Interacting with Computers 20 (2008) 562–573

Contents lists available at ScienceDirect

Interacting with Computers

journal homepage: www.elsevier.com/locate/intcom

Extending drag-and-drop to new interactive environments: A multi-display, multi-instrument and multi-user approach

Maxime Collomb, Mountaz Hascoët *

Maître de Conférences, Université Montpellier II – LIRMM – CNRS, 161 rue Ada, 34392 Montpellier Cedex 5, France

ARTICLE INFO

Article history:
Received 11 April 2008
Received in revised form 14 June 2008
Accepted 23 July 2008
Available online 29 August 2008

Keywords:
Distributed display environments
Wall-sized displays
Drag-and-drop
Multi-user interaction models
Plasticity

ABSTRACT

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 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 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; Hascoët et al., 2004. 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 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 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 builds upon an analysis of requirements for adapting to new emerging interactive environ-

* Corresponding author. Tel.: +33 4 67 41 86 56; fax: +33 4 67 41 85 00.
E-mail addresses: collomb@lirmm.fr (M. Collomb), mountaz@lirmm.fr (M. Hascoët).

0953-5438/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.intcom.2008.07.004

ments and 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 different displays are handled by several 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 ComputerScience, 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. Furthermore, 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 Paul, 1954.

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 and Hutchins, 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.

Other works are devoted to manage topological issues by stitching displays in a more realistic way Nacenta et al., 2006 than a two dimension arrangement, by taking into account the offsets between screens and differences in resolutions Baudisch et al., 2004, or by defining lightweight personal bindings between displays Ha et al., 2006.

2.4. Multi-user issues

Amongst all problems that arise with new display environments, multi-user issues can be regarded as a particular case of issues that traditionally belong to the computer-supported collaborative work domain at large. However, it is important to stress the very specific situations in which we consider useful to extend drag-and-drop to multi-user settings.

The UDP notation Lecolinet, 2003 is very helpful to better characterize the specificities and limits of our approach. UDP notation was originally made to compare multi-pointer situations. In this notation, systems are characterized according to three dimensions: U, the number of users, D the number of displays and P the number of pointers per display. In this notation, systems falling in typical classes usually share the same notation. For example, single display groupware systems are all denoted by N-1-N in UDP.

As far as drag-and-drop operations are concerned, our approach would fall in the N-N-N of UDP. This corresponds to the fact that at one point, several pointers per display correspond to several users performing several drag-and-drop like interactions across several displays. However, our approach shows some specificities and limits compared to other more general cases of N-N-N systems or toolkits like for example Ubit Lecolinet, 2003 or I-Am Lachenal, 2004. Indeed, our approach makes the hypothesis that each user uses his personal laptop to interact with others. Therefore, contrary to more general approaches the number N of pointers per display is limited to the number of computers/laptops involved.

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 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. 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. Furthermore, 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: irregular lags between the user's hand movement and the feedback would significantly decrease the usability of such interaction style.

Throwing, *drag-and-throw* and *push-and-throw* are recent extensions that belong to this category. They are described briefly in this Section.

Geißler Geißler, 1998; Streitz et al., 1999 proposed three techniques to work more efficiently on interactive walls. The goal was to limit physical displacement of the user on a 4.5×1.1 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 in which the focus remains around the starting point of interaction or source object's original location. Contrary to target-oriented interaction, source-oriented interactions are less dependent on the visibility of items located far away from the user. In most cases, ghosts or proxies of relevant items located far away are displayed around the original source object's position.

Recent extensions that can be considered source-oriented include *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, in the case of putting a file in the recycle bin, the user starts the drag gesture toward the recycle bin (Fig. 1-d). 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 disap-

