Extending a Graphical Toolkit for Two-Handed Interaction

Stéphane Chatty

Centre d'Études de la Navigation Aérienne
7 avenue Edouard Belin
31055 TOULOUSE CEDEX
FRANCE
chatty@dgac.fr

Laboratoire de Recherche en Informatique
CNRS & Université de Paris Sud
91405 ORSAY CEDEX
FRANCE
chatty@lri.fr

ABSTRACT

Multimodal interaction combines input from multiple sensors such as pointing devices or speech recognition systems, in order to achieve more fluid and natural interaction. Two-handed interaction has been used recently to enrich graphical interaction. Building applications that use such combined interaction requires new software techniques and frameworks. Using additional devices means that user interface toolkits must be more flexible with regard to input devices and event types. The possibility of parallel interactions must also be taken into account, with consequences on the structure of toolkits. Finally, frameworks must be provided for the combination of events and status of several devices. This paper reports on the extensions we made to the direct manipulation interface toolkit Whizz in order to experiment two-handed interaction. These extensions range from structural adaptations of the toolkit to new techniques for specifying the time-dependent fusion of events.

Keywords: interaction styles, multimodal interaction, two-handed interaction, graphical toolkit, direct manipulation.

INTRODUCTION

Though many aspects of their construction are still a matter of research, graphical interfaces are now well known. Most of them make use of a pointing device that users manipulate with their dominant hand. This has led to the introduction of a number of interaction styles centered around that pointing device: buttons, menus, point-and-click, drag-and-drop, and so on. Such interaction styles enable interface designers to build systems that are fairly efficient and easy to use. However, the efficiency of such interfaces can probably be improved. In the real world, we perform many tasks with both hands, because it is more efficient. Because of these natural skills, drawing pictures with a MacDraw-like tool is sometimes frustrating: a significant part of the time is spent in moving the mouse around to select tools, locking objects so that they do not move when working on them, and so on. This is very similar to handcrafting with one's hand behind one's back: tools make it possible, but at the cost of a considerable waste of time. When considering graphical software, it is interesting to note that keyboard short-cuts are a way for us to use our non-dominant hand when drawing, and to avoid unnecessary movements with the dominant one. A recent study shows that carefully designed two-handed graphical interaction can improve the efficiency of interfaces [13].

Apart from drawing tools, a number of application domains could benefit from such interfaces. Among these are the domains where users are well-trained professionals, whose attention is focused on the task they are performing. We believe that air-traffic control is a good example of such a domain. The interfaces provided to air-traffic controllers essentially consist of a presentation of the situation in air-space: it is the so-called "radar image", which is composed of maps and a number of symbols representing way-points, aircraft, and other useful information. Many countries are currently working on new interfaces that allow controllers to manipulate these representations with modern interaction techniques. At CENA, we are exploring the hypothesis that controllers might be able to plan their work by manipulating future trajectories of aircraft. This is why we are investigating efficient techniques for interaction with curves and objects moving along them. Among other techniques, we are developing two-handed interfaces in order to test their efficiency with experiments and measurements.

Graphical interaction provides designers with many degrees of freedom, but also with many possibilities to build bad systems. This is even more true for two-handed interaction, which can even be made less efficient than single-handed equivalents. In order to explore possible interaction styles and determine the most efficient ones for a given task, studies on two-handed interaction must be supported by sufficiently versatile software tools. Some of the currently available graphical toolkits provide enough support for building highly interactive interfaces. However, they do not support, or they even impede, the construction of two-handed interfaces. This paper reports on the extensions that were made to the Whizz graphical toolkit so as to handle two-handed interaction. We first review a number of two-handed interaction styles and identify three classes of technical issues raised by their construction. These classes are closely related to a more general classification of multimodal interfaces. We then give a brief description of Whizz and how it supports the construction of single-handed graphical interfaces. The last three sections are devoted to the three classes of technical issues raised by

two-handed interfaces, and to the solutions to these issues that were implemented in Whizz.

RELATED WORK

Graphical interaction and the related software issues have been widely explored for more than ten years. A number of graphical toolkits have been proposed to ease the construction of graphical presentations and the description of mouse and keyboard-based dialogues. The X Toolkit and InterViews [15] are such toolkits. Systems such as Garnet [16] and X_{TV} [1, 2] pay great attention to the description of direct manipulation interfaces. However, until now these systems have been dedicated to interfaces based on a single pointing device and a keyboard.

The notion of multimodal interaction was identified by Bolt [6]. Different opinions still exist about the exact definition of the term multimodal. However, most authors who recently wrote on that subject considered it as defining systems that feature multiple input devices (multi-sensor interaction) or multiple interpretations of input issued through a single device. A number of studies have dealt with the combination of voice recognition and graphical interaction [20, 10]. Other authors studied the combination of direct manipulation and 2D gesture recognition [19, 14]. Some graphic toolkits such as Sassafras [11] and Grandma [19] support the construction of multi-threaded interfaces. At the French IHM'91 workshop [12], a classification of multimodal systems, later refined by Nigay and Coutaz [17], was devised. This classification is organized along two axes: the sequential or concurrent use of modalities, and their independent or combined interpretation. Systems which feature two or more modalities in parallel, with combined interpretation of input tokens, are called synergistic. The "Put That There" style of interaction is an example of synergistic multimodal input. Other possibilities are exclusive, concurrent and alternate modalities. Nigay and Coutaz also stress the distinction between combining low-level input tokens, such as phonemes and mouse events, and high-level ones. An example of high-level fusion is the "Put That There" style, where the meaning of words and the identity of the designated objects are necessary to the fusion of input data.

