediately (*Continue* condition). In both cases, when she was told to restore the icon to its original location (*Cancel* and *Undo* conditions), the instruction explicitly mentioned that "various techniques may be available" to help her.

In both cases, trials were organized into 4 blocks of 24 trials. For instance, when DRAGGING, a block contained 12 trials in the *Cancel* condition and 12 trials in the *Continue* condition, presented randomly. The 12 trials in the same condition always involved an icon of 48x48 pixels, located 800 pixels from the target location. Only the direction of movement varied across trials to take into account the fact that the spring's orientation depends on the movement direction. To vary movement direction, we laid out icons and target locations in a circular way.

There were two phases for each case: *discovery* and *use*. For the first two blocks, participants did not receive any indication about available techniques. They were simply encouraged to explore the interface. After completion of these two *discovery* blocks, the experimenter demonstrated each available technique before the participant ran into the two other *use* blocks. These last two blocks were aimed at observing what strategy participants adopted once they had been exposed to all techniques, with clear instructions about how to use them.

Because we were interested in observing how people behave with the Dwell-and-Spring technique in a traditional desktop environment, but also in contexts where the hardware does not feature additional physical buttons or keys (e.g., a touchscreen), the environment only proposed techniques that rely on "single-point input". The environment only proposed: the Dwell-and-Spring technique (*Dwell-and-Spring*), the technique that consists in dropping the icon in the top menu bar (*MenuBar*), an undo menu item (*EditMenu*) and, of course, the manual technique that basically consists in dragging the icon back manually (*Manual*).

Discovering Techniques

We first analyze data we collected in the *discovery* phase of both cases DRAGGING and DROPPED.

DRAGGING *Case*

4 out of 12 participants discovered how to use the *Dwell-and-Spring* technique. This is less than we expected since the spring popped up in 96% of the trials and we thought that the spring offered powerful feedforward. The experimenter's observations help explain this low rate of discovery: several participants only used the spring as a visual guide, and not as a reactive graphical object. The same participants spontaneously said to the experimenter that the spring was useful because it showed the icon's original position. Quantitative evidence backs this interpretation: the participants who did not discover the technique grabbed but dropped the spring in about 70% of all cases (under the *Cancel* condition; this happened in only 3% of the cases under the *Continue* condition).

The four participants who discovered the *Dwell-and-Spring* technique understood how to use it during the first block: at first try for two of them, at second and sixth try for the two others. Once discovered, they used the technique a lot: in 100%, 92%, 79% and 70% of all cases (*Cancel* condition). They also made a few errors in the first block where they activated the spring under the *Continue* condition, but no such accidental spring activation was observed in the second block.

This suggests that feedforward about spring activation should be stronger. A simple solution consists in making the spring more difficult to drop, to offer more opportunities to activate the spring (e.g., by enlarging the area where the spring is visible around the activation point or by making it more difficult to compress). Another approach would consist in removing

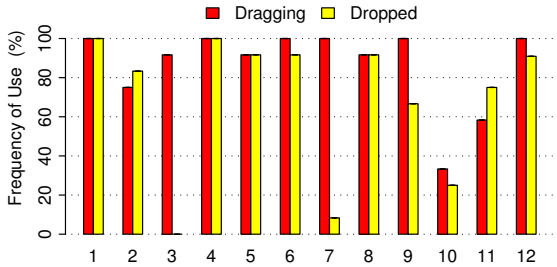


Figure 5. Use of Dwell-and-Spring in the last block for both DRAGGING and DROPPED, per participant.

the need for a release event to activate the spring (the spring would get activated as soon as the cursor enters the spring's handle). However these design solutions are in contradiction with criterion C_3, which stipulates that the spring should be easy to discard. An interesting trade-off might be to come up with a way of discarding the spring that is a function of expertise: the spring could be made difficult to drop only the first few times the user explicitly interacts with it.

