Extending drag-and-drop to new interactive environments: a multi-display, multi-instrument and multi-user approach
Research report RR-08012

Maxime Collomb, Mountaz Hascoët
Univ. Montpellier II – LIRMM – CNRS
161 rue Ada, 34 392 Montpellier Cedex 5 - France.
collomb@lirmm.fr, mountaz@lirmm.fr

Drag-and-drop is probably one of the most successful and generic representations of direct manipulation in today's WIMP interfaces. At the same time, emerging new interactive environments such as distributed display environments or large display surface environments have revealed the need for an evolution of drag-and-drop to address new challenges. In this context, several extensions of drag-and-drop have been proposed over the past several years. However, implementations for these extensions are difficult to reproduce, integrate and extend. This situation hampers the development or integration of advanced drag-and-drop techniques in applications.

The aim of this paper is to propose a unifying implementation model of drag-and-drop and of its extensions. This model –called M-CIU– aims at facilitating the implementation of advanced drag-and-drop support by offering solutions to problems typical of new emerging environments. The model builds upon a synthesis of drag-and-drop implementations, an analysis of requirements for meeting new challenges and a dedicated interaction model based on instrumental interaction. By using this model, a programmer will be able to implement advanced drag-and-drop supporting (1) multi-display environments, (2) large display surfaces and (3) multi-user systems. Furthermore by unifying the implementation of all existing drag-and-drop approaches, this model also provides flexibility by allowing users (or applications) to select the most appropriate drag-and-drop technique depending on the context of use. For example, a user might prefer to use pick-and-drop when interacting with multiple displays attached to multiple computers, push-and-throw or drag-and-throw when interacting with large displays and possibly standard drag-and-drop in a more traditional context. Finally, in order to illustrate the various benefits of this model, we provide an API called PoIP which is a Java-based implementation of the model that can be used with most Java-based applications. We also quickly describe Orchis, an interactive graphical application used to share bookmarks and that uses PoIP to implement distributed drag-and-drop like interactions.

1. Introduction

Even though *drag-and-drop* has been integrated widely, implementations vary significantly from one windowing system to another or from one toolkit to another (Collomb and Hascoët, 2005). This situation is worse for drag-and-drop extensions such as pick-and-drop, drag-and-pop, push-and-throw, etc. As far as these extensions are concerned, very little support if any is usually provided and implementations are hard to reuse, generalize or extend as new needs arise.

As the number of drag-and-drop extensions is increasing, it is important to propose a unified framework that clarifies the field and offers benefits from at least three perspectives: user's perspective, design perspective and implementation perspective. From a user's perspective, such a unification will hopefully make it possible for users to choose the type of drag-and-drop they like best or that best suits some specific environment or task. From a designer's perspective, a unifying framework will help better understand differences between the possible techniques and the design dimensions at stake. Lastly, from the programmer's perspective, a unifying implementation model should save a lot of time and efforts in the development of different of drag-and-drop extensions in new emerging and challenging environments such as large displays and distributed

display environments. The aim of this paper is to propose the basis for building such a framework, including a unified and open implementation model.

In the following section, we discuss how emerging interactive environments bring new challenges for the drag-and-drop paradigm, we review most extensions proposed so far and propose a unified framework for comparing them. In the next section, we present a new implementation model that not only builds upon an analysis of requirements for adapting to new emerging interactive environments but also accounts for implementation models of existing solutions in most widespread windowing systems or toolkits (Collomb and Hascoët, 2005). This model is based on the definition of instruments (Beaudouin-Lafon, 2000) that embody interaction techniques. The presentation of the implementation model at a generic level is based on a generic type of instrument that embodies basic drag-and-drop interaction styles. We further discuss how specific emerging drag-and-drop extensions can be implemented as specific instruments that are smoothly integrated into the model.

2. New challenges for the drag-and-drop paradigm

Most challenges that drag-and-drop has faced recently can be attributed to the emergence of new display environments such as wall-size displays or distributed display environments (DDE). DDE have been defined (Hutchings et al., 2005) as

> "Computer systems that present output to more than one physical display."

This general definition covers a broad range of systems. Consequently, systems that can be studied in this field can exhibit huge differences. An example of such a huge difference is the difference between two typical types of DDE: (1) multiple displays attached to the same machine and (2) multiple displays attached to different machines. While both configurations can be considered as DDE, the degree of integration of the distinct displays is very different. In the first case, the different displays are handled within the same windowing system, offering a single workspace or desktop with full communication capacities between windows of the different displays. In the second case, on the contrary, the two displays are handled by two distinct and potentially heterogeneous windowing systems, making it much more difficult to offer similar single workspace spanning the different displays. These two different configurations lead to significantly different types of problems when implementing drag-and-drop in these environments.

2.1. Preliminary definitions

In order to better characterize the types of problems that are most challenging for drag-and-drop and its variants, preliminary definitions are useful. In this paper, we use a generally agreed definition of the term *display*: a physical device used to display information. The term *window* is used to refer to an area of a display devoted to handle input and output from various programs.

Based on these definitions, we define a *surface* as a set of windows. We further define the term *distributed surface* as a surface with two additional properties: (1) windows of the surface potentially appear on different displays attached to different machines handled through different windowing systems, (2) interactions between windows handled by different windowing systems is transparent for the user, e.g it is similar to single interactions that happen in a single workspace.

For example, a set of windows spanning three different displays handled by three different machines, such as a laptop running MacOS, a PC running X-Windows and a tablet PC running Windows can be considered as a distributed surface as soon as a system makes it possible to surpass the boundaries of each windowing system to support some interactions between different windows on different displays. It is important to stress that systems handling distributed surface environments are de-facto potentially multi-user systems. Indeed, each machine involved in the surface can be controlled by a different user. To

some extent they can be considered as a specific type of groupware environment.

Our aim is to propose solutions for typical problems arising when designing and implementing a drag-a-drop like interaction in a distributed surface environment. These problems can be structured in three categories: (1) usability and scalability problems, (2) multi-computer and interoperability problems and (3) multi-user and concurrency problems.

