Implementing Drag-and-Drop Like Manipulations in a Collaborative and Large Screen Surface Context

Maxime Collomb, Mountaz Hascoët

Implementing drag-and-drop like manipulations in a collaborative and large screen surface context

Maxime Collomb, Mountaz Hascoët

LIRMM, UMR 5506 du CNRS
161 rue Ada
34 392 Montpellier Cedex 5
{*collomb, mountaz*}*@lirmm.fr*

Abstract

Drag-and-drop is probably one of the mostly successful and generic illustrations of direct manipulation technique available in today's WIMP interfaces. Yet it does not adapt without major evolutions to the emerging new devices and environments. Over the past years, several evolutions of *drag-and-drop* have emerged. The diversity of implementations strategies makes them difficult to integrate into an existing windowing system. At the same time, standard existing *drag-and-drop* implementations vary widely across different windowing systems or toolkit and do not facilitate integration or evolution.

The aim of this paper is to propose a unifying implementation model capable of handling most drag-and-drop extensions in a consistent framework while supporting distributed environments. The proposed model aims at facilitating the implementation and evolution of drag-and-drop in future windowing systems. Its benefits are shown with a Java-based implementation. This work builds upon a synthesis of *drag-and-drop* implementations, an analysis of requirements of recent evolutions and a dedicated interaction model based on instrumental interaction.

Key words: distributed display environment, augmented surfaces, drag-and-drop, interaction model, Java API

1 INTRODUCTION

For years, *drag-and-drop* has enhanced the WIMP (Windows, Menus, Icons, Pointer) paradigm. Indeed, *drag-and-drop* has very good degrees of indirection, integration and compatibility [10] while many other interaction paradigms

found in WIMP are rather *indirect* manipulation techniques heavily based on menus or dialogs.

Recent evolutions, like *pick-and-drop*, further extend the capacities of *drag-and-drop*. However, such evolutions also introduce a problem related to compatibility, implementation and integration.

As the number of *drag-and-drop* extensions is increasing, it is important to have a framework for designer to understand better the difference between each technique. Such a framework is also very meaningful as a basis for a unified interface, which let users choose the technique they prefer to use or like best.

Additionally, *drag-and-drop* is limited to basic contexts (a single user on a single computer). However, some extensions like *pick-and-drop*, *hyperdragging* and stitching work over a network, *drag-and-drop* and all other extensions do not support distributed environments.

2 DRAG-AND-DROP EXTENSIONS

Over the past few years, progress on multiple or augmented display surfaces has contributed to the emergence of several new techniques for moving objects around a desktop or within an application. Indeed, *drag-and-drop*, as we know it, reaches its limits when using more complex systems than the traditional "one screen, one keyboard and one mouse". And, if *drag-and-drop* can be used on interactive surfaces in theory, it is in fact tedious especially on large surfaces (such as iWall [19] or DynaWall [5]).

2.1 Pick-and-drop

Pick-and-drop [7] has been developed to allow users to extend *drag-and-drop* to distributed environments. While *drag-and-drop* enforces the user to remain on the same computer while dragging objects around, *pick-and-drop* let 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 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 are associated with this unique ID and stored on a *pick-and-drop* server.

2.2 Hyperdragging

Hyperdragging [8] is part of a computer augmented environment. It helps users smoothly interchange digital information among their laptops, table or wall displays, or other physical objects.

Using *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 its 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).

2.3 Stitching

Stitching [9] is an interaction technique designed for pen-operated mobile devices that allow 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 start dragging an object on the source screen, reach 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 then bound devices are able to transfer data.

2.4 Shuffle, Throw or Take it

Geißler [6,16] proposed three techniques to work more efficiently on interactive walls. The goal was to limit the physical displacement of the user on a 4.5 x 1.1 meters triple display (the DynaWall [5]).

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 of users on the appropriate widget.

Next, the author proposes a *throwing* technique. To throw an object, the user has to achieve a short stroke in the opposite direction that the object should

be moving, followed by a longer stroke in the correct direction. 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 the previously described *pick-and-drop* to the DynaWall.

2.5 Drag-and-pop and vacuum

Drag-and-pop [17] 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 case of putting a file to the recycle bin, the user starts the drag gesture toward the recycle bin (fig. 1-left). After few pixels, each valid target on the drag motion direction creates a linked tip icon that approaches the dragged object. User 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 [1], a variant of *drag-and-pop*, is a circular widget with a user controllable arc of influence that is centred 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 centre in the form of proxies that can be manipulated in lieu of the original.

2.6 Drag-and-Throw & Push-and-Throw

Drag-and-throw and *push-and-throw* [15,11] are throwing techniques designed for multiple displays (one or more computers). They address the limitation of throwing techniques [6,16] 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-

center). Three types of visual feedback are used: trajectory, target and take-off area (area that matches to the complete display).

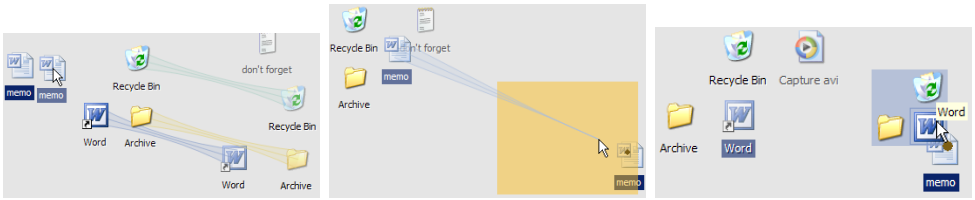

Fig. 1. (Left to right) Examples of *drag-and-pop*, *push-and-throw* and *push-and-pop*.