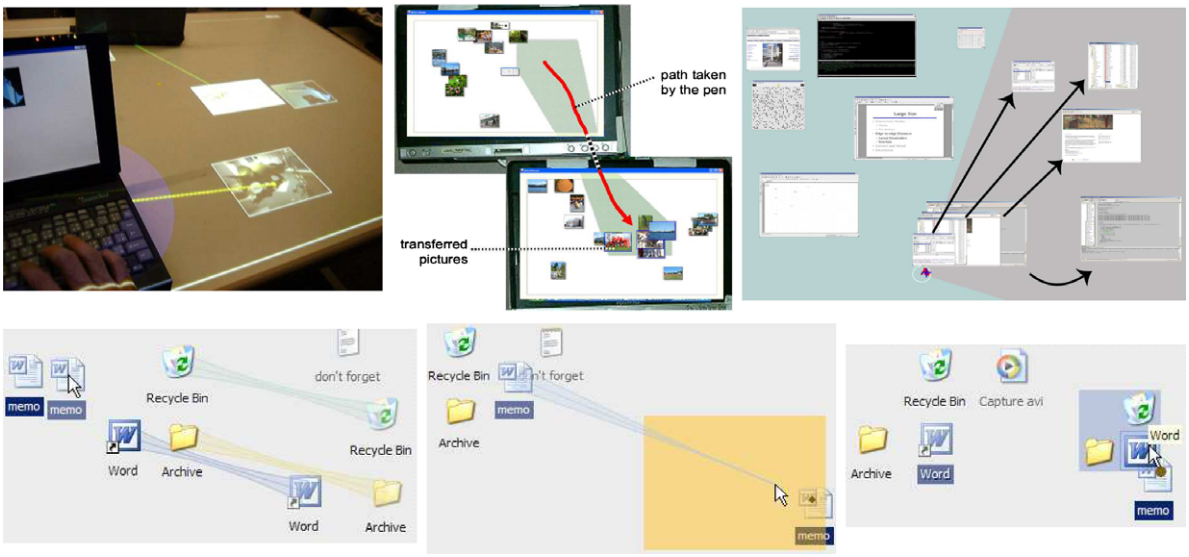

Fig. 1. (Left to right, top to bottom) Examples of (a) hyperdragging, (b) stitching, (c) vacuum (black arrows are added), (d) drag-and-pop, (e) push-and-throw and (f) push-and-pop. (Reproductions with authors' permission).

pear. 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 (Fig. 1-c), 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 Jun 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 Jun Rekimoto and Masanori Saitoh, 1999 (Fig. 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, see Fig. 1-a).

Stitching Hinckley et al., 2004 (Fig. 1-b) is an interaction technique designed for pen-operated mobile devices. These devices have to support networking and allow starting a drag-and-drop gesture on a screen and ending the gesture on another screen. 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 briefly 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 feedback), we further draw a comparison of previous approaches which is summarized in Fig. 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 Fig. 2).

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 or 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 beginning with drag-under feedback made by modifying its shape and color for example.

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 previously and will be described in the next section.

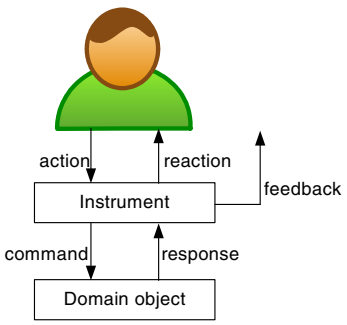

Fig. 2. Interaction instrument mediates the interaction between user and domain objects Beaudouin-Lafon (2000).

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 Dan et al. (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. Fig. 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. The concept of coverage is related to the nature and reachability of the potential targets: can a target be any location on the display identified by x and y coordinates, or should a target necessary be a graphical object, an icon or component? Are all areas of all displays reachable?

Instruments with *full coverage* make it possible for users to reach any positions, objects, icons or components. We further denote by *partial coverage* the type of coverage found in situation where the instrument may fail to reach certain areas under certain circumstances. For example, in wall-size displays with touch/pen input, regular drag-and-drop coverage is partial since some areas may be out of reach e.g. situated too high for example. Other types of instruments, like for example, push-and-pop, only support moves to potential targets. We denoted this limitation as "limited to target". 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.

It is interesting to notice that all extensions designed so far that fall in the source-oriented category do not support full coverage. This leaves an interesting open design space for source-oriented instruments capable of supporting full coverage. Our future work includes designing new instruments of that type.

Fig. 3 shows the different types of coverage (partial coverage, full coverage and coverage limited to targets) 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-and-drop like manipulations in different environments and can be used in the development of most Java-based applications[3]. Even though we used PoIP to illustrate the M-CIU model and to provide more details when useful, the M-CIU model aims at being general enough to be

[1] *M*ulti-Computer, multi-*I*nstrument, and multi-*U*ser.

[2] *P*ointer *O*ver *IP*.

[3] PoIP is implemented in Java and relies on RMI (Remote Method Invocation) for network communication and AWT for events.

	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

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

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), and 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 5.3.2.

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.
- 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).

