Multiple input support in a model-based interaction framework

Stéphane Chatty, Alexandre Lemort, Stéphane Valès

Multiple input support in the IntuiKit framework

Stéphane Chatty[1,2] Alexandre Lemort[2] Stéphane Valès[2]

[1]ENAC
7 avenue Edouard Belin
31055 Toulouse Cedex, France

[2]IntuiLab
Les Triades A, rue Galilée
31672 Labège Cedex, France

{surname}@intuilab.com

Abstract

IntuiKit is a programming framework aimed at making the design and development of post-WIMP user interfaces more accessible. Developing for tabletops puts special requirements on such a framework: managing parallel input, device discovery, device equivalence, and the description of combined interactions. We describe how the model-based architecture of IntuiKit supports these, either through features that are native to IntuiKit, or through extensions for managing multiple input. We illustrate these features through examples developed in several tabletop projects, including one application aimed at improving collaboration between air traffic controllers.

Keywords: interaction framework, tabletop, multiple input, model-driven architecture, event model, data-flow

1 Introduction

Developing user interfaces for tabletops is both very similar to desktop or touchscreen user interface development, and very different from it. The similarities are obvious: apart from orientation issues the graphics are the same, and there is no reason that interacting with a single finger on a tabletop should be different from interacting on a touchscreen. Therefore, reusing interactive components developed for single input (mouse, touch screen) is a legitimate wish. But there are differences, and the most obvious are not always the most important. The most obvious is the ability to handle multiple flows of input, though it is not much different from handling animation in parallel with interaction. More unusual is the ability to handle a variable number of contact points, or the implied hierarchical structure of devices that have both multi-touch and multi-user detection. Then the connection between the graphics system and the input system (often called "picking") must be made extensible. The differences between devices must also be accounted for: some do user identification but lack correlation between X and Y coordinates of touch points, others are limited to two pointers, etc. Furthermore, devices are still costly and developers often need to simulate them with mice because they cannot get easy access for unit tests. Therefore, a framework for adapting, filtering and simulating devices must be provided. Finally, support must be provided for programmers to design interaction styles that combine several input flows.

IntuiKit is a framework for prototyping and developing post-WIMP user interfaces (that is user interfaces that do not rely on the Windows-Icon-Mouse-Pointing paradigm) as well as multimodal user interfaces. The purpose of IntuiKit is not only to make the programming of such interfaces feasible, but to make their design, development and deployment cost-effective enough for non specialised industries to afford it. In particular, in order to allow the reuse of design elements when switching from prototyping to final development, IntuiKit relies on a model-driven architecture: as much as possible of the user interface is made of data obtained by instantiating models [6]. The resulting interface models can be "rendered" by different implementations of the framework: one aimed at prototyping (which currently offers a Perl programming interface), and the other aimed at industrial deployment (which currently offers a C++ interface). The modelling concerns graphical objects, but also elements of architecture, behaviour and control, animation trajectories, speech grammars, etc. Adding multiple input support to IntuiKit therefore consists in enriching the core models used in the framework, and adding new models that represent new concepts if necessary. This article describes these models. We first describe the basic concepts of IntuiKit. We then describe how IntuiKit supports device addressing, input configuration and parallel interaction. We finally describe a full application implemented using those mechanisms, before reviewing related works.

2 IntuiKit basics

Figure 1. A set of tabs for a car display

2.1 The IntuiKit tree

IntuiKit considers an interactive application as a tree of elements. The management of the tree, its traversals and the communication between elements are managed by the IntuiKit core. Terminal elements are provided by IntuiKit modules, that are in charge of rendering them or updating their rendering upon modifications. For instance, the GUI module manages graphical elements from the SVG standard; the Speech module manages elements that represent grammar rules; the Animation module manages trajectories and collision detection; the Base module manages elements that help describe the behaviour of the interface, such as clocks and finite state machines. More complex elements named components can be built by assembling other elements. For instance, a simple button component can be obtained by combining a finite state machine (FSM), a special element that only renders one of its subtrees depending on the state of the FSM, and a series of rectangles and texts that represent the different states of the button. Programmers can also build their own components in code, for instance to create functional core objects that can communicate with the rest of the application using the architecture patterns supported by the IntuiKit core. Parts of a tree that constitute an application can be loaded at run time from XML files, thus allowing for skinning or customisation. For instance, the tabs from an in-car comfort and navigation system in Figure 1 are obtained by loading a given SVG file into the tree presented in Figure 2, and Figure 3 is obtained by loading another SVG file. The same type of customisation based on element loading can be used to adapt to touchscreens an application made for desktops: for a given button, one can choose to load either a FSM that implements a “take-off” or one that implements a “land-on” strategy [16].

