Reciprocal Drag-and-Drop

Caroline Appert, Olivier Chapuis, Emmanuel Pietriga, Maria Lobo

▶ To cite this version:

Caroline Appert, Olivier Chapuis, Emmanuel Pietriga, Maria Lobo. Reciprocal Drag-and-Drop. ACM Transactions on Computer-Human Interaction, 2015, 22 (6), pp.29:1–29:36. 10.1145/2785670 . hal-01185805v2

HAL Id: hal-01185805
https://hal.science/hal-01185805v2

Submitted on 30 Sep 2015

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.

Reciprocal Drag-and-Drop

CAROLINE APPERT, CNRS & Univ. Paris-Sud, Inria
OLIVIER CHAPUIS, CNRS & Univ. Paris-Sud, Inria
EMMANUEL PIETRIGA, Inria, Inria Chile, Univ. Paris-Sud & CNRS
MARÍA-JESÚS LOBO, Inria, Univ. Paris-Sud & CNRS

Drag-and-drop has become ubiquitous, both on desktop computers and touch-sensitive surfaces. It is used to move and edit the geometry of elements in graphics editors, to adjust parameters using controllers such as sliders, or to manage views (e.g., moving and resizing windows, panning maps). Reverting changes made via a drag-and-drop usually entails performing the reciprocal drag-and-drop action. This can be costly as users have to remember the previous position of the object and put it back precisely. We introduce the DND^{-1} model that handles all past locations of graphical objects. We redesign the Dwell-and-Spring widget to interact with this history, and explain how applications can implement DND^{-1} to enable users to perform reciprocal drag-and-drop to any past location for both individual objects and groups of objects. We report on two user studies, whose results show that users understand DND^{-1}, and that Dwell-and-Spring enables them to interact with this model effectively.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces - Graphical user interfaces

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Undo model; Direct manipulation; Dwell-and-Spring.

ACM Reference Format:
Caroline Appert, Olivier Chapuis, Emmanuel Pietriga and María-Jesús Lobo, 2015. Reciprocal Drag-and-Drop. *ACM Trans. Comput.-Hum. Interact.* 22, 6, Article 29 (September 2015), 36 pages.
DOI: http://doi.acm.org/10.1145/2785670

1. INTRODUCTION

Most graphical user interfaces rely heavily on drag-and-drop interactions for view management. Drag-and-drop is the primary method for moving and resizing windows on a desktop, for laying out icons, for panning a map or a very large image, for browsing a long document. But objects manipulated via drag-and-drop often have to be restored to one of their previous positions. For instance, a user will carefully lay out windows on his desktop but will then temporarily move or resize one of them to access content hidden behind it, such as an icon or another window of lesser importance that was left in the background; he will then want to restore the foreground window to its earlier configuration. The reader of a document will scroll down to an appendix or check a reference, and will then want to come back to the section he was reading. Current systems do not enable users to easily restore windows or viewports to their earlier configuration; users have to manually reposition and resize the corresponding objects. Such actions can be costly. From a motor perspective, the cost of *repairing* a drag-and-drop manipulation can be higher than that of the original manipulation depending on how precisely the object has to be positioned. This is especially true for touch-based

Authors' address: Université Paris Sud - Bat. 660, 91405 ORSAY Cedex, FRANCE.

© ACM, 2015. 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 was published in ACM Transactions on Computer-Human Interaction, ToCHI, 22(6), Article No. 29, 36 pages, September 2015. DOI: http://doi.acm.org/10.1145/2785670

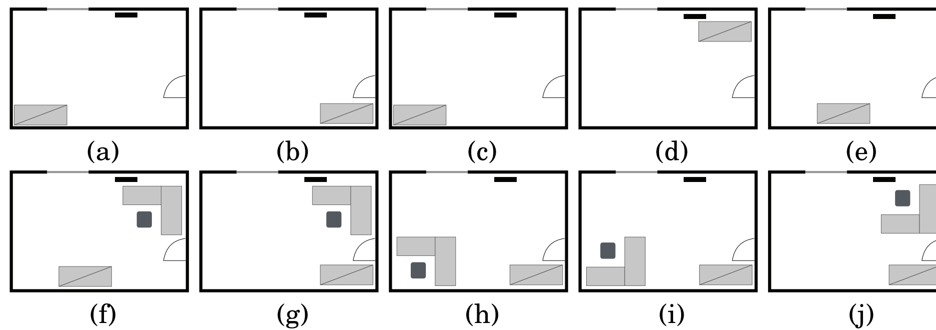


Fig. 1. Exploring different office layout alternatives on a floor plan. (a) Placing a cupboard in the SW corner. (b) When moving the cupboard to the SE corner, it is difficult to access it when the door is open. (c) Cupboard back to the SW corner. (d) Cupboard in the NE corner. The heater is partially occluded. (e) Cupboard almost centered along the S wall. (f) Adding a desk in the NE corner, composed of two tables and a chair. The heater is partially occluded. (g) Cupboard back in the SE corner to free space for the desk in the SW corner. (h) Desk in the SW corner. (i) Changing the relative placement of the desk elements. (j) Desk back in the NE corner with the new relative layout between the two tables and the chair.

interfaces, which can make precise manipulations challenging [Siek et al. 2005]. The cost can also be high from a cognitive perspective, as users may have difficulty remembering what was the previous state of a particular object [Katifori et al. 2008].

Users also rely heavily on drag-and-drop for content manipulation in WYSIWYG applications such as vector graphics editors and slide show presentation programs. The precise positioning of elements is particularly important in such contexts, where users perform advanced graphical layout task. But it can be challenging. For instance, graphical shapes vary in their size and may be very close to one another. Accidental selections are likely to occur, and users may want to revert a subset of objects, selected by mistake, back to their original position; and this without having to cancel the manipulation for the objects they actually had intended to move. Some shapes can also overlap other shapes, or even completely cover them. While the shapes below can be visible through alpha-blending, they will often be difficult to access. In this situation, users will often temporarily move the shape on top to access the ones below and modify them, but will eventually want to revert to the original layout.

From a user perspective, such graphical layout tasks are often part of an exploratory process. For instance, Figure 1 illustrates a scenario in which a person rearranges furniture in an office and tests alternative layouts. The software allows her to explore different arrangements by selecting and moving either a single piece of furniture, or multiple pieces together. Direct manipulation strongly contributes to making such exploratory design activities easy. But effectively supporting users also entails enabling them to easily revert back to past states from which to try other design options. Most graphical editing software provides an undo command to restore a past state of the entire document but, unfortunately, the underlying undo model is usually a global linear one that does not keep track of branches in the history of manipulations. Such a basic undo mechanism has two strong limitations, as detailed below.

The first limitation is that some previous states in the history can become inaccessible [Berlage 1994; Yoon et al. 2013]. If a user applies a command that turns state A into state B, reverts back to state A using undo, and then applies a new command that turns state A into state C, she will no longer be able to get back to state B. For instance, in Figure 1, the user moves the cupboard (a-b) but then undoes this move (b-c) when she realizes that this location might not be so convenient because of its proximity to the door. Later, after having considered the different constraints (window,

heater, additional furniture), she finally decides that putting the cupboard behind the door (as in (b)) is the best option. She wants to revert it to this location, but as she has moved it to other locations (c-d-e) after her undo operation (b-c), she is no longer able to get back to this configuration other than by manually moving it back there.

The second limitation comes from the lack of integration of object selection mechanisms with the history of direct manipulations. When adjusting layouts, users often want to apply direct manipulation actions to multiple objects simultaneously, typically selected using a rubber-band rectangle or by clicking on all objects in turn while keeping a modifier key pressed. Multiple selection allows users to manipulate groups of objects simultaneously while preserving their relative layout. But this notion of group is transient, as graphical editors usually support only one active selection at a time. Undoing an action performed on multiple objects will no longer be possible once the selection has changed. Users then have to select these objects again, and manually revert them to their earlier position using the reverse drag-and-drop action. Coming back to Figure 1, the user moves the two tables and the chair that make her workstation (g-h), and then changes their relative layout, thus breaking the previous multiple selection (i). Because there can only be one single active selection at a time, testing a location of the workstation that has already been explored (f), but with the new relative layout made in (i-j), requires selecting all its elements again and manually dragging-and-dropping them in the right place. Some graphical editors feature a command to group objects together. But this makes the exploratory design process much more cumbersome, as groupings have to be anticipated and created explicitly. In addition, groupings set persistent links between objects, which impede single-object editing operations.

In all examples above, users have to put back individual objects or groups of objects to one of their past locations. In other words, they have to perform *reciprocal drag-and-drop actions*. Such actions occur frequently, and their associated cognitive and motor cost can be high. We present the DND^{-1} history model, that keeps track of all past locations for both individual and multiple object selections. DND^{-1} is based on a direct selective undo model for drag-and-drop actions that works for all past selections of both single and multiple objects. We extend the Dwell-and-Spring widget, that was introduced in [Appert et al. 2012], to let users quickly restore single objects and groups of objects to any past location[1]. DND^{-1}, together with this extended version of the Dwell-and-Spring widget, better support exploratory tasks in direct manipulation interfaces, as they let users revert arbitrary objects to a previous configuration, while preserving the result of drag-and-drop actions that happened later in the editing process. For instance, DND^{-1} lets users perform the edits described in Figure 1 in a lesser number of actions, as they can undo actions performed on a selection of objects without having to undo the direct manipulations that were performed later on in the workspace.

We first describe the DND^{-1} model and detail how we extended the Dwell-and-Spring widget to support reciprocal drag-and-drop. We illustrate DND^{-1} on both simple and advanced cases, and discuss implementation details. Finally, we report on two user studies which indicate that: (i) users are faced with situations in which they would like to have better support for reciprocal drag-and-drop, (ii) they can successfully use the Dwell-and-Spring widget in such situations and (iii) they can take advantage of DND^{-1} to solve advanced layout problems with reciprocal drag-and-drop.

[1]The original Dwell-and-Spring technique was supporting only a basic linear undo model for single objects.

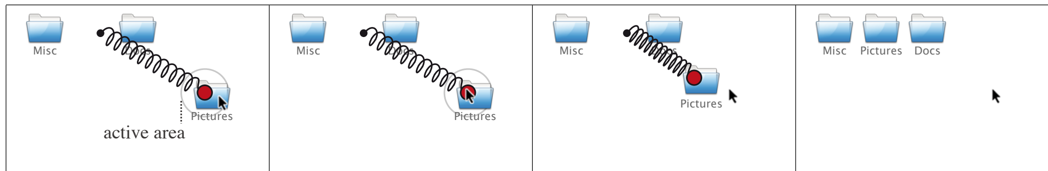


Fig. 2. The Dwell-and-Spring technique (DS). A red circular handle pops up close to the cursor when the user presses the mouse button and remains still for 500ms (i.e., dwells) over an icon. Releasing the mouse button while the cursor is over the spring handle will undo the last move of this icon.

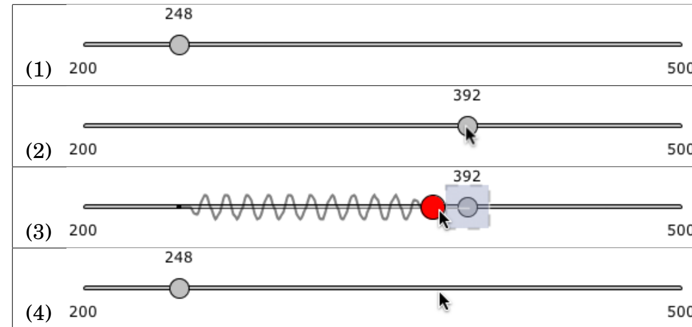


Fig. 3. Evaluating the impact of different values of a parameter controlled with a slider. (1) The current value of the slider is 248. (2) The user tests value 392. (3) Unsatisfied with it, he sets the value back to 248.

2. RECIPROCAL DRAG-AND-DROP

Situations that call for reciprocal drag-and-drop can be simple: for instance, putting a window back to its last location or reverting it to its previous size. They can also be much more elaborate: for instance, putting back a group of shapes to an earlier position on the drawing canvas after having manipulated other shapes, and putting them back there while preserving the new relative position that was given to them after they were initially moved away. This section illustrates both simple and more advanced situations, and explains how we have redesigned the Dwell-and-Spring widget on top of the DND^{-1} model to provide users with a flexible and powerful way of reverting various types of drag-and-drop actions (the companion video shows the technique in action).

2.1. Last Location of an Object

The basic Dwell-and-Spring technique, as described in [Appert et al. 2012], readily applies to all simple cases of reciprocal drag-and-drop. Figure 2 illustrates it on a very simple case, where an icon gets restored to its last position. A red circular handle pops up close to the cursor when the user presses the mouse button and remains still for 500ms (i.e., dwells) over the icon. Bringing the cursor or finger onto this handle will make a spring appear, showing what the center of the icon will become if the user releases the mouse button or lifts his finger over the spring handle. If the user dwells without having initiated any movement, the spring shows the last move that was applied to the icon. If the user has already initiated a drag-and-drop, the spring proposes the reciprocal drag-and-drop for the current move. The user can either move over the spring handle and select it, activating the spring and thus bringing back the object to its previous location; or he can discard the widget by getting out of the active area.