2.2. Usability and scalability issues

Usability problems derive from the complexity of new environments and the usage variety: heterogeneity of displays both in terms of size, number and nature, heterogeneity of users in terms of abilities, experience and style. Over years, drag-and-drop basic paradigm has been extended with new interaction styles. Most of these extensions address specific needs for particular display environments. Consequently, we now have interesting alternatives to the drag-and-drop original style that best suits some contexts. These alternatives will be discussed in section 2. One benefit of our approach is to support different styles of interaction in a unified implementation model so that shifting from one interaction style to another is facilitated. This feature can be seen as a particular support for plasticity:

> "the capacity of a user interface to withstand variations of both the system physical characteristics and the environment while preserving usability"

(Thevenin and Coutaz, 1999).

Other types of usability problems that arise with emerging environments can be considered as *scalability* due to the increasing space available on large wall-sized displays: to what extent a technique originally designed to work on one single and relatively low resolution display will adapt to an increasing number, size and resolution of displays? When such displays are used with direct pointing devices, e.g in the iRoom (Stanford University, 2008) or DynaWall (Fraunhofer institute, 2008), the original drag-and-drop paradigm reaches its limits : interactions that involve dragging objects tend to be particularly tedious and error-prone (Collomb and Hascoët, 2004; Collomb et al., 2005) and can be further complicated by the bezels separating screen units (Baudisch et al., 2003). Drag-and-drop might even fail when targets are out of reach, e.g. located too high or too low on a display. Further, user performances in terms of time necessary to complete a task are known to decrease as the size of displays increases because they induce greater distances between targets and sources and target acquisition time is known to increase with distance (Fitts, 1992).

2.3. Distributed display surfaces: transparently integrating multiple computers and multiple windowing systems

Multi-computer and multi-windowing system problems are probably the most challenging problems that have to be addressed to handle distributed surfaces. Because of the difficulty in surpassing the boundaries of windowing systems associated with each display, there is still very little support for making windows of a distributed surface behave as if they were part of the same workspace. Some systems (Shoeneman, 2008; Johanson et al., 2002; Lachenal, 2004) aim to support communication between displays based on redirection of input/output mechanisms, but support is still at its early stage.

Other approaches like those found in distributed visualization environments provide multi-head support for multiple displays attached to different machines (Xdmx project, 2008; Humphreys et al., 2002). These systems provide advanced support for distributed surfaces. However, they do not have the flexibility needed to handle heterogeneous or dynamic distributed surfaces. Indeed, they impose strict constraints on the architecture of clusters of machines used and on the windowing systems or graphic toolkit that is run by these machines. Clearly they are not aimed at handling evolving sets of machines running heterogeneous windowing systems and graphical toolkits. Our model, on the contrary, imposes no particular constraints on the machines involved in distributed surfaces

and it also supports evolving configurations so that windows from new machines can be dynamically added or removed from a distributed surface.

2.4. Multi-user issues

Amongst all problems that arise with new display environments, some multi-user problems are typical of the field of computer-supported collaborative work (CSCW) and have been studied over the past decades (Greenberg and Marwood, 1994). As far as drag-and-drop is concerned, the important aspect of multi-user support is that one is able to distinguish event streams from different users. By fulfilling this requirement, our model can also be used by a programmer willing to integrate advanced drag-and-drop features in a CSCW context.

3. Drag-and-drop extensions

Problems listed in the previous section are partially addressed by a set of drag-and-drop extensions that have been proposed over the past 10 years. In this domain, it is possible to distinguish between different types of approaches depending on whether the underlying interaction model is *target-oriented*, *source-oriented* or *undirected*. This section rapidly reviews the various extensions proposed recently according to these three categories. The next section further compares these extensions according to more detailed dimensions.

3.1. Target-oriented interaction

Hereafter *target-oriented* interaction refers to an interaction style in which the main focus and feedback is located around potential targets locations. In most cases, with target-oriented interaction, users have to adjust their move continuously around potential target locations to finally acquire the right target at its real location. These continuous adjustments may have marked impacts on the systems since they imply high refresh rates: a little lag between the user's hand movement and the feedback would significantly decrease the usability of such interaction style.

Furthermore, to be effective, target-oriented instruments should be used with displays where targets are all roughly equally within sight. In very large wall-size displays it might be difficult to clearly distinguish targets very far away from the source location. In such environments, target-oriented instruments would fail and source-oriented instruments would be more suitable.

In this section, we quickly review most recent extensions that belong to this category: *throwing*, *drag-and-throw* and *push-and-throw*.

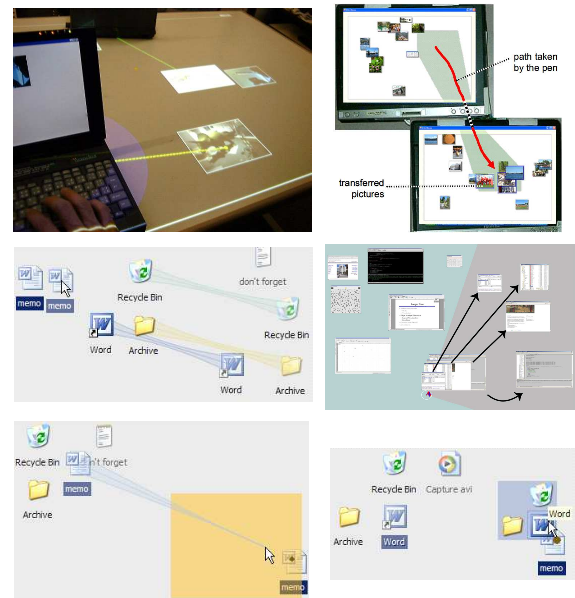

Figure 1. (Left to right, top to bottom) Examples of (a) hyperdragging, (b) stitching, (c) drag-and-pop, (d) vacuum (black arrows are added), (e) push-and-throw and (f) push-and-pop. (reproductions with authors permission)

Geißler (Geißler, 1998; Streitz et al., 1999) proposed three techniques to work more efficiently on interactive walls. The goal was to limit phys-

ical displacement of the user on a 4.5 x 1.1 m triple display (the DynaWall (Fraunhofer institute, 2008)). The first technique is *shuffling*. It is a way of re-arranging objects within a medium-sized area. Objects move by one length of their dimensions in a direction given by a short stroke by the user on the appropriate widget. Next, the author proposes a *throwing* technique. To throw an object, the user has to achieve a short stroke in direction opposite to which the object should be moving, followed by a longer stroke in the correct direction. The length ratio between the two strokes determines the distance to which the object will be thrown. According to the author, this technique requires training to be used in an efficient way. The third technique, *taking*, is an application of pick-and-drop (see section 3.3) tailored to the DynaWall.