DROPPED *Case*

7 participants discovered how to use *Dwell-and-Spring*. This may seem like a lot, given that contrary to case DRAGGING, the spring would not spontaneously pop up; participants had to explicitly press and dwell on the object to see the spring. But once a participant had found how to invoke the spring, they already knew how to use it as they had all learnt how to do so in the DRAGGING case. This observation tends to support our expectation of an asymmetrical transfer between the cancel and undo conditions during the experiment.

As in the DRAGGING case, participants discovered the spring technique in the first block: 2 at first try, 2 at third try, 2 at fourth try, and 1 at eighth try. Then, as in the DRAGGING case, they used the *Dwell-and-Spring* technique a lot: 95%, 92%, 68%, 100%, 88%, 87% and 86%.

Using Techniques

The above analysis reveals that users who discovered the *Dwell-and-Spring* technique made extensive use of it. In the following, we analyze data collected in the *use* phase (after *discovery*), to find out whether the other participants, who did not discover the technique by themselves, eventually adopted *Dwell-and-Spring* once exposed to it and to the other techniques (*MenuBar*, *EditMenu* or *Manual*) by the experimenter.

Frequency of Use & Qualitative Results

Figure 5 shows the frequency of use of *Dwell-and-Spring* in the last block[2] for conditions *Undo* (DROPPED case) and *Cancel* (DRAGGING case). The only other technique that was used significantly is *Manual*: *MenuBar* was used only twice and *EditMenu* was used 6 times.

Except for P_3, P_7 and P_{10}, participants used *Dwell-and-Spring* very often, with P_1 and P_4 using it systematically. The three participants who used *Dwell-and-Spring* in less than 50% of the trials in the DROPPED case said that they were not willing to wait for the spring to pop up to precisely reposition the icon, as precision did not matter much. They also stated that they would have used *Dwell-and-Spring*, had precision been an issue, e.g., had the task been to reposition a scrollbar knob. We discuss this speed-accuracy trade-off in the next section. We can also observe that the frequency of use is a bit lower in the DROPPED case than in the DRAGGING case. This is probably due to the fact that doing a long press on an object to undo its last move is less natural than making a pause during a movement the user wants to cancel.

The above results show that most participants quickly adopted *Dwell-and-Spring*. Of course, the Hawthorne effect [25] may have led to higher frequency of use than we would have observed in a real setting. However, the qualitative comments we collected at the end of the experiment were very positive and showed a real interest for the technique. Several participants spent a lot of time discussing design issues with the experimenter. Interestingly, more than half of the participants suggested that *Dwell-and-Spring* should enable users to trigger multiple undos in a single "spring step".

Errors

We define an error as a trial that ends while the icon is more than 400 pixels away from the ideal position it would have been at, had the participant correctly followed instructions (the distance between start and target icon locations is initially 800 pixels). These errors represent 2.78% of all trials in the DRAGGING case and 2.09% in the DROPPED case. In the DROPPED case, errors correspond to trials where the participant pressed the space bar before putting the icon back to its original position in the *Undo* condition (i.e., instruction ignored, possibly because of mechanical routine). In the DRAGGING case, 75% of all errors were made under the *Cancel* condition: 75% of those correspond to trials where the participant ignored the instruction, and 25% to trials where the participant dropped the spring before activating it. Errors in the *Continue* condition correspond to trials where the participant activated the spring while she should have continued her current interaction. In the DRAGGING case, the overall error rate caused by springs is about 1%.

We also recorded occurrences of accidentally spring grabbing in the *Continue* condition. This happened in 3.86% of the trials in the DRAGGING case and in only one trial in the DROPPED case. All these trials ended without any error, indicating that participants were able to drop the spring. In the DROPPED case, a cancel spring popped up in about 6.62% of the trials under the *Continue* condition (typically just before the end of the task), but participants never activated it. These observations tend to show that the Dwell-and-Spring technique fulfills design criterion C_3 (minimize accidental triggers and enable easy repair).