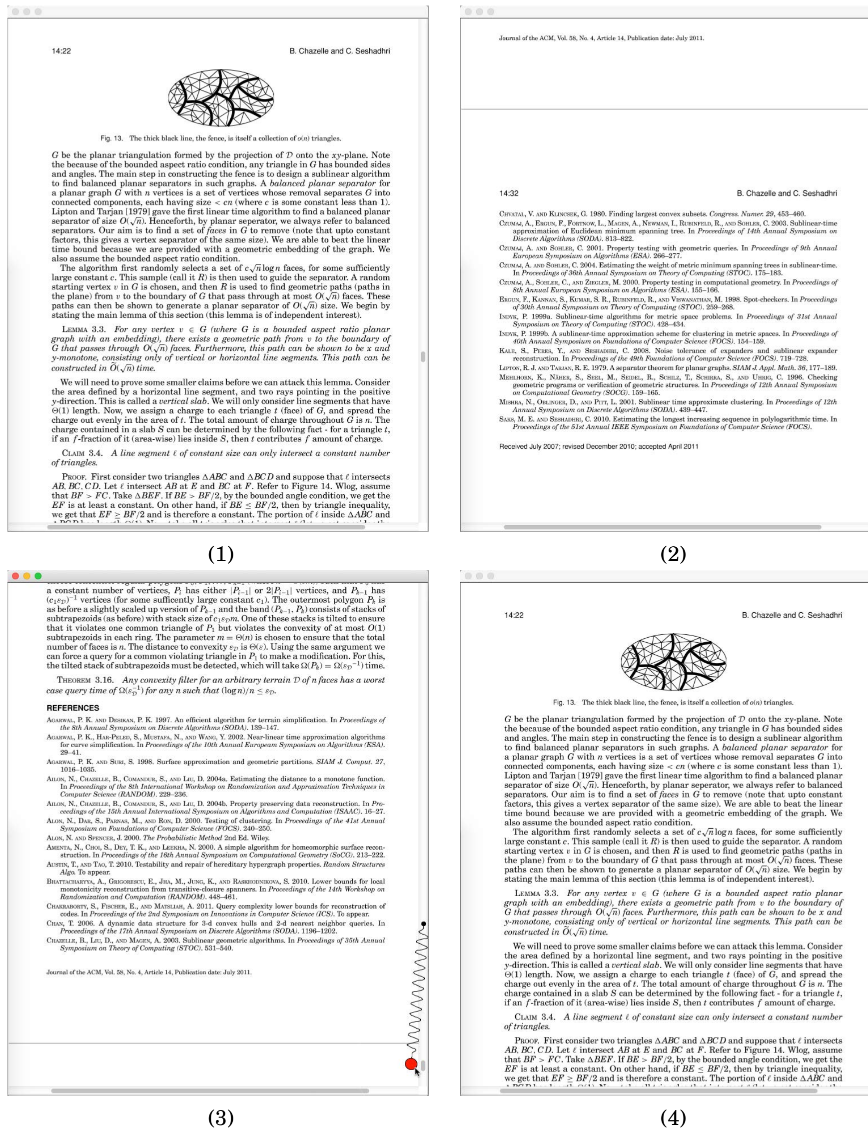


Fig. 4. Reading a document. (1) The user finds a bibliographic reference while reading. (2) He drags the scrollbar knob to the end of the document to check that reference. (3-4) Once he has checked it, he invokes Dwell-and-Spring on the scrollbar knob to get back to the page he was reading.

This simple technique already applies to many cases of reciprocal drag-and-drop: manipulating icons on the desktop (Figure 2), navigating documents using a scrollbar (Figure 4) or with a swipe gesture on a touch-sensitive surface (Figure 5), moving and resizing windows (Figure 6), or any other action where the spring's actions are equivalent to what the user would manually do to revert to the original state, like moving a slider knob (Figure 3) or a manipulation handle (Figure 7). However, in its original version [Appert et al. 2012], Dwell-and-Spring was only able to revert the

(1) (2) (3) (4)

Fig. 5. Panning a map on a tabletop. (1-2) The user swipes on the touch screen to set the viewport over the region of interest on the map, revealing more of Central Park in New York City. (3-4) She touches the screen and remains still to invoke Dwell-and-Spring on the map and go back to her view on Midtown.

(1) (2) (3) (4) (5) (6)

Fig. 6. Managing the desktop. (1) The user is making a transcription of a sketch into a vector-based document. (2-3) She resizes the window in order to copy and paste some elements that she had already drawn somewhere else in her document. (4-5) She restores the initial window size to get back to an ideal window layout for her transcription task. (6) She adjusts the location of the just pasted elements.

current or the last drag-and-drop, as it was only keeping track of the previous location of each object, based on a per-object linear undo model.

2.2. All Past Locations of an Object

The first enhancement made to the Dwell-and-Spring technique is to provide users with extra spring handles that allow them to apply a series of reciprocal drag-and-drop actions quickly. As illustrated in Figure 7, the spring widget features additional handles that are horizontally aligned with the main spring handle (which was the only handle in the original design). Users can navigate through these handles to get

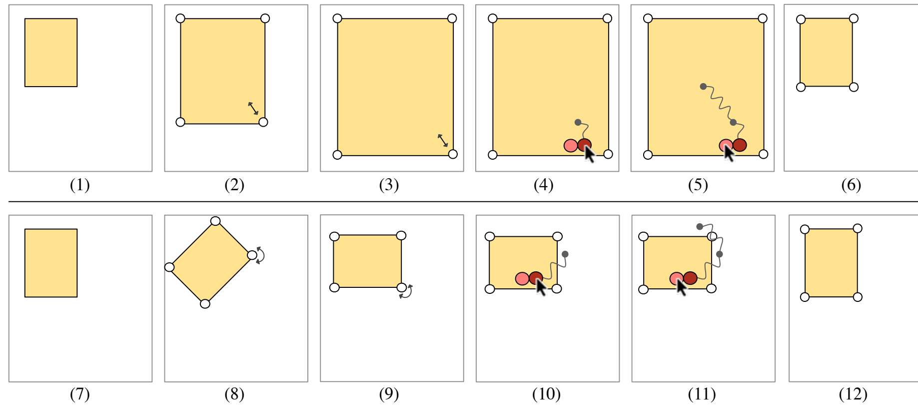


Fig. 7. DND^{-1} applied to shape manipulation handles. (1-2-3) The user resizes a rectangle twice. (4) She invokes Dwell-and-Spring on the resizing handle and enters the spring's main handle. This shows where the resizing handle was prior to the last resizing manipulation. (5) She moves the cursor to the next spring handle in the DND^{-1} history. This shows where the resizing handle was prior to the last two resizing manipulations. (6) She releases the mouse button on that second spring handle to revert back to the configuration in (1). Steps (7–12) illustrate a similar scenario on a rotation handle.

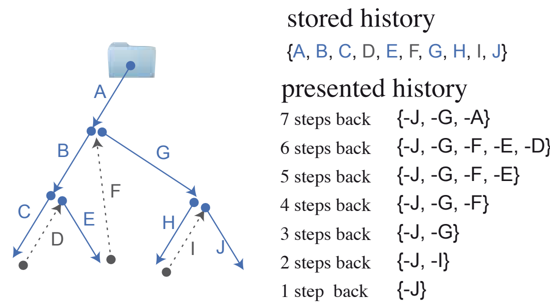


Fig. 8. DND^{-1} stores all repositioning actions applied to an object, including those performed via a reciprocal drag-and-drop (D, F and I, shown as dashed black lines). It presents the shortest path to all past locations.

a preview of where the selected object(s) would go if they released the mouse button or lifted their finger on one of them. Releasing the mouse button while the cursor is over a spring handle will invoke the series of reciprocal drag-and-drop actions that are associated with this handle. As explained below, the series of past moves is managed by the DND^{-1} model, which differs from the linear undo stack that is implemented in most systems. With DND^{-1}, users can revert back to any past location, while keeping the length of the history as short as possible.

Applications that support undo typically store the history of actions as a tree whose nodes are the different states of the application. Performing an operation means creating a novel child state to the current node. Undoing an operation means getting back to the parent node. Some systems make several branches active at the same time: in collaborative work settings (e.g., [Edwards and Mynatt 1997]) or for comparing variations of an image design that have minor differences [Terry et al. 2004]. However, the linear undo model only supports one *single active path*. All nodes outside this path are inaccessible via undo. For instance, in Figure 8, the user moves the icon three times

successively (displacements A, B then C), reverts C, and then moves the icon again by E. At this point, she can no longer recover the position the icon had after displacement C, since this one no longer belongs to the active path ({A,B,E}). A few applications, such as Emacs [Gosling 1982; Yang 1992], make users able to recover any state. However, this might require chaining a long series of interactions to reach a given state, as the history stack is presented to them as a *full sequential path* in the history tree.

Figure 8 illustrates DND^{-1}, the local undo model we propose to navigate in the history tree of displacements performed on a graphical object. This model lies in-between the *single active path* and the *full sequential path* models described above. It stores the full history of repositioning actions, but provides users with shortcuts to quickly access nodes in the tree, so as to make them able to recover any past location, in the spirit of the Selective Undo model [Berlage 1994]. Each object remembers the sequence of moves that were applied to it, including reciprocal drag-and-drop actions. All past locations are accessible. Also, when a user invokes a reciprocal drag-and-drop action to restore a past position P of an object O, DND^{-1} adds the straight move between O's current location and P to the end of the history, in the spirit of the *inverse model* for selective undo (used in e.g., [Berlage 1994; Myers 1998; Yoon et al. 2013]), rather than inserting all reciprocal moves after the corresponding moves in the history, as the *script model* (used in, e.g., [Kurlander and Feiner 1988; Myers et al. 2015]) would have done.

For example, in Figure 8, the user moves an icon using three standard drag-and-drop actions (moves A, B and C). He then moves the icon back to the location where it was before reaching C, using a reciprocal drag-and-drop (move D). At this point, there is no difference between DND^{-1} and the *script model*: both append D (i.e., -C) right after C at the end of the history: {A, B, C, D}. However, when the user moves the object back to the location it had before move B (following move E), DND^{-1} handles this reciprocal drag-and-drop as any regular drag-and-drop by appending F to the history ({A, B, C, D, E, F}), while the *script model* would have turned the history into {A, B, -B, C, -C, E, -E}. This entails that a drag-and-drop that has been undone with DND^{-1} can be easily redone. However, keeping a trace of all past moves also means that the number of steps to revert to a past location can be very long. In order to present an object's history of past locations in a compact way, we have implemented a navigation algorithm that computes the shortest path in the history to reach any of these past locations. This is basically achieved by removing cycles, i.e., series of moves that bring the object back to a location already present on the path. For instance, navigating two steps back with DND^{-1} after move J in Figure 8-b entails following path {-J, -I}; but navigating three steps back entails following path {-J, -G}, as {-J, -I, -H} would have led to the same location than {-J}, which is already proposed for a one-step-back navigation. This simplification decreases the number of steps that should be presented to the user, while ensuring that he can reach any past locations. For the scenario in Figure 8, thanks to this simplification, our extended version of Dwell-and-Spring presents seven spring handles while it would have presented ten (i.e. the length of the stored history) otherwise.

2.3. Groups of Objects

The second enhancement made to Dwell-and-Spring is to provide users with another type of extra spring handles that allow them to apply reciprocal drag-and-drop actions to groups of objects that were moved simultaneously in the context of a multiple selection. Figure 9 illustrates what Dwell-and-Spring looks like after the user has played the scenario of Figure 10. These additional square handles act on the groups that contain object O, on which Dwell-and-Spring has been invoked. Handles are organized into several rows, one per group. The primary handle of a row is aligned with the main

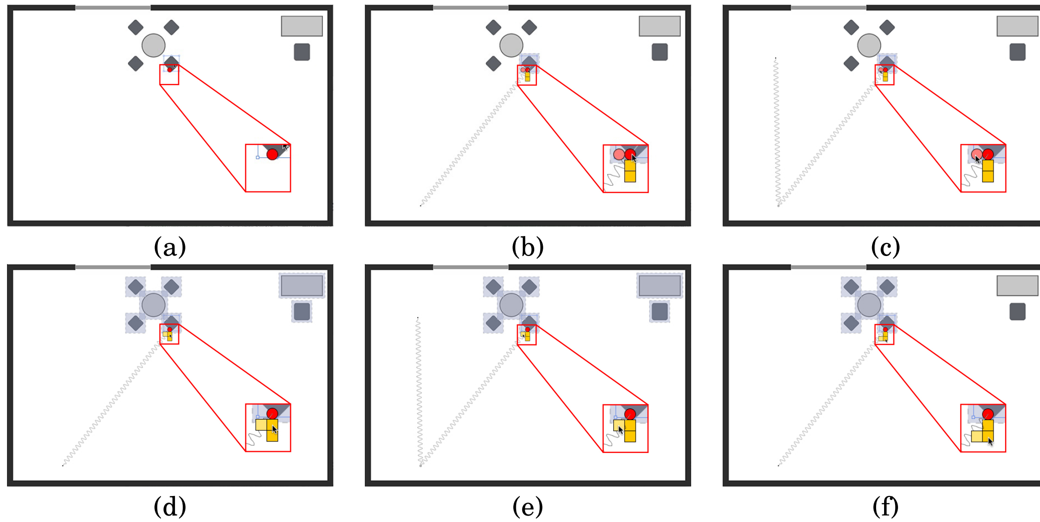


Fig. 9. Extended version of Dwell-and-Spring (the zoomed-in inset is added on top of UI screenshots for illustration purposes only). (a) The user presses the mouse button and remains still over object O; the main circular handle appears after a short delay. (b) The user enters this handle; additional handles appear as a result. (c) Navigating the row of circular handles lets her undo sequences of moves for O as an individual object. (d) Square handles allow her to restore any past location for groups in which O was – or is – involved. The first row of square handles will act on group $G_{meeting+desk}$, that contains the 7 objects in the scene. (e) Navigating along this row lets the user undo sequences of moves for group $G_{meeting+desk}$. (f) The second row of square handles will act on group $G_{meeting}$, that contains the 4 chairs and the circular table.

spring handle that initially popped out. When the cursor enters the handle associated with a group G, additional handles appear on its left. There are as many handles as the number of locations this group has visited. To help users anticipate what will happen if they activate a given handle, Dwell-and-Spring gives some feedforward when the mouse cursor or the finger enters that handle: objects in the group are highlighted, and the sequence of moves that will be reverted is shown as a series of springs that ends on what will become the center of object O after completion of the sequence of reciprocal drag-and-drops.