Two-handed interaction was suggested a long time ago [7], and has gained more popularity since recent work by Xerox PARC and the University of Toronto [4]. Two-handed interaction is a special case of multi-sensor interaction, and therefore of multimodal interaction. As other multimodal interfaces, two-handed interfaces may use exclusive, alternate, concurrent or synergistic modalities. Similarly, such interfaces feature high-level fusion: for instance, one could imagine a two handed iconic interface where one hand would select objects and the other would choose operations in menus. They may also feature low-level fusion, as in the well known shift-clicks of your favorite desktop interface. We will see later in this paper that other forms of low-level fusion in two-handed interaction deeply involve time. Finally, two-handed interaction also shares concerns with multi-user interaction. Some multi-user systems, such as MMM [3], allow several users to edit the same object in parallel. This raises the same need for low level fusion as two handed interaction.

TWO-HANDED INTERACTION

Let us now review some styles of graphical interaction involving both hands. Single-handed input offers a number of degrees of freedom to designers, and two-handed input increases that freedom. First, the debate on the choice of input devices will reappear: is it better to use two mice, or a mouse and a trackball, for instance? We will not open that debate here, but acknowledge the diversity of choices. For example, virtual reality designers will want to use two digital gloves. In their 1986 paper, Buxton and Myers explored the use of a graphics tablet and a slider box (a sort of 1-D mouse). Today's keyboards can definitely be considered as two-handed input devices for text input, even if their management is straightforward.

In the domain of graphics manipulation, a quick survey reveals potential applications to several combinations of devices. We all are used to shift-clicks, which combine actions on the mouse with one hand to actions on the keyboard with the other hand. One-dimensional input devices may also be combined with the mouse: a slider or a rotary knob controls the zoom factor of the display, while the mouse is used to draw. Using these devices in parallel would save time when drawing precise figures, as one often switches from high scale (to draw details) to normal scale (to see the overall result). Finally, two pointing devices may be used, as in Xerox PARC's Toolglass. With the exception of digital gloves, the use of two pointing devices has the greatest power of expression, and is the most demanding in terms of software complexity. For that reason, the majority of our examples will use two pointers, considering that other two-handed interaction will be handled in a similar way.

Guidelines for two-handed interaction

There is probably no task for which two-handed input should be the only way to perform operations: there will always be situations in which one hand is used for another task, such as holding a sheet of paper or a glass of water. This means that all systems based on two-handed input should be usable with one hand only. Obvious design rules suggest that one-handed and two-handed actions for the same operation should be similar, and that one should be easily inferred from the other. We suggest that this requirement is most easily met when using paradigms from the real world. In our opinion, interfaces based on such paradigms just need to be extended for two-handed input according to their paradigm. For instance, we all have a good idea of what happens if we pick an object with one hand and drag it; similarly, something predictable should happen if we pick an object with both hands and stretch it. We claim that two-handed interaction styles should generally follow that rule.

This being stated, there still are many possibilities for two-handed interaction. The classification of multimodal interfaces provides a good framework for exploring these possibilities, because it defines a kind of hierarchy among them. The simplest usage of several modalities is their exclusive usage, and it is the basis of other usages. If a system allows parallel or combined modalities, it is obviously able to provide independent interactions with these modalities, except if that possibility has explicitly been disabled. The next step in

complexity is the use of parallel interactions: the two hands work at the same time. Finally, the most complex interfaces are those which combine the input from both hands. Using that classification, we will see how single-handed graphical interaction can be extended.

Independent interaction

A simple way to smoothly extend one-handed interfaces consists of adding a second pointing device that can be used in the same way as the first. This enables users to save a considerable amount of time when pressing buttons or selecting tools: for instance, the non-dominant hand can select tools while the dominant one rests on the object which is being manipulated. Such interfaces can still be used with one hand: they are just more efficient with both hands. Xerox PARC's Toolglass is a sophisticated version of that: having the tools located on a transparent palette that can be moved around allows users to keep their focus on the object of interest. Similar interactions may be used to control global parameters of the display, such as the zoom factor, without moving the dominant hand.

Finally, the second pointing device could be used for drawing pictures or moving icons. However, people are slower in performing precision tasks with their non-dominant hand. The designers of Toolglass solved this issue by assigning the task of moving a palette to that hand, whose designation is easy. The size of the target compensates for the relative imprecision of the hand. Another possibility may be suggested: using the non-dominant hand for the designation of small objects with large cursors. Such large cursors would have to be everywhere "hot", as opposed to traditional cursors that only have a hot spot. However, even if the system may be designed to nicely take care of imprecision, there is no real benefit in using the non-dominant hand where the dominant one can be used, except if actions can be performed in parallel.

Parallel interaction

Parallel interaction is the natural next step as soon as two-handed interaction is possible. Even though most people are not trained to perform real independent tasks in parallel, we all unconsciously use our non-dominant hand for secondary tasks, such as bringing a tool to the dominant one. The utility of Toolglass, for instance, would be limited if interactions had to be strictly serialized: our hands are not used to waiting for each other before performing operations, and imposing it would be frustrating. Therefore, parallelism is inherent to two-handed interaction, because of our natural habits. Consequently, all the examples of interaction styles we mention in this paper use parallel interaction, in a more or less obvious way.

Some applications can also be found to real parallel interaction, where the two hands perform independent tasks of the same importance. We should of course mention games. Simulation games will make use of parallel actions when the tasks they simulate make use of them: driving a car, or piloting a plane, for example. Other games may be designed to challenge human capabilities: juggling games or two-handed action games can be imagined. When exploring the technical issues associated to parallel interaction, we will also mention the use of parallelism for manipulating cards in a game of Patience. Nevertheless, we believe that parallelism is more a necessity than a goal in itself, and that it will mainly be useful when combining actions of the two hands.