Movement Time & Precision

Figure 6 shows movement time and precision (distance between the icon's original location and its position at trial end time) for trials in the *Cancel* and *Undo* conditions, after having removed the errors mentioned above. We do not use any statistical test (e.g., ANOVA) on purpose. We designed an observational experiment, not a strict experimental protocol to quantitatively compare *Manual* and *Dwell-and-Spring*.

[2]We analyze data in the last block only, as it is more likely that participants had made a "definitive" choice by then.

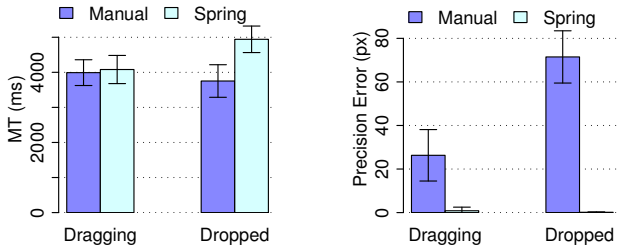


Figure 6. Movement time (left) and precision error in pixels (right) for *Manual* and *Dwell-and-Spring*, in both cases DRAGGING and DROPPED (under *Cancel* and *Undo*). Error bars show the 95% confidence limits.

In the DRAGGING case, we observe very similar movement time for *Dwell-and-Spring* and *Manual*, and a better precision (distance close to zero) for *Dwell-and-Spring* than for *Manual*. In the DROPPED case, *Manual* was about 1.2 seconds faster than *Dwell-and-Spring*, but *Manual* was far less precise than *Dwell-and-Spring*. It is not surprising that *Dwell-and-Spring* offers a much better precision since its implementation performs the ideal reverse manipulation, putting the object back to its exact original location. The average precision error with *Manual* was 71.5 pixels (median 69 pixels). There is thus a trade-off between movement time and precision when comparing *Dwell-and-Spring* and *Manual* in the DROPPED case. Note that precision error in the DRAGGING case is somewhat lower than the one we would have observed in a more realistic context of use, as the spring visually helped participants recover the exact icon's location.

Summary

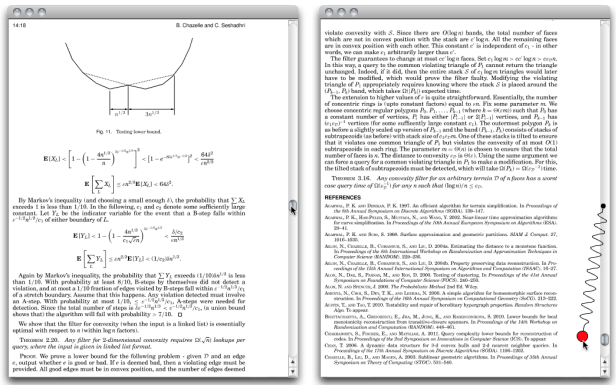
Collecting users' habits in different contexts of use revealed that they always repair their direct manipulation errors manually, except when the direct manipulation acts at the functional level of the corresponding application. Observing users when they are in an environment where *Dwell-and-Spring* is available revealed that one third of users spontaneously tried to make use of it, and that demonstrating the technique even a single time is sufficient for users to understand and adopt it. Our quantitative analysis highlighted the speed-accuracy trade-off that users may face with such a technique. While it may be a bit slower in some cases, *Dwell-and-Spring* allows users to accurately cancel or undo a direct manipulation, which can be a significant advantage for precise positioning.

APPLICATIONS

The basic Dwell-and-Spring technique readily applies to many cases of direct manipulation: manipulating icons on the desktop, moving and resizing windows, navigating documents using a scrollbar, or any other action where the spring's actions are equivalent to what the user would manually do to revert to the original state. The metaphor can also be extended to more advanced cases. We first give examples of application in a WIMP environment, and then discuss extensions to the original design to support more advanced interactions that facilitate other actions associated with direct manipulation, such as text selection and editing.