Drag-and-throw and *push-and-throw* have different trajectories: *drag-and-throw* uses the archery metaphor (user performs a reversed gesture - to throw an object on the right, pointer has to be moved to the left) while *push-and-pop* uses the pantograph metaphor (user's movements are amplified).

The main strength of these techniques is to visualize and control the trajectory of the object before actually sending the object. 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 [11].

2.7 Push-and-Pop

Push-and-pop [12] 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-right): 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 the *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 a better readability (icons are part of the take-off area), target acquisition is easier [12] and user can focus on the take-off area.

All these extensions have been developed individually as prototypes with *ad-hoc* event models. To represent them in a unified way, we propose to use the instrumental interaction model.

3 Instrumental interaction model for drag-and-drop and its evolutions

Drag-and-drop and its evolutions can very well be described in the instrumental interaction model proposed by the author of [10]. The first advantage of

this model is that it can be used to clarify the concepts of interaction at stake. Furthermore, while instrumental interaction was initially introduced as an interaction model, according to the author himself, it can also be considered as the basis for implementation models. This leads us to the second advantage of this model. Indeed, the implementation model we propose in the next section could easily be derived from the interaction model described in this section. We believe that this ensures both adequacy and consistency of the implementation to the interaction needs.

3.1 Instrumental interaction principles

Instrumental interaction model is based on the concept of *interaction instrument* [10]. An *interaction 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* enabling users to control their *actions* on the instrument, and provide *feedback* as the command is carried out on target objects.

An instrument decomposes interaction into layers (fig. 2):

- Interaction between the user and the instrument: the *action* of the user on the instrument and the *reaction* of the instrument,
- Interaction between the instrument and the domain object: the *command* sent to the object and his *response*, which the instrument may transform into *feedback*.

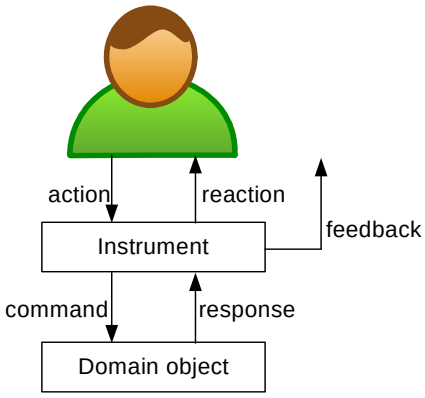

Fig. 2. Interaction instrument mediates the interaction between user and domain objects [10].

3.2 Interaction model

Our objective is to define the set of commands and responses that happen between an instrument and an object when performing *drag-and-drop* like operations. Thus, by specifying the exchanges that occur between the domain object and the instrument, we aim at making the instrument modular: a *drag-and-drop* instrument could be replaced by a *pick-and-drop* instrument, changing the *actions* and *reactions* set between the user and the instrument but keeping the same set of *commands* and *responses* between the instrument and the object. Thus the object is not affected by this swapping of instrument.

3.2.1 Instrument and domain objects

As far as standard *drag-and-drop* is concerned, the instrument part is reduced to minimum: a pointer that let the user move objects around. But as soon as one considers extensions such as *drag-and-pop*, throwing techniques or *push-and-pop*, exhibiting an instrument is mostly useful.

In throwing extensions, the metaphors used based on pantograph and archery metaphor directly suggest the use of an instrument. In *pick-and-drop*, a real physical instrument, the pen, is used to perform the operation from one screen to another. In other cases, the instrument is less visible but still, reactions and feedback provided with these techniques can be embodied by an instrument.

Domain objects can be of various kinds: icons, pieces of text, images, etc. but they have to be differentiated depending on the part they play in the operation. Indeed, domain objects will be acting either as *source* or as *target*: a source is being moved over potential targets until the user ends the operation by dropping the source over the target.

3.2.2 Actions, Reactions and Feedback

There are mainly four actions that occur when a user performs dragging and dropping:

- Source selection and target selection,
- Specification of type of action,
- Data transfer (validation of the selected target) and
- Cancellation.

As soon as *drag-and-drop* extensions become available in windowing systems, a fifth action has to be considered. This action consists of:

- Choice/activation of one type of *instrument*.

There are many ways the latter action could be performed. In most cases, the user might want to specify he uses one type of instrument as default and thus the choice might appear as one aspect of his profile. But in some cases he might want to shift between two instruments on the fly. This is mostly useful for more elaborated instruments where some instruments perform better for some task than for other. For example, *push-and-pop* has proved to be faster and more accurate than *push-and-throw* for sending on source object to one target object, but at the same time it does not make possible to reach any area in the display. With *push-and-pop*, only potential targets can be reached. When a user just wants to move one object to a non-target location (a position on the desktop, for example), he has to use another instrument. In order to keep the interaction fluid, this change should be performed on the fly. In the case of *push-and-pop* the user can switch between two instruments by just bringing the source object back to its initial position. By doing so, the current instrument deactivates and hands the rest of the operation over to a *push-and-throw* instrument.

In most existing windowing systems, the four core actions mentioned above are performed very easily. To select one or several sources, the user just moves the pointer over the source(s) and clicks on the mouse (or other pointing device) and remains pressed, thus obtaining a ghost (also called *drag icon*). This ghost can further be dragged over the target to select it. While dragging, users can change the type of action they want to perform by using modifier keys such as CTRL or ALT in Windows/Linux systems, OPTION or COMMAND on MacOS systems. They are usually three types of actions available: copy, move, make a link (or alias). Lastly, while dragging, users can also cancel the operation by pressing ESC so that the drag is interrupted and all dragged sources return to their original location.