Combined interaction

The most elaborate way to use two pointing devices is to combine their actions. In the real world, we often use our non-dominant hand to hold objects while performing precise operations on them. We also use it in coordination with the dominant hand to provide additional strength, or to manipulate objects that are more precisely moved when held from two distant points. Traditional interfaces have replaced the second hand by a form of magic: in a drawing tool, when we move one end of a segment, the other end is held by an invisible hand. What we suggest here is to disable that magic when two hands are at work. The non-dominant hand can hold the end of the segment, with no need for magic. This leads to an interaction style based on a physical metaphor: if one hand picks the end of a segment and drags it, the whole segment moves; if the second hand holds the other end during that operation, the segment is deformed, like a metal stick would be. This is what we call "hold-and-pull".

Another example of combined interaction is the simultaneous designation of two objects. This type of interaction is used in the real world as a security for critical operations [18]: an operation will be performed only if two buttons are pressed simultaneously, for instance. This can be immediately transposed to graphical interfaces. For instance, the designer of a drawing editor could decide that, by clicking on two graphical buttons simultaneously, a user may quit the editor without saving the edited files. The role of time in such interactions is important: as in double-clicking, a reasonable tolerance must be specified. Therefore, time has to be taken into account when performing the fusion of input data.

We now have identified several two-handed interaction styles, which illustrate the different aspects of multi-sensor interaction: independent, parallel and combined interaction. In the rest of this article, we will see how support for such interaction styles was added to Whizz.

AN OVERVIEW OF WHIZZ

Whizz is a toolkit aimed at describing the behaviour of highly interactive or animated user interfaces. It was designed with three main goals in mind:

- homogeneity: we consider that direct manipulation by users, animation, and data visualization are different aspects of the dynamic behaviour of an interface. The design of an interactive object and its graphical behaviour should be reusable in different contexts: for instance, a scrollbar always has the same behaviour, whether driven by a user's actions on the mouse, or by a clock (when one of the arrows at the ends of the scrollbar is "depressed"), or by the variations of some piece of data (when the size of a document changes, for instance). This is illustrated by figure 1.
- straightforward visual representation: Whizz was designed to allow the development of visual user interface construction tools, with the goal of applying such tools to the design of highly interactive user interfaces. This led us to identifying a number of basic building blocks that are to the behaviour of an interface what graphical objects are to its visual appearance.

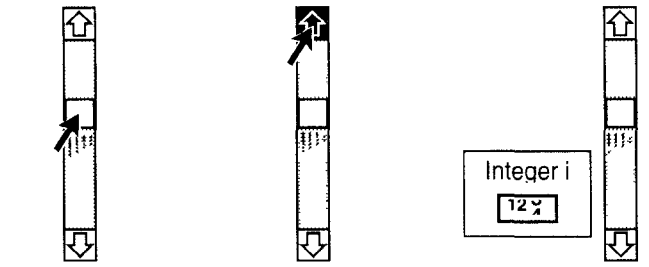

Figure 1: A scrollbar has the same graphical behaviour, whether driven by time, the user's action, or data variations

Graphical interfaces are obtained by assembling a number of these building blocks.

• extensibility: Whizz implements a number of graphical behaviours and handles mouse and keyboard input, but we also wanted its paradigm to be usable for other media such as sound, and to other input devices such as rotating knobs or 3D devices.

In order to achieve these goals, Whizz has an object oriented structure, and uses a data-flow model, as many signal processing and music synthesis systems [9]. All the objects manipulated in Whizz are *modules* that can be connected together with links that carry pieces of information. Among these modules, Whizz makes a distinction between graphical objects and graphical behaviours; the latter are further decomposed into movement shape and movement source. Movement shapes are implemented by objects that manage trajectories: straight lines, circles, paths, etc. Similar objects manage other visual variations, such as color changes. Movement sources are clocks, active values or representation of the user's actions. A simple movement can thus be achieved by connecting a source, a trajectory and a graphical object.

The programming interface of Whizz uses a musical metaphor in order to make its structure easy to learn: movement sources are tempos, trajectories are instruments, and graphical objects are dancers; the small pieces of information circulating from tempos to instruments, then from instruments to dancers, are notes. Depending on their type, dancers have a number of input slots which control their position, shape and appearance: for instance, a segment has slots that control its two ends. A note reaching one of these slots will change the position of the corresponding end. Similarly, instruments have an input slot that control the step-by-step emission of notes on their output slots; they also have other input slots to allow random access. Figure 2 illustrates how a simple animation scene can be built with Whizz. Figure 3 shows another simple Whizz construction that describes (with a minor simplification) the action of dragging an icon. This construction may be built when a "button down" event occurs on an icon, by connecting the two modules. It may be destroyed, by removing the connection, when a "button up" event is detected.

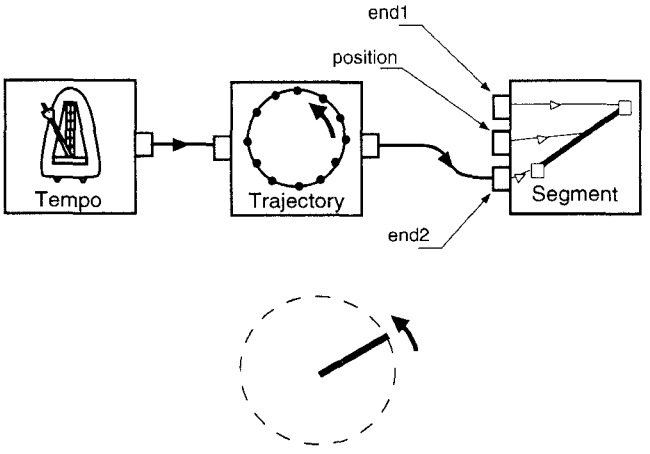

Figure 2: Animating a segment: one of the ends of the segment is connected to the output of a circular trajectory. The trajectory emits positions when it receives pulses from the tempo.