Manipulating Widgets

The technique straightforwardly applies to widgets such as sliders and scrollbars. Figure 7 illustrates a simple yet powerful example, where Dwell-and-Spring is used to navigate back and forth between two distant pages in a document. Going back to the original page would typically be a tedious task as the scrollbar knob would have to be positioned very precisely. Dwell-and-Spring makes such a task very easy by enabling the user to undo her last manipulation of the scrollbar. Similarly, the technique can be used to freely browse the values of a variable controlled by a slider and effortlessly revert to the original setting. This can be very useful when performing dynamic queries or any other visual analysis task.



(a) (b)

Figure 7. While reading a document, a user wants to have a quick look at a reference cited on the current page. (a) She drags the scrollbar knob to the end of the document. (b) Once she has checked this reference, she dwells on the knob to invoke a spring that will automatically take her back to the page she was reading.

One issue with the straightforward application of the basic Dwell-and-Spring technique to these uni-dimensional widgets is that the spring's hook can be in the way of the cursor if the user chooses to reverse course but does not want to activate the spring. This can be especially bothersome when the user's focus of attention is elsewhere than on the widget itself, as is typically the case when scrolling a document (attention focused on the pages) or performing dynamic queries (attention focused on the result-set). To avoid the spring becoming an annoyance, its hook is offset orthogonally as illustrated in Figure 7-b. With this minor change, the spring's hook no longer interferes with direct manipulation of the scrollbar knob, even when reversing course.

To further address potential problems of accidental triggering due to the focus of attention not always being on the widget, the spring is discarded, even if acquired, when the user dwells a second time after its appearance. Indeed, such a second dwell likely means that the user is focusing on another part of the interface and did not intend to activate the spring.

Manipulating Objects and Navigating Spaces

In addition to widgets, Dwell-and-Spring can be used to undo direct manipulations of arbitrary objects in applications such as vector graphics editors like Adobe Illustrator or InkScape. This is especially useful in cases where the user wants to restore the manipulated object to its exact former position. Similarly, Dwell-and-Spring applies directly to pan & zoom navigation. Panning a map or image can be seen as a displacement of the map *object*, displacement that can be undone as simply as that of any other object.

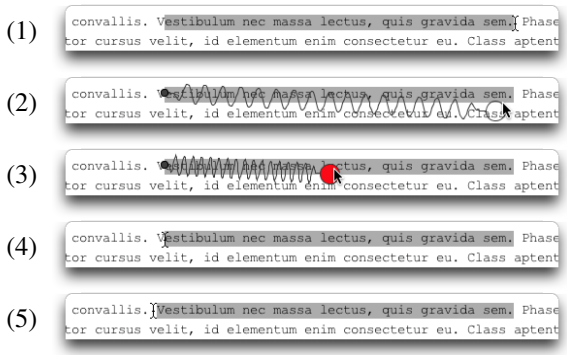


Figure 8. Adjusting a text selection: (1) the user starts selecting a sentence, but misses the first character. (2) She dwells to get a spring. (3) Entering the spring handle immediately triggers the undo action: (4) the cursor gets sent to the selection's start point coordinates; that point becomes the active text selection endpoint controlled by the cursor. (5) The user can freely adjust her selection to include the missing character, all this in a single press-drag-release sequence.

Dwell-and-Spring maintains a per-object history of direct manipulations that enables users to undo actions on specific objects regardless of when they happened, as in *regional undo* [21]. The user simply selects and dwells on the object to be reverted. For instance it is possible to remove a first object from an alignment of objects, then a second object, and finally revert the first object to its aligned position without having to undo anything about the second object. Dwell-and-Spring can also address the problem of editing occluded objects [28]. The user simply moves the occluding objects so as to be able to edit the object of interest, and then restores the objects' positions by activating a spring on each one of them.