2.2 Event model

Because user interface programmers deal with interaction and behaviour, and not only rendering, a very central

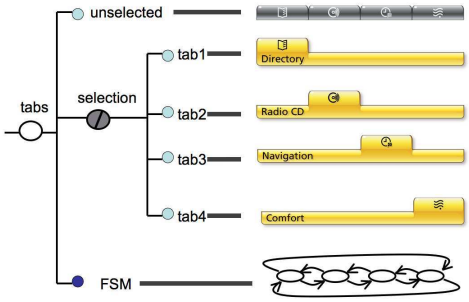

Figure 2. The tree corresponding to the tabs

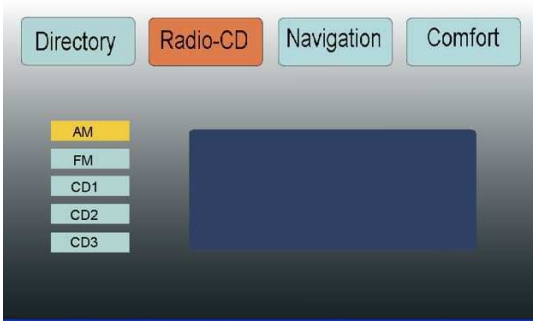

Figure 3. The same tree with another SVG file

feature of IntuiKit is its communication between elements. It has since long been established that event-based communication and control is well adapted to managing user input. Event communication is a pattern of control flow alternative to function call, that can be implemented on top of classical function calls. This is used by most interaction frameworks to juxtapose event communication and function calls in applications. However, a characteristic of user interaction, and especially post-WIMP interaction, is that the same actions can be triggered from very different contexts: user input, animation, or computations from the functional core. Being able to reuse components thus requires that one provide a unique model of communication for all these contexts: fragments of code that use different communication mechanisms cannot be connected easily. Furthermore, because a user interface has a parallel execution semantics [5], the overall semantics may become unclear if several control transfer mechanisms are used. That is why IntuiKit uses event communication as the only mechanism for transferring control from components to others: in the same way as functional programming splits all code into functions that communicate through function calls, IntuiKit splits all code into elements that communicate through events.

IntuiKit uses the concept of event source. Each event source has two features: its event specification format, and its event contents. For instance, the `'clock'` source has a specification format that distinguishes between one-shot

timeouts and periodic timers, and accepts one argument in the first case and two in the second case. The event contents is a timestamp, with an additional repetition number in the second case. Some event sources like the clock are global to the application and brought by IntuiKit modules. Elements are also sources; this includes graphical objects or FSMs, but also any component built by programmers. In the latter case, the event specification format and the event contents depend on the structure of the component, and can be customised by the programmer. For instance, one can build a dialog box and make it emit empty "OK" and "Cancel" events, or a wallclock component that emits "SECOND", "MINUTE" and "HOUR" events that contain the time of the day. Finally, element properties, such as the width of a rectangle, are also event sources. This serves as the basis for data-flow communication which is therefore a special case of event communication.

Event subscription is obtained by adding special elements to the IntuiKit tree, which all use the same pattern: a reference to an event source, an event specification, and an action. The simplest element is the Binding, which links an event specification to an action. FSMs add state to this: their transitions are Bindings that are only active in given states. A Connector has an implicit event specification and action: its event source is an element property, whose value is used to update another element property when it changes. A Watcher is used inside components; it accepts several properties as event sources, and triggers an action when one or more of them changes. Watchers are used to implement rendering; Watchers and Connectors are used to produce data-flow communication: data-flow filters modify their output properties in the actions triggered by their watchers. Figure 4 shows a FSM used to control a stick-button.