5.2.1. Interaction

An instrument receives an input stream (e.g. events from a mouse and a keyboard) and processes them to implement interactions. Fig. 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 concern the processing of input events and associated feedbacks.

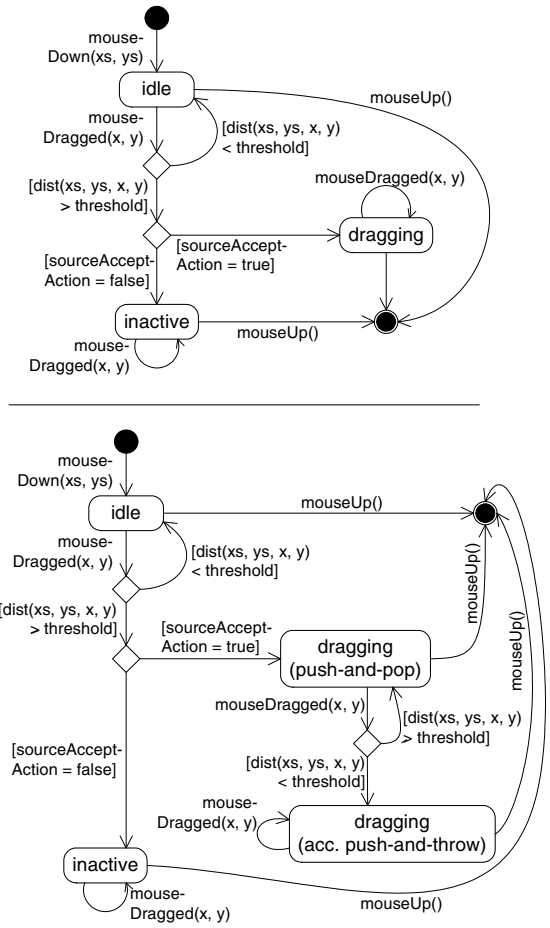

Fig. 4. State diagram for the push-and-throw (top) and push-and-pop (bottom).

5.2.2. Implementation

The PoIP API requires the instruments to be implemented following the master/slave scheme described in the next section. We introduced no other constraints on instrument in order to keep the maximum of possibilities for the implementation of various future instruments.

The instrument implementations are facilitated by two means: (1) a communication mechanism is provided between slave and master instruments so instrument developers do not need to deal with network aspects and (2) instrument classes' inheritance. PoIP heavily relies on object oriented concepts and instrument classes form a hierarchy which allows an instrument to reuse some behaviors of its ancestors. For example, the *accelerated push-and-throw* instrument inherits from the *push-and-throw* instrument and reuses most of it.

Instruments implement interactions described in state diagrams (Fig. 4). The M-CIU model makes it possible for several instruments (e.g. several state diagrams) to run simultaneously. This corresponds to several users performing drag-and-drop like operation simultaneously. Most of the time, the different instruments run independently one from another and do not need to synchronize, nor share resources. However it might happen that at some point several different instruments might need to share a given resource. This specific situation will be discussed in section about multi-user and concurrency issues.

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.

5.3.1. 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.

5.3.2. 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.

5.3.3. Distributed surface server

A distributed surface server 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.

5.3.4. Topology manager

Our model includes a topology manager which is handled by the distributed surface server. It manages the topology of shared windows by offering the following services:

- Getting the shared windows which is at a given position relatively to another shared window.
- Computing the new position of the pointer when it goes beyond the limits of a shared window toward another shared window.

[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.

A default topology manager is provided in the PoIP API. The default topology manager works with shared windows absolute virtual coordinates in a two dimensions space. This means that any shared window can be mapped against a location in the virtual 2D space. The default manager provides a basic graphical user interface for users to specify the position of any new shared window relatively to the other previously registered shared windows. This user interface is very limited but at the same time it makes it possible for users to manipulate the topology very easily. However, our default interface would not scale if the number of display were to increase significantly. Therefore, in the future work we plan to improve this default topology handling. More scalable approaches such as lightweight personal bindings for example Ha et al., 2006 or others where topology is not based on absolute virtual coordinates in 2D space but rather on the definition of relationships between pairs of shared windows will be useful to improve the default topology manager in the future. Note that the independency between the topology manager strategy and the rest of the model makes it possible to develop more advanced topology manager to replace the default one and still benefit from the rest of the model for drag-and-drop like interaction support.