To keep track of all multiple selections an object has belonged to, the DND^{-1} model stores the whole history of moves into a hashtable ($History$) whose keys are groups of objects (which can be singletons) that index lists of timestamped movements. Figure 10 illustrates this on a simple example. However, there can be some ambiguity when deciding what a group is, as the same objects can be involved in different multiple selections (Figure 10-(a-d)). In order to maintain a coherent history, we have designed a strategy for handling groups, that allows users to recover any past state without breaking any previous group.

When a group G has been moved by d, we first add G and all the singletons for objects in G that are not already in $History$. We associate an empty history with each of them (lines 1-8). We then review the history of all groups, as detailed in Algorithm 1. Each group G_i is split into two parts: $G_\cap$, that contains the objects that belong to both G and G_i, and $G_\setminus$, that contains the objects that only belong to G_i (lines 10-11). Each of these groups, if not empty, are updated in $History$ (after creation if needed). The history of $G_\cap$, which is empty if just created, is merged with the one of G_i and is then enriched with the last move d (line 15). The history of $G_\cap$, which will be empty if it just got created, is merged with the one of G_i.

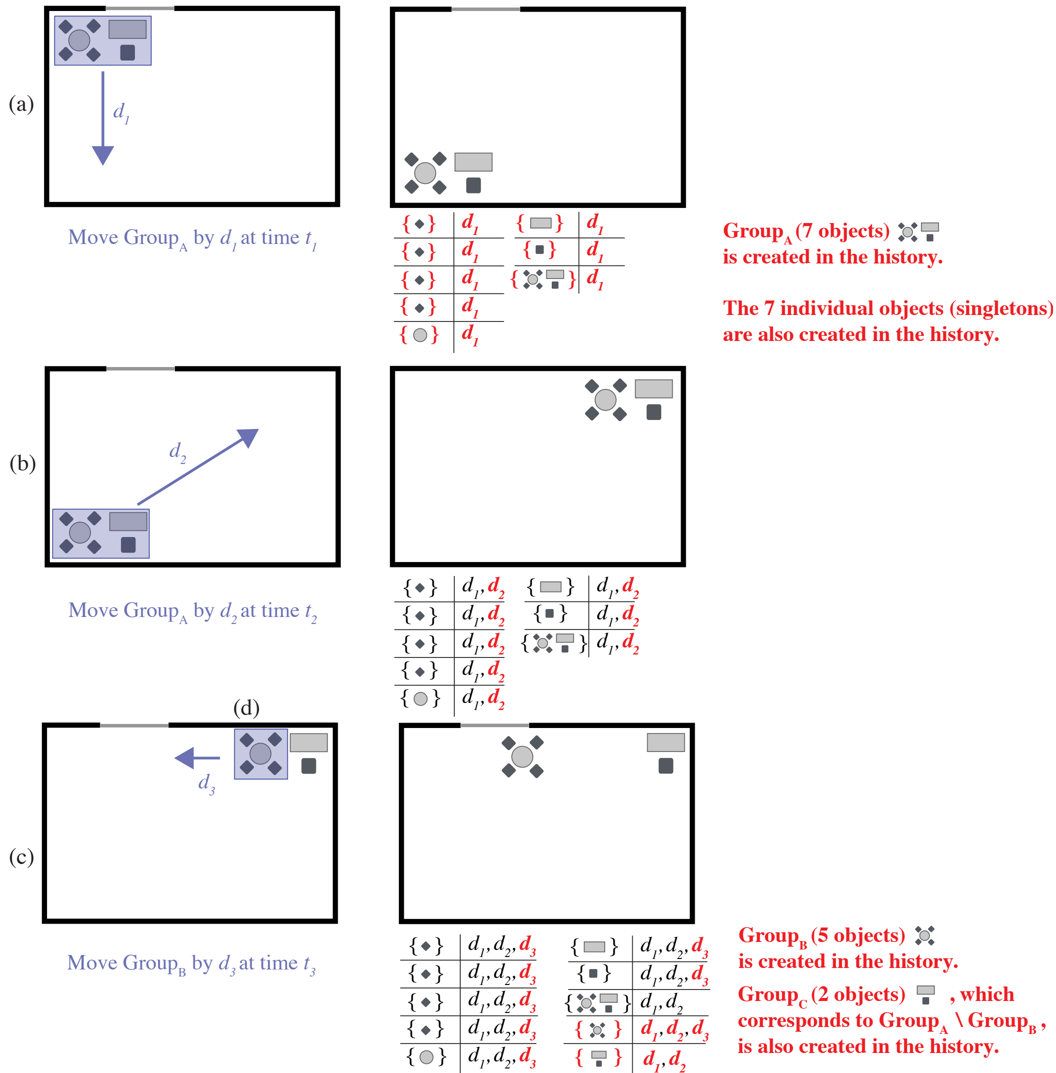


Fig. 10. A scenario involving multiple selections: arranging furniture on an office's floor plan. (a-b) Moving Group$_A$ by d_1 then d_2. Group$_A$ consists of all 7 pieces of furniture: the circular table and its associated 4 chairs + the rectangular table and its associated chair. (c) Moving group Group$_B$ by d_3. Group$_B$ consists of the 2 pieces of furniture: the rectangular table and its associated chair

Figure 10 illustrates this on a concrete example. Starting from an empty $History$, Group$_A$, that consists of 7 objects, is moved by d_1 as illustrated in step (a). Our algorithm creates 7 singletons and Group$_A$ in $History$. Then, during the process of revising existing groups, the individual histories are populated with d_1. In step (b), d_2 is added to all histories. In step (c), the user moves Group$_B$ by d_3. This latter group is created (line 3 in Algorithm 1) and, when revising the existing groups, Group$_A$ is split into Group$_B$ (line 10) and Group$_C$ (line 11). Group$_B$'s empty history is merged with the one of Group$_A$, which is (d_1, d_2), and d_3 is added to the resulting history (line 15). Group$_C$ is also created (line 18) and its empty history is merged with the one of Group$_A$ (line 19).

```
// Let h be the function that returns the local history of G stored in History as a
   list of timestamped past movements (d_i, t_i), merge(H1, H2) be a function that
   creates a chronologically ordered list of all timestamped movements of H1 and
   H2 (removing duplicates), and append(H, (d, t)) be a function that appends (d, t) at
   the end of H.
```

1 **if** G *does not exist in* $History$ **then**
2 | $h(G) = \emptyset$
3 **foreach** *object* O *in* G **do**
4 | **if** $\{O\}$ *does not exist in* $History$ **then**
5 | | $h(\{O\}) = \emptyset$
6 |
7 **end**
8 **foreach** *existing group* G_i *in* $History$ **do**
9 | $G_\cap = G_i \cap G$
10 | $G_\setminus = G_i \setminus G_\cap$
11 | **if** $G_\cap \neq \emptyset$ **then**
12 | | **if** $G_\cap$ *does not exist in* $History$ **then**
13 | | | $h(G_\cap) = \emptyset$
14 | | $h(G_\cap) = append(merge(h(G_\cap), h(G_i)), (d, t))$
15 | **if** $G_\setminus \neq \emptyset$ **then**
16 | | **if** $G_\setminus$ *does not exist in* $History$ **then**
17 | | | $h(G_\setminus) = \emptyset$
18 | | $h(G_\setminus) = merge(h(G_\setminus), h(G_i))$
19 |
20 **end**
21 $h(G) = append(h(G), d)$

ALGORITHM 1: Revising $History$ after a group G has been moved by d at time t.

The creation of singleton objects (line 4 in Algorithm 1) gives more flexibility to users by allowing them to revert moves on individual objects, as in the scenario of Figure 11, detailed below. However, we do not add all sets that belong to the power set of group G (the one that just moved). This is to keep the number of groups in $History$ reasonable, while ensuring that any revert operation is possible. This simplicity vs. flexibility tradeoff means that, if users want to revert a move for a subset g of objects that belong to group G, they have to revert that move on each of the individual objects. Finally, when users delete a graphical object, this object is not removed from $History$, but is simply tagged as passive, so as to keep a memory of it in case it gets restored by means of the application's functional undo model or by drag-and-drop across applications. Its passive state makes it ignored by our algorithm for revising groups.

Support for groups in DnD^{-1} means that users can keep a trace of previous multiple object selections even if other selections happened afterwards. In particular, the current relative layout between objects within a group G is preserved in case they want to restore a past location of G, making transitions such as the one in Figure 1-(i-j) very easy: the user can put back all desk elements to a past location while preserving the rearrangement of the elements that was made afterwards. It also overcomes some limits of the *single active selection* model used in most applications. With the latter, the current selection gets cleared as soon as users click on a region or object that does not belong to the selection. DnD^{-1} should save a lot of time and effort when trying to revert complex selections (small objects, objects scattered all over the workspace, partially occluded objects, etc.). For example, Figure 11 shows a scenario where a user wants to test alternative placements for the legend of a bar chart. The user accidentally selects a bar along with the legend, but notices it only after he has moved the

legend to another location (a-c). As each object is also added individually in $History$, he can easily restore the bar's position (d), which has the effect of creating the group that contains only the graphical elements of the legend in $History$. As the novel placement of the legend overlaps with the y-axis label, he applies a reciprocal drag-and-drop to this group to put it back where it initially was (e). Later, he decides to slightly offset the label of the y-axis (f) and can easily test the alternative placement with the legend on the left again (g-h).

2.4. Implementing DND^{-1}

DND^{-1} is designed to be implemented at the system level, in an application-independent manner. We developed a Java prototype using SwingStates [Appert and Beaudouin-Lafon 2008] to demonstrate the approach, in which each client application implements the `ReciprocalDnDProtocol` interface and runs in a `JInternalFrame`. The `ReciprocalDnDManager` communicates with these applications exclusively through messages, while the interaction techniques to navigate in DND^{-1} (Dwell-and-Spring or DnD-List – described later in Experiment 2) are hosted on a full-screen `GlassPane`.

The central object in the implementation is the `ReciprocalDnDManager`, that handles the entire $History$ across client applications. Application developers can add support for DND^{-1} simply by registering their application with the `ReciprocalDnDManager` and implementing the simple protocol described below.

Client applications send a message each time an object is created or deleted. The `ReciprocalDnDManager` can then tag this object as either active or passive. They also send a message `moved(`G`, `d`)` each time a group of objects G is moved by a displacement vector d. The `ReciprocalDnDManager` then updates $History$ according to the algorithm described earlier. All these client-to-server messages contain the sender application's id, and the ids of objects that are created, deleted or moved. The `ReciprocalDnDManager` organizes object ids using namespaces (one per client application), meaning that unicity is ensured across applications but that applications still have to guarantee the unicity of their objects' ids.

Techniques for interacting with DND^{-1} are connected to the `ReciprocalDnDManager` to expose $History$ to users and let them select a group G and a past location p to restore. When users invoke such a reciprocal drag-and-drop, the `ReciprocalDnDManager` updates $History$ by adding δ to the history of G, with δ being the vector between the current location of G and p. It then sends a `translate(`G`, `δ`)` message to the right client application with the ids of objects in G and the translation vector as arguments. The client application is then in charge of applying this displacement. To enable the implementation of feedforward mechanisms such as object highlighting, client applications must also be able to reply to two other messages: `id:pick(screenPoint)` and `rectangle:bounds(`G`)`. The first returns the id of the object that is picked at a given location, the second returns the bounding box of a group of objects. We have used this prototype to implement the experiments that we report on in the next sections.