Text Selection

Dwell-and-Spring can also be extended to support more elaborate scenarios that go beyond a simple undo. For instance, when selecting text, it is not uncommon for the user to miss the first character of the string she actually intended to select. She will often realize this only after the cursor has reached the selection's end point. Dwell-and-Spring can offer an alternative to the tedious solution that consists in starting the selection from scratch.

Figure 8 shows how the spring's behavior can be modified so that the user can switch back and forth between selection endpoints and adjust them at will. The spring gets activated as soon as the cursor enters the spring's handle (no need to release the mouse button) and the new cursor position becomes the current selection end point. This mechanism can apply to any rubber band selection, enabling smooth adjustment of endpoints in a single press-drag-release sequence.

Window Management & Buttons

The examples presented above were implemented with our Java library. This library allows the developer to put an overlay on top of any Java Swing application, enabling Dwell-and-Spring for any press-drag-release interaction. In order to assess the value of Dwell-and-Spring in a more ecological setting, we implemented it in an actual window system, Metisse [9], and used it for two months. We wanted to find out how much Dwell-and-Spring can improve interaction for each possible window operation that usually does not support

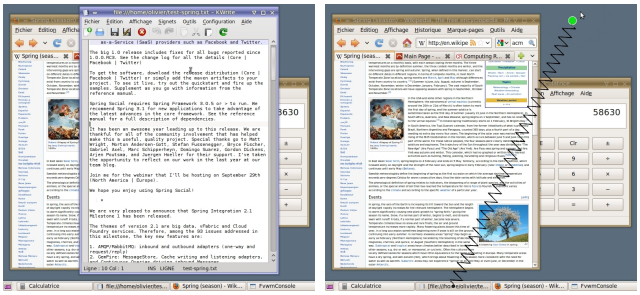


Figure 9. Pressing the Iconify button and dwelling makes a spring pop up, which allows the user to easily cancel the iconification operation.

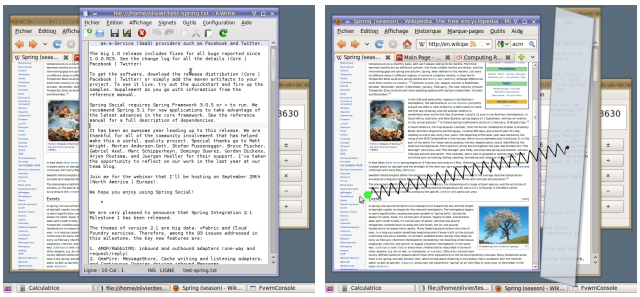


Figure 10. Pressing the mouse button in a window overlapped by another makes the latter roll to reveal the former. Dwelling triggers a spring that can be used to restore the windows' original z-ordering.

undo: move, resize, iconify, and z-reordering, i.e., bring a window to front.

Move and resize operations rely on standard press-drag-release interactions, and are compatible with the original design of Dwell-and-Spring. Iconify and z-reordering commands are more challenging, because the user usually invokes them through interactions located on a single point (a click or a press). The spring metaphor can apply to such operations by thinking in terms of displacement of graphical objects as explained below.

To iconify a window, the user clicks on the Iconify button located in the window's title bar. Even though the user's action is located on the button, the operation can be seen as if the window were shrinking and moving to the taskbar. Restoring the original position and size of the window is achieved through a click on its taskbar icon. In our implementation, the window is smoothly animated into an icon as soon as the user presses the Iconify button. If she waits a little with the mouse button still pressed (dwelling), a spring shows up to link the current cursor position to the taskbar icon (Figure 9). Grabbing the spring's handle and releasing the mouse button then restores the window to its state before it got iconified. Contrary to the basic Dwell-and-Spring design, which continuously attracts the cursor toward the spring's origin, the cursor location remains unchanged so as to stay close to the window of interest (the spring actually attracts the window).