The default manager further accounts for private and public aspects of shared windows. During the registration, users may specify if a shared window is private or public. When a shared window is declared private, it cannot be accessed by other pointers nor allow any drag-and-drop operations on it. However, it still handles redirection of pointers from the given private shared window to any other public shared window. For that reason, the topology manager ensures *teleportation* of pointers from private windows to any public shared window. Hence by declaring a shared window as private, a user can participate in the interaction occurring on several distant public shared windows without having other people interfering with his own private shared window. Note that several private shared windows are allowed to be located at the same place in the 2D space handled by the topology manager. Indeed, since only one pointer can reach a given private shared window there is no risk of ambiguity. As a consequence, the number of private shared window is not an issue regarding the default topology manager approach.

5.3.5. 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 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. Master instruments are used to implement appropriate state diagrams (such as for example those depicted in Fig. 4). Each master instrument can implement a different state diagram (this is the requirement for a multi-instrument support). Master instruments are also responsible for the creation and suppression of slave instruments. 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. Master instruments dispatch orders to slaves using the communication mechanism offered by PoIP so that slaves can do the real job.

Let's consider an example where two users meet in a room where a wall-sized display with touch capacities is available. User A is using the wall-sized display and the associated computer and user B is using his own laptop. At one point, both users want to exchange data using different drag-and-drop like extensions. User A uses a push-and-pop interaction style while user B, on the contrary, prefers to use a push-and-throw interaction style. User A starts a push-and-pop from computer A and the target component is handled by application B running on the laptop of user B. At the same time, user B starts a push-and-throw with his pen from his laptop B toward an application running on computer A. This example of a typical distributed drag-and-drop like operation.

Fig. 5, describes how masters and slave instruments communicate when the two users of this example interact simultaneously. In this situation, Window A of the Fig. 5 (respectively Window B) is a shared window handled within the windowing system of user A (respectively user B). This shared window receives events thanks to the drag-and-drop manager installed on machine A (respectively machine B). When user A interacts with his input device, the master instrument associated with window A handles events according to a push-and-pop state diagram described in (Fig. 4). The Fig. 5 shows that this master further dispatches orders to its associated slave instruments to surpass the windowing system boundaries. In turn, slave instruments do the real job: e.g. find which component is at a given location or perform adequate feedback or action in the relevant shared window.

At the same time, user B also interacts and as depicted on Fig. 5, similar behavior occurs simultaneously on window B. The difference is that the master instrument of window B implements the push-and-throw state diagram instead of a push-and-pop. The M-CIU model makes this different transparent and what happens on window B is very similar to what happens on window A: the master instrument of window B handles events according to its associated state diagram (namely push-and-throw of Fig. 4) and further dispatches orders to its slaves so that adequate feedback and actions be handled by them.

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 streams with the ID of the input device from which they originally started. This feature enables multiple users to interact using several different instruments simultaneously to perform drag-and-drop-like operations. However, PoIP implementation is limited by the input restrictions of the underlying windowing system. As a consequence, in the current PoIP implementation of the model, the number of simultaneous users cannot exced the number of machines available for handling input.

In many cases, multi-user support brings to the scene concurrency issues. Concurrency problems for collaborative editing have been widely documented by Ellis and Gibbs, 1989; Greenberg and Marwood, 1994; Allison, 1994; Campbell, 2006 and others. However, there are several reasons why the M-CIU model is not responsible for concurrency management. Firstly, from the M-CIU model perspective, the interactions that occur simultaneously are independant from one another. Indeed, the M-CIU model handles multiple instruments and feedback. It does not handle the actions on objects which are performed at the level of applications (see Section 5). When several instruments run simultaneously, only their actions might lead to critical sections e.g. sections of code where different actions might need to simultaneously work on a

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

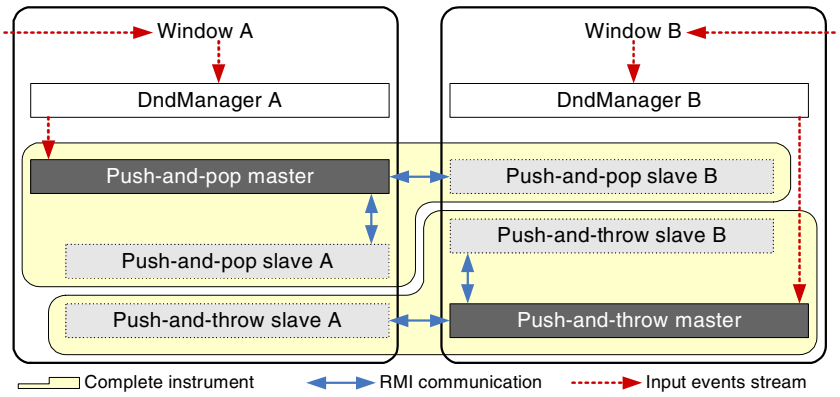

Fig. 5. Example of two concurrent drag-and-drop-like manipulations on a distributed surface containing two windows.