3. EXPERIMENT 1: UNDERSTANDING USERS' DRAG-AND-DROP HABITS AND EVALUATING THE DISCOVERABILITY OF DWELL-AND-SPRING

For completeness, we include an experiment that was reported in [Appert et al. 2012], in which we introduced the original, simpler version of the Dwell-and-Spring widget. We conducted this first experiment to capture what users typically do in situations where they want to revert a drag-and-drop. We also wanted to evaluate whether the spring metaphor implemented in Dwell-and-Spring is a viable alternative or not. The experiment lasted around 45 minutes and contained two parts: an interactive *in-situ* questionnaire to gather data about how users currently revert drag-and-drop actions,

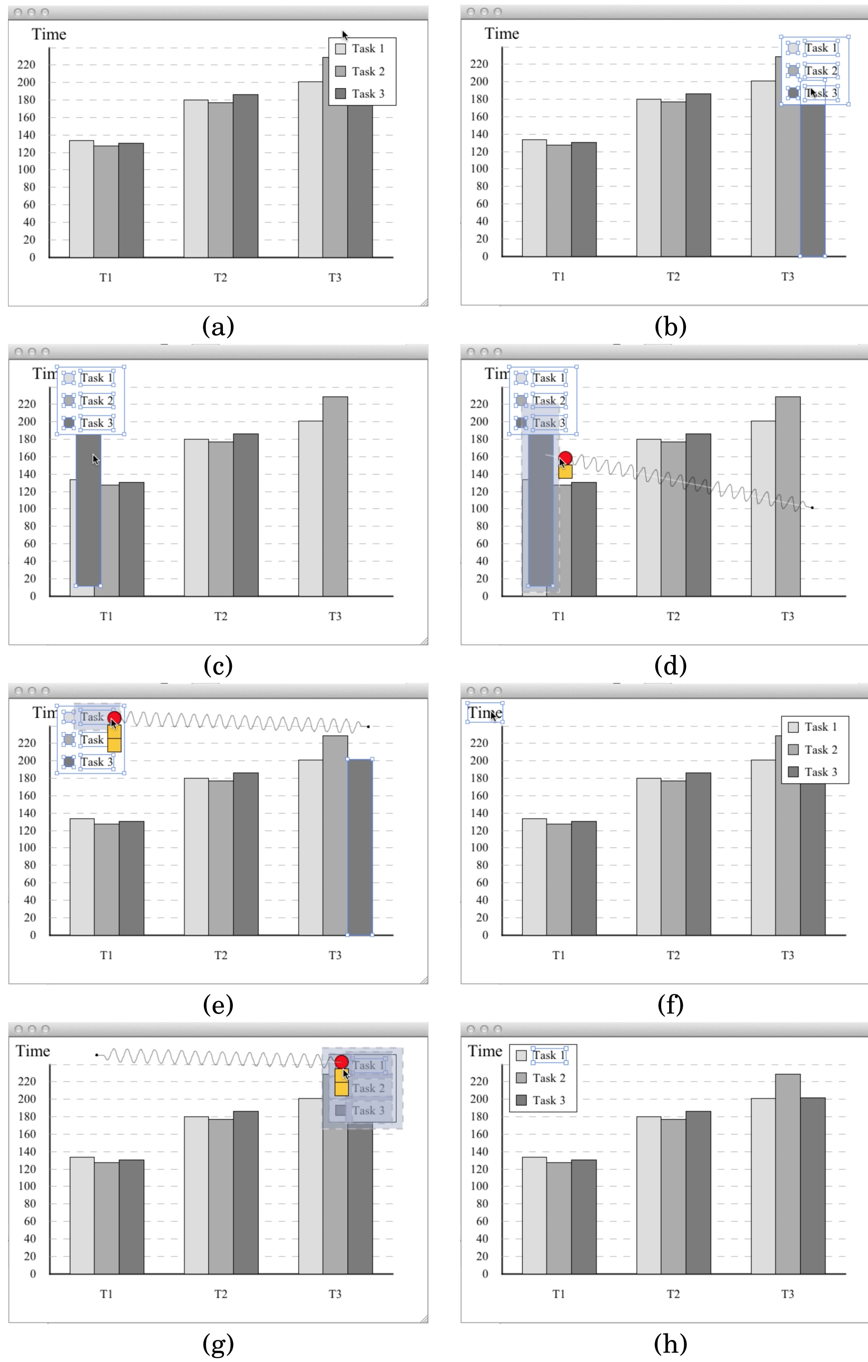


Fig. 11. Positioning the legend of a bar chart. (a-b) The user makes a selection of all elements of the legend. (c) He notices that he has accidentally included a bar in his selection and moved it along with the legend. (d) He uses Dwell-and-Spring to restore the bar to its original position. (e) The novel placement of the legend is not satisfying, and he puts the legend back where it was initially. (f) He moves the y-axis label, and (g-h) reverts the legend back to the left position.

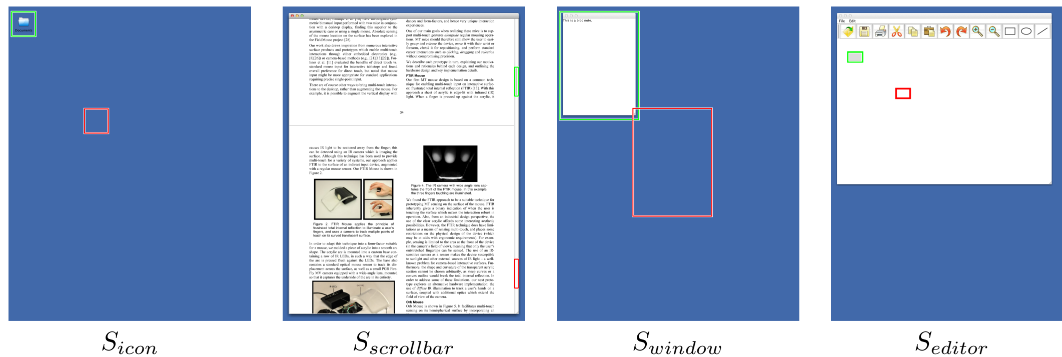


Fig. 12. Scenarios used for collecting users' habits in different situations of reciprocal drag-and-drop. The current location of the object (icon, scrollbar knob, window or vector shape) is highlighted in green and the past position to restore is highlighted in red.

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.

3.1. 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 of them 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 mouse with 400 dpi resolution and default system acceleration.

3.2. Capturing Users' Habits

To gather data about how users revert drag-and-drop actions more specifically, 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 (Figure 12): 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 an-

DRAGGING case (would like to do, usually do)

Strategies \ Scenarios	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)

Strategies \ Scenarios	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

Fig. 13. Strategies reported in the interactive questionnaire for each scenario in both the DRAGGING case (top) and the DROPPED case (bottom). 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.

swer 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 *telling 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 presentation order 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 scenario presentation order 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.

Figure 13 summarizes the answers from our participants. It first shows a clear difference between the graphical editor scenario (S_{editor}), in which participants manipulate content (functional level), and the other scenarios, in which they modify the view configuration (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

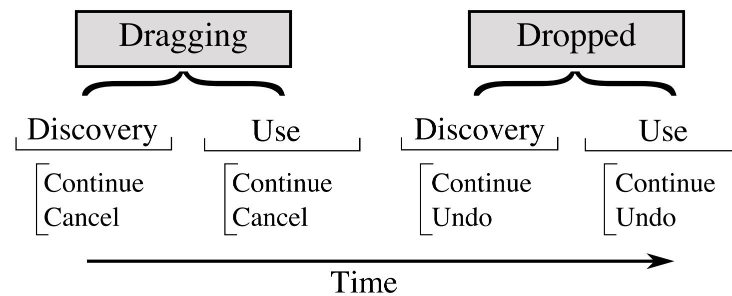


Fig. 14. Outline of the discovery-and-use experiment's design.

`Escape` key and no participant used the right-click technique to abort and cancel the current action. In the other three 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 participants told us that they usually use `Cmd-Z` to move back a desktop icon (S_{icon}) to its original position, and four thought they were using it to restore the past position of a window (S_{window}). 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 view level and the functional level is not always clear to users: in one case, the path of the file remains unchanged in the file system, while in the other case, the same interaction causes the file to be moved to a new folder.

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. Expression of such a need for better revisitation mechanisms supports findings reported in previous studies (such as, e.g., [Ko et al. 2006]). Topaz [Myers 1998] includes scrolling operations in its history to allow users to selectively undo them. But users have to locate these specific navigation actions within the whole history of commands. As we will see later, Dwell-and-Spring is particularly well-suited to canceling on-going, or undoing just-performed, scrolling actions directly on the scrollbar, which could be additionally augmented with some colored marks, as in [Alexander et al. 2009].

3.3. Discovery and Use: Experiment Design

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

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

Figure 14 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 or-

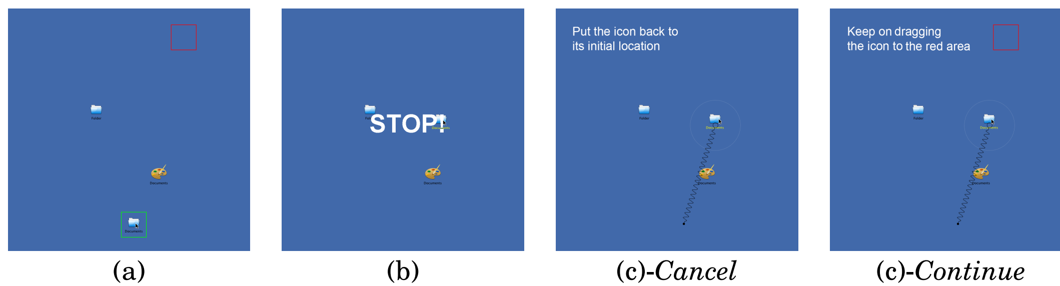


Fig. 15. Discovery task used in Experiment 1. (a) Participants are instructed to move an icon, highlighted in green, towards a target location, which is highlighted in red. (b) In the middle of their drag-and-drop, a message pops up to ask them to stop their movement. (c)-*Cancel*-(c)-*Continue* Participants are instructed to either put back the icon to its initial location or keep on dragging the icon to the target location.

der 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 (Figure 15). In the DRAGGING case, participants were told before starting that they would be interrupted in the middle of their move by a pop-up message that would give them 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 they had followed the new instructions, participants had to press the Space bar to end the trial.

In the DROPPED case, participants also had to drag-and-drop a desktop icon to a target location. As soon as they had dropped the icon, they got a message asking them to either put it back where it was before they moved it, and then press the Space bar (*Undo* condition), or to just press the Space bar immediately (*Continue* condition). In both cases, when they were 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 them.

In both cases, trials were organized into 4 blocks of 24. 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 participants ran into the two other *use* blocks. These last two blocks were aimed at observing what strategy par-

ticipants 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 as in Figure 5), 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*).

3.4. Discovering Techniques

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

3.4.1. 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. When the spring popped up after a dwell, some participants brought the cursor over the spring handle but did not activate the spring by releasing the mouse button as soon as the cursor was over the handle. Instead, they brought the spring handle towards the spring's origin (thus compressing the spring) and only then released the mouse button. 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: 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 it (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 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 would make the spring hard to discard in case the user does not want to activate it. 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.

3.4.2. 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 a learning effect between the cancel and undo conditions during the experiment.

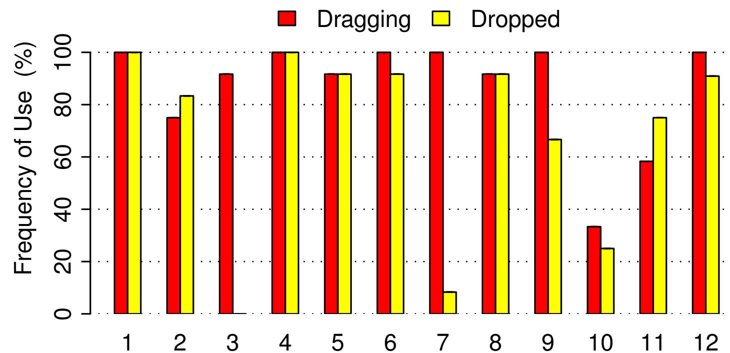


Fig. 16. Use of Dwell-and-Spring in the last block for both DRAGGING and DROPPED, per participant.

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%.

3.5. 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.

3.5.1. Frequency of Use & Qualitative Results. Figure 16 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 six 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 [Landsberger 1958] 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".

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

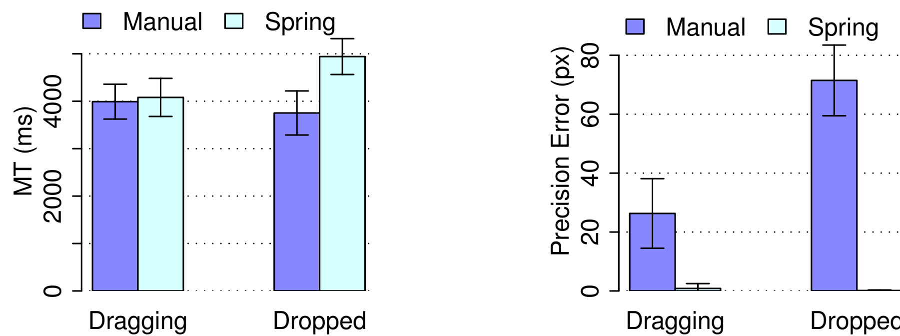


Fig. 17. Movement time (left) and precision error in pixels (right) for *Manual* and *Dwell-and-Spring*, in both cases DRAGGING and DROPPED (under *Cancel* and *Undo*). Error bars show the 95% confidence limits.

3.5.2. Outliers and Errors. For our analyses, we first filter out trials that end while the icon is more than 400 pixels away from the ideal position it should have been put at. As the distance between start and target icon locations is initially 800 pixels and the message pops up either at the end of the movement in the DROPPED case or in the middle of the movement in the DRAGGING case, this 400-pixel criterion captures outlier trials where something unexpected occurred. In the DROPPED case, these are trials where participants 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). This happened in 2.09% of the trials in the DROPPED case. In the DRAGGING case, these are trials where participants either ignored the instruction that asked them to put the icon back where it was (*Cancel* condition) or activated the spring while they should have continued their current move (*Continue* condition). This happened in 2.78% of the trials in the DRAGGING case (∼2% in the *Cancel* condition and ∼0.75% in the *Continue* condition). This shows that participants activated the spring while they should not have in less than 1% of the cases. All remaining trials, which ended with the icon being less than 400 pixels away from the target location, are kept for further analyses.

Regarding accidental interactions with the spring, we also recorded occurrences of participants grabbing the spring without activating it while they should not have used it (*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 minimizes accidental triggers and enables easy repair.

3.5.3. Movement Time & Precision. Figure 17 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 outliers mentioned above. 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*[3]. 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 it automatically performs the ideal reciprocal manipulation, putting the object back to its exact orig-

[3]The small imprecision observed in Figure 17-right is due to accidental clicks before pressing the space bar.