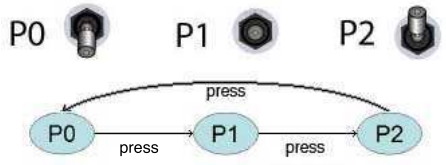

Figure 4. Behaviour of a 3-state button

3 Managing multiple devices

In most user interfaces, input devices can remain implicit in the code. In WIMP user interfaces, for instance, programmers usually subscribe to event types that are related to the mouse or others that are related to the keyboard: no need to introduce any further reference to the keyboard or the mouse. If one is missing, the programs does not run ; if there are other devices, either they can safely be ignored or the operating system manages to mix them so as to emulate one mouse and one keyboard. On a tabletop, multiplicity is the rule and one wants to distinguish among devices: "I want to subscribe to touches from this user", or "I want to subscribe to touches from any user, then to moves from the user that has just touched the table", etc. The event model of IntuiKit offers a framework for this: one just needs to have different event sources representing the different input among which one wants to select, or a source with an appropriate event specification format. By adapting the sources and event specification used in a given component, one can easily adapt it from single-touch to multi-touch interaction. However, some questions still remain: how to reference a given input source? how to check if a given device is present? how to handle the fact that the number of input flows, for instance the number of users or the number of fingers, is not known in advance? how to manage complex sources such as a multi-user multi-pointer table?

3.1 Devices as external elements

The model chosen as the basis for answering the above questions is partially an extension of the semantics of the IntuiKit tree. It consists of stating that input devices are IntuiKit elements, but elements that are located *outside* of the application tree. By stating that devices are elements, we go a long way towards allowing the programmer to reference them and towards handling new input flows. By stating that they are outside of the application tree, we give account of the fact that the programmer does not control their existence: they are context elements that may or may not be there, and it is up to the programmer to query them. This model actually applies to many resources that are external to applications: operating system resources, other applications, context capture, etc. Users interact with a system represented by a large tree, part of which represents the application being developed. This is consistent with, for instance, the way the Unix file system is extended in the Linux kernel to make various information about the kernel or programs accessible to other programs.

3.2 Referencing and finding devices

Since devices are elements, they are event sources and can be used in event subscriptions. This can be done by using an extension to the reference system previously available in the IntuiKit tree. The reference system, in a similar way to the source/specification couples used for events, uses namespaces and names. Precedently, all namespaces referenced elements in the application tree. The `'input'` namespace is used to reference devices. For instance, one can use

```
new Binding(-source => 'input:diamondtouch', ...);
```

in Perl or

```
<binding source="input:diamondtouch" ... />
```

in XML to subscribe to a DiamondTouch device plugged on the computer. Device names are dependent on the configuration of the computer. Therefore one may wish to use an IntuiKit property instead of a literal for the device name so as to set the device name in a configuration file, in CSS format for instance.

The above referencing scheme leads to an error if the device requested is not present. If a programmer wishes to handle this situation, it is possible to use references from the programming language as an intermediate. In Perl, for instance:

```
$d = get Element(-uri => 'input:diamondtouch', ...);
if (defined $d) {
  new Binding(-source => $d, ...);
} else {
  ...
}
```

It is possible to express typing constraints on the device:

```
$d = get DiamondTouch(-uri => 'input:diamondtouch', ...);
```

To express richer constraints, such as "any pointing device", it is planned to add new namespaces that handle more complex queries than a name: XPath-like or SQL-like requests for instance.

3.3 Dynamicity

A feature of multi-touch or multi-user systems is the dynamicity of input. The number of touch points or pointers usually starts at zero when starting the system. It then increases as soon as a new user starts interacting, a new finger lands on the surface or a new pointer is used. When this happens, the programmer usually wishes a new feedback and possibly new behaviours to be added to the user interface. This situation is very similar to the addition of a new device, for instance a new mouse, to the computer. Actually, plugging mice successively onto the computer is a simple way of simulating a tabletop during unit tests. The dynamicity of users and touch points is very similar to hot-plugging.