Drag-and-throw and *push-and-throw* (Hascoët, 2003; Collomb and Hascoët, 2004) are throwing techniques designed for multiple displays (one or more computers). They address the limitation of throwing techniques (Geißler, 1998; Streitz et al., 1999) providing users with a real-time preview of where the dragged object will come down if thrown. These techniques are based on visual feedbacks, metaphors and the explicit definition of trajectories (fig. 1–e). Three types of visual feedback are used: trajectory, target and take-off area (area that matches to the complete display). Drag-and-throw and push-and-throw have different trajectories: drag-and-throw uses the archery metaphor (user performs a reverse gesture - to throw an object on the right, the pointer has to be moved to the left) while push-and-throw uses the pantograph metaphor (user's movements are amplified). The main strength of these techniques is that the trajectory of the object can be visualized and controlled before the object is actually sent. So users can adjust their gesture before validating it. Therefore, contrary to other throwing techniques, drag-and-throw and push-and-throw have very low error rates (Collomb and Hascoët, 2004).

3.2. Source-oriented interaction

We use the term *source-oriented* to characterize drag-and-drop extensions where the focus remains around the source object's original location. Contrary to target-oriented interaction, a source-oriented interaction usually does not involve continuous adjustments: a single fairly dramatic movement from the user is usually sufficient to complete the task. However, in some cases, e.g. in drag-and-pop, different drag directions cause the tip cluster to be a little different every time, users might need to re-orient once or twice to identify the correct tip icons. For such interaction styles, additional feedback (e.g. rubber bands) is usually provided to help in identifying the real location of the objects that correspond to the ghost.

In this section we review quickly most recent extensions that can be considered target-oriented: *drag-and-pop*, *vacuum*, and *push-and-pop*.

Drag-and-pop (Baudisch et al., 2003) is intended to help drag-and-drop operations when the target is impossible or hard to reach, e.g., because it is located behind a bezel or far away from the user. The principle of drag-and-pop is to detect the beginning of a drag-and-drop and to move potential targets toward the user's current pointer location. Thus, the user can interact with these icons using small movements. As an example, the case of putting a file in the recycle bin, the user starts the drag gesture toward the recycle bin (fig. 1–c). After a few pixels, each valid target on the drag motion direction creates a linked tip icon that approaches the dragged object. Users can then drop the object on a tip icon. When the operation is complete, tip icons and rubber bands disappear. If the initial drag gesture has not the right direction and thus the target icon is not part of the tip icons set, tip icons can be cleared by moving the pointer away from them but the whole operation has to be restarted to get a new set of tip icons.

The vacuum (Bezerianos and Balakrishnan, 2005) (figure 1–d), a variant of drag-and-pop, is a circular widget with a user controllable arc of influence that is centered at the widget's point of invocation and spans out to the edges of the display. Far away objects standing inside this influence arc are brought closer to the widget's center

in the form of proxies that can be manipulated in lieu of the originals.

Push-and-pop (Collomb et al., 2005) was created to combine the strengths of drag-and-pop and push-and-throw techniques. It uses the take-off area feedback from push-and-throw while optimizing the use of this area (fig. 1–f): it contains full-size tip icons for each valid target. The notion of valid target and the grid-like arrangement of tip icons are directly inherited from drag-and-pop's layout algorithm. The advantage over drag-and-pop is that it eliminates the risk of invoking a wrong set of targets. And the advantage over push-and-throw is that it offers better readability (icons are part of the take-off area), target acquisition is easier (Collomb et al., 2005) and users can focus on the take-off area.

3.3. Undirected interaction

Lastly, some interaction styles are neither source-oriented nor target-oriented. We call them *undirected* since feedback will not be concentrated in one particular area of the display. *Pick-and-drop*, *stitching* and *hyperdragging* are the main examples of extensions that fall into this category, and we review them in this section.

Pick-and-drop (Rekimoto, 1997) has been developed to allow users to extend drag-and-drop to distributed environments. While drag-and-drop requires the user to remain on the same computer while dragging objects around, pick-and-drop lets him move objects from one computer to another using direct manipulation. This is done by giving the user the impression of physically taking an object on a surface and laying it on another surface. Pick-and-drop is closer to the copy-paste interaction technique than to drag-and-drop. Indeed like the copy/paste operation, it requires two different steps: one to select the object to transfer, and one to put the object somewhere else. But pick-and-drop and drag-and-drop share a common advantage over copy-paste techniques: they avoid the user having to deal with a hidden clipboard. However, pick-and-drop is limited to interactive surfaces which accept the same type of touch-pen devices and which are part of the same network. Each pen has a unique ID and data is associated with this unique ID and stored on a pick-and-drop server.

Hyperdragging (Rekimoto and Saitoh, 1999) (figure 1–a) is part of a computer augmented environment. It helps users smoothly interchange digital information between their laptops, table or wall displays, or other physical objects. Hyperdragging is transparent to the user: when the pointer reaches the border of a given display surface, it is sent to the closest shared surface. Hence, the user can continue his movement as if there was only one computer. To avoid confusion due to multiple simultaneous hyperdragging, the remote pointer is visually linked to the computer controlling the pointer (simply by drawing a line on the workspace).