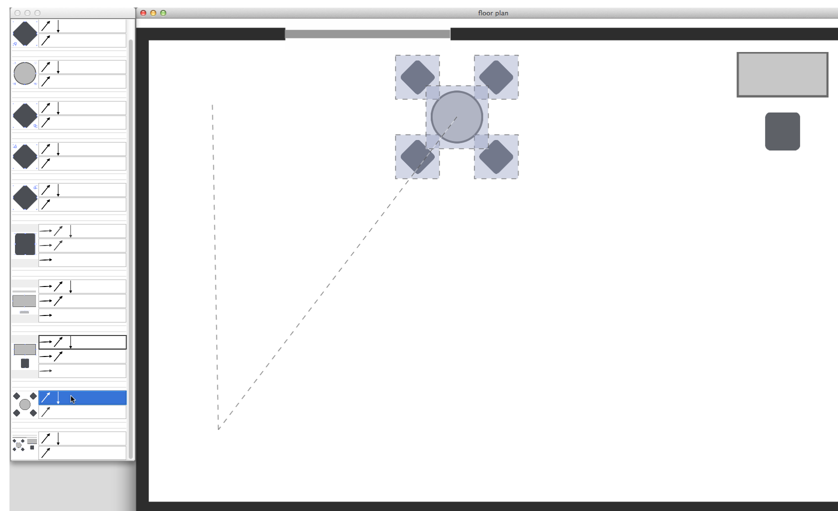


Fig. 18. The DnD-List (*List*) technique (after playing the scenario in Figure 10).

inal location. The average precision error with *Manual* was 71.5 pixels (median 69 pixels). Our experimental design allowed participants to choose which technique they wanted to use. This resulted in an unbalanced number of collected measures between the *Manual* and *Dwell-and-Spring* conditions, thus violating the assumptions made when running a statistical test comparing conditions. However, Figure 17 suggests a trade-off between movement time and precision when comparing *Dwell-and-Spring* and *Manual* in the DROPPED case. The precision error is lower in the DRAGGING case than in the DROPPED case as the spring was always popping up at the moment participants had to pause in the middle of their drag movement. In such cases, the spring visually helped participants revert the icon back to its exact previous location.

3.6. 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.

4. EXPERIMENT 2: USABILITY OF DND^{-1} COMBINED WITH DWELL-AND-SPRING

We conducted a second experiment to test if users could understand and use the DND^{-1} model effectively when they have to restore the position of either a single object or a group of objects. We also wanted to assess the relevance of the Dwell-and-Spring technique (abbreviated *Spring*) in comparison with a baseline technique which exposes the full history (abbreviated *List*).

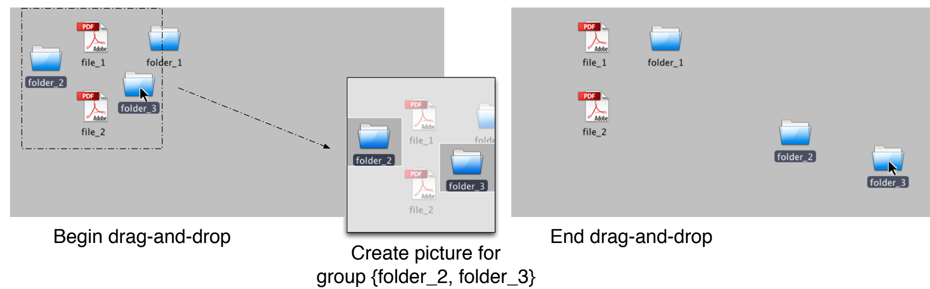


Fig. 19. Thumbnail creation process for DnD-List. When users initiate a drag-and-drop on a multiple selection (here, {folder_2, folder_3}) for the first time, DnD-List takes a screen shot, crops it and applies a mask to emphasize the objects that actually belong to the group.

4.1. The DnD-List (*List*) technique

As the DND^{-1} model is novel, we wanted to gather observations about its usability independently from its combination with Dwell-and-Spring. We thus designed a baseline technique that exposes the full history of DND^{-1} in a standard list presentation, close to the one found in, e.g., Adode Photoshop. This technique, DnD-List (*List*), consists of a separate window, that remains always visible on top of other windows. As illustrated in Figure 18, this window shows a scrollable list. Each row displays the history of moves for a given group of objects G. An image of G is on the left, and the list of paths that lead to all past locations of G is on the right. A path is displayed as a series of arrows whose orientation matches that of the actual movements that brought G where it currently is. The user can click on any of these paths to actually execute the reciprocal drag-and-drop that reverts this series of movements. As DnD-List is implemented according to the DND^{-1} model, a reciprocal drag-and-drop is appended to the history, following the inverse model of selective undo (as Dwell-and-Spring does – see Section 2.2).

As illustrated in Figure 19, *List* makes a screen capture each time users press the mouse button, and creates a thumbnail picture of a group G when users select and move it for the first time. *List* crops the minimal square that fully contains the group and adds a white translucent mask on top of this image to emphasize the objects that actually belong to the group. Objects that fall within the bounding box of the multiple selection but that are not part of the group are still visible, but faded out.

Groups are sorted by the number of objects they contain (largest groups at the bottom of the list). Groups that feature the same number of objects are sorted according to the timestamp of their last move (a group is below another group if it has been moved more recently). Each time a move occurs, resulting from either a manual drag-and-drop or a reciprocal drag-and-drop, the path item that has just been inserted is highlighted and the window auto-scrolls to make it fully visible inside the list's viewport. Also, when the mouse cursor hovers an item, the bounding boxes of objects that belong to the associated group appear, along with a polyline showing what will become the center of this group if the user selects this item. Figure 18 illustrates this. G_{desk} has just been moved (Figure 10-e) and the cursor hovers a path item to restore a past position of $G_{meeting}$. This hovering feedforward mechanism, which is intended to prevent errors, is hosted on a transparent layer on top of any application that implements the communication protocol described in the implementation section above.

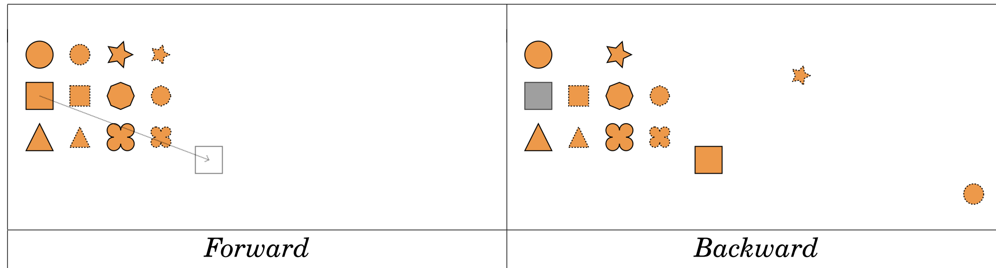


Fig. 20. Move and Undo instructions for a single object.

4.2. Hypotheses

Our main hypothesis is that users can interact with DnD^{-1} using both techniques (factor TECH $= \{Spring, List\}$). We hypothesize that DnD^{-1} enables them to minimize the number of operations required to revert a sequence of moves.

Our secondary hypothesis is that the cost of using a technique depends on the structure of the $History$ relative to the `reciprocalDnD(`G, P`)` operation to do. This cost is a function of both local and global depths. Let us consider a group G at position P. The performance of *Spring*, which operates object-wise, will be mainly impacted by local depth, i.e., the number of positions that G visited until it was eventually put in P (LOCAL-DEPTH). The performance of *List*, on the contrary, will also and mainly be impacted by the number of manipulations that have been performed on other objects since G left position P. Indeed, this entails that the right item is deeper in the list and that retrieving it requires more scrolling. To operationalize this notion of cost, we consider two secondary factors: LOCAL-DEPTH and LIST-SCROLL, as detailed later.

The experiment was divided into two parts and took 70 minutes on average. In each part, participants had to both perform movements of objects using drag-and-drop operations and apply undo operations in order to reach a specific graphical layout. In the first part, the difference between the target and the current layouts can be corrected with a reciprocal drag-and-drop on an individual object, whereas in the second part, solving the difference requires performing a reciprocal drag-and-drop on a group of objects. In both cases, if users make an optimal use of the DnD^{-1}, they are able to solve the difference with a single (optimal) reciprocal drag-and-drop.

4.2.1. Participants. Twelve volunteers (5 female), all right-handed, aged 24 to 40 years-old (average 30.0, median 28.5), daily computer users, participated in the experiment.

4.2.2. Apparatus. We used a Mac Pro workstation running Mac OS X, equipped with a high-end graphics card connected to a 30” LCD display (100 dpi, 2560×1600 pixels) and an Apple Mighty Mouse set with the default transfer function.

4.3. Part 1: Multi-step Reciprocal Drag-and-Drop for Individual Objects

4.3.1. Experiment Scene and Task. Starting from a graphical scene composed of 16 different shapes (Figure 20), participants are instructed to follow a scenario that consists of a series of *Forward* and *Backward* instructions. Participants have to press the space bar in order to get the specific instruction and perform it. In the case of a *Forward* instruction (Figure 20-a), the next position P_{next} where the shape S has to be moved is shown by displaying an outline of S in P_{next} with an arrow linking the center of S to P_{next}. In the case of a *Backward* instruction (Figure 20-b), the past position P_{past} (that needs to be restored) of shape S is illustrated with a grayed out copy of S centered on P_{past}. To avoid any ambiguity when interpreting instructions, any two shapes either differ in their geometry (circle, star, square, hexagon, triangle or clover) or in both

Participant	Block 1		Block 2	
P01	*Spring*	$S_{pratice}$, S_4, S_2, S_5	*List*	$S_{pratice}$, S_6, S_1, S_3
P07	*List*	$S_{pratice}$, S_4, S_2, S_5	*Spring*	$S_{pratice}$, S_6, S_1, S_3
P02	*Spring*	$S_{pratice}$, S_1, S_6, S_4	*List*	$S_{pratice}$, S_3, S_2, S_5
P08	*List*	$S_{pratice}$, S_1, S_6, S_4	*Spring*	$S_{pratice}$, S_3, S_2, S_5
P03	*Spring*	$S_{pratice}$, S_3, S_5, S_1	*List*	$S_{pratice}$, S_2, S_4, S_6
P09	*List*	$S_{pratice}$, S_3, S_5, S_1	*Spring*	$S_{pratice}$, S_2, S_4, S_6
P04	*Spring*	$S_{pratice}$, S_5, S_4, S_6	*List*	$S_{pratice}$, S_1, S_3, S_2
P10	*List*	$S_{pratice}$, S_5, S_4, S_6	*Spring*	$S_{pratice}$, S_1, S_3, S_2
P05	*Spring*	$S_{pratice}$, S_6, S_3, S_2	*List*	$S_{pratice}$, S_4, S_5, S_1
P11	*List*	$S_{pratice}$, S_6, S_3, S_2	*Spring*	$S_{pratice}$, S_4, S_5, S_1
P06	*Spring*	$S_{pratice}$, S_2, S_1, S_3	*List*	$S_{pratice}$, S_5, S_6, S_4
P12	*List*	$S_{pratice}$, S_2, S_1, S_3	*Spring*	$S_{pratice}$, S_5, S_6, S_4

Fig. 21. Experiment 2. Presentation order of the 6 scenarios and TECH conditions across participants.

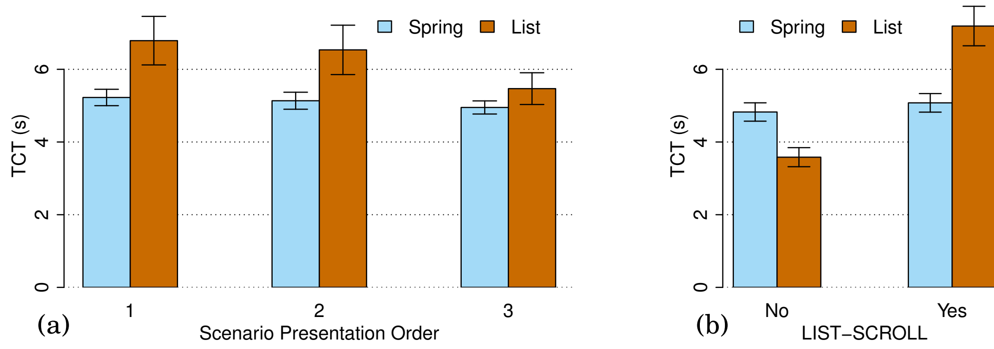
their size (small or large) and outline stroke (dotted or solid). In the *List* condition, the size of the history list window is set to 250 × 300 pixels, meaning that a maximum of four objects with up to two past locations fit in its viewport at the same time. To avoid introducing variability in the way participants navigated the list, its window could not be resized and it could only be browsed using the scroll bar.

The experiment program also controls the $History$'s structure, ensuring that it is exactly the same across participants. Drag-and-drop interactions are enabled only for *Forward* instructions. Conversely, the technique for restoring a past position (*Spring* or *List*) is enabled only for *Backward* instructions. During a *Forward* instruction, participants are allowed to move only the shape that has to be moved. Any other move makes the application beep and cancel the move without recording it in $History$. Similarly, during a *Backward* instruction, participants are allowed to restore a past position only for the shape that differs from the indicated layout. Any other reciprocal drag-and-drop causes a beep and is ignored. However, participants are not forced to perform the reciprocal drag-and-drop in a single (optimal) operation. They can choose to perform a series of reciprocal drag-and-drop operations as long as they perform them on the right shape. The program logs the number of operations and proposes the next instruction as soon as the shape is placed in the indicated position. Each time the program beeps, it counts a misinterpretation and reinitializes the timer used to record task completion time. We logged 3.8% such errors for *Spring* and 3.6% for *List*.

4.3.2. Design and Procedure. The experiment is divided into two blocks. One group of 6 participants see the *Spring* block first, followed by the *List* block, while the other half start with *List* and then see *Spring*. At the beginning of each block, the operator introduces the technique that participants are about to use (3 min). Participants then have to complete four scenarios, with the first one serving as a practice session ($S_{practice}$).

We collect measures (for analysis) on 6 scenarios (S_1, ..., S_6) per participant, i.e., 3 scenarios per TECH block. We use the 6 same scenarios for all participants. We compute 6 possible scenario presentation orders with a Latin Square. Each order is assigned to two participants, one participant starting with *List* and another participant starting with *Spring* (Figure 21). Overall, each scenario is played 6 times with *List* and 6 times with *Spring*.