Complex graphical interfaces can be achieved by establishing links between a number of dancers, instruments and tempos, thus utilizing the underlying data-flow structure of Whizz. Other modules than the ones we described may be added to build more complex behaviours: filters that perform numeric or geometric operations, logic gates, and so on. For instance, the construction of figure 3 does not exactly describe the dragging of an icon as it is usually performed: it does not take into account the relative position of the cursor on the icon when the mouse was clicked. In practice, two more modules should be used: one for storing the offset, and the other for combining it with the position of the mouse.

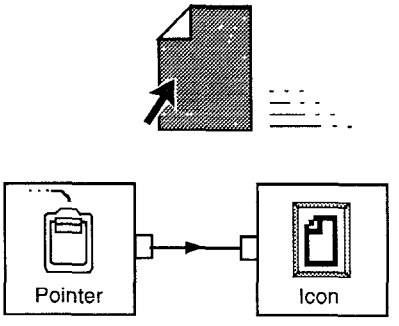

Figure 3: Dragging an icon: the module representing the user's action with the mouse is connected to the position of the icon.

Flow graphs built this way give account of the continuous evolutions of the display. Isolated evolutions such as those traditionally associated to input events can also be handled by Whizz. Events such as button clicks can either be converted to notes, or they can be associated to reconfigurations of the flow graph, in order to change the behaviour of the

interface or create new graphical objects, for instance. This is what happens in the example of figure 3, where connections and disconnections are associated to "button down" and "button up" events. Such connections and disconnections are currently performed by programming, but techniques for graphically describing them are being studied.

An experimental visual construction tool based on Whizz was developed, and is still under development. This tool, called Whizz'Ed, uses the graphical notation that we used in figures 2 and 3 for describing data flows. Techniques for representing and specifying reactions to events and reconfigurations of the flow graph are currently being devised. These representations, even though they were not yet implemented in Whizz'Ed, are used in the the rest of this article. More details and other features of Whizz can be found in [8]. Whizz was implemented in C++ on top of X_{TV}, a graphical toolkit designed at the University of Paris Sud to serve as the basis for experiments on interaction styles and user interface software [1]. Whizz has already been used by programmers in several user interface development projects involving direct manipulation, such as an aeronautical map editor or a radar image featuring gesture recognition. We will now see how Whizz can be extended to support the development of two-handed interfaces.

INDEPENDENT INTERACTION

Before thinking of using two devices in parallel and combining their actions, the first step towards building two-handed interfaces consists in being able to handle two devices. This will only allow independent, exclusive interaction, but other interaction styles depend on this one being properly supported. We will not consider here the low-level details of connecting and managing devices other than the traditional mouse and keyboard. Such issues can be solved by using operating system facilities or extensions to a window server. We will rather focus on issues related to interaction management, which affect the structure of graphical toolkits. These issues are the dynamic management of event types, the event handling scheme, and the support for handling the imprecision of the non-dominant hand.

New event types

A great effort has been put in graphical toolkits to provide a homogeneous framework for manipulating input from the keyboard and the mouse. This homogeneity is the key to the construction of maintainable interactive software, and must not be lost when adding new devices. For that reason, our first requirement for a multi-sensor interaction toolkit is the smooth integration of the signals from new devices with those from traditional ones. Event-based systems provide an elegant solution to this requirement: one just needs to insert events in the event queue. However, this supposes that new event types can be created. Such new event types include those associated to the new devices or modalities (an event type for each gesture type, for instance). They may also include synthetic event types resulting from the combination of primitive event types. For two-handed interaction, no new primitive type is needed, but synthetic ones will be necessary to describe simultaneous clicks with both pointing devices, for instance.

That ability to manage new event types is rarely found in user interface toolkits; it is a serious problem when using a statically typed language such as C++: if all event types cannot be statically defined, the typing scheme of the language cannot be used. In order to solve this problem, Whizz provides a mechanism for dynamically defining new event types. In Whizz, an event type is a full blown object, that can be instantiated when necessary. Every event type contains a description of the fields found in events of that type. When creating an event, one only needs to provide a reference to the desired event type, and Whizz allocates the necessary space. This mechanism is compatible with the existence of default event types (which are globally defined objects), and with the creation of new event types.

Event selection and handling

Once events are created and integrated in the event queue, they have to be dispatched to graphical objects and handled. This means that the event selection and distribution mechanism provided by the toolkit has to be extensible to dynamically defined event types. For example, a programmer may wish to bind "circling" gestures on a graphical object to a callback function, and "underlining" gestures to another callback function. Two-handed interaction also introduces a need for flexibility in event selection. If a two-handed interface is to offer an equal treatment to both hands, we can expect the graphical toolkit to handle events from both pointing devices in the same way. For instance, let us consider the graphical button and the callback function of figure 4. The toolkit must allow programmers to bind "button down" events on the button to the callback function, without mentioning a specific device. The expected behaviour is illustrated in figure 5: clicking with either pointer results in the callback function being called.


```
void OK (WhzEvent* ev) {
      printf ("ok\n");
}
```

Figure 4: A graphical button and the associated callback function.