common resource. Therefore it is natural that concurrency be handled at the level of application, not at the level of the M-CIU model.

Orchis is a good example of an application based on M-CIU and PoIP where concurrency can be handled relatively simply. See Section 6 for more details. At the application level, concurrent access on shared resources can be handled at a finer grain. Furthermore, the semantic of concurrent access can be more precisely defined. Depending on the semantic of the drag-and-drop associated actions, different applications may choose between very different strategies to handle concurrency including leaving concurrency management to users, implementing a floor control mechanism, using mutual exclusion, other database like approaches, or optimistic locking or multi-versioning like in collaborative editing systems Greenberg et al., 1992; Sun and Chen, 2002; Moran et al., 1995. Our experience with Orchis showed that managing critical sections at the level of application induced flexibility and could even lead to relatively simple solutions (see Section 6).

5.5. The five steps of typical drag-and-drop like interaction

Using the M-CIU model, all interactions are handled through the main entities of the model described in the previous sections. These interactions involve 5 steps typical of most drag-and-drop existing implementation: initialization, drag detection, drag, drop, and finalization. In this section, we illustrate how these five steps are handled in the M-CIU model with the example introduced in the previous section where two users interact simultaneously.

5.5.1. Initialization

Before any drag-and-drop-like operation starts, application A and B are launched and 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.

5.5.2. Drag detection

The beginning of the push-and-pop 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 push-and-pop for user A and one for push-and-throw for user B.

5.5.3. Drag

During the drag process, master instruments notify slave instruments that the pointer is moving. This type of redirection ensures that 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.

5.5.4. 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.

5.5.5. 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 (Shneiderman, 1981) 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 *distributed display environments*, 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 Fig. 6.

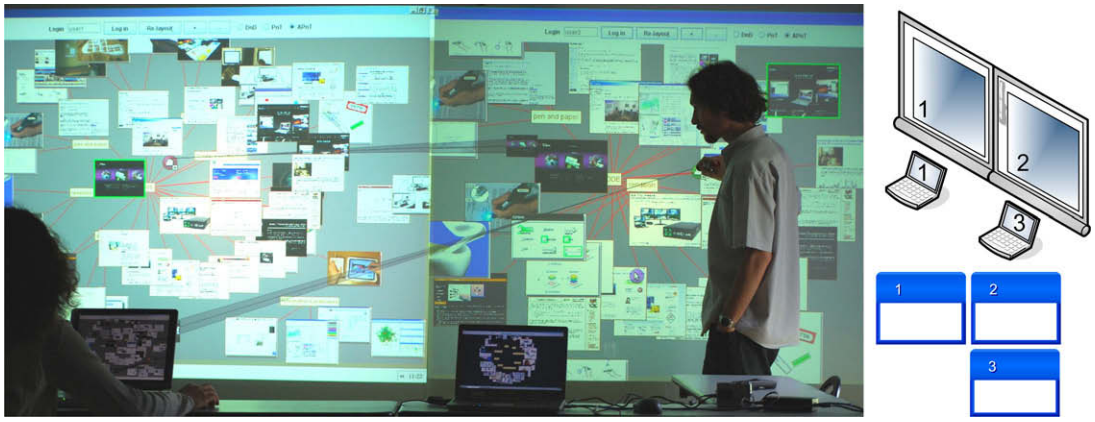

Fig. 6. Two users working with Orchis on which two bookmarks are copied using accelerated push-and-throw. Topology of the three Orchis windows (top right) in the meeting room and (bottom right) as managed by the topology manager.

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 Fig. 6 for two users and one wall-sized display. In Orchis, the topology is handled by the default topology manager contained in PoIP. It can be customized by the user interface provided with it.

At one point, one user using regular drag-and-drop copies part of his bookmarks to the wall-sized display. Using accelerated push-and-throw, another user does the same.