As explained above, a scenario consists of a series of *Forward* and *Backward* instructions. More precisely, a scenario contains twelve *Backward* instructions interleaved with some *Forward* instructions, 4.75 in average (min=1 and max=8), for a total of 57 *Forward* instructions. Each scenario is generated in a pseudo-random manner, ensuring a balanced number of measures per LOCAL-DEPTH × LIST-SCROLL and satisfying



Effect for *TCT*	n, d	$\mathrm{F}_{n,d}$	p	η^2_G
TECH	1,11	4.70	0.0530	0.04
LOCAL-DEPTH	2,22	8.89	0.0015	0.12
LIST-SCROLL	1,11	102	< 0.0001	0.45
TECH × LOCAL-DEPTH	2,22	1.55	0.2346	0.02
TECH × LIST-SCROLL	1,11	105	< 0.0001	0.38
LOCAL-DEPTH × LIST-SCROLL	2,22	1.74	0.1949	0.01
TECH × LOCAL-DEPTH × LIST-SCROLL	2,22	2.73	0.0873	0.02

(c) TCT ∼ TECH × LOCAL-DEPTH × LIST-SCROLL × Rand(PARTICIPANT)

Fig. 22. (a) *TCT* by scenario presentation order for each TECH. (b) *TCT* by TECH × LIST-SCROLL for the third scenario. (c) ANOVA results.

three criteria: (1) the twelve *Backward* instructions of a scenario are distributed among the same four shapes; (2) each shape is involved in three *Backward* operations with LOCAL-DEPTH = $\{1, 3, 5\}$; (3) half of the *Backward* instructions can be performed by clicking an item that is initially visible when using *List* (LIST-SCROLL=*NO*), while the other half requires scrolling the viewport (LIST-SCROLL=*YES*).

4.3.3. Results for Part 1. Among the 864 (12 instructions × 6 scenarios × 12 participants) *Backward* instructions measured, 850 were completed by invoking a single reciprocal drag-and-drop. This shows that participants were able to use DND^{-1} in an optimal way for 98.38% of the *Backward* instructions they had to perform. The distribution of non-optimal trials plays against *List* (10 for *List* and 4 for *Spring*), and a pairwise Wilcoxon test ($n = 12$) confirms a significant effect of TECH on the number of non-optimal trials ($p = 0.048$). However, there is no significant effect of the structure of $History$ on the number of such trials: neither LOCAL-DEPTH nor LIST-SCROLL has a significant effect.

For trial completion time (*TCT*) analyses, we only keep trials completed in an optimal way. Among the 850 trials that were completed with a single reciprocal drag-and-drop, participants scrolled the viewport even though it was not required in 11.11% of the cases in the *List* condition. In these trials, participants failed to recognize the right shape in the list's thumbnails even though they were visible without scrolling (LIST-SCROLL=*NO*). This is not surprising, as searching through a list of graphical objects to find a target is a difficult task, that is costly in terms of both visual and cognitive processing [Salvucci 2001]. We filter out these trials, as they may be an artefact related to our experimental task. In a real context of use, objects may be either easier or harder to recognize in these thumbnails, depending on the type of graphics in the scene.

We look at the evolution of performance over the three scenarios per technique in order to check for a potential learning effect. As illustrated in Figure 22-a, we observe

such an effect only for *List*, not for *Spring*. An ANOVA shows a significant interaction between TECH and the scenario presentation order on *TCT* ($F_{2,22} = 3.76$, $p = 0.0392$, $\eta^2_G = 0.06$). A post-hoc (Holm corrected) t-test reveals that *TCT* significantly differs between each pair of conditions that vary in their scenario presentation order for *List*, while this is not the case for *Spring*. A t-test also shows that *Spring* is significantly faster than *List* in the first ($p = 0.0133$) and second ($p = 0.0188$) scenarios, but not in the third scenario ($p = 0.2136$).

So as not to disadvantage *List* by ignoring the fact that users' performance will benefit from learning, we analyze effects on *TCT* only for trials collected in the third scenario. Figure 22-c reports the results of an ANOVA for model *TCT* ~ TECH × LOCAL-DEPTH × LIST-SCROLL × Rand(PARTICIPANT). It shows that, in our experiment, the structure of $History$ impacts the performance of both *Spring* and *List*.

More specifically, both LOCAL-DEPTH and LIST-SCROLL have an impact on *TCT* for both techniques. First, the local depth of $History$ for an object (LOCAL-DEPTH), which sets either the number of spring handles to traverse with *Spring* or the number of items per object thumbnail with *List*, has an impact on the performance of both *Spring* and *List*. Second, LIST-SCROLL, which is related to the global depth of $History$, has a large effect on *TCT*. This mainly comes from the fact that the performance of *List* strongly degrades when users need to scroll the list of items, as illustrated in Figure 22-b. Indeed, the effect of TECH is marginal, whereas interaction effect TECH × LIST-SCROLL is very large. A post-hoc t-test (with Bonferroni correction) actually shows that (i) *List* is significantly faster than *Spring* when LIST-SCROLL=*NO* ($p = 0.0003$) and that (ii) *Spring* is significantly faster than *List* when LIST-SCROLL=*YES* ($p = 0.0001$).

While our analyses consider trials that better reflect the behavior of expert users, results are similar when analyzing the entire set of measured trials. The only difference lies in the performance of *List*, which is slightly worse than that of *Spring*. For instance, interaction effect TECH × LIST-SCROLL remains very strong, but there is a significant difference between both techniques only when LIST-SCROLL=*YES*, whereas *List* is not significantly faster than *Spring* when LIST-SCROLL=*NO*.

In summary, DND^{-1}, combined with either *Spring* or *List*, lets users restore past locations of individual objects in an optimal way. However, the performance of *List* suffers from a high variability depending on the reciprocal drag-and-drop considered and on the history structure. On the opposite, by adopting a per-object navigation strategy, *Spring* limits this cost, yielding more constant performance figures across reciprocal drag-and-drop actions.

4.4. Part 2: Reciprocal Drag-and-Drop for Groups of Objects

4.4.1. Experiment Scene and Task. In the second part of the experiment, the application enables users to perform multiple rectangular selections through rubber-band interaction. It also allows them to add and remove individual objects using Shift+Click. The graphical scene is the same as in Part 1, but every *Forward* and *Backward* instruction involves several objects. Also, as opposed to Part 1, in which *Forward* and *Backward* instructions were interlaced, a scenario in Part 2 gives a series of *Forward* instructions and ends with a single *Backward* instruction. This final instruction indicates a target layout that can always be reached with a single (optimal) DND^{-1} operation.

As in Part 1, the application ensures that $History$ is the same across participants by constraining their interactions. For a *Forward* instruction, participants have to move all shapes involved at the same time, using the correct multiple selection. Otherwise, the application beeps and ignores the last move. For the final *Backward* instruction, the application enables any sequence of reciprocal drag-and-drop actions. It records them until the target layout is reached. Participants were also allowed to skip a trial

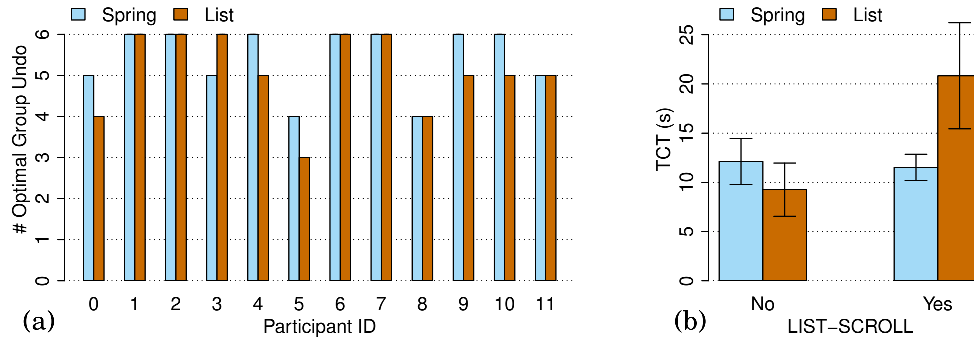


Fig. 23. (a) Total number of trials completed with a single optimal DND^{-1} among the six trials each participant performed with each TECH. (b) *TCT* by TECH × LIST-SCROLL.

on demand, if they were confused and unable to figure out how to reach the target layout. This could have happened if they had chained a large number of reciprocal drag-and-drop actions.

4.4.2. Design and Procedure. As in Part 1, each participant has to complete two blocks, one per TECH. When a block starts, the operator gives a 2-minute introduction about how to use the technique on a group of objects. Participants then complete a practice scenario, followed by six measured scenarios. We generate 12 scenarios (2 series of 6) in advance, that we present to all participants. To make sure each scenario is performed with each technique across participants, the presentation order between the two series of 6 scenarios is always the same, but the presentation order between the two TECH conditions varies (6 participants start with *Spring*, the 6 others start with *List*). We also counterbalance the presentation order of the 6 scenarios within a series by using a 6×6 Latin square.

In Part 2, each scenario contains from 9 to 11 *Forward* instructions that involve 3 or 4 subgroups of 6 specific graphical shapes among the 16 shapes in the scene. A subgroup consists of 2, 3, 4 or 6 shapes and is involved in 1 to 5 *Forward* instructions. The direction and amplitude of the drag-and-drop is randomly generated while avoiding overlap between shapes. The final *Backward* instruction is generated by picking one of a subgroup's past locations (the chosen subgroup must have been moved at least four times). The past location is either 1, 2 or 3 step(s) backward (LOCAL-DEPTH $\in \{1, 2, 3\}$). Half of the scenarios require scrolling with *List* (LIST-SCROLL=*YES*), while the other half does not (LIST-SCROLL=*NO*). In order to ensure an equivalent difficulty among the 2 sets of 6 scenarios, we generated six pairs of scenarios. Paired scenarios feature the same number of subgroups, which have the same size, and a similar sequence of instructions. They only differ in the shape of objects they contain, and in the amplitude and direction of each drag-and-drop.

4.4.3. Results for Part 2. As explained above, each final *Backward* instruction can be performed in a single optimal reciprocal drag-and-drop. The percentage of trials that were completed in such an optimal way was high in our experiment, with no significant difference between *Spring* (90.3%) and *List* (84.7%). Figure 23-a reports the number of trials that were completed in such an optimal way, per participant. Eight of the twelve participants made either 0 or 1 error. Moreover, among the 18 trials that were not completed with an optimal use of DND^{-1}, 9 required only 2 or 3 reciprocal drag-and-drop operations, and only 3 were considered as too difficult and skipped at the participant's request. These results show that, even if the notion of group necessarily

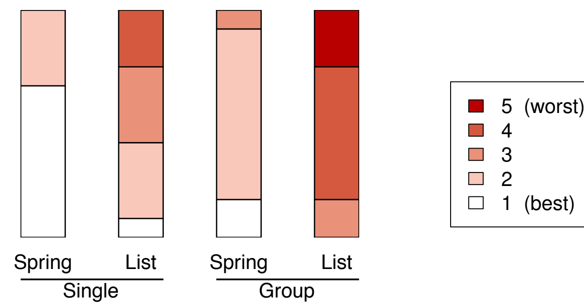


Fig. 24. Participants' subjective rating of difficulty with each TECH for both the single and group *Backward* instructions on a five-point Likert scale (1: best, ..., 5: worst).

introduces some difficulty in the task, participants succeed in using the DND^{-1} model to apply reciprocal drag-and-drop on groups of objects.

For our analysis of *TCT*, we first remove the 18 non-optimal trials mentioned above. As we did in Part 1, we also remove 11.45% of trials in the *List* condition, which correspond to cases where participants scrolled the viewport even though this was not necessary (LIST-SCROLL = *NO*). However, as opposed to our earlier observations about reciprocal drag-and-drop on single objects, there was no significant learning effect on *TCT*, neither for *Spring* nor *List*. Figure 23-b reports *TCT* by TECH for both LIST-SCROLL conditions and shows the same TECH $\times$ LIST-SCROLL interaction effect we already observed in Part 1. A t-test shows that *Spring* is significantly faster than *List* when LIST-SCROLL = *YES* ($p = 0.006$), while this difference is not significant when LIST-SCROLL = *NO* ($p = 1.0$). Finally, LOCAL-DEPTH did not have a significant effect on *TCT*.

4.5. Qualitative Questionnaire

At the end of the experiment, participants had to fill a questionnaire. They were asked about their preferred technique, and about their perception of how easy it was to use each technique for undoing moves on both individual objects and on groups of objects. All participants said that Dwell-and-Spring was their preferred technique, which is in accordance with the perceived difficulty of the different experimental tasks. Figure 24 illustrates this perception of difficulty as a score on a five-point Likert scale ([1: very easy ... 5: very difficult]). A pairwise Wilcoxon test with Holm correction reveals that this score significantly differs for all six pairs of tasks. In particular, participants found that *Spring* was easier to use than *List* for tasks on both individual objects and groups of objects.

4.6. Summary

Our empirical results suggest that the DND^{-1} model can effectively support undo for direct manipulations on both individual objects and multiple selections. This does not mean that participants understood all the details of DND^{-1}, but that they were able to use it effectively for the layout problem that they had to solve.

All participants expressed a preference for technique *Spring* over *List*. This probably comes from the fact that *List* is a global technique that displays the whole history of all objects, consequently suffering from issues related to the difficulty in identifying the right manipulation in the history. The first issue is the cost of searching in terms of motor control: the longer the history, the larger the number of required scrolling actions. The second issue is the cost of recognizing an object or a group of objects as what can be shown in the thumbnails is limited. Designing more meaningful thumbnails by emphasizing differentiating features between groups would help, but this is difficult to

achieve in an application-independent manner since we do not have any information regarding the semantics of the objects manipulated. By using an object as a reference to provide a contextual history in place, *Spring* does not suffer from such problems. In particular, *Spring* scales better than *List* with the size of the history, and better fits with the per-object nature of the DND^{-1} model.