For regular *drag-and-drop*, when these actions are performed, windowing systems provide feedback usually referred to as *drag-under visual effect* and *drag-over visual effect*.

Drag-over effects consist mainly in 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 colour 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: to indicate that the action was cancelled, they may animate ghosts back to their original position. It is interesting to note that even if *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 its cancellation.

Drag-under visual effects represent feedback 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 colour 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 certain 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* operation, they are not for most of its recent extensions. In the latter case, more feedback is added that cannot really be considered as drag-under or drag-over. This additional feedback will be referred to as *instrument feedback*. For example, in the case of *drag-and-pop* rubber bands are used to help user with locating/identifying potential targets. Furthermore, when a user initiates a drag, for both *drag-and-pop* and *push-and-pop* computation and display of a rearrangement of all potential targets is needed. This will be part of the reaction and feedback provided by the related instrument. In the case of throwing, take-off areas as well as trajectories are displayed to help user adjust its target selection. Similar additional feedback is used in other extensions.

As a summary, in an instrumental view of *drag-and-drop* techniques at large, five main actions can be exhibited and feedback can be decomposed into three types of feedback: drag-under feedback, drag-over feedback and instrument feedback. When a user needs to change from one instrument to another one, drag-under and drag-over visual effects might be preserved, only instrument feedback would vary.

3.3 Comparison of instruments

In this section we consider that all potential *drag-and-drop* techniques can be embodied in an instrument. These instruments let the user perform some or all of the five actions mentioned in previous section and provide different types of feedback. All instruments differ in several ways. Our aim is to exhibit the dimensions that matters in a comparison.

An instrument mediates the interaction between user and domain objects [10] (fig. 2). The interactions between the instrument and a domain object (commands/responses) are the same for all *drag-and-drop* like instruments. On the contrary, interactions between the user and the instrument (actions / reactions / feedback) are specific to each instrument.

We can now compare all the techniques described in the previous section: *drag-and-drop*, *pick-and-drop* [7], *hyperdragging* [8], *throwing* [6,16] *drag-and-throw* [15], *push-and-throw* [15], *accelerated push-and-throw* [12], *drag-and-pop* [17] and *push-and-pop* [12].

3.3.1 Actions

The users' actions are different: for a *drag-and-throw*, a *push-and-throw* or an *accelerated push-and-throw* instruments, the pointer is moved all the way to the target. Users have to constantly monitor the reactions and feedbacks of the instrument and reorient themselves as it is virtually impossible to guess upfront where their pointer (finger, mouse pointer, etc.) has to be in order to acquire a target.

In contrast, *drag-and-pop* and *push-and-pop* instruments involve a single fairly dramatic movement on the screen. Once users have identified the target tip icon, they can complete the interaction easily: the target is at a stable location and acquiring it requires only very little attention. However, since different drag directions cause the tip cluster to be a little different every time, users need to re-orient once to identify the correct tip icon. In *drag-and-pop*, the rubber bands have been designed to help users in that task.

Actions might also differ along another dimension: the direction of the hand motion and the direction of the throw is either opposite or not. In *drag-and-throw*, the user drags the pointer in the opposite direction of the throw until the target is selected. This is consistent with the archery metaphor. On the contrary, with other instruments such as *push-and-pop* or *drag-and-pop*, the user aims by moving his hand in the actual direction of the target.

Drag-and-drop and *hyperdragging* involve basic dragging and dropping actions. The difference is that *hyperdragging* allows the pointer to leave its display. *Pick-and-drop* is also very close to *drag-and-drop* except that two clicks are necessary to complete an operation.

In the *throwing initial technique*, users needed to make two strokes which direction and length determined the position where the involved object was thrown.

3.3.2 Reactions

Reactions are trivial for *drag-and-drop*, *pick-and-drop* and *hyperdragging* instruments.

There is a major difference between *push-and-throw* and *drag-and-pop* instru-

ments: while reactions of *push-and-throw* based instruments are visible all over the display, reactions of *drag-and-pop* based instruments are visible in the users' motor space. Hence, the focus of the user will not be the same when using these techniques.

Indeed, when using *drag-and-throw* or *push-and-throw*, the pointer moves in a small area (the take-off area), but the source object is replicated as a ghost and moves all over the display. Hence, users mostly focus on ghost and targets all over the display.

On the contrary, using *drag-and-pop* keeps users' focus in their motor space. Indeed, potential targets are replicated close at hand near the pointer thus in the space reachable by the user, that we call motor space.

We call *'to target'* instruments the instruments of the type of *drag-and-throw*. They replicate source object with a ghost that is pushed toward the potential targets real position. They can be opposed to *'to pointer'* instruments - such as *drag-and-pop* - that replicate targets as ghosts and move ghosts toward the pointer real position (e.g. the source object real position). *Throwing* uses a 'to target' approach since it throws the object in the direction of the target.

3.3.3 Feedback

Only drag-under and drag-over visuals are used for *drag-and-drop* and *pick-and-drop* instruments.

Hyperdragging adds a trajectory feedback that links the remote pointer to its original display.