Interpreting a z-reordering operation as a displacement of a local part of a window makes the operation amenable to undo with the spring metaphor. As described in [10], when the user presses the mouse button on an overlapped window and starts dragging, the top window rolls to make the selected window fully visible (Figure 10-b). The rationale for not putting the

top window below is that it may be the destination of a press-drag-release interaction (drag-and-drop). In our implementation, the user can dwell to make a spring pop up, that will allow her to roll back the window on top. Note that if the press is immediately followed by a release without any drag, the window rolls back behind the selected window.

This new way of implementing operations to iconify and change windows' depth allows users to have a look at what is behind a window without changing the current layout of all the opened windows. It provides a low-cost alternative to the technique described in [6], where the user has to manipulate the windows' borders, which are rather small targets. The above examples also show how Dwell-and-Spring can apply to buttons that change the view. This may, however, require the introduction of additional graphical feedback as we did with the window rolling effect in the last example.

RELATED WORK

The introduction of direct manipulation in our interfaces has enabled users to easily invoke commands. The need for an undo mechanism then rose very quickly. Indeed, the first personal computer, the Xerox Star [20], already had a dedicated undo physical key. This means of trying some commands while relying on the assumption that they can always be undone allows users to adopt an exploratory behavior [12]. While an intensive use of the undo command for a problem solving task is not surprising, it can also give an indication of the usability of an interface by detecting critical incidents in other contexts [1]. The undo mechanism in an interface is thus more than a simple command and has received a lot of attention in both academia and industry.

Most applications implement a linear model of undo. The application manages a stack of *command objects* that implement both the effect and the reverse effect for that command. The user can thus navigate in the application history by sequences of undo and redo actions. A few applications exhibit this stack as a textual list so that users can directly jump back to a given state (e.g., Adobe Photoshop). However, this has the effect of undoing all the commands that were invoked after it. Some research tools offer a more sophisticated model of undo. For instance, selective undo [7] allows users to isolate a command in the history so that the revert operation will be pushed on top of the current state. This mechanism requires a sophisticated implementation so as to propose only the commands that make sense for the current state. Edwards et al. [14] also propose a clever implementation to handle undo in Flatland, an application that allows users to specify behaviors a posteriori, that will reinterpret past input. The approach consists in introducing commands that will be nested with past commands already stored in the history. Chimera [23] exhibits the history as a list of graphical panels that allow users to edit an object in the history. Chimera propagates these changes to the current state. Collaborative environments also raise specific challenges because they involve both shared and individual history so that tools may even propose *time* as a first class object (e.g., [15] and [30]).

All these approaches require a complex model of the history of commands of an application, while the model of history of Dwell-and-Spring is a simple list of per-object displacements. All the tools and applications mentioned above work at the functional level, while we work at the interaction level. Those levels do not interfere, since Dwell-and-Spring actually simulates direct manipulations of objects users would have manually performed to undo a previous manipulation on top of the current state. If this movement corresponds to a command at the functional level, it will be integrated in the application history. Also, keeping a per-object history makes Dwell-and-Spring a kind of spatial selective undo by allowing users to undo the displacements of a given object without undoing more recent displacements of other objects. Some editors also support such *regional undo* by making the command apply within a certain region (e.g., Emacs, DistEdit [27], and the spreadsheets described in [21]).

Complementary to the above, several tools propose advanced visualizations of the interaction history. For example, Chronicle [18] or the application-independent approach presented in [26] propose visualizations of the user's workflow by emphasizing relevant areas to better explain the current state. These tools are very powerful to visualize, and thus reflect on, the current state to either communicate or retrieve sequences of interest. However they are not intended to support undo navigation by, e.g., restoring a previous state as Rekimoto envisioned with the Time-Machine concept [29].