Stitching (Hinckley et al., 2004) (figure 1–b) is an interaction technique designed for pen-operated mobile devices. The devices that allow it to start a drag-and-drop gesture on a screen and to end the gesture on another screen. Devices have to support networking. A user starts dragging an object on the source screen, reaches its border, then crosses the bezel and finishes the drag-and-drop on the target screen. The two parts of the strokes are synchronized at the end of the operation and then bound devices are able to transfer data.

4. Comparison of extensions

The extensions presented in the previous section differ in several ways. The instrumental interaction model (Beaudouin-Lafon, 2000) can be useful to exhibit dimensions for a better comparison of their interaction styles. In this section, we quickly review the instrumental interaction model. Based on this model, we exhibit dimensions such as instrument feedback and instrument coverage. By considering these dimensions and others (drag-over and drag-under types of feedback), we further draw a comparison of previous approaches which is summarized in Table of Figure 3.

4.1. Instrumental interaction

Instrumental interaction consists of describing interactions through *instruments*. An instrument can be considered as a mediator between the user and domain objects. The user acts on the instrument, which transforms the user's *actions* into *commands* affecting relevant target objects. Instruments have *reactions* that enable users to control their *actions* on the instrument, and provide *feedback* as the command is carried out on target objects (see figure 2).

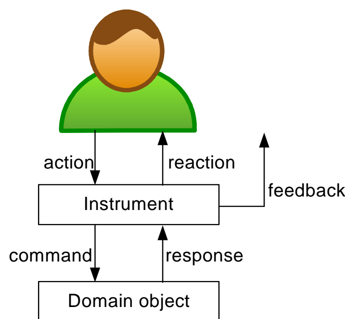

Figure 2. Interaction instrument mediates the interaction between user and domain objects (Beaudouin-Lafon, 2000).

Different extensions of drag-and-drop can be embodied through different instruments. Interactions between an instrument and domain objects (commands/responses) are the same for drag-and-drop and all the extensions presented previously, i.e. all instruments support primitive and generic commands: source selection, target selection, specification of type of action, data transfer (validation of the selected target) and cancellation.

The most important part of typical drag-and-drop interactions concerns interactions between the user and the instrument (principally reactions and feedback). Reactions and feedback of instruments involve three types of feedback: drag-under feedback, drag-over feedback and instrument feedback. When a user needs to change from one instrument to another, drag-under and drag-over visual effects might roughly be preserved, but instrument feedback and reaction vary significantly. For simplification, in the following sections we assimilate instrument feedback and reaction into a single concept we refer to as instrument feedback.

4.2. Drag-under and drag-over feedback

In regular drag-and-drop operations, feedback is usually referred to as *drag-under feedback* and *drag-over feedback*. Drag-over feedback consists mainly of feedback that occurs on a source object. Typically, during a regular drag on a source, the pointer shape changes into a drag icon or ghost that represents the data being dragged. This icon can change during a drag to indicate the current action (copy/move/alias). Hence, drag-over feedback mainly consists of shape and color of source ghost changes when the user changes the type of action, or when drop becomes possible/impossible. Some windowing systems may go a step beyond by providing animation, e.g. to indicate that the action was canceled, they may animate ghosts back to their original location. It is interesting to note that even though the drag-and-drop model is mature, not all windowing systems offer this feature. When no animation is provided, it is significantly more difficult for the user to follow the effect of a cancel operation.

Drag-under feedback denotes the visual effects provided on the target side. It conveys information when a potential target has a drag icon passing through it. The target can respond in many ways: by modifying its shape and color or even in more sophisticated cases by performing actions. For example, when moving a file over a set of folders, when the file remains above a particular folder a sufficient amount of time, the folder might open up to let the user recursively explore the file hierarchy to the desired target folder.

If drag-over and drag-under visual effects are sufficient to describe feedback in the case of regular drag-and-drop operations, they are not for most of its recent extensions. In the latter case, more feedback is needed. This additional feedback is the instrument feedback mentioned previ-

	Issues primarily addressed			Interaction style		
	Distributed surface	Multi-User	Scala-bility	Instrument feedbacks	Instrument Coverage	Category
Pick-and-drop [20]	✓	✓		Ghost	Partial	undirected
Hyperdragging [21]	✓	✓		Line	Full	undirected
Stitching [14]	✓			Trajectory, screen frame, pie menu	Partial	undirected
Throwing [11]			✓	None	Full	Target-oriented
Drag-and-pop [1]	✓		✓	Rubber-bands, tip icons	Targets	Source-oriented
Vacuum [3]			✓	Arc of influence, proxies	Full	Source-oriented
Drag-and-throw [13]	✓		✓	Take-off area, trajectory	Full	Target-oriented
Push-and-throw [13]	✓		✓	Take -off area, trajectory	Full	Target-oriented
Push-and-pop [6]	✓		✓	Take-off area, tip icons	Targets	Source-oriented

Figure 3. Comparison of drag-and-drop extensions.

ously and will be described in the next section.

4.3. Instrument feedback

Instrument feedback is useful to provide users with better control over their actions. Instrument reactions or feedback can be considered as a specific type of recognition feedback. As suggested by (Olsen and Nielsen, 2001),

> "Recognition feedback allows users to adapt to the noise, error, and miss-recognition found in all recognizer-based interactions"

. Such feedback includes, for example, the rubber bands that are used in the case of drag-and-pop to help users in locating/identifying potential targets. Another example is the case of throwing, where take-off areas as well as trajectories are displayed to help users adjust target selection, etc. Such feedback is used in other extensions and varies significantly from one particular instrument to another. Table of Figure 3 summarizes these differences.

4.4. Instrument coverage

All instruments described above do not support full coverage. By coverage, we mean: areas of a surface where an instrument can drop an object.

Some instruments have *partial coverage*. For example, in wall-size displays with touch/pen input, coverage is partial since some areas might be out of reach e.g situated too high.