Throwing initial technique does not provide any instrument feedback. This is probably the critical point that yields high error rates and low precision. Indeed, in the context of throwing, instrument feedback can be considered a specific type of recognition feedback. As it was suggested by [4],

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

Drag-and-throw and *push-and-throw* introduce two feedbacks: the take-off area and the trajectory. *Accelerated push-and-throw* can not offer the take-off area feedback so only trajectory is used.

Drag-and-pop offer two kinds of visual feedbacks: the tip icons and the rubber bands that link the tip icons to their originals.

Push-and-pop combines strengths of *push-and-throw* and *drag-and-pop*. There-

fore, it uses the take-off area **and** the tip icons feedbacks.

Table 1 sums up the design dimensions of all the instruments discussed in this section.

technique	**Action** (need to reorient)	**Reaction** (approach)	**Feedback**
drag-and-drop	never	-	-
pick-and-drop	never	-	-
hyperdragging	never	-	trajectory
throwing	never	to target	-
drag-and-throw / push-and-throw	constantly	to target	take-off area and trajectory
acc. push-and-throw	constantly	to target	trajectory
drag-and-pop	once	to pointer	tip icons and rubber bands
push-and-pop	once, later never	to pointer	take-off area and tip icons

Table 1
Design dimensions of direct manipulation instruments.

4 Toward an open unified implementation model

So far, the extensions discussed above are developed individually with *ad-hoc* event models. Most of them remain as prototypes. However, *Hyperdragging* was implemented for a specific system in the Sony products under the *Flying-Pointer* name [18].

It is striking to observe the diversity of implementations of actual regular *drag-and-dro*p in different windowing systems [14,13]. The lack of a unified and open implementation model over different platforms hampers the integration of evolutions and the support of distributed environments. So far, only *pick-and-drop* and *hyperdragging* have been implemented in a multi-users environment.

The objective of this work is to propose a unified and open model for all *drag-and-drop* like manipulations. This includes *drag-and-drop* of course, but

also its recent evolutions described above. These evolutions include additional reactions and enriched feedback. They also require multi-machine support. Our aim is to build a model out of past experiences and new requirements to facilitate the integration of recent and future extensions of *drag-and-drop* like operation. In order to illustrate the use of our model, we provide a Java-based implementation and illustrate its benefits with the implementation of two techniques: *drag-and-drop* and *push-and-pop* (which can be deactivated into *push-and-throw*).

4.1 Principles

The core of our model is the event model. One important aspect of our model is modularity: source and target components of the operation should be notified of relevant events and their behaviour should not be reconsidered with every new technique.

In order to achieve this modularity, the key elements are a *DmManager* and a *DmAbstractInstrument*, which are responsible for managing all the operations: notifying the components of the different events that occur during the operation, and eventually displaying some visual clues (depending on the technique that is used). The *DmManager* is visible to the components which support *drag-and-drop* like operations. It manages register of source and target components. On the other hand, the *DmAbstractInstrument* manages the actual behaviour of an instrument.

Amongst existing windowing systems, MacOS is the only one to provide a similar entity [13], which is called *DragManager* in the Carbon toolkit. In the Cocoa toolkit, the *drag-and-drop* mechanism is deeply included in the system and such entity does not appear to the developer.

DmManager and *DmAbstractInstrument* entities ensure a consistent behaviour between the different applications that use the technique: for example, it is very distracting to have two applications performing different actions (*copy*, *move*, etc.) when the same modifier key (*shift*, *ctrl*) is used. Another example of such inconsistent behaviour can be found when different shortcuts are used for the same commands (e.g. cut/copy/paste actions are usually shortcut by *ctrl+x*, *ctrl+c* and *ctrl+v*, but we can still find some applications that use *shift+delete*, *ctrl+insert* and *shift+insert*).

The whole *drag-and-drop* behaviour lies in the *DmAbstractInstrument* entity. This allows modularity. To change from one interaction technique(say *drag-and-drop*) to another one (say *push-and-pop*), only the instrument needs to be changed. The components (sources or targets) do not need to be aware of it. Whatever technique is used, source and target components act the same and

are notified of same types of events.

4.2 Architectural model

The model that we propose was kept as simple as possible. As shown in fig. 3, we introduced two interfaces that have to be implemented by the source (*DmSourceListener*) and the target (*DmTargetListener*) components. We further introduce two event types and a *DmManager* class that manages the registration process for source and target components. Finnaly, we define a *DmAbstractInstrument* that contains the instrument behaviour.

Fig. 4 shows an UML sequence diagram of a successful object manipulation. It uses a source and a target component that can be any visible component which is capable of receiving a full set of pointing device events.