However, we also want to avoid interferences: depressing a button with the left hand, then releasing another button with the right one, must not be interpreted as a button click, as shown in figure 6. As soon as an action is started, the symmetry is broken, and the toolkit must allow programmers to specify how. The event selection mechanism of X_{TV}, which is used in Whizz, permits such precise definitions. First, X_{TV} has an explicit notion of devices: there are classes of devices such as mice or keyboards, that are instantiated to represent

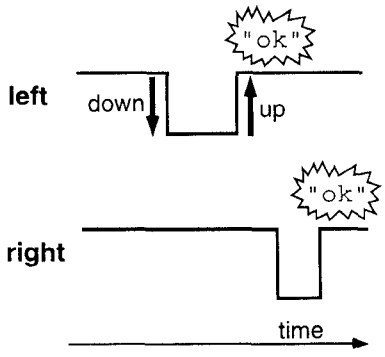

Figure 5: Clicking on the button with either pointing device produces the same result

the physical devices used in a program. Then, a programmer can bind a number of reactions to a graphical object. When doing so, one must specify the set of event types and the set of devices or device types whose events will be managed by the reaction. Reactions can be dynamically created and destroyed, bound and unbound, so that objects can be made temporarily sensitive to specific events. This makes it easy to specify the behaviour of the graphical button of figure 4, as illustrated in figure 7: a reaction is permanently bound to "button down" events from any device on the button; when this reaction is triggered, it binds a new reaction to the button; this second reaction, associated to "button up" events from the device that emitted a "button down", activates the button, then destroys itself. In single-handed interfaces, this second reaction could be permanent like the first one: mechanical constraints impose that a "button up" event necessarily follows a "button down" event, and there is only one pointing device to select events from. But here, there is no such mechanical constraint. Programmers have to specify which device they are interested in, and they only have that information when a button has been depressed. This is why the second reaction has to be bound, or even created temporarily.

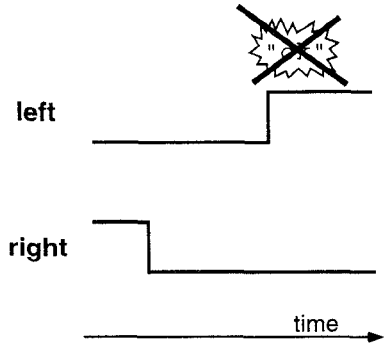

Figure 6: Interferences between pointing devices must be avoided.

```
/* a button */
Button but (100, 50, 200, 100, "OK");

/* a reaction bound to any button-down event */
XtvReaction down (&down_callback);
down.Bind (but, MouseButtonDn, XtvMice);

void down_callback (XtvReaction* r, XtvEvent* ev) {
    /* provide visual feedback */
    Button* b = (Button*) ev->GetTarget ();
    b->Select ();
    /* a reaction to button-up from the same device */
    XtvReaction* up = new XtvReaction (&up_callback);
    up->Bind (*but, MouseButtonUp, ev->GetDevice ());
}

void up_callback (XtvReaction* r, XtvEvent* ev) {
    /* trigger action associated to the button */
    Button* b = (Button*) ev->GetTarget ();
    b->Fire ();
    /* no more reaction bound to button-up events */
    delete r;
}
```

Figure 7: Using X_{TV} to make a button sensitive to all "button down" events, but only to "button up" events emitted from the corresponding device.

Handling imprecision

Finally, irrespective of the amount of symmetry being desired between the two hands, the imprecision of the non-dominant hand has to be accounted for. We proposed to use bigger cursors and ensure that all of their points are active: for example, clicking while the tip of an arrow is over an object or when its tail is on it should have the same consequences. This challenges another assumption heavily used by graphical toolkits: the fact that designations occur with a precision of one pixel. All mouse events, and more significantly their dispatching, are built around this assumption: the cursor has a "hot spot", and events contain the position of that hot spot. The dispatching mechanism usually considers every graphical object in a view where an event occurred, and asks that graphical object whether it contains the pixel where the hot spot is located. In order to support less precise cursors, toolkits would have to replace this test of a point against a shape with the test of a shape against another shape. This would of course require more complex and more costly geometric computations. In the current version of Whizz, we only implemented circular cursors, or circular zones around arbitrary cursors. Nevertheless, we believe that the present speed of computers allows the implementation of a more general mechanism.

The use of cursors that are "everywhere hot" also raises the issue of choosing the selection when the cursor is over two non-overlapping objects. The usual technique that consists in selecting the first graphical object under the cursor cannot be applied here: it relies on the fact that other objects under the cursor are necessarily under the first object. Here, two objects with no obvious order can be under the cursor at the same time. This issue can be addressed with techniques such as selecting the object that is closest to the center of the cursor, and giving a visual feedback of that selection. Other, more complex solutions that we have not explored yet, involve the selection of several objects at the same time. This technique could allow more realistic interaction, but its usability and feasibility still have to be assessed.

PARALLEL INTERACTION

One of the novelties of two-handed input that has an impact on user-interface construction is the possibility to perform parallel actions. Isolated actions, which are associated with a single event, pose no real problem. For instance, let us consider a two-handed MacDraw-like tool in which the dominant hand draws figures, while the non-dominant one selects tools in a palette. The parallelism here lies in the ability to move both hands at the same time, and to click with the non-dominant pointing device at any time: this only relies on the correct management of the event queue. However, many actions performed with a pointing device are made of several events rather than a single one: a click is a "button down" followed by a "button up", a drag has additional "mouse move" events. Furthermore, these actions are generally associated with a visual feedback. Performing two or more of these actions in parallel imposes constraints on the underlying toolkit.