Other instruments, mainly target oriented instruments, may offer coverage *limited to targets*. For example, with push-and-pop, only potential targets can be reached. With such instruments, source objects cannot be dropped to any other area of the surface. There are many contexts of drag-and-drop situations where no specific target is aimed and where such instruments would fail, so it is important to consider this issue.

Finally, instruments that make it possible to reach all areas of the display will be considered *full coverage* instruments. Table of Figure 3 shows the different types of coverage (partial coverage, full coverage and coverage limited to tar-

gets) of most existing drag-and-drop extensions.

5. M-CIU model and PoIP API

The implementation model we propose is called M-CIU[1] and addresses the different issues discussed in section 2. It offers a unified framework that makes it possible to implement every drag-and-drop extension discussed previously. Hence shifting from one interaction style to another is facilitated.

The M-CIU model is implemented as an API (application programming interface) called PoIP[2]. PoIP supports drag-an-drop like manipulations in different environments and can be used in the development of most Java-based applications. PoIP is implemented in Java[3]. Even though we used PoIP to illustrate the M-CIU model and to provide more details when useful, we believe that our model is general enough to be implemented at other levels or in other programming languages or toolkits. PoIP is available for download with an example of use (Collomb, 2008).

5.1. Overview of M-CIU model

The M-CIU model is based on four key entities: instruments, drag-and-drop managers, shared windows, distributed surface server and topology manager. We quickly present these entities in this section and will provide more details in the next subsections.

Instruments embody interaction techniques. A hierarchy of instruments (see section 5.2) is provided to factorize most common implementation details shared by different interaction techniques. In order to support distribution, each instrument includes one master and several slave instruments which will further be described in section 5.2.

Drag-and-drop managers play a central part as they are used to coordinate all other main entities of the model. Every computer involved in a distributed drag-and-drop in our model has to run a drag-and-drop manager. These managers are responsible for the registration of source and target components (e.g. UI components involved in the interaction), they handle the creation/destruction of slave instruments and also help with the redirection of event streams. A *drag-and-drop manager* is the single interface for a set of instruments. Indeed, a master instrument can change depending on the context, and slave instruments are created and destroyed upon users activities. The drag-and-drop manager provides a stable interface for this set of instruments and further simplifies communications between the main model entities.

Shared windows are created to display most feedback and will be described further in section 5.3.

Source and target components can be any basic UI components involved in the interaction provided that they have the capacity of registering to a drag-and-drop manager.

Distributed surface server and topology manager are the entities responsible for handling shared windows distributed over displays and their associated topology. They will be described in more detail in section 5.3.

5.2. Multi-instrument support and genericity

Our approach to usability and scalability problems mentioned in section 2.2 consists of providing a unified implementation model that embodies drag-and-drop-like interaction techniques in instruments. Even though the instrumental interaction (Beaudouin-Lafon, 2000) model is primarily devoted to describing interactions, some aspects of the model are well suited to structuring implementations.

Hence, our approach consists of proposing a multi-instrument model which meets the following requirements:

- Instruments act upon objects transparently: objects are notified about the manipulation as usual but they are not aware of the type of instrument in use. The effort needed to introduce new instruments is minimal.

[1] *M*ulti-Computer, multi-*I*nstrument, and multi-*U*ser
[2] *P*ointer *O*ver *IP*
[3] PoIP relies on RMI (Remote Method Invocation) for network communication and AWT for events

- Users can choose the instrument they want to use, depending on their preferences and the context (touch display, large display, small display). This choice can be part of the user's profile. It can also be made on the fly to adapt to an evolving context.

- Several users can manipulate objects at the same time with different instruments.

Practically, instruments are defined through a hierarchy of classes, all inheriting from a very generic instrument class in the same way that in most UI toolkits all widgets or graphical components usually inherit from a generic window class.

It is important to note that an instrument embodies the implementation of both the interaction and the distribution (multi-computer support).

Interaction

An instrument receives an input stream (e.g. events from a mouse and a keyboard) and processes them to implement interactions. Figure 4 presents two state diagrams (Muller and Gaertner, 2003) that depict interaction models for push-and-throw and push-and-pop. Actually, the first state diagram could also stand for the drag-and-drop and drag-and-throw interaction models and the accelerated push-and-throw interaction model is very close to the second diagram.

drag-and-drop-like state diagrams share several common points due to the actual nature of the interaction techniques that they embody. However, some differences between instruments are visible in these diagrams, e.g. an additional state for push-and-pop. The most important differences between instruments concerns the processing of input events and associated feedbacks.

5.3. Multi-computer support and interoperability

In order to address the multi-computer issues discussed in section 2, one preliminary requirement is to support some sort of interoperability between windowing systems. In our model, interoperability is based on (1) implementation of distributed surfaces and (2) slave instruments.

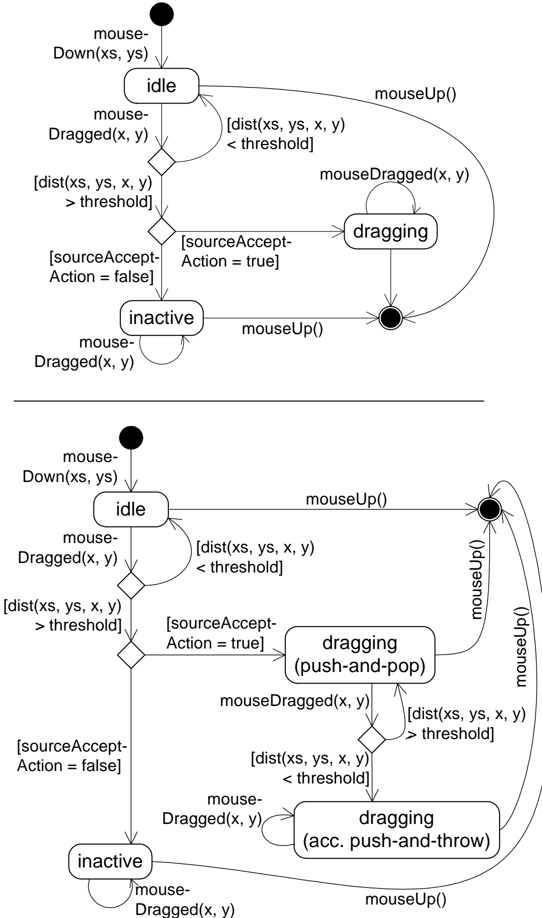

Figure 4. State diagram for the push-and-throw (top) and push-and-pop (bottom).