5. RELATED WORK

Most applications handle a history of commands, as well as meta commands UNDO and REDO to navigate it. We review the different models implemented by existing applications to offer such functionalities. However, the DND^{-1} model should not be directly compared to any of these undo models. It is not intended to replace any of them, but rather to complement them. DND^{-1} helps users make a precise drag-and-drop and, if this manipulation actually invokes an application command, it will be handled by the application's undo model. In that regard, DND^{-1} is rather an enhancement to direct manipulation than an undo model.

5.1. Undo Models

In the most common case, the history of commands can be represented as a tree whose nodes are interface states. In the linear undo model, only a single child is associated with each node (the most recently visited one). This makes it impossible to access some previous states of the interface via undo. A few applications like Emacs [Gosling 1982; Yang 1992] provide users with a better experience, in terms of exploration, by making any previous state recoverable. This is achieved by considering UNDO as a regular command that also gets stored in the history. While this model is very powerful – any state can be restored – it remains a bit confusing, as users have to figure out how to break a flow of undo commands. Also, this model is global: reverting an object to a given past state entails undoing all commands that were performed afterwards, no matter which objects they were performed on. As the US&R model [Vitter 1984], the DND^{-1} model makes the whole branching of drag-and-drop history accessible. However, DND^{-1} provides a direct path to any past location, while the US&R model prompts the user each time a redo command is issued on nodes that have several children.

Selective Undo [Prakash and Knister 1994; Berlage 1994] lets users select a command to undo anywhere in the history. The original mechanism [Prakash and Knister 1994] basically removed the command to undo from the history, and then redid the commands that were coming after it in the history. However, this interpretation of the history of commands as a script may not always match users' mental model of the undo command [Cass et al. 2006]. With direct selective undo [Berlage 1994], application developers can implement an undo behavior that is more appropriate in this context, by adding an *inverse* command at the end of the history. The Amulet toolkit [Myers and Kosbie 1996] provides a good architecture to enable support for such undo mechanisms in graphical applications. However, selective undo remains difficult to implement since what the semantics of a reverse command should be is difficult to predict, as it depends on the current interface state. The DND^{-1} model can be described as direct selective undo for drag-and-drop at the level of individual objects and group of objects. It defines the inverse move to be appended to the history of an object or group as the straight movement from the latter's current location to one of its past locations.

Edwards et al. [2000] experimented with a clever approach to handle undo in Flatland. It consists of introducing commands that will be nested with past commands already stored in the history. As Flatland is a system intended to be used by several users, undo commands can also be region-based, regardless of when they happened. This concept of regional undo has also been applied to spreadsheet editors [Kawasaki and Igarashi 2004], interactive large displays [Seifried et al. 2012], code editors [Yoon

et al. 2013], VLSI systems [Zhou and Imamiya 1997], sketch-based interfaces [Oe et al. 2013] and painting applications [Myers et al. 2015]. However, to support such an advanced model of undo, applications must take into account a complex system of dependencies and causal relationships. The CAUSALITY model [Nancel and Cockburn 2014] identifies the different components and relations that an application should use to store their interaction history in order to support any advanced undo model.

Our model, DND^{-1}, focuses on direct manipulation. It is not intended to replace the functional undo model of applications, but to run in parallel with them. DND^{-1} uses a direct selective undo model to provide users with shortcuts that bring objects back to their exact past positions, which is something that would otherwise be difficult to do. In a sense, DND^{-1} offers a quick way to perform *forward* recovery [Abowd and Dix 1992], i.e., to do the right manipulation from the present state to reach a novel state that is similar to a past state for some objects' location. Actually, when users restore object O to a past location P, DND^{-1} simply sends a message asking the underlying application to translate O from its current location to P. But DND^{-1} does not have any idea of what the semantics of this object's move are in the application. If this move triggers the invocation of a command in the application, the command will be added to the undo model of that application, no matter the complexity of this model. If this move does not invoke any command, it will simply be ignored by the application's undo model. For instance, in a graphical editor, if DND^{-1} moves a slider that controls transparency while an object is selected on the canvas, the `setTransparency` command will be stored in the application's command history of the graphical editor. But if there is no object selected when DND^{-1} moves the slider, the application will ignore it. Object movements performed by DND^{-1} can have functional effects in a given application. Those effects can be appended to the application's own history of commands, as any object movement that is performed by users, no matter the complexity of the underlying history model.

5.2. Navigation in Command Histories

A few applications allow users to access their interaction's history. The most basic history representation is a list of text items as in, e.g., Adobe Photoshop. Clicking on an item reverts the document back to the state in which it was before this command got invoked, following a linear undo model. The most elaborate history representation is probably the Chronicle system [Grossman et al. 2010] that was also designed for image editing. This system proposes a sophisticated timeline of user actions and serves *chronicle* widgets on demand. A chronicle is an augmented video clip of what happened between two time stamps on a given image area. This approach provides a very exhaustive visualization of commands performed, that supports reflection and communication with others for, e.g., creating tutorials. But it is not intended to support undo navigation by restoring a previous state as Rekimoto envisioned with the Time-Machine concept [Rekimoto 1999].

Other approaches have been designed more specifically for selective undo. In the GINA system [Berlage 1994], history is a list of text commands. Users can filter them out using string pattern matching expressions. As it may be difficult to refer to a graphical object textually, users can also drag-and-drop an object onto the list to get the partial history specifically related to it. Other systems represent the history graphically. For instance, Meng et al. [1998] propose two types of history widgets for selective undo, with which users can browse a collection of snapshots that illustrate the different interface states. Representing the history as a picture list has also been investigated in a collaborative web site design tool [Klemmer et al. 2002], in a painting application to perform selective undo [Myers et al. 2015] and for representing users' operations on large and complex visualizations [Heer et al. 2008]. Chimera [Kurlander

and Feiner 1992] also exposes the history as a list of graphical panels, allowing users to edit an object in the history and propagate changes to the current state. Chimera relies on an application-dependent visual language to create a graphical representation of the command. This representation conveys the command's semantics better than a scaled-down snapshot of the full interface would, while using less screen real-estate. Nakamura and Igarashi [2008] later showed how to adopt such a comic strip metaphor in an application-independent manner, by emphasizing low-level user events and by using the Phosphor afterglow effect [Baudisch et al. 2006] on affected widgets. These systems can be powerful, but browsing collections of pictures requires heavy-weight widgets, which makes them ill-suited to performing fast iterations in graphical layout design tasks.

Dwell-and-Spring is a light-weight widget, that works in an application-independent manner, both in terms of graphical rendering and interaction with the widget. This is made possible by the fact that Dwell-and-Spring works at the local level (focus on objects or groups of objects), and has a more focused role: contrary to the heavy-weight widgets mentioned above, Dwell-and-Spring is not intended to navigate the whole history of an application, but is designed to navigate the history of past locations of the object it has been invoked on.

5.3. Enhancements to Direct Manipulation

As mentioned above, DND^{-1} focuses on the history of direct manipulations, providing shortcuts to drag-and-drop interactions that move objects and groups of objects. UIMarks [Chapuis and Roussel 2010] also offers shortcuts for graphical interfaces, enabling users to explicitly put some marks on the user interface and configure them to, e.g., facilitate invocation of frequently-used widgets. The drag-and-pop technique [Baudisch et al. 2003] also accelerates drag-and-drop interactions by using the direction of movement to predict and bring potential targets close to the dragged object.

Dwell-and-Spring implicitly records the start and end points of any press-drag-release interaction 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 [Hinckley et al. 2005], that suggests the use of 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 [Dragicevic 2004] proposes to cross window borders to fold windows during a drag-and-drop to facilitate navigation over windows. The trailing widget [Forlines et al. 2006] follows the cursor and can be grabbed with a quick movement to access a menu. Boomerang [Kobayashi and Igarashi 2007] allows users to suspend a drag-and-drop by using a throwing gesture.

Most graphical editors support an explicit grouping command, but very few systems support more lightweight grouping definitions as DND^{-1} does by keeping a trace of multiple selections. Among them, Bubble clusters [Watanabe et al. 2007] automatically cluster objects that have been brought close-by via direct manipulation. The Dynomite notebook application [Wilcox et al. 1997] groups graphical strokes that are spatially and temporally related, to enable users to undo the creation of a set of hand-written letters, in the same spirit as a text editor does with a series of characters that belong to the same word unit. QuickSelect [Su et al. 2009] proposes a similar propagation mechanism for enlarging the current selection based on previous multiple selections. Finally, Handle Flags [Grossman et al. 2009] also proposes extra grouping widgets (handles) that appear when the stylus approaches hand-written strokes. There are as

many handles as the number of potential groups, which are computed based on how the strokes are spatially clustered and how the user actually refined proposed groups to perform previous multiple selections. However, if these techniques also keep trace of previous selections as DND^{-1} does, they focus on the quick selection of these groups and do not address interaction with their history.

6. DISCUSSION AND LIMITATIONS

DND^{-1} provides users with a simple way of putting individual objects and groups of objects back to any location in their respective histories, regardless of what other movements were applied to other objects in the scene. As graphical user interfaces heavily rely on direct manipulation and object movements at large, DND^{-1} can be helpful in many cases, at both the system level and the application level: as discussed in this article (Section 2), reciprocal drag-and-drop actions may be useful for desktop and window management, view navigation, direct manipulation in vector graphics editors, and control of widgets such as sliders, scrollbars, color wheels, manipulation handles, etc.

Both the operating system and applications can register object moves to DND^{-1}. A naive implementation would consist in systematically registering the translation vector between the mouse press and release events (or the finger touch and lift events). However, this will fail in some cases, as graphical objects feature some tolerance with respect to the input movement, in order to make the motor action easier to perform. For example, when a snapping mechanism is implemented, or when users manipulate an object that only has a single degree of freedom (such as a slider using a 2D input device), the input movement performed by the user does not necessarily match the actual graphical object's movement. In such cases, what should be registered in DND^{-1} is the object's actual move rather than what the input device receives.

Another issue to consider is that some object movements are performed using another input device than the pointing device. For instance, users can scroll a document or move the currently selected object using arrow keys. The resulting object movements can still be registered in DND^{-1}. For example, when scrolling a document by maintaining the UP key pressed, the system can send a `moved`($scrollbar_{knob}$, d) to the model when the key is released. By doing so, the scrolling operation is featured in DND^{-1}'s history and can be reverted. To avoid registering successive micromovements, application designers can implement any policy for aggregating moves before sending a message to DND^{-1}. Sending `moved` messages would be triggered as soon as an input event of another type occurs, including those related to DND^{-1}'s invocation. For example, several small moves, applied in sequence to an object using arrow keys, can be aggregated into a single `moved` message with the overall amplitude corresponding to the sum of the small successive moves. The message would get sent as soon as users hit any other key or use a mouse button.

DND^{-1} has been designed to be *application-independent*. The downside of this choice is that it has no knowledge of the semantics of object movements in the underlying applications. In some cases, a reciprocal drag-and-drop will thus not correspond to the undo of the command that was invoked by the initial drag-and-drop. For example, when an icon is moved from one window to another, and the window arrangement is changed afterwards, moving back this icon using the inverse displacement might not put it back in the window (and folder) from which it was taken. In that regard, DND^{-1} shares one of the limitations of scripting systems that replay sequences based on cursor movements (an issue raised in [Myers 1998]). This is why the interaction techniques that are proposed to navigate DND^{-1} must provide a clear feedforward of what the result of a reciprocal drag-and-drop will be (as Dwell-and-Spring and DnD-List do), so as to help users anticipate and prevent such errors.

DND^{-1} can also be qualified as *selection agnostic*. There are two main reasons for making this choice. First, an application-independent approach requires establishing a common vocabulary across all applications for the communication protocol between the model and the applications. As selection mechanisms can be different from one application to another[4], establishing a common protocol that would consider an orthogonal notion of *selection* could have been confusing and would have inherently led to more complexity in the communication protocol. Second, objects belonging to a given selection may have different histories. This is an issue with regional undo that is far from trivial to address [Li and Li 2003; Yoon et al. 2013]. As our approach is focused on drag-and-drop, the notion of group is independent from the current selection, and is only implicitly defined when users move several objects together by mean of a drag-and-drop on a multiple selection. While this is a restriction of our model, it has the advantage of containing its complexity.

Even if the DND^{-1} model itself does not conceptually interfere with the notion of selection in applications, the choice of interaction technique used to navigate the history may still have a side effect on the application's selection, if the input events it relies on interfere with those used by the selection mechanism. For example, a reciprocal drag-and-drop invoked using DnD-List does not require any direct interaction with the application's objects and will thus always let the current selection unmodified. On the contrary, invoking Dwell-and-Spring requires performing a long press directly on the object or group that should be reverted. If the application relies on a click (quick press-release sequence) to select an object, Dwell-and-Spring will not interfere and will leave the selection as is. But if the application relies on a simple press to select an object, the object on which Dwell-and-Spring has been invoked will become the new selection. Also, on touch screens that have a limited input vocabulary, long press events may already be mapped to another action, such as entering an *edit* mode (e.g., for moving app icons on home screens on a smartphone) or invoking a contextual menu. The spring handle is offset with respect to the user's finger and should allow him to ignore it if he does not actually want to trigger a reciprocal drag-and-drop. However, additional UI design work may be required to handle some specific cases where Dwell-and-Spring's footprint may still interfere with the application's controls.