To illustrate this issue and experiment on it, we extended a simple graphical application to two-handed interaction. The application we chose is a game of Patience (Solitaire), played with all cards aligned in four rows, face up. An empty space is left in each row, and a card can be moved to this space according to a simple rule, thus leaving a new empty space (figure 8). We had already implemented a simple version of this game, where cards were successively moved with drag actions. However, players of this game usually think of several movements ahead. When playing with real cards, they often use both hands and move two cards at a time, one occupying the space left empty by the other. We decided to experiment with that technique using two pointing devices, and to allow users to drag two cards at a time.

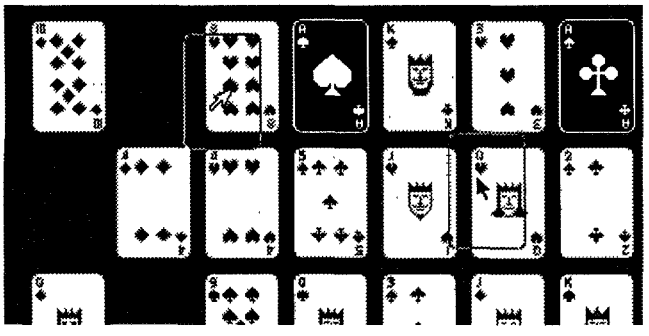

Figure 8: Two cards are moved at the same time.

Actions as independent entities

Implementing parallel drags has consequences on input management. A drag is a long action, composed of several events. During the drag, data has to be stored to maintain the status of the action and its visual feedback. This storage of data is a form a dialogue control. Depending on the architectural model implemented by the toolkit, this control may be global, associated to the visual representation, or independent. Parallel actions are only possible if the control is independent, as stressed by Rubine [19]. If the visual feedback associated to a dragged card were managed as a property of the window, only one card could be moved at a time (or the feedback would keep blinking from one pointer to the other). The issue here is that we want to manage several actions at the same time, and therefore we need to store the status of several dialogue controls. The most straightforward solution consists in considering actions as full-blown objects, dynamically created and destroyed when needed. Facilities such as Garnet's interactors or the action modules of Whizz are well suited for such situations. With Whizz, every new user's action results in the instantiation of a module that emits the positions of the pointer, and of a graphical object connected to that module. For instance, in figure 9, a user clicked on Icon 1 with the left pointer, and on Icon 2 with the right pointer, thus obtaining the construction shown. The interface allowing parallel interactions on cards is shown in figure 8.

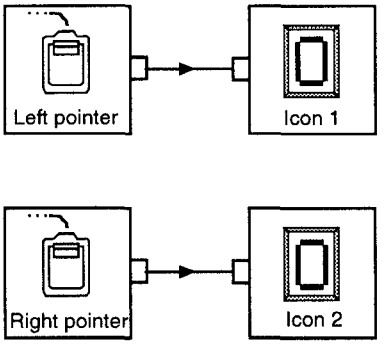

Figure 9: The Whizz construction obtained when two cards are moved in parallel. The flow graph is composed of two independent parts.

Two actions on the same object

A natural extension of parallel interactions consists in using both hands to manipulate a single object. This may be understood as a form a combined interaction, but actually it is not. For instance, a pointer may be used to control one end of a segment, while the other pointer controls the other end. As long as one end can be moved independently from the other (which is usually the case), no combination is required: when an event is received from a pointer, the corresponding end is moved. Two events will result in two moves. When using Whizz, this is obtained by simply connecting the two pointers to the two ends of the segment. Figure 10 shows this construction applied in the context of air-traffic control.

This type of parallel interaction sheds more light on what is called parallelism in multimodal interfaces. The parallelism is only present at the higher levels of interaction management, when two visual feedbacks have to be maintained at the same time, for instance. At the lowest level, which deals with events, everything is sequential. We will see that this has consequences when events have to be combined.

COMBINED INTERACTION

The last and most complex task for supporting two-handed input is supporting combined interactions. It should be noted that the combination of input data generally occurs at low

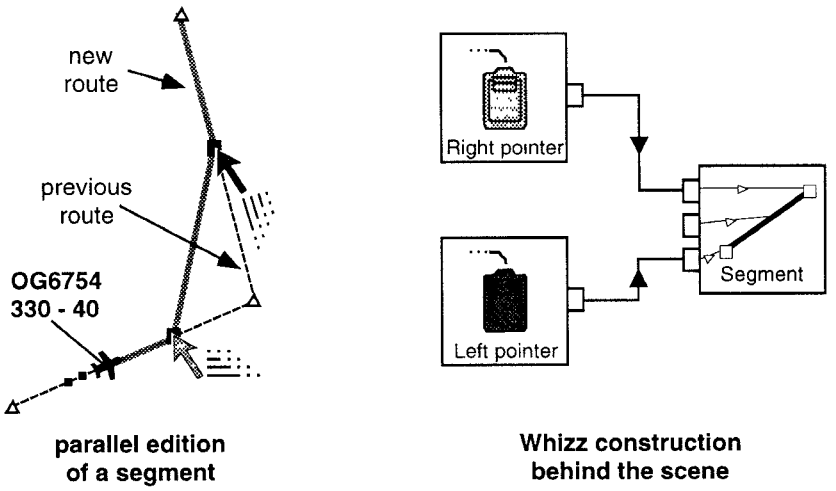

Figure 10: Two actions in parallel on the same object.

level, in contrast with multimodal interactions such as "Put That There". High level fusion may only occur when actions have been given a meaning, ie. when they have been completed. This is incompatible with the fact that two-handed interaction generally involves parallel actions performed by both hands: we do not want the system to wait for the completion of our actions before combining them, at least because we want some feedback on the operation we are performing. For that reason, a toolkit for two-handed interaction must support the combination of input events, and not only the fusion of semantic operations.