Furthermore, one of the users organizes bookmarks gathered on the wall-sized display by directly using the touch device associated with the display (Fig. 6). 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. Software architecture and implementation

Orchis is a Java/Swing application which uses the PoIP API described in the previous section. The Fig. 7 presents an example with two Orchis applications running and accessing data from a bookmark server. Other web-based clients have been developed to provide access to the same bookmark server. The Fig. 7 shows two clients running Orchis and one web based client (bottom left of the figure). The two Orchis instances contain shared windows which automatically create a continuous shared workspace and drag-and-drop like operations are performed from one shared window to the other.

In order to support multi-instrument, multi-display and multi-user drag-and-drop operations between different instances of

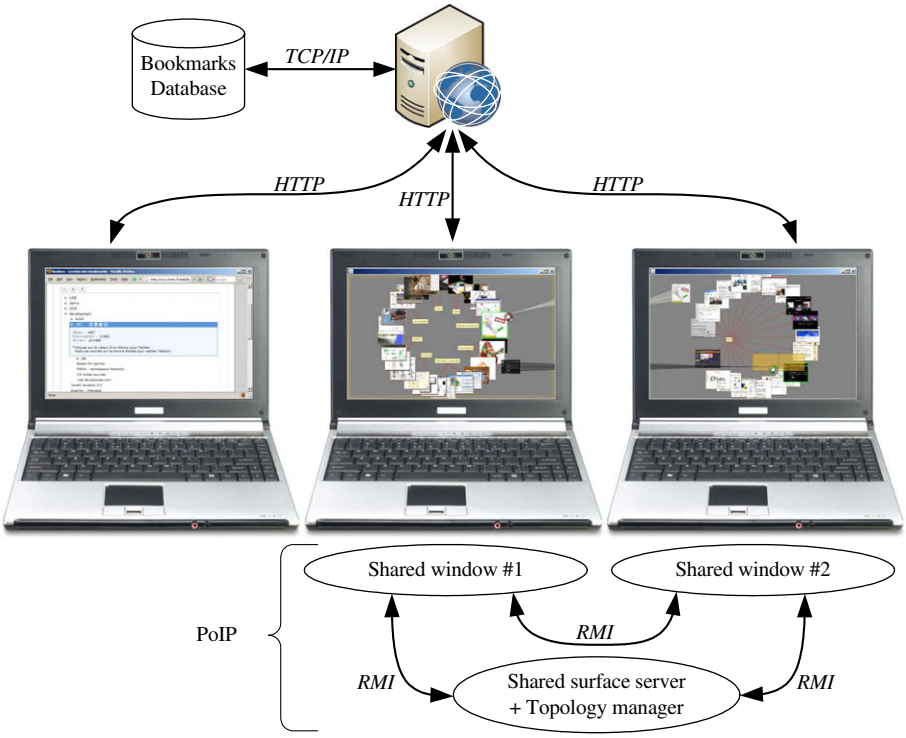

Fig. 7. Orchis architecture: three clients access to the bookmark server through one web based application (left laptop) and two instances of Orchis (center and right laptops).

Orchis the work was minimal thanks to PoIP. As shown on Fig. 7, the PoIP API manages all the communications between shared windows. Each instance of Orchis creates a shared window object at launch and then has to deal with identified input events. It further implements the interfaces required by PoIP to perform drag-and-drop like operations. The GUI components used in Orchis are not aware of which drag-and-drop like technique is currently used. Orchis further provides an interface for changing the technique on the fly and forwards the user's choice to the *DnDManager* handled by PoIP.

6.3. Concurrency in Orchis

In Orchis, the only critical sections that have to be handled occur when simultaneous interactions lead to move actions on related bookmarks or folders. These situations are very rare. Indeed, as noted by Stefik et al., 1987 among others, social protocols decrease the probability of such critical sections. However, the situation could occur and is handled by Orchis relatively simply.

Our approach in Orchis is based on the replication of objects when they are dragged. Hence, when a drag starts, source objects are replicated for each instrument performing the interaction. When the interactions end, the actual move actions are performed on the object in the order they were received by the surface server. Consequently, the final position of the moved object is the position last received by the surface server. Due to the very limited number of situations where such concurrent access occur, this replication strategy was found to be satisfactory. It keeps the interaction fluid since no locking is performed and it does not compromise the consistency of the data handled. However, other more sophisticated strategies might be considered in the future work as new needs arise.