Distributed surfaces

As defined previously, a distributed surface is a set of windows possibly displayed by different windowing systems and behaving as if they were part of the same workspace. In particular, a drag-and-drop interaction can transparently start with one window of the distributed surface handled in one windowing system and ends on another window operated on another windowing system.

Shared windows

So far we have used the term window in a general way. We now need to refine the concept and introduce the term *shared window* to provide more details on implementation. A shared window is used to make it possible for a given common window or graphic component to be part of a distributed surface. A shared window has a name and a unique ID. Shared windows act on their associate windows or components in two ways:

- **Redirection** of input events received in the associated window[4]. This mechanism allows a pointer to move across the distributed surface, thus transparently surpassing windowing system boundaries.
- **Rendering feedbacks**. A transparent pane is laid on top of the window in order to render multiple pointers. This pane is also made available for instruments to implement different types of feedbacks, especially to perform drag-over feedback and instrument feedback.

Distributed surface Server

A distributed surface is useful for establishing connections between the different shared windows independently of their associated windowing system. Shared windows make a continuous workspace using a given topology. This workspace can be distributed between multiple computers, used by multiple users, each with different input devices (i.e. multi-computer, multi-user).

The distributed surface server is used to:

- manage windows IDs. An ID identifies both windows and associated input devices. An ID is assigned to a window when the window is registered on the server.
- maintain a list of shared windows to ensure that each time a shared window registers or unregisters all other windows are notified.
- handle the topology of windows within the surface according to a topology manager.

[4]Input redirection is the transmission of an input stream so it can be treated on a remote window. Input redirection involves (1) capturing events on a source window (2) transmitting events from the source window to the target window and (3) analyzing events on the target window. Source and target windows can be the same window.

Master and slave instruments

In order to support multi-computer environments, instruments are decomposed into one master instrument and several slave instruments. The number of slave instruments depends on the context and more specifically on the number of different computers potentially involved in the distributed surface. Most generic levels of communication between slaves and masters are handled at the most abstract classes of instruments, but more specific communication is left to more specific classes of instruments. Indeed, communications between masters and slaves may vary a lot both in terms of nature and of frequency from one instrument to another.

A given drag-and-drop manager handles one and only one master instrument and a variable number of slave instruments, depending on the number of running drag-and-drop interactions. As shown in figure 5, a master instrument is mainly devoted to dispatching input events and handling associated slave instruments, while slave instruments do the real job, e.g. find which component is at a given location or perform adequate feedback in the relevant shared window. A master instrument requests the creation of slave instruments when a drag-and-drop-like manipulation is detected and asks for their destruction at the end of the manipulation. Thus, a master instrument handles n slave instruments during manipulation where n is the total number of shared windows in the distributed surface.

5.4. Multi-user support and concurrency

The M-CIU model allows multiple users to interact simultaneously on a distributed surface[5]. This can be achieved by augmenting event

[5]The number of users cannot be higher than the number of computers involved since only one input stream is managed on each computer.

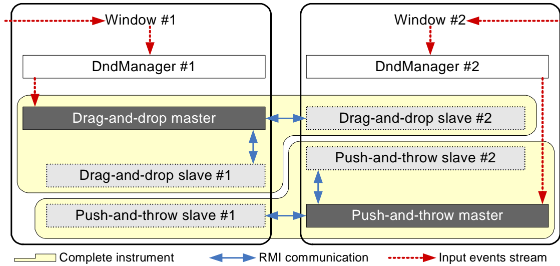

Figure 5. Example of two concurrent drag-and-drop-like manipulations on a distributed surface containing two windows.

streams with the ID of the input device from which they originally started. This feature certainly enables multiple users to interact using several different instruments to perform drag-and-drop-like operations but it does not solve all problems that should be handled at the level of each application to ensure that concurrency results in a coherent behavior. Handling further concurrency issues is beyond the scope of this work.

5.5. A complete example of a typical multi-instrument, multi-display and multi-user interaction with the M-CIU model

Lets consider an example where two users are discussing a project and meeting in a room where a wall-sized display with touch capacities is available. User A is using his own laptop and user B is using the wall-sized display and the associated computer. At one point, user A wants to give user B data displayed on the screen of his laptop. Conversely, user B wants to give user A data displayed on the wall-sized display. With his laptop, user A uses a simple regular drag-and-drop interaction style while user B, on the contrary, prefers to use a push-and-throw interaction style with the wall-sized touch display. To handle drag-and-drop-like operations between both users, two different computers are involved with two different applications and two different interaction styles. User A starts a drag-and-drop from computer A and the target component is handled by application B running on computer B. At the same time, user B starts a push-and-throw with his pen from computer B toward an application running on laptop A. This example of a typical distributed drag-and-drop like operation is depicted in figure 5. Using the M-CIU model, all interactions are handled through the main entities of the model described in the previous sections and involve 5 steps: initialization, drag detection, drag, drop, finalization.

Initialization

Before any drag-and-drop-like operation starts, application A and B are launched and a few elements needed for drag-and-drop-like operations are created once: the source listener, the target listener, and the master instruments. While these operations are required only once, it is still possible to change these elements later, e.g. a master instrument can be changed dynamically according to user preferences. In this initialization step, source and target components have to register themselves on the drag-and-drop Managers.

Drag detection

The beginning of the drag-and-drop of user A is detected by the master instrument of computer A (which receives all input events of computer A). Respectively, the beginning of the push-and-throw of user B is detected by the master instrument of computer B. When the beginning of the interaction is detected, associated source components are notified and respond. Once source components have accepted the operation, the master instruments ask for the creation of all necessary slave instruments. As a result, each drag-and-drop manager handles two slave instruments: one for drag-and-drop of user A and one for push-and-throw of user B.