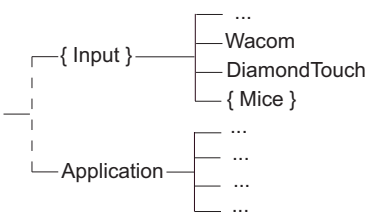

Figure 5. The set of input devices in the tree

In conformance with the choice to make event communication the core mechanism of control in IntuiKit, the dynamicity of input devices is handled through events. IntuiKit has *Set* elements, that contain other elements; adding or removing an element from a set triggers an event. By subscribing to a Set as an event source, one can associate an action to the addition or the removal. The set of all input devices is accessible through the global name `'input'`; subscribing to it allows to be notified of all new devices connected to the computer:

```
new Binding(-source => 'input', -spec => 'add', ...);
```

3.4 Hierarchical devices

So far we have been indiscriminate about the exact nature of input devices, referring to multiple mice in the same way as multiple pointers on a tabletop. However, there are differences. When using mice, every of them is an input device of its own. When using a Wacom Intuos 2 tablet, which is able to handle two styluses, the tablet itself is the device: plugging it on the computer is a legitimate hot-plugging event, and one may wish to subscribe to all events from the tablet. Nevertheless, one would still be interested in being notified when a new stylus is used on the tablet, and to subscribe to events from this stylus. The same holds for a DiamondTouch and its four users; it gets even more complex when one uses a tracking algorithm to distinguish among multiple pointer from a same user on the DiamondTouch.

The IntuiKit tree provides the appropriate framework for describing this. By considering devices as IntuiKit elements that can contain other elements, one can build a model of devices that takes into account all of the above situations: devices are either atomic or made of subdevices, like interaction components can be made of subcomponents. For instance, a mouse is made of a locator and several buttons. A Wacom Intuos 2 contains a set of pointers. A DiamondTouch contains a set of users, which each possibly contain a set of pointers. One can subscribe to the apparition of a new pointer by subscribing to the set of all devices in the first case, or the set of pointers contained in the device in the two other cases.

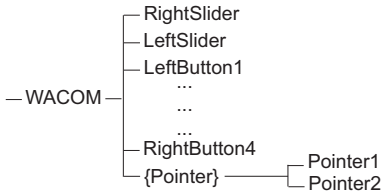

Figure 6. Internal structure of a Wacom device

This hierarchical view of devices not only matches the underlying model of IntuiKit where all interactive components can be made of sub-components. It also matches theoretical models of input devices [3] and provides a useful basis for dealing with device equivalence and combination when detecting and adapting the input configuration for reusing components or applications.

4 Input configuration

The gist of software reuse is the ability to reuse a piece of code or a component in contexts that did not need to be explicit at the time of writing it. For instance, a sorting function can be used to sort all types of lists. This usually requires a way to impose constraints on the context of reuse. A type system is often used for that purpose in programming languages. This also requires that the reusing programmer be be proposed a way to adapt the original code to a context that is not exactly compatible. Functional languages make it easy to create wrapper functions or to pass functions as arguments to the original code, for instance.

With tabletops, the central reuse issue is currently that of device structure and specification. There is no standard for devices as there has been, explicitely or not, for mice and keyboards. Whatever its actual protocol, a basic mouse always consists of a locator and one or more two-state buttons. A tabletop device can distinguish users or not, detect proximity or not, provide X and Y histograms of the contact surface or contours of all contact points, etc. Furthermore, these devices are still expensive and programmers often have to simulate them with mice when developing applications. And finally, there is no reason why an interactive component developed for a touch screen or a desktop computer could not be reused on a tabletop. For all these reasons, the ability to detect and adapt the input configuration is important for reusing applications or components. Concepts such as equivalence, adaptation, and complementarity [15] play important roles in this.

4.1 Device inspection and encapsulation

Detecting the input configuration available to an application on a computer starts with being able to locate the devices plugged onto the computer. We have seen earlier how this can be done and how typing constraints can be applied. But we have seen that there is no standard on how tabletops are structured. In addition, several devices from the same brand may have different features. For instance, a Wacom Intuos 2 tablet accepts two styluses at a time, whereas a Wacom Cintiq has a similar protocol and can be managed with the same driver but can only handle one stylus. Therefore, in order to use a given device to an application, one needs to access its structure to inspect it or to connect to it.