Another distinction should be introduced between the combination of status and events, and the combination of events. The former is useful for implementing hold-and-pull interactions, whereas the latter is needed for simultaneous clicks on buttons.

Combining status and events

Let us consider a segment that we want to deform with a hold-and-pull interaction. Only one event is received from the pointer that grabs the segment, and this event (usually a "button down" event) changes the status of the segment, now considered as held. It is not that initial event which is combined with the "mouse move" events from the other pointer to produce the deformation of the objet, but rather the status of the segment. Depending on this status, "mouse move" events will result in the segment being moved or in its being deformed. This is why the combination of events and status is at least as important as the combination of events.

In systems built using Whizz, the status of the interface and its components is stored in modules and in the configuration of the flow graph. For instance, when an animated object follows a circular trajectory, its current position is stored in the module that manages the trajectory; when the user clicks on an icon to drag it, the fact that a drag has started is materialized by an action module and its connection with a slot of the icon. In order to support the combination of status and events, we added a number of facility modules that both store a status and modify the structure of the flow graph according to that status. For instance, we introduced switches, which are modules with one input slot, a control slot, and two output slots. Notes received on the input slot are emitted on one of the output slots; notes received on the control slot change the output slot that will emit the notes received by the input slot. This new module is used in figure 11 to implement hold-and-pull interactions. In that figure, the positions emitted by the right pointer are used to move the segment or deform it, depending on the state of the switch. An extension was added to the module implementing the segment. This extension is a reaction to "button down" and "button up" events from the left pointer. The reaction converts events into notes, which are used to control the switch: when a "button down" event occurs, the switch moves to Position 2, and the segment is ready to be deformed; when a "button up" occurs, the switch moves back to Position 1, and the segment can be moved. This illustrates how the status of an input device can be easily made to control input from another device.

Combining events

Finally, interactions such as simultaneous clicks on buttons need a real combination of events. In the case of two graphical buttons, three high level classes of events may occur: clicks on the first button, clicks on the second one, and simultaneous clicks on the two buttons. Events of the third class are obtained by merging events of the first two classes.

In order to combine events, one needs to introduce a notion of simultaneity of events. This notion is complex to implement: the only solution consists in delaying the handling of events, as it is usually done for multiple clicks. If an event from the other device is received during the delay, we have got two simultaneous events. If nothing happens, the delayed events may be released and handled. We added modules to Whizz for supporting that notion of simultaneity. These modules, called temporal filters, have two input slots and

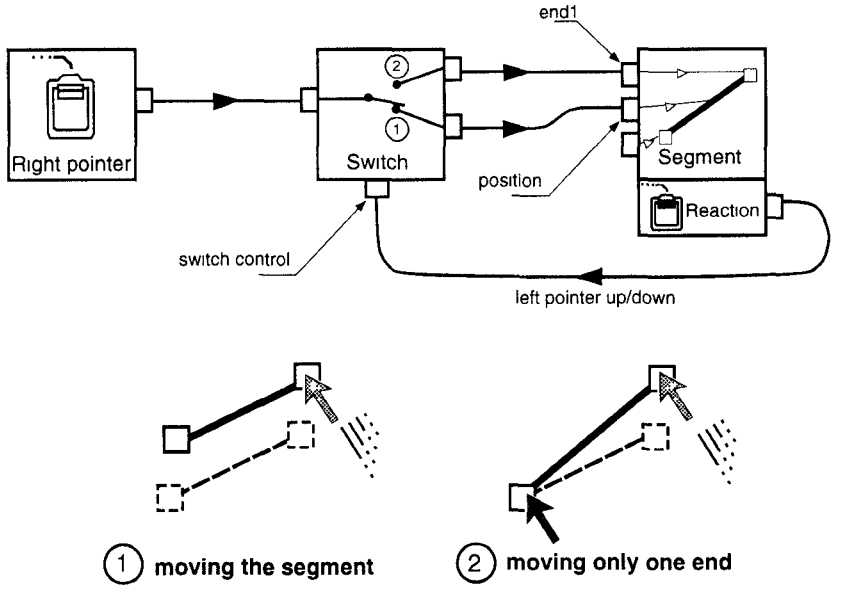

Figure 11: A partial view of the Whizz construction for supporting two-handed interaction on a segment. The right pointer is used to move the whole segment. When the left pointer is clicked on the icon that materializes the end of the segment, the flow is redirected so that the segment is deformed.

three output slots. If two notes are received on the two input slots "at the same time", they are merged and emitted on the central output slot. Notes that were not correlated are emitted separately on the two other output slots. As for multiple clicks, application designers then have the responsibility of hiding the delay that was introduced by the temporal filter. This can be done by using the same technique as for double clicks in desktop interfaces, where the first click of a double click triggers the same action as a simple click, usually the selection of the icon. Figure 12 shows how temporal filters can be used to implement simultaneous clicks: only the notes received simultaneously result in a note emitted towards the on/off module. With this construction, users may quit the application by clicking on the two icons at the same time.

The same temporal filters can be used for synchronizing flows of data. For instance, they can be connected to two action modules: this will allow programmers to decide that an object (probably a heavy one) can be moved only if pulled with both hands at the same time. With such tools, we expect to be able to explore new kinds of combined interactions in the future.