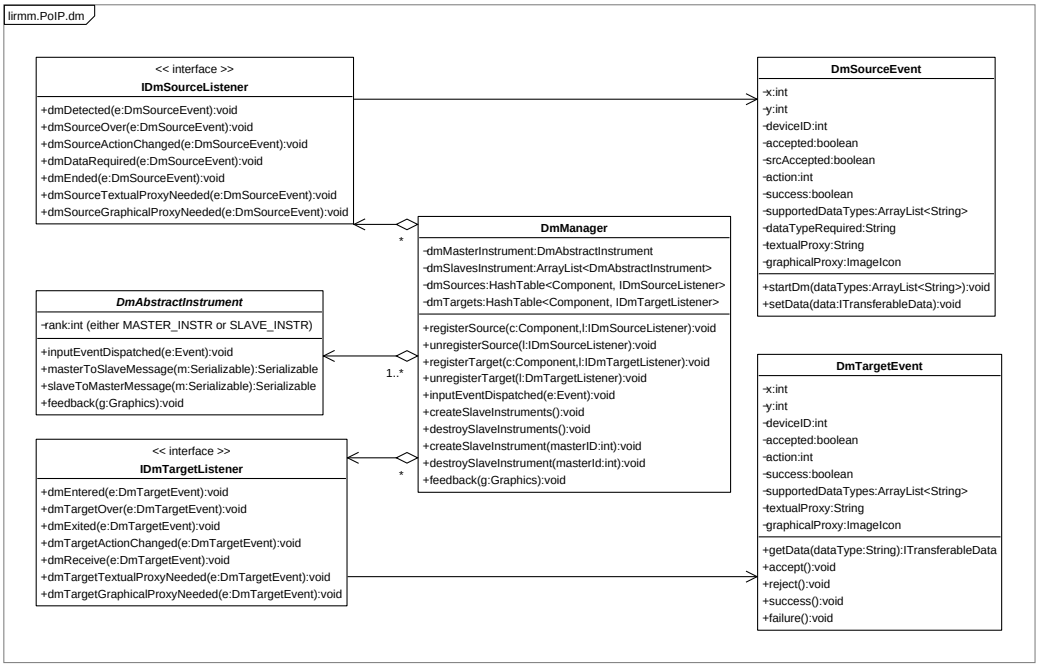

Fig. 3. Class diagram of *drag-and-drop* model.

4.2.1 DmSourceListener

The *DmSourceListener* interface contains all the methods needed by the source component to manage a *drag-and-drop* like operation.

The activation gesture must be detected by the instrument because it could be different from one technique to another (e.g. the beginning of a drag for *drag-and-drop*, a click for *pick-and-drop*, etc.). When this gesture is detected,

the source component is notified by *dmDetected()* and should return the types of the data to be transferred by specifying the supported data types and by calling the *startDm()* method of the received event. Note that there can be several data types.

During the operation, the source component is notified by *dmSourceOver()* and *dmSourceActionChanged()* methods while the pointer is moving or modifier keys (*alt*, *ctrl*, *shift*) are enabled or disabled.

When the object is dropped, the source component is asked for the actual data to transfer through the *dmDataRequired()* method that receives the data type requested by the target in parameter. This method should return the data to transfer in the specified format using the *setData()* method of the event. And, after the data is submitted to the target component, the source is notified of the end of the operation through the *dmEnded()* method. So the source knows if target has successfully received and handled data and is able to process some updates if necessary.

Note that it is important that *dmDetected()* method returns one or more data formats and that the actual data is transferred later through the *dataRequired()* method. Indeed, generating the data can trigger a long process and it could be awkward for the user to have a delay at the beginning of the drag. Furthermore, why generating the data on several formats since most likely only one format will be used by the target? Some current systems (e.g. Java/Swing) do not offer this possibility [13].

Finally, a source component of a *drag-and-drop* like manipulation should be able to give a graphical and a textual proxy of it. Such proxy can be used for displaying a *ghost* or *tip icon* at pointer location while dragging for example.

4.2.2 DmTargetListener

The *DmTargetListener* interface has to be implemented by a component that wants to be a potential target for *drag-and-drop* like techniques.

When an object is being dragged over a target, the component is notified by *dmEntered()* method. The target component should then call the *accept()* or the *reject()* method of the received event depending on its abilities to handle the supported data types (*getSupportedDataTypes()* method of the event). While the object is moved over the target, the component is notified through the *dmTargetOver()* method so it can display some visual feedbacks (drag under) and when the object leaves the target, the component's *dmExited()* method is called. If ever the user changes the way to transfer the object (by pressing *alt*, *shift* or *ctrl* modifiers), the component is notified by the *dmTargetActionChanged()* method. The target component is notified of this event