The component encapsulation mechanism in IntuiKit provides support for this. The programmer of a component can choose what properties and children of the component are visible to the outside, and under what name. Programmers who use an element can list its exported properties, children and events. It is also possible to directly access them: events by directly subscribing, and properties and children as follows:

```
$set = $dev->get (-child => 'pointers');
$x = $pointer->get (-property => 'x');
```

This allows application programmers to check that the element they are using is as expected, and to access its parts so as to subscribe from them or connect to them. It is also planned for the future to check that the sequence of events that an element may emit (and declared as a regular language for instance) is compatible with the input language of a component that subscribes to it, as in [1]. This would provide an equivalent of type checking for the event-driven architecture of IntuiKit, and would help detect if a given device can be used with a given application.

Accessing the internal structure of a device, when it exports it, also allows to perform source selection. On a Wacom Intuos 2 or a DiamondTouch, if one is interested in a given stylus or user one can directly subscribe to events from this stylus or user rather than using a complex event specification or subscribing to all events and performing a test afterward. Finally, one can use component encapsulation to masquerade a device as another type of device. For instance, given a DiamondTouch device, one can create a component that contains it as the only child, and that exports only the events and properties from the child of the DiamondTouch that represent one user. The result is a component that emulates a touch screen.

4.2 Source filtering

Encapsulation works when the equivalence between devices can be obtained by simple renamings and information hiding. However, most often the situation is more complex: to make a given device compatible with another, one often needs to combine event attributes, to memorise information from one event to the next, or to perform computations. For instance, the locator in a mouse is a relative device. To make the mouse compatible with a component that expects events from a touch screen, one needs to apply an acceleration law to translations of the mouse, to store an absolute position that is incremented at each move, to apply a scaling function, and to crop the result so as to stay within the bounding box of the screen. To simulate a multi-user touchscreen with a DiamondTouch, one needs to compute the barycentres of the X and Y histograms it provides for each user. One even sometimes needs to combine several devices to simulate another, for instance four mice to simulate a DiamondTouch.

The architecture proposed for this in IntuiKit is *source filtering*. It uses the fact that all IntuiKit elements can be event sources to have an organisation similar to data-flow connections: an element stands as a front end for another element and acts as a relay for event subscriptions. Filtering can be combined with encapsulation: a filter can be a component that encapsulates the original source, thus hiding it from the rest of the application. In all cases, the filter element has its own event specification format, which allows it to give its own version of what types of events are

available. For instance, a filter for a DiamondTouch may provide the subscription language of a mouse if it applies the barycentre computation mentioned above. Some filters just relay event subscriptions and do not perform any operation on events: they just provide another way of subscribing. But most filters apply transformation to events: renaming of event attributes, or computations to create new attributes. Those transformations are performed in an action similar to the Watcher's action in a data-flow element.

Filtering is the mechanism used in IntuiKit to account for gesture recognition: the positions provided in events by pointer devices are memorised in the filter, then a classification algorithm is applied when the button or pressure is released. Filtering also accounts for picking, the operation in which each pointer event is associated to a graphical object based on geometrical considerations, and which allows to subscribe to pointer events by addressing graphical objects themselves. Having this mechanism explicit is important in multi-touch systems: it allows all pointers to be associated with the graphical scene, by connecting them to the picking filter as soon as they are detected. Filtering is also used to perform input configuration. As already mentioned, a DiamondTouch can be filtered by applying a barycentre computation to its events so as to simulate the behaviour of a mouse or touch screen. It can also be filtered by using a tracking algorithm that extracts multiple contact points from the data provided by the DiamondTouch, thus providing into a multi-user multi-touch source.

4.3 Remote devices

Interaction designers sometimes wish to access input devices through inter-process communications. This may be because the device is actually remote, for instance when experimenting with two remote tables coupled with a video-conferencing system [7]. This may be because there is no driver available yet for a given device on the target platform. Or this may be because a given application has been built without support for changing the configuration, and the configuration needs to be adapted from the outside. For these situations, IntuiKit's ability to transport events over message buses such as the Ivy software bus [2] or Open Sound Control [18] is appropriate. One can build, with IntuiKit or with their tool of choice, a software agent that implements IntuiKit's text protocol for input devices. The software agent can represent the actual device or use source filtering to simulate the device expected by the application.