6.4. Discussion

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

- 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 Fig. 3).

One limitation is that only one input stream is managed on a computer. The number of simultaneous users is therefore limited to the number of computers involved in the distributed surface. This limitation is not due to the M-CIU model but to the implementation of PoIP. PoIP deals with input events once they are treated by the windowing system and therefore cannot identify if these events are generated by different input devices. A better solution would be to implement the M-CIU model at the windowing system level Hutterer and Thomas, 2007. However, this requires important development resources and is beyond the scope of our work.

7. Conclusion and future work

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 the necessary support for larger and distributed display surfaces, heterogeneous interaction styles and multi-user simultaneous interactions. 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 some of 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 have also provided an implementation of the model as an API called PoIP that implements the M-CIU model at the toolkit level. PoIP can be used in the development of most Java-based graphical 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 performance with the Java applications tested.

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 support drag and drop over distributed surfaces displayed by several computers 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.

However, there are several issues that were left open for future work. First, as mentioned earlier, even though it is possible for several users to interact on the same display simultaneously, we do not address problems typical of single display groupware. Our implementation is limited by the input restrictions of the underlying windowing system. Therefore in PoIP the number of simultaneous interacting users is limited by the number of laptops available. This is coherent with our main target scenario where each user comes with his personal laptop. It is important to stress that this limitation is *in the implementation* not *in the model*. PoIP implementation is constrained by the difficulty to account for multiple concurrent input streams in today's windowing systems. Extending our implementation in PoIP to typical cases of SDG would be an interesting further development. This would however ask the question of the level at which the model should be implemented. Our approach in PoIP was to implement the model at the toolkit level but another interesting option would be to consider deeper levels that might benefit from recent advances in this area Hutterer and Thomas, 2007.

Another limitation left for future work is the question of scalability in terms of users. M-CIU model was designed for small groups of users and we have not yet investigated how it would scale to considerably larger numbers of users. Future work in the direction probably also need to consider this issue in a broader context of social mediation in large groups.

Apart from this, several improvements of the model are necessary to include more support for instrument implementation without burdening the model with unnecessary functions. Since most instruments can be implemented from state diagram descriptions, augmenting the model with an approach similar to the approach of Swingstates Appert and Beaudouin-Lafon, 2006 would be an interesting future direction for this work. This might significantly reduce the code necessary to introduce new instruments without falling into problems typical of monolithic approaches Bederson and Hollan, 1994.

Lastly, an important future work concerns topology management. This problem is very important in distributed display environments. Since it was not the primary focus of the M-CIU model, the default topology manager provided in our model is very primitive. Nevertheless, it offers two advantages: (1) it already makes it possible to manage the relative positions of shared windows in the distributed surface and (2) it is very independent of

the rest of the model and can be replaced with more advanced topology management in the future. Recently, completely different approaches can be found in the literature to address topological issues in distributed display environments. These approaches range from completely user-driven approaches like for example in Swordfish Ha et al., 2006 to approaches where topology is configured more automatically like in I-am for example (Lachenal, 2004). This opens the space for very interesting alternatives that could replace or augment the current default topology manager of our model.

References

Allison, C., 1994. Concurrency control strategies for real time groupware. In: Proceedings of Concurrent Engineering: Research and Applications, pp. 163–170.

Appert, C., Beaudouin-Lafon, M., 2006. SwingStates: adding state machines to the swing toolkit. In: ACM UIST 2006 Proceedings, ACM Press, pp. 319–322.

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

Baudisch, P., Cutrell, E., Hinckley, K., Gruen, R., 2004. Mouse ether: accelerating the acquisition of targets across multi-monitor displays. In ACM CHI'04 Extended Abstracts, 1379–1382.

Beaudouin-Lafon, M., Instrumental interaction: an interaction model for designing post-wimp user interfaces. In: Proceedings 2000 ACM CHI'00, ACM Press, pp. 446–453.

Bederson, B.B., Hollan, J.D., 1994. Pad++: a zooming graphical interface for exploring alternate interface physics. In: Proceedings ACM UIST 1994, ACM Press, pp. 17–26.

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

Campbell, Jeffrey D., 2006. Coordination for multi-person visual program development. In J. Vis. Lang. Comp. 17 (1), 46–77.