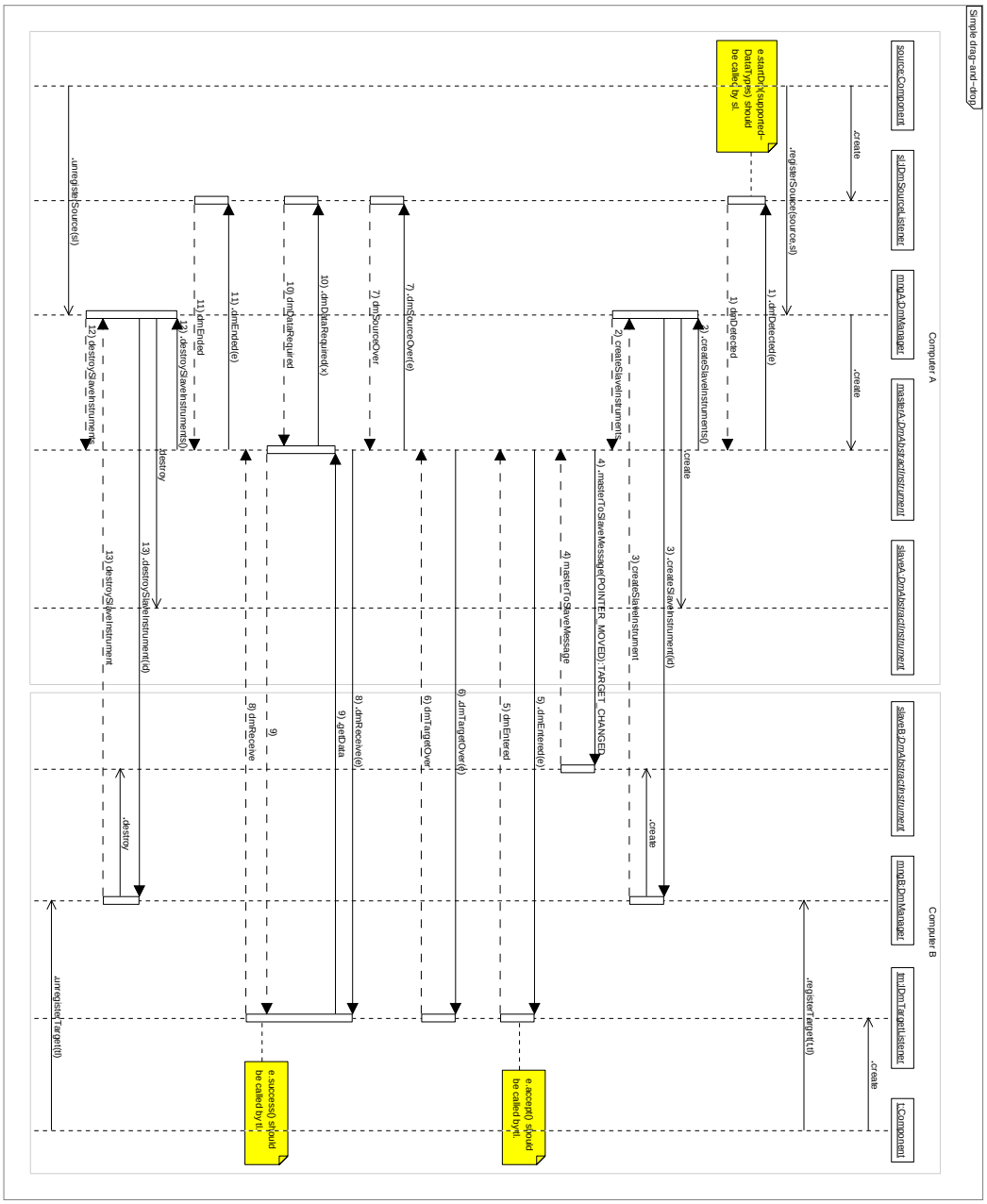

Fig. 4. Sequence diagram of *drag-and-drop* model.

before the source component.

At the end of the operation, the *dmReceive()* method is called for the listener of the target component. The target component is able to know the supported data types (event method *getSupportedDataTypes()*) and can ask for the data in a given format thanks to the method *getData()* of the event. Once the data is received, either *success()* or *failure()* method of the event has to be called. So the system and the source component are aware of whatever data transfer succeeded or not.

Like the source component, the target must be able to create graphical and textual proxies of itself. It can be used by techniques like *drag-and-pop*. *Drag-and-pop* creates new ghosts near the mouse pointer for a set of selected targets.

4.2.3 DmManager

The *DmManager* aims at managing the core of the instrument. It is the interface between the user and the domain objects. It manages *actions* of the user and responds with *reactions* and *feedbacks*. To do so, it uses a *command/response* set of methods to communicate with the domain objects (source and target components). This set of methods is defined by *DmSourceListener* and *DmTargetListener* and are meant to stay unchanged. It is the counterpart to have a modular instrument: interfaces have to be standard and therfefore fixed *a priori*.

4.2.4 DmAbstractInstrument

DmAbstractInstrument is an abstract instrument that implements the behaviour of the instrument. It knows the registered source and target component thanks to *DmManager*. An instrument can be a master instrument or a slave instrument. A *DmManager* owns one and only one master instrument and a variable number of slave instruments. Slave instruments were introduced mainly for multi-machine support but they are also used in a single machine environment for consistency purposes. Slave instruments are usually used for finding which component is at a given position or for visual feedbacks.

A master instrument asks for 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 is linked to n slave instruments during a *drag-and-drop* like manipulation. n is the number of accessible surfaces (an accessible surface owns a *DmManager*).

For example, suppose two computers (A and B) share their surfaces and a *drag-and-drop* instrument is used on computer A while a *push-and-pop* instrument is in use on computer B. Then, if the user of A starts a *drag-and-drop* from A, the *DmManager* of A, which already owns a master *drag-and-drop* instrument, will create a slave *drag-and-drop* instrument. *DmManager* of B will also create a slave *drag-and-drop* instrument for A. Now, if at the same time the user of B starts a *push-and-pop* on B then a slave *push-and-pop* instrument will be added to both *DmManager* of A and *DmManager* of B, though *DmManager* of B already owns a master *push-and-pop* instrument. In other words, if the two users of A and B perform *drag-and-drop* like operations at the same time, each *DmManager* will three instruments: one master and two slaves per *DmManager*.

Several types of instruments can be used without any problem, but only on type of instrument is linked to a given surface (i.e. an input device). At any time, a *DmManager* owns exactly one master instrument and as many slave instruments as *drag-and-drop* like manipulations are running.

4.3 Multi-computer support

Implementing an interaction technique, such as drag-and-drop or its extensions across machines, is a real challenge. Multi-computer environments are particularly difficult to handle especially when interaction takes place dynamically. In this context, where the user starts dragging on one computer and drops on another machine that he selects "on the fly", several well-known problems arise: (1) the spontaneous device sharing problem [9], (2) incompatibility of heterogeneous windowing systems implementations and (3) data transfer problem.