5 Interacting in parallel

The device addressing and input configuration mechanisms in IntuiKit provide programmers with a flexible way of accessing and managing multiple input devices. However, the programmer's main goal is to implement interaction styles and not only access devices. What makes multiple input special regarding interaction is obviously the ability to have several interactions in parallel, and sometimes to combine several input flows to produced synergistic interaction. As IntuiKit is aimed at multimodal interaction, we will see how these possibilities are built in its core mechanisms.

The first requirement for interacting in parallel is that the programming framework provide a way of adding new input sources that can emit events asynchronously. Most modern frameworks or toolkits have this feature, either through a multi-threading system or through an extensible "main loop". IntuiKit has an extensible main loop that can handle in an asynchronous way any input source managed by the underlying operating system.

Then the software architecture promoted by the framework must allow programmers to build interactors that can be manipulated in parallel. The event communication model is intrisically parallel, but any incitation to store interaction state globally, for instance in global variables, is fatal. It is even worse if the framework itself proposes interactors that do store state globally. The architecture proposed by IntuiKit, based on a tree of components that contain properties and communicate through events, is a strong incitation to building programs as a collection of components that store their state locally and behave as parallel agents. When dialogue control requires that some state is shared by several components, such as the state of a dialogue box, the hierarchical organisation of the IntuiKit tree incites to manage this by storing the state in a parent common to all components concerned; here, for instance, the dialogue box would managed the "global" state and its children (buttons for instance) would manage their local states and communicate with it through events that change its "global" state. Elements such as IntuiKit's FSMs provide help for creating such locally managed behaviours, at different levels in the tree if need be. By cloning a component one clones all its subcomponents and properties, and thus several identical components will be able to work in parallel.

Finally, one sometimes needs to merge input flows: zooming with two fingers, button that is only triggered when two users press it together, etc. The two main ways of describing interactive behaviours in IntuiKit provide for such combined behaviours. On the one hand, data flows can come from two origins and end up connected to different properties of the same element, like in the Whizz or ICon toolkits [4, 9]. This allows for instance to control one angle of a rectangle with one finger and the opposed angle with another finger. On the other hand, the transitions of finite state machines can be labelled with any event specification from any event source. This allows to build FSMs that rely on events from different pointers, for instance a three-state button that is fully pressed only when two fingers have been pressed on it.

6 Example application

IntuiKit is used in several research or commercial projects using tabletops. We describe here the Multi-Actor Man Machine Interface (MAMMI) project carried out for Eurocontrol in the domain of air traffic control. The project aims at designing a large horizontal surface where two or more controllers could share tools and data, exchange information (figure 7). The sought benefit is for them to be able to adjust their repartition of tasks in real-time so as to adapt to situations and improve their overall performance.

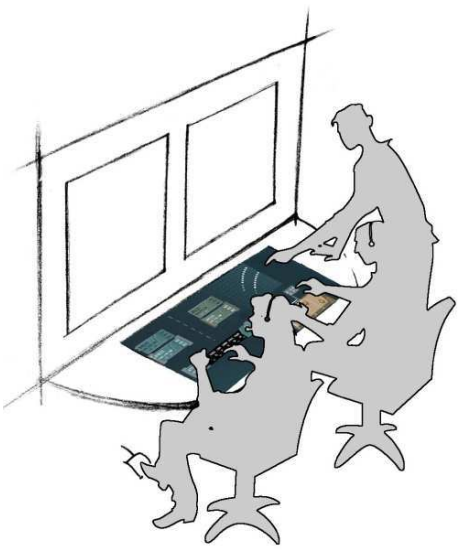

Figure 7. A tabletop for air traffic control

Two hardware devices have been investigated: the DiamondTouch and a prototype built by a European company. Both have input resolutions that enable to test realistic solutions, and can be plugged to display devices that support rich graphics. Each has advantages and drawbacks regarding interaction. The DiamondTouch distinguishes between up to four users. The other device detects an indefinite number of inputs represented by sets of points.

The project team was composed of a graphic designer, UI designers and hardware experts. The model-based architecture of IntuiKit allowed us to applied iterative and concurrent design. First, participatory design led to producing paper prototypes that explored design variations based on the features of each device: with and without user identification, with and without multi-touch for each user. The paper prototypes then served as a reference to split the interface into a tree of components that represent the software architecture of the application as well as its graphical structure. The tree then served as a contract between all project actors, especially between UI designers and the graphic designer. From then on, team members started to work independently. The hardware expert exploited IntuiKit's ability to handle devices as remote sources: he produced a software agent for handling the prototype device remotely, because no driver was available on the target platform. The graphic designer produced graphical elements using a professional drawing tool (Figure 8).