While the visualization tools mentioned above focus on long term history, Phosphor [5] proposes a lightweight visualization of the recent interactions by showing an afterglow that explains the transition made. Phosphor is close to Dwell-and-Spring in that it can facilitate the manual undo of a recent action. However, it does not propose interactive features to support these undo operations and provides support for only very recent interactions. On the opposite, UIMarks [11] enables users to explicitly leave some marks on the user interface that facilitate going back to specific positions. Dwell-and-Spring lies in-between those two approaches: it implicitly records the start and end points of any press-drag-release with which users can interact by using a dwell time to trigger a widget. Using the time dimension during a drag-and-drop to avoid having to rely on an additional modality (e.g., the keyboard) is not new and is, for example, used in some systems to reveal the content of a folder when dwelling over its icon. Another example is Scriboli [19], that suggests the use dwelling after a lasso selection to pop up a contextual menu.

Allowing users to interact during a press-drag-release interaction can also be addressed with other approaches such as crossing or gesture dynamics. For instance, Fold-and-Drop [13] proposes to cross window borders to fold windows during a drag-and-drop to facilitate navigation over windows. The techniques presented in [16] rely on the use of a trailing widget [17], which can be grabbed with a quick movement, to access a menu. Boomerang [22] allows to suspend a drag-and-drop by using a throwing gesture. Finally, while Dwell-and-Spring facilitates interaction with previous states, Drag-and-pop [4] accelerates interaction with the future by using the direction of movement to predict and bring the potential targets close to the dragged object.

CONCLUSION

In this paper we present Dwell-and-Spring, a novel approach for undoing and canceling direct manipulation actions. The key ideas consist in using dwell events, that do not interfere with standard press-drag-release interactions, and a spring widget that users can directly manipulate. Press-drag-release interactions being ubiquitous in current interfaces, Dwell-and-Spring can improve the usability of many applications, as we demonstrated with actual implementations. Our empirical evaluation showed that users adopt it and appreciate it as soon as they discover it. We plan to run a field study that will focus on the potential distractions Dwell-and-Spring can cause. However we envision an implementation where the more a user uses the spring, the more transparent it gets. The analogy with a physical spring is especially useful in the discovery phase of the technique, but it probably becomes less so when the user actually knows how to use it and wants to optimize time.

As a first step, we focused on cancel and undo operations. An a priori straightforward generalization is the inclusion of redo by making two springs pop up in case a redo makes sense. However, it can be confusing to users, and alternatives need to be designed and studied. We also plan to explore how Dwell-and-Spring can provide users with the ability to undo several steps at a reduced cost by refining the interaction with the spring's handle, which could for instance interpret some specific gestures. Finally, implicit graphical objects such as groups or selections are not handled by Dwell-and-Spring. Implementing a clever history mechanism that makes these objects explicit is an interesting challenge.

ACKNOWLEDGMENTS

This work has been partially funded by ANR (ANR-11-JS02-004-01). We also thank David Bonnet, Wendy Mackay and Michel Beaudouin-Lafon for fruitful discussions.