Drag

During the drag process, master instruments notify slave instruments that the pointer is moving. This type of redirection ensures that the the slave instrument can perform adequate feedback wherever the pointer moves. When the pointer moves over a potential target component, both

the source and target component are notified.

Drop

At the end of the operation targets are notified and data transfer from source to target can take place. At this stage, master instruments also ask for the destruction of all slave instruments in the same way as they previously asked for their creation.

Finalization

When application A and B are closed, source and target components unregister themselves from associated drag-and-drop Managers. This happens only once per session whereas creation and deletion of slave instruments happens every time a new drag-and-drop-like operation is performed.

6. An application: Orchis

Orchis is an interactive and collaborative graphical application designed to share bookmark collections. Its architecture is client/server and most data is stored on the server side. Other web-based bookmarking clients have been implemented as well to access the same data. However, Orchis offers most graphical features associated with more direct manipulation style. Hence we consider Orchis as a good place to illustrate the use of our multi-instrument, multi-user, multi-computer model.

6.1. Scenario

Let's consider the situation where several users gather bookmarks that deal with *web site design*, for example. Each of them owns a laptop and they regularly meet and use an interactive wall-sized display to compose a common repository.

In this context, one distributed surface server runs on the computer associated with the wall-sized display and Orchis runs on every computer involved in further interactions (e.g. all laptops and the computer associated with the wall-sized display). Orchis displays windows such as the windows depicted on Figure 6. The topology of shared windows determines how one pointer can move from one window on one computer to another window on another computer and is shown on Figure 7 for two users and one wall-sized display.

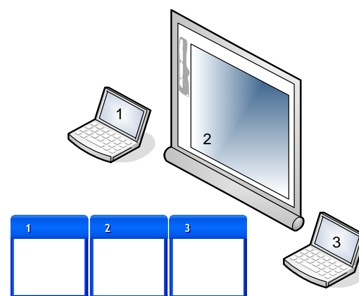

Figure 7. Topology of three Orchis windows (1) in the meeting room and (2) as managed by the distributed surfaces server.

At one point, one user using regular drag-and-drop copies his bookmarks dealing with web design to the wall-sized display. Using accelerated push-and-throw, another user does the same (Figure 6).

Further, one of the users organizes bookmarks gathered on the wall-sized display by directly using the touch device associated with the display. Accelerated push-and-throw is defined as the preferred technique for the wall-sized display so this user operates using accelerated push-and-throw. By using the mouse of his laptop, another user can bring his pointer on the shared window of the wall-sized display and occasionally helps the first user in the organization task.

At the end of the process, the resulting bookmark collection is saved in the database so it can be reused later by both Orchis and all other connected web clients.

6.2. Discussion

Orchis is an excellent example of what can be done with PoIP API:

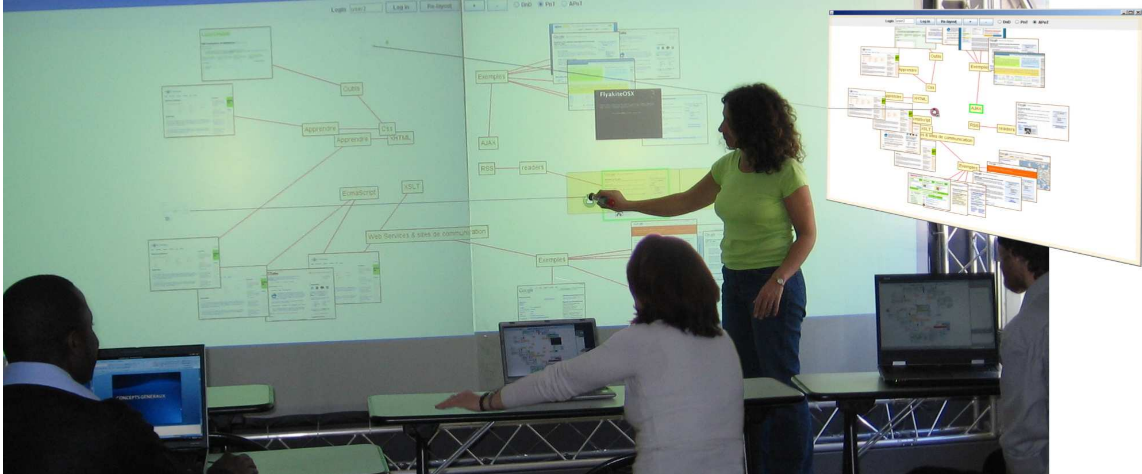

Figure 6. Three users working with Orchis on which two bookmark are copied using accelerated push-and-throw and regular push-and-throw. A view of the right user's window is added to render the workspace as seen by the right user.

- Several windows from different computers can set up a seamless distributed surface.
- This surface can be used simultaneously by several users.
- Each user can choose his preferred drag-and-drop-like interaction technique. Note that Orchis only uses instruments that offer full coverage (see table of Figure 3).

One limitation is that only one input stream is managed on a computer. The number of simultaneous user is limited to the number of computers involved in the distributed surface.

7. CONCLUSION

In this paper, we have shown the necessary changes to drag-and-drop to meet requirements of new emerging interactive environments. We have pointed out issues that arise with these new environments. We have further reviewed, compared and discussed recent drag-and-drop extensions that partially address these issues. Finally, we have proposed the M-CIU model, which is an implementation model that builds upon these analyses to meet the challenging requirements of new emerging interactive environments and to make it possible for a programmer to support most extensions of drag-and-drop in a single unified framework.

We provide an API called PoIP that implements the M-CIU model and that can be used in the development of most Java-based applications. The API has been used in the development of a collaborative bookmarking application called Orchis. Overall, PoIP was found to be robust, and even though PoIP uses a layered pane to display pointers and feedbacks, we did not notice any significant reduction in performances with the Java applications tested.