Hence, existing drag-and-drop implementations do not provide any support for distributed display environments. In related work, *pick-and-drop* and *hyper-dragging* are designed and implemented for interactions between computers. They do not aim at providing a general implementation that would solve the problems mnetioned above and extend easily to other *drag-and-drop* like techniques.

Pick-and-drop is probably the most emblematic example as it is the first multi-computer direct manipulation technique. Nevertheless as for others that followed, the implementation is ad-hoc and cannot easily extend to support the requirements needed by most other extensions discussed earlier in this paper. Indeed, based on a combination of Pen-ID associated with a specific device (a pen) and a server (called pen manager), the implementation of *Pick-and-drop* is not only device dependent but also very specialized. It cannot extend easily to handle most of the feedback needed in other drag-and-drop extensions.

Providing a framework that would solve all of the three problems mentioned above in a general way goes beyond the scope of this paper. In particular, over the past decades, progress with data transfer protocols and formats have been made. When this field comes to maturity, solutions for the data transfer problem will come somewhat naturally. More recently, the need for binding together devices "on the fly" to share information [2], collaborate or communicate has already led to partial solutions for the "spontaneous device sharing problem" [9]. The aim of our work is to propose a solution for the second problem mentioned above: incompatibility of heterogeneous windowing systems implementations. However, we have designed a simple basis for supporting multiple users to work collaboratively on distributed display environments.

The concept of *SharedSurface* was introduced. *SharedSurface* shares the surface of a given GUI component by a registration on a *RemoteSurfaceServer*. All the shared surfaces compose one continuous shared space where pointers can move transparently from one to another and perform *drag-and-drop* like operations as well. In order to support these operation each shared surface handles one unique *DmManager* and each *DmManager* handles one unique master instrument.

4.3.1 Remote surface server

A *RemoteSurfaceServer* has a very simple behaviour: it maintains a list of shared surfaces and manages the topology of surfaces. For example, if a surface needs to know which surface is on its left, it asks the RemoteSurfaceServer.

4.3.2 Shared surface

A *SharedSurface* owns a name and a unique ID. It receives all input events that occur on the linked component and controls a fake pointer. The fake pointer has the same unique ID as the shared surface that controls it. The fake pointer can move over all shared surfaces registered on the *RemoteSurfaceServer*.

This implies that several pointers can act on the same surface at a given time. The model described here allows identifying each input device thanks to its ID. Thus, components (or domain objects) have to be able to deal with multiple input devices. It is possible for a *DmManager* to handle simultaneous *drag-and-drop* like manipulations that end on the same target component, but this target component must be able to handle mixed events from the two simultaneous interactions.

4.4 Pros & Cons of the model

As already said, an asset of this model is to offer a consistent behaviour through the different applications that use *drag-and-drop* or *drag-and-drop-like* techniques unlike X-Window/Motif and Windows/OLE systems [13]. Downside, customizations (e.g. visual feedbacks like *drag-over* effects) are not allowed for source and target components. Such customizations could conflict with the visual feedbacks or hints managed by the *DmAbstractInstrument* itself. Moreover, cancellation of the operation can only be made by the *DmAbstractInstrument* and therefore the source application has no control over it.

Another asset of this model is to simplify the concepts. A developer who wants his component to support *drag-and-drop* like manipulations only deals with

five classes.

However, the data management has not been simplified. If transfered data is of basic type (i.e. text, files list, image), then data creation is very simple. Else, developer will have to write its own class to manage data or to convert data into a textual form. As mentioned earlier, we do not focus on data management in this paper: the proposed model concentrates on the event and interaction models. The definition of a scalable way to represent data goes past our establishment of an interaction model. Furthermore, such data representations already exist (e.g. OLE, MIME, etc.).

The main asset of this model is to allow use of any *drag-and-drop* extensions by only replacing the *dmMasterInstrument* entity. Source and target components do not need to be aware of the interaction technique to be used. The *dmMasterInstrument* property can be changed dynamically. The *DmManager* serves as an intermediary between source and target components, the instrument and the user.

And finally, this model can support multi-device systems and thus multi-computer systems thanks to the management of a device identifier.

5 IMPLEMENTATION

The model previously described has been implemented in a Java API. Three type of instrument have been included as this API: a *drag-and-drop* instrument and a *push-and-pop* instrument that can be deactivated into *push-and-throw*. This combination has been chosen because *push-and-pop* was found to be the most efficient [12] and thanks to the *push-and-throw* deactivation, all possible locations on a workspace can be reached (contrary to *push-and-pop*).

This implementation has a limitation: system-wide *drag-and-drop* like operations are not allowed. It is only possible to manipulate domain objects that are registered in the *DmManager* thus created in a Java environment. The reason for that is that doing system-wide *drag-and-drop* like operations properly would require this system to be implemented at the windowing system level. This, in turn requires a redesign and major reimplementation of very low level system routines that goes beyond our scope.

5.1 Implementation choices

5.1.1 The communication technology

The communication technology chosen is the Remote Method Invocation (RMI) of Java. This led us to introduce several interfaces. Indeed, a stub interface is needed for each class for which methods can be invoked remotely (e.g. *SharedSurface*, *RemoteSurfaceServer*, *DmManager*, etc.).

When invoked remotely methods are likely to throw *RemoteException*. It seemed inadequate to ask that source and target components' listeners' methods throw *RemoteExceptions*. For this reason, a proxy is created for each listener.

On the other hand, parameters of remotely invokable methods have to be serializable (implement the *Serializable* interface). This includes particularly events and data.