REFERENCES

1. Akers, D., Simpson, M., Jeffries, R., and Winograd, T. Undo and erase events as indicators of usability problems. In *Proc. CHI '09*, ACM (2009), 659–668.
2. Alexander, J., Cockburn, A., Fitchett, S., Gutwin, C., and Greenberg, S. Revisiting read wear: analysis, design, and evaluation of a footprints scrollbar. In *Proc. CHI '09*, ACM (2009), 1665–1674.
3. Appert, C., and Beaudouin-Lafon, M. SwingStates: Adding state machines to Java and the Swing toolkit. *Software Pract. Exper. 38*, 11 (2008), 1149–1182.
4. Baudisch, P., Cutrell, E., Robbins, D., and Czerwinski, M. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch-and pen-operated systems. In *Proc. INTERACT '03*, IOS & IFIP (2003), 57–64.
5. Baudisch, P., Tan, D., Collomb, M., Robbins, D., Hinckley, K., Agrawala, M., Zhao, S., and Ramos, G. Phosphor: explaining transitions in the user interface using afterglow effects. In *Proc. UIST '06*, ACM (2006), 169–178.
6. Beaudouin-Lafon, M. Novel interaction techniques for overlapping windows. In *Proc. UIST '01*, ACM (2001), 153–154.
7. Berlage, T. A selective undo mechanism for graphical user interfaces based on command objects. *ACM ToCHI 1*, 3 (1994), 269–294.
8. Buxton, W. A three-state model of graphical input. In *Proc. INTERACT '90*, North-Holland (1990), 449–456.
9. Chapuis, O., and Roussel, N. Metisse is not a 3D desktop! In *Proc. UIST '05*, ACM (2005), 13–22.
10. Chapuis, O., and Roussel, N. Copy-and-paste between overlapping windows. In *Proc. CHI'07*, ACM (2007), 201–210.
11. Chapuis, O., and Roussel, N. UIMarks: quick graphical interaction with specific targets. In *Proc. UIST '10*, ACM (2010), 173–182.
12. Dix, A., Mancini, R., and Levialdi, S. Alas i am undone-reducing the risk of interaction. In *HCI '96 Adjunct Proceedings*, London Imperial College (1996), 51–56.
13. Dragicevic, P. Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows. In *Proc. UIST'04*, ACM (2004), 193–196.
14. Edwards, W. K., Igarashi, T., LaMarca, A., and Mynatt, E. D. A temporal model for multi-level undo and redo. In *Proc. UIST '00*, ACM (2000), 31–40.
15. Edwards, W. K., and Mynatt, E. D. Timewarp: techniques for autonomous collaboration. In *Proc. CHI '97*, ACM (1997), 218–225.
16. Faure, G., Chapuis, O., and Roussel, N. Power tools for copying and moving: useful stuff for your desktop. In *Proc. CHI '09*, ACM (2009), 1675–1678.
17. Forlines, C., Vogel, D., and Balakrishnan, R. Hybridpointing: fluid switching between absolute and relative pointing with a direct input device. In *Proc. UIST'06*, ACM (2006), 211–220.
18. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: capture, exploration, and playback of document workflow histories. In *Proc. UIST '10*, ACM (2010), 143–152.
19. Hinckley, K., Baudisch, P., Ramos, G., and Guimbretiere, F. Design and analysis of delimiters for selection-action pen gesture phrases in scriboli. In *Proc. CHI'05*, ACM (2005), 451–460.
20. 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* (1989), 11–26, 28–29.
21. Kawasaki, Y., and Igarashi, T. Regional undo for spreadsheets. UIST '05 Demonstration abstract, 2004.
22. Kobayashi, M., and Igarashi, T. Boomerang: suspendable drag-and-drop interactions based on a throw-and-catch metaphor. In *Proc. UIST '07*, ACM (2007), 187–190.
23. Kurlander, D., and Feiner, S. A history-based macro by example system. In *Proc. UIST '92*, ACM (1992), 99–106.
24. Kurtenbach, G., and Buxton, W. User learning and performance with marking menus. In *Proc. CHI '94*, ACM (1994), 258–264.
25. Landsberger, H. *Hawthorne Revisited*. Cornell University, Ithaca, 1958.
26. Nakamura, T., and Igarashi, T. An application-independent system for visualizing user operation history. In *Proc. UIST '08*, ACM (2008), 23–32.
27. Prakash, A., and Knister, M. J. A framework for undoing actions in collaborative systems. *ACM ToCHI 1*, 4 (1994), 295–330.
28. Ramos, G., Robertson, G., Czerwinski, M., Tan, D., Baudisch, P., Hinckley, K., and Agrawala, M. Tumble! splat! helping users access and manipulate occluded content in 2d drawings. In *Proc. AVI '06*, ACM (2006), 428–435.
29. Rekimoto, J. Time-machine computing: a time-centric approach for the information environment. In *Proc. UIST '99*, ACM (1999), 45–54.
30. Rhyne, J. R., and Wolf, C. G. Tools for supporting the collaborative process. In *Proc. UIST '92*, ACM (1992), 161–170.