Finally, the extended version of Dwell-and-Spring that we introduced in Section 2 may generate some visual clutter, especially if the object on which it is invoked has been moved a large number of times or if it has been involved in numerous multiple selections. How much visual interference this might cause depends on the nature of the graphical scene below, and on how past locations and target locations are distributed over it. To reduce potential clutter, the DND^{-1} model can be implemented with a maximum length of past steps per group of objects kept in history[5]. But a more interesting solution would consist in designing a graphical footprint that progressively becomes less intrusive as users get increasingly familiar with the widget. The feedforward of a reciprocal drag-and-drop could be displayed as a simple straight line, rather than as a series of springs that explicitly represent the full sequence of drag-and-drop that will get reverted. We believe that a richer graphical representation helps explore the history and understand the widget, but that it is not necessary to effectively use it. Our experience with it makes us think that what is the most important, once fa-

[4]Consider, e.g., advanced selection mechanisms like that of Adobe Illustrator, where there are two levels of selection – vertex and shape – in comparison with the absence of the notion of selection on graphical controllers such a slider knob.

[5]The number of past steps has an impact on the widget's footprint, which may overflow the screen when invoked close to its left or bottom edge. In such cases, the widget's layout could be mirrored vertically and/or offset in the spirit of how contextual menus behave in desktop interfaces.

miliar with how the widget works, is the overlaid feedforward that helps anticipate the result when brushing through the different spring handles. We plan to enhance DND^{-1}'s communication protocol in order to let the application developer specify, at the object level, the best type of feedforward (a series of lines or springs, or a simple line or spring). This could be especially relevant for objects that are always moved along a single axis, such as scrollbar knobs or sliders.

7. CONCLUSION

We introduced DND^{-1}, a model that keeps track of all past locations of objects that can be moved through drag-and-drop interactions in a graphical user interface, including windows and other view management widgets. We have extended the Dwell-and-Spring technique and combined it with DND^{-1} in order to cover a large number of scenarios in which users need to perform a reciprocal drag-and-drop action and that were identified as limitations in [Appert et al. 2012]. With this extended version of Dwell-and-Spring, users can now recover any past location of a single object or group of objects. First, they have access to the whole history of locations that an object has visited, and can thus revert a series of drag-and-drop actions in a single step. Second, they have access to the history of drag-and-drop actions on past multiple object selections, and can thus revert a drag-and-drop on a group of objects without breaking the current active selection.

In the experiments that we conducted, participants were able to understand how the technique works and how it makes it easier to perform reciprocal drag-and-drop actions. They were also able to use it to solve graphical layout tasks in which advanced reciprocal drag-and-drop actions are required. We plan to run a field study that will focus on the potential distractions that the widget popping up might cause. However, as discussed in this article, we advocate for an implementation where the graphical representation of Dwell-and-Spring gets lighter as the user becomes familiar with it. The analogy with a physical spring is especially useful in the discovery phase of the technique, but it probably becomes less so when users actually know how to use it and want to optimize time.

REFERENCES

Gregory D. Abowd and Alan J. Dix. 1992. Giving undo attention. *Interacting with Computers* 4, 3 (Dec. 1992), 317–342. DOI:http://dx.doi.org/10.1016/0953-5438(92)90021-7

Jason Alexander, Andy Cockburn, Stephen Fitchett, Carl Gutwin, and Saul Greenberg. 2009. Revisiting read wear: analysis, design, and evaluation of a footprints scrollbar. In *Proceedings of the 27th international conference on Human factors in computing systems (CHI '09)*. ACM, 1665–1674. DOI:http://dx.doi.org/10.1145/1518701.1518957

Caroline Appert and Michel Beaudouin-Lafon. 2008. Swingstates: Adding state machines to java and the swing toolkit. *Software: Practice and Experience* 38, 11 (Sept. 2008), 1149–1182. DOI:http://dx.doi.org/10.1002/spe.v38:11

Caroline Appert, Olivier Chapuis, and Emmanuel Pietriga. 2012. Dwell-and-spring: Undo for direct manipulation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, 1957–1966. DOI:http://dx.doi.org/10.1145/2207676.2208339

Patrick Baudisch, Edward Cutrell, Dan Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson, and Alex Zierlinger. 2003. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch-and pen-operated systems. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT '03)*. IOS & IFIP, 57–64. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.9085

Patrick Baudisch, Desney Tan, Maxime Collomb, Dan Robbins, Ken Hinckley, Maneesh Agrawala, Shengdong Zhao, and Gonzalo Ramos. 2006. Phosphor: explaining transitions in the user interface using afterglow effects. In *Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06)*. ACM, 169–178. DOI:http://dx.doi.org/10.1145/1166253.1166280

Thomas Berlage. 1994. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction* 1, 3 (Sept. 1994), 269–294. DOI:http://dx.doi.org/10.1145/196699.196721

Aaron G. Cass, Chris S. T. Fernandes, and Andrew Polidore. 2006. An empirical evaluation of undo mechanisms. In *Proceedings of the 4th Nordic conference on Human-computer interaction (NordiCHI '06)*. ACM, 19–27. DOI:http://dx.doi.org/10.1145/1182475.1182478

Olivier Chapuis and Nicolas Roussel. 2010. UIMarks: Quick graphical interaction with specific targets. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology (UIST '10)*. ACM, 173–182. DOI:http://dx.doi.org/10.1145/1866029.1866057

Pierre Dragicevic. 2004. Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows. In *Proceedings of the 17th annual ACM symposium on User interface software and technology (UIST '04)*. ACM, 193–196. http://doi.acm.org/10.1145/1029632.1029667

W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. 2000. A temporal model for multi-level undo and redo. In *Proceedings of the 13th annual ACM symposium on User interface software and technology (UIST '00)*. ACM, 31–40. DOI:http://dx.doi.org/10.1145/354401.354409

W. Keith Edwards and Elizabeth D. Mynatt. 1997. Timewarp: techniques for autonomous collaboration. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '97)*. ACM, 218–225. DOI:http://dx.doi.org/10.1145/258549.258710

Clifton Forlines, Daniel Vogel, and Ravin Balakrishnan. 2006. Hybridpointing: Fluid switching between absolute and relative pointing with a direct input device. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, 211–220. DOI:http://dx.doi.org/10.1145/1166253.1166286

James Gosling. 1982. *Unix Emacs Reference Manual*. Carnegie-Mellon University, Pittsburgh, PA, USA.

Tovi Grossman, Patrick Baudisch, and Ken Hinckley. 2009. Handle flags: Efficient and flexible selections for inking applications. In *Proceedings of Graphics Interface (GI '09)*. Canadian Information Processing Society, 167–174. http://dl.acm.org/citation.cfm?id=1555880.1555918

Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2010. Chronicle: Capture, exploration, and playback of document workflow histories. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology (UIST '10)*. ACM, 143–152. DOI:http://dx.doi.org/10.1145/1866029.1866054

Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. 2008. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics* 14, 6 (Nov. 2008), 1189–1196. DOI:http://dx.doi.org/10.1109/TVCG.2008.137

Ken Hinckley, Patrick Baudisch, Gonzalo Ramos, and Francois Guimbretiere. 2005. Design and analysis of delimiters for selection-action pen gesture phrases in Scriboli. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*. ACM, 451–460. DOI:http://dx.doi.org/10.1145/1054972.1055035

Akrivi Katifori, George Lepouras, Alan Dix, and Azrina Kamaruddin. 2008. Evaluating the significance of the desktop area in everyday computer use. In *First International Conference on Advances in Computer-Human Interaction (ACHI '08)*. IEEE, 31–38. DOI:http://dx.doi.org/10.1109/ACHI.2008.27

Yoshinori Kawasaki and Takeo Igarashi. 2004. Regional undo for spreadsheets. In *UIST '04 Demonstration Abstract*. ACM, 2 pages. http://www-ui.is.s.u-tokyo.ac.jp/~kwsk/undo/kawasaki_uist04_regional.pdf

Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. 2002. Where do web sites come from?: Capturing and interacting with design history. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '02)*. ACM, 1–8. DOI:http://dx.doi.org/10.1145/503376.503378

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987. DOI:http://dx.doi.org/10.1109/TSE.2006.116

Masatomo Kobayashi and Takeo Igarashi. 2007. Boomerang: Suspendable drag-and-drop interactions based on a throw-and-catch metaphor. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, 187–190. DOI:http://dx.doi.org/10.1145/1294211.1294243

David Kurlander and Steven Feiner. 1988. Editable graphical histories. In *IEEE Workshop on Visual Languages*. IEEE, 127–134. DOI:http://dx.doi.org/10.1109/WVL.1988.18020

David Kurlander and Steven Feiner. 1992. A history-based macro by example system. In *Proceedings of the 5th annual ACM symposium on User interface software and technology (UIST '92)*. ACM, 99–106. DOI:http://dx.doi.org/10.1145/142621.142633

Henry A. Landsberger. 1958. *Hawthorne Revisited: Management and the Worker, Its Critics, and Developments in Human Relations in Industry.* Cornell University, Ithaca, New York, NY, USA.

Rui Li and Du Li. 2003. A regional undo mechanism for text editing. In *Proceedings of the 5th International Workshop on Collaborative Editing Systems (IWCES '03)*. Citeseer. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.4754&rep=rep1&type=pdf

Chii Meng, Motohiro Yasue, Atsumi Imamiya, and Xiaoyang Mao. 1998. Visualizing histories for selective undo and redo. In *Proceedings of the Third Asian Pacific Computer and Human Interaction (APCHI '98)*. IEEE, 459–464. DOI: http://dx.doi.org/10.1109/APCHI.1998.704487

Brad A. Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*. ACM Press/Addison-Wesley Publishing Co., 534–541. DOI: http://dx.doi.org/10.1145/274644.274716

Brad A. Myers and David S. Kosbie. 1996. Reusable hierarchical command objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '96)*. ACM, 260–267. DOI: http://dx.doi.org/10.1145/238386.238526

Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt. 2015. Selective undo support for painting applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, 4227–4236. DOI: http://dx.doi.org/10.1145/2702123.2702543

Toshio Nakamura and Takeo Igarashi. 2008. An application-independent system for visualizing user operation history. In *Proceedings of the 21st annual ACM symposium on User interface software and technology (UIST '08)*. ACM, 23–32. DOI: http://dx.doi.org/10.1145/1449715.1449721

Mathieu Nancel and Andy Cockburn. 2014. Causality: A conceptual model of interaction history. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, 1777–1786. DOI: http://dx.doi.org/10.1145/2556288.2556990

Tatsuhito Oe, Buntarou Shizuki, and Jiro Tanaka. 2013. Undo/redo by trajectory. In *Proceedings of the 15th International Conference on Human-Computer Interaction: Interaction Modalities and Techniques - Volume Part IV (HCI'13)*. Springer-Verlag, 712–721. DOI: http://dx.doi.org/10.1007/978-3-642-39330-3_77

Atul Prakash and Michael J. Knister. 1994. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction* 1, 4 (Dec. 1994), 295–330. DOI: http://dx.doi.org/10.1145/198425.198427

Jun Rekimoto. 1999. Time-machine computing: a time-centric approach for the information environment. In *Proceedings of the 12th annual ACM symposium on User interface software and technology (UIST '99)*. ACM, 45–54. DOI: http://dx.doi.org/10.1145/320719.322582

Dario D Salvucci. 2001. An integrated model of eye movements and visual encoding. *Cogn. Syst. Res.* 1, 4 (Feb. 2001), 201–220. DOI: http://dx.doi.org/10.1016/S1389-0417(00)00015-2

Thomas Seifried, Christian Rendl, Michael Haller, and Stacey Scott. 2012. Regional undo/redo techniques for large interactive surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, 2855–2864. DOI: http://dx.doi.org/10.1145/2207676.2208690

Katie A. Siek, Yvonne Rogers, and Kay H. Connelly. 2005. Fat finger worries: How older and younger users physically interact with PDAs. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT '05)*. Springer-Verlag, 267–280. DOI: http://dx.doi.org/10.1007/11555261_24

Sara L. Su, Sylvain Paris, and Frédo Durand. 2009. QuickSelect: History-based selection expansion. In *Proceedings of Graphics Interface 2009 (GI '09)*. CIPS, 215–221. http://dl.acm.org/citation.cfm?id=1555880.1555929

Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. Variation in element and action: Supporting simultaneous development of alternative solutions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, 711–718. DOI: http://dx.doi.org/10.1145/985692.985782

Jeffrey S. Vitter. 1984. US&R: A new framework for redoing (extended abstract). *ACM SIGSOFT Software Engineering Notes* 9, 3 (April 1984), 168–176. DOI: http://dx.doi.org/10.1145/390010.808262

Nayuko Watanabe, Motoi Washida, and Takeo Igarashi. 2007. Bubble clusters: an interface for manipulating spatial aggregation of graphical objects. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. ACM, 173–182. DOI: http://dx.doi.org/10.1145/1294211.1294241

Lynn D. Wilcox, Bill N. Schilit, and Nitin Sawhney. 1997. Dynomite: a dynamically organized ink and audio notebook. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems (CHI '97)*. ACM, 186–193. DOI: http://dx.doi.org/10.1145/258549.258700

Yiya Yang. 1992. Anatomy of the design of an undo support facility. *International journal of man-machine studies* 36, 1 (Jan. 1992), 81–95. DOI: http://dx.doi.org/10.1016/0020-7373(92)90053-N

YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. IEEE, 119–126. DOI:http://dx.doi.org/10.1109/VLHCC.2013.6645254

Chunbo Zhou and Atsumi Imamiya. 1997. Object-based nonlinear undo model. In *Proceedings of the Computer Software and Applications Conference (COMPSAC '97)*. IEEE, 50–55. DOI:http://dx.doi.org/10.1109/CMPSAC.1997.624739

Received May 2015; revised July 2015; accepted August 2015