Collomb, M., Hascoët, M., 2004. Speed and accuracy in throwing models. HCI2004 Design for life, vol. 2. British HCI Group, pp. 21–24.

Collomb, M. PoIP: an API for implementing advanced drag-and-drop techniques In: http://edel.lirmm.fr/dragging/. 2008.

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

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

Ellis, C.A., Gibbs, S.J. 1989. Concurrency control in groupware systems. In: Proceedings of SIGMOD, pp. 399–407.

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

Paul, M., 1954. Fitts The information capacity of the human motor system in controlling the amplitude of movement. In J. Exp. Psychol. 47 (6), 381–391. Reprinted in J. Exp. Psychol. General, 121(3):262–269, 1992.

J. Geiler, 1998. Shuffle, throw or take it! working efficiently with an interactive wall In: ACM CHI'98 Proceedings, ACM Press 265–266.

Greenberg, S., Roseman, M., Webster, D., Bohnet, R., 1992. Issues and experiences designing and implementing two group drawing tools. In: Proceeding of Hawaii International Conference on Systems Sciences, pp. 138–150.

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

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

Hascoët, M., 2003. Throwing models for large displays. HCI2003, Designing for society, vol. 2. British HCI Group, pp. 73–77.

Ha, V., Inkpen, K., Wallace, J., Ziola, R., 2006. Swordfish: user tailored workspaces in multi-display environments. In: CHI'06 Extended Abstracts on Human Factors in Computing Systems (Montréal, Québec, Canada, 22–27).

Hinckley, K., Ramos, G., Guimbretiere, F., Baudisch, P., Smith, M., 2004. Stitching: pen gestures that span multiple displays. In: Proceedings of AVI'04, ACM Press 23–31.

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

Hutchings, D.R., Stasko, J., Czerwinski, M., 2005. Distributed display environments. In interactions 12 (6), 50–53.

Hutterer, P., Thomas, B.H., 2007. Groupware support in the windowing system. In: Piekarski, W., Plimmer, B., (Eds.), Proceedings of the Eighth Australasian User Interface Conference (AUIC2007) Balarat, Bic, Australia.

Johanson, B., Hutchins, G., Winograd, I., Stone, M., 2002. Pointright: experience with flexible input redirection in interactive workspaces. In: ACM UIST'02 Proceedings, ACM Press 227–234..

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

Lecolinet, E., 2003. Multiple pointers: a study and an implementation. In: ACM IHM Proceddings, ACM Press 134–141.

Lecolinet, E., 2003. A molecular architecture for creating advanced GUIs. In: ACM UIST Proceedings, ACM Press 135–144.

Moran, T.P., McCall, K., Melle, B.V., Pedersen, E.R., Halasz, F.G., 1995. Some design principles for sharing in Tivoli, a whiteboard meeting support tool. In: Greenberg, S., Hayne, S., Rada, R. (Eds.), Groupware for Real- Time Drawing: A Designer's Guide. McGraw-Hill, London.

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

Nacenta, M.A., Sallam, S., Champoux, B., Subramanian, S., Gutwin, C., 2006. Perspective cursor: perspective-based interaction for multi-display environments. In: ACM CHI 2006 Proceedings, ACM Press 289–298.

Dan, R., Olsen, S., 2001. Travis Nielsen. Laser pointer interaction. In: ACM CHI'2001 Proceedings, ACM Press 17–22.

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

Rekimoto, J., Saitoh, M., 1999. Augmented surfaces: a spatially continuous work space for hybrid computing environments. In: ACM CHI'99 Proceedings, ACM Press 378–385.

Shneiderman, B., 1981. 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, ACM Press.

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

Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., Tatar, D., 1987. WYSIWIS revisited: early experiences with multiuser interfaces. ACM Trans. Office Inform. Syst. 5 (2), 147–167.

Streitz, N.A., Geiler, J., Holmer, T., Konomi, S., Müller-Tomfelde, C., Reischl, W., Rexroth, P., Seitz, P., Steinmetz, R., 1999. i-LAND: an interactive landscape for creativity and innovation. In: ACM CHI'99 Proceedings, ACM Press 120–127.

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

Sun, C., Chen, D., 2002. Consistency maintenance in real-time collaborative graphics editing systems. ACM Trans. ComputerHum. Inter. 9 (1), 1–41.

Thevenin, D., Coutaz, J., 1999. Adaptation and plasticity of user interfaces. In: Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users.

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