The use of RMI is efficient (no lag) and transparent for users. There is no sensible difference when a fake pointer moves on its local surface or on a remote shared surface.

5.1.2 Input events

All input events are captured by adding an *AWTEventListener* to the default *Toolkit*. Events are consumed and re-generated while adding a device ID and moving the fake pointer.

To make sure that the system pointer does not exit the component linked to the *SharedSurface*, a *Robot* is used to re-centre the system pointer. However, it is possible to avoid the use of a *Robot*. Removing the *Robot* presents one advantage but also one drawback. The advantage is that it avoids potential conflicts with direct pointing devices (e.g. Mimio, SmartBoard). Hence without the *Robot*, direct pointing devices can be used safely. The problem though is that in this case, *drag-and-drop* is bound to a single surface. Only *push-and-pop* or other such techniques can be used to move an object from one surface to another. This is consistent: a direct pointing device is bound to its surface.

5.1.3 Visuals

In order to display fake pointers and instrument feedbacks, a layered pane is created on top of the component linked to a *SharedSurface*.

Fig. 5. (Top to bottom) (a) *drag-and-drop* (b) *push-and-pop* (c) *push-and-throw* (d) two concurrent manipulations: a *push-and-throw* from A to B and a *push-and-pop* from B to A.

5.1.4 Drag-and-drop instruments

The *drag-and-drop* instrument (fig. 5 - (a)) has been implemented without using the existing *drag-and-drop* mechanism included in Java. It directly analyzes the mouse event on the source and target components.

A *ghost*, provided by the source component, is displayed at the pointer location by a slave instrument.

Communications between the master instrument and its slaves occur at different stages: when the source component listener is needed (at the beginning of a *drag-and-drop*), when the location of the fake pointer changes (so the ghost should also move), when the target component changes and when the ghost image changes (occurs only once).

Concerning the events, if a target refuses an operation when the *dmEntered()* method is triggered, it will not be notified of any other event (*dmExited()*, *dmTrgOver()*, *dmReceive()*).

5.1.5 Push-and-pop instrument

The *push-and-pop* instrument (fig. 5 - (b) and (d)) creates an array of targets around the pointer [12,17].

The set of targets only contains targets that accept the manipulated domain object. Indeed, when creating the set, each target's *dmEntered()* and *dmExited()* methods are called. If the event's *accept()* method is called, the potential target component is included in the array. Otherwise, if *reject()* method is called, the component is ignored.

The array of targets is laid out with the mechanism described by authors of [17]. The original algorithm was enhanced by the ability to manage a set of icons representing targets of different sizes. If a target provides a graphical proxy which is too large, the image is reduced. This can be done either by scaling the entire image or by cropping it. The best method has still to be specified. Examples of too large images are returned by components like text areas or text fields.

The *take-off area* (i.e. array of targets) contains a circle at the starting point of the manipulation. By pointing back to this circle, the *push-and-pop* is deactivated into *push-and-throw* [12] (fig. 5 - (c) and (d)). As mentioned earlier, *Push-and-throw* is another technique that allows the user to reach any part of the workspace and not only on a pre-selected set of targets. This is useful when rearranging icons on a desktop or when moving some text inside a text area, for example.

5.2 Modularity

In order to use a new *drag-and-drop* like technique, developers need to create a new class that inherits from *DmAbstractInstrument*. Events described in the *DmSourceListener* and *DmTargetListener* have to be triggered by this new instrument.

In order to enable components for *drag-and-drop* like interactions, they have to implement *DmSourceListener* and/or *DmTargetListener* interfaces and register themselves by calling the *(Un)RegisterSource()* and/or *(Un)RegisterTarget()* methods.

It is possible to switch instruments easily by setting the *dmMasterInstrument* property of the *DmManager*. Note that a *drag-and-drop* like manipulation should not be running while switching instruments.

Multi-machine manipulations introduce a new issue: topology of the different displays has to be known because relative positions of components are used to generate the *take-off area* [15]. Such topology is used by authors of [3] but do not manage mobile displays (e.g. laptops). A much more dynamic topology is proposed by authors of [8], but it uses a complex camera-driven system. At this point, a compromise has to be found between functionalities and complexity.

6 CONCLUSION

We proposed a new implementation model for *drag-and-drop* and its extensions. The main strengths of the model are: (1) openess and modularity for the programmer - programming a new technique requires little effort (2) modularity for the user - who can choose his preferred technique a little like one chooses his favourite X-Window manager and (3) multi-computer support for *drag-and-drop* like operations.

Compared to many *drag-and-drop* implementations, we introduced more consistency by adding the notion of instrument and a *DmManager* entity that ensures the same behaviour for every *drag-and-drop* like operation.

In addition to the model, we implemented a Java-based API that implements three instruments: drag-and-drop, push-and-throw, and push-and-pop. This API and an example of use are available at http://edel.lirmm.fr/dragging/.

References

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

[2] Jolle Coutaz, Stanislaw Borkowski, and Nicolas Barralon Coupling Interaction Resources: an Analytical Model. In EUSAI'2005, pages 183–188, 2005.

[3] 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.

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

[5] Fraunhofer institute. The dynawall project. *http://www.ipsi.fraunhofer.de/ambiente/english/projekte/projekte/dynawall.html*.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] Sony Japan. Flyingpointer. *http://www.sony.jp/products/consumer/pcom/software_02q1/flyingpointer*.

[19] Stanford University ComputerScience. The stanford interactive workspaces project. *http://iwork.stanford.edu/*.