CONCLUSION

In this paper, we identified several two-handed interaction styles, and classified them in terms of the two main characteristics of multimodal interaction: parallelism, and combination. We then exposed the technical issues raised by the implementation of these interaction styles: issues related to independent interaction, then parallel interaction, and finally combined interaction. These issues range from structural problems in graphical toolkits, such as the extensibility of their input mechanism, to the need of new abstractions for describing combined interactions. We explained how our toolkit Whizz solves these problems, or was extended to solve them. With these extensions, Whizz now supports the use of two-handed input, and we strongly believe it is easily extensible to other kinds of multi-sensor input. The extended Whizz was used by the author and another programmer to develop prototypes of two-handed interfaces, and to extend an existing application (a map editor) to two-handed interaction. We are currently using it to develop a two-handed graphics editor, in order to explore new interaction techniques.

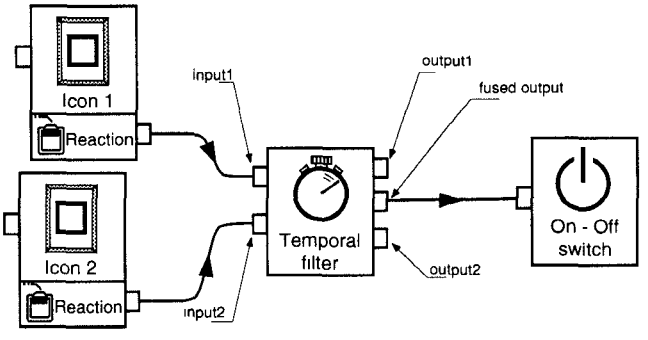

Figure 12: Temporal fusion of events. Clicks on icons are transformed into notes. The temporal filter emits merged notes only if it receives two notes on its two slots in a specified interval of time.

Future directions for that work will include:

- the integration of the new synchronization modules in Whizz'Ed, our experimental visual tool for interface programming, still under development. This should allow the easier exploration of new interaction styles.

- experiments with other modalities such as speech recognition, to determine how Whizz and the abstractions it provides can be applied to other multimodal interactions.
- the evaluation of the interaction styles we proposed for air traffic control. We will first evaluate these styles on simple drawing tasks, then integrate them in realistic environment to test their impact on the work of controllers.

ACKNOWLEDGEMENTS

Michelle Jacomi helped to implement and test the features described in this article. Bo Overgaauw and Forrest Colliver helped with English. The author also wishes to thank Michel Beaudouin-Lafon and Thomas Baudel (LRI, University of Paris), Philippe Palanque (LIS, University of Toulouse), and the anonymous reviewers for their useful comments on this paper.

REFERENCES

1. M. Beaudouin-Lafon, Y. Berteaud, and S. Chatty. Creating direct manipulation interfaces with X_{TV}. In *Proceedings of EX'90, London*, pages 148–155, 1990.
2. M. Beaudouin-Lafon and M. Thiellement. A tour through AVIS. *ACM SIGCHI Bulletin*, 23(4), Oct. 1991.
3. E. Bier and S. Freeman. MMM: a user interface architecture for shared editors on a single screen. In *Proceedings of the ACM UIST*, pages 79–86. Addison-Wesley, 1991.
4. E. Bier, M. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and magic lenses : the see-through interface. In *Proceedings of the ACM SIGGRAPH*, pages 73–80. Addison-Wesley, 1993.
5. M. M. Blattner and R. B. Dannenberg. *Multimedia interface design*. ACM Press, 1992.
6. R. A. Bolt. Put-That-There: voice and gesture at the graphics interfaces. *ACM Computer Graphics*, 14(3):262–270, 1980.
7. W. Buxton and B. Myers. A study in two-handed input. In *Proceedings of the ACM CHI*, pages 321–326. Addison-Wesley, 1986.
8. S. Chatty. Defining the behaviour of animated interfaces. In *Proceedings of the IFIP WG 2.7 working conference*, pages 95–109. North-Holland, Aug. 1992.
9. P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the ACM Conference on Lisp and Functional Languages*, 1984.
10. A. Gourdol, L. Nigay, D. Salber, and J. Coutaz. Two case studies of software architecture for multimodal interactive systems: VoicePaint and voice-enabled graphical notebook. In *Proceedings of the IFIP WG 2.7 working conference*, pages 271–284. North-Holland, Aug. 1992.
11. R. D. Hill. Supporting concurrency, communication and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Transactions on Graphics*, 5(2):179–210, Apr. 1986.
12. Groupe multimodalité. In *Actes d'IHM'91 - Troisièmes journées sur l'ingéniérie des interfaces homme-machine*. 1991.
13. P. Kabbash, W. Buxton, and A. Sellen. Two-handed input in a compound task. In *Proceedings of the ACM CHI*, pages 417–423. Addison-Wesley, 1994.
14. G. Kurtenbach and T. Baudel. Hypermarks: issuing commands by drawing marks in Hypercard. In *CHI'92 Posters and Short Talks*, page 64, 1992.
15. M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, Feb. 1989.
16. B. A. Myers. A new model for handling input. *ACM Transactions on Office Information Systems*, pages 289–320, July 1990.
17. L. Nigay and J. Coutaz. A design space for multimodal systems: concurrent processing and data fusion. In *Proceedings of the ACM CHI*, pages 172–178. Addison-Wesley, 1993.
18. D. A. Norman. *The Design of Everyday Things*. Doubleday, New York, 1990.
19. D. H. Rubine. *The automatic recognition of gestures*. PhD thesis, Carnegie Mellon University, 1991.
20. D. Weimer and S. Ganapathy. *Interaction techniques using hand tracking and speech recognition*, pages 109–126. In Multimedia Interface Design [5], 1992.