However, the question of the level at which the model should be implemented is left open. Our approach with PoIP was to implement the model at the toolkit level. However, considering lower levels (windowing system level, window manager or middleware level) would certainly be costly but also most likely to provide great benefits. Even though our model is implemented at the toolkit level, we made it general enough to be imple-

mented at other levels as well.

Our model proposes a multi-instrument approach which is important to address problems of usability and scalability mentioned in section 2. In that context, modularity and genericity were used to minimize the cost of introducing new drag-and-drop extensions as new needs arise. Our model further supports multi-computer environments transparently. This is useful to ensure that drag and drop operations will be able to occur on distributed surfaces displayed over several distinct computers that are possibly running different windowing systems. Multi-computer support is achieved thanks to the combination of shared surface management and slave instruments. Finally, our model supports multiplexing of input streams thanks to input device ID. Hence, we make it possible for multiple users to perform different types of drag-and-drop operations simultaneously.

References

P. Baudisch, E. Cutrell, D. Robbins, M. Czerwinski, P. Tandler, B. Bederson, and A. Zierlinger. Drag-and-pop and drag-and-pick: techniques for accessing remote screen content on touch and pen-operated systems. In *Proceedings of Interact 2003*, Sep. 1–5 2003.

Michel Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *ACM CHI '00 proceedings*, pages 446–453, ACM Press, 2000.

A. Bezerianos and R. Balakrishnan. The vacuum: facilitating the manipulation of distant objects. In *ACM CHI '05 Proceedings*, pages 361–370, ACM PRESS, 2005.

M. Collomb and M. Hascoët. Speed and accuracy in throwing models. In *HCI2004, Design for life, Volume 2*, pages 21–24. British HCI Gr oup, 2004.

M. Collomb PoIP: an API for implementing advanced drag-and-drop techniques In http://edel.lirmm.fr/dragging/.

M. Collomb, M. Hascoët, P. Baudisch, and B. Lee. Improving drag-and-drop on wall-size displays. In *Proceedings of Graphics Interface 2005*, Victoria, BC, 2005.

M. Collomb, M. Hascoët. Comparing drag-and-drop implementations. Technical Report RR-LIRMM-05003, LIRMM, University of Montpellier, France, 2005.

Joëlle Coutaz, Stanislaw Borkowski, and Nicolas Barralon Coupling Interaction Resources: an Analytical Model. In *EUSAI'2005*, pages 183–188, 2005.

Fraunhofer institute. The dynawall project. *http://www.ipsi.fraunhofer.de/ambiente/-english/projekte/projekte/dynawall.html*. 2008.

Paul M. Fitts The information capacity of the human motor system in controlling the amplitude of movement. In *Journal of Experimental Psychology*, volume 47, number 6, June 1954, pp. 381-391. (Reprinted in Journal of Experimental Psychology: General, 121(3):262–269, 1992).

J. Geißler. Shuffle, throw or take it! working efficiently with an interactive wall. In *ACM CHI '98 proceedings*, pages 265–266, ACM Press, 1998.

S. Greenberg, and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *ACM CSCW '94 proceedings*, pages 207–217, ACM Press, 1994.

M. Hascoët, M. Collomb, and R. Blanch. Evolution du drag-and-drop : du modèle d'interaction classique aux surfaces multi-supports. *revue I3, 4(2)*, 2004.

M. Hascoët. Throwing models for large displays. In *HCI2003, Designing for society, Volume 2*, pages 73–77. British HCI Group, 2003.

K. Hinckley, G. Ramos, F. Guimbretiere, P. Baudisch and M. Smith. Stitching: Pen Gestures that Span Multiple Displays. In *Proceedings of AVI'04*, pages 23–31, ACM PRESS, 2004.

G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings SIGGRAPH '02*. Pages 693–702, ACM Press, 2002.

D. R. Hutchings, J. Stasko, and M. Czerwinski. Distributed display environments. In *interactions 12*, 6 Nov. 2005, pages 50–53.

Brad Johanson, Greg Hutchins, Terry Winograd, and Maureen Stone. Pointright: experience with flexible input redirection in interactive workspaces. In *ACM UIST '02 proceedings*, pages 227–234, ACM Press, 2002.

C. Lachenal. Modèle et infrastructure logicielle pour l'interaction multi-instrument multisurface. PhD these, Université Joseph Fourier, Grenoble, France, 2004.

P.A. Muller, and N. Gaertner. Modlisation objet avec UML. Edition Eyrolles. 2003.

Dan R. Olsen and S. Travis Nielsen. Laser pointer interaction. In *ACM CHI'2001 proceedings*, pages 17–22, ACM PRESS, 2001.

Jun Rekimoto. Pick-and-drop: a direct manipulation technique for multiple computer environments. In *ACM UIST '97 proceedings*, pages 31–39, ACM Press, 1997.

Jun Rekimoto and Masanori Saitoh. Augmented surfaces: a spatially continuous work space for hybrid computing environments. In *ACM CHI '99 proceedings*, pages 378–385, ACM Press, 1999.

B. Shneiderman. Direct manipulation: A step beyond programming languages. In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems*. (Part - II): Human interface and the User interface - Volume 1981, ACM Press, 1981.

C. Shoeneman. Synergy. *http://synergy2.sourceforge.net/*. 2008.

N. A. Streitz, J. Geißler, T. Holmer, S. Konomi, C. Mller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, R. and Steinmetz. i-LAND: an interactive landscape for creativity and innovation. In *ACM CHI '99 proceedings*, pages 120–127, ACM PRESS, 1999.

Sony Japan. Flyingpointer. *http://www.sony.jp/products/consumer/-pcom/software_02q1/flyingpointer*. 2002.

Stanford University ComputerScience. The stanford interactive workspaces project. *http://iwork.stanford.edu/*. 2008.

D. Thevenin and J. Coutaz. Adaptation and Plasticity of User Interfaces. In *Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users*, 1999.

Xdmx project. Distributed multihead X. *http://dmx.sourceforge.net/*. 2008.