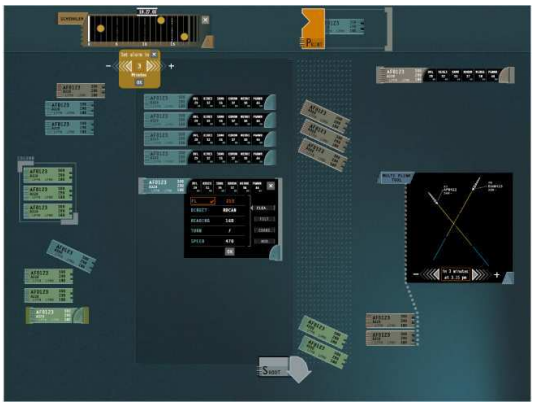

Figure 8. The designer's work

Meanwhile, the UI designers programmed the interaction. For each component, they defined the behaviours and connection to the functional core. Because they used mice for testing, they used IntuiKit's device encapsulation mechanism to emulate the target devices with multiple mice. Using device inspection, their were able to test the two design variants: the application checks if user identification is available and adapts the way it handles conflicts. If the device does not identify users, the application solely relies on existing social protocols to prevent and resolve conflicts. With user identification, the application activates software

Figure 9. Different feedback for each user

coordination policies, such as access permission or explicit sharing, and provides different feedback for each user (Figure 9) to improve mutual awareness. They also used a gesture recognition filter to implement gestures in the interface.

Finally, after a period of graphic design, test and integration, the final application was operational and able to run with multiple mice, a DiamondTouch, and the prototype device. Switching devices is just a matter of changing a reference in a configuration file.

7 Related work

Interaction libraries such as DiamondSpin [17], or the Grouplab DiamondTouch Toolkit [8] have been developed

for tabletop interaction, but they are focused on one particular hardware or on graphical features such as rotation.

The Input Extension to the X Window Server protocol [10] provides inspection of devices, but no support for dynamicity, input configuration and interaction description. More recently, the TUIO protocol [13] addressed tabletops and tangible user interfaces, with a low level of asbtraction. Multiple input has been addressed in the Whizz [4] and ICon [9] toolkits, using the data-flow paradigm to support interaction description. ICon addresses the issues of input configuration, by allowing users to edit the data-flow graph. However, these toolkits do not address the dynamicity of devices and the data-flow paradigm alone sometimes makes state-based interaction complex to handle.

Concurrency in user interaction tools has been studied as early as in Sassafras [11]. The architecture model in IntuiKit also has some similarities with interactors in Garnet [14] or with VRED [12]. Its originality lies in the model-driven architecture, the unifying tree structure, and the unification of events and data-flow.

8 Conclusion

In this article, we have described how the IntuiKit environment supports device adressing and detection, input configuration and the description of parallel interaction in tabletop systems. Beyond the mere ability to produce a given type of interaction, what makes an interaction style available to the large public of users is the ability to manage this style according to basic software engineering criteria: reusability, encapsulation, orthogonality. The IntuiKit model was extended to support tabletops without having to introduce additional concepts. This strongly suggests that the properties already demonstrated by IntuiKit in terms of reusability, customisation and concurrent engineering are transferred to application development for tabletops.

Acknowlegements

This work was supported by the French *Agence Nationale de la Recherche* through project DigiTable, and by *Eurocontrol* through project Mammi. The event source model was designed with Pierre Dragicevic and David Thevenin. The tracking algorithm mentioned in section 4 was designed by François Bérard at LGI/IIHM and has not yet been published. Carole Dupré and Sébastien Meunier have helped with examples.

References

[1] J. Accot, S. Chatty, S. Maury, and P. Palanque. Formal transducers: models of devices and building bricks for the design of highly interactive systems. In *Proc. of the 4th Eurographics DSVIS workshop (DSVIS'97)*. Springer-Verlag, 1997.

[2] M. Buisson, A. Bustico, S. Chatty, F.-R. Colin, Y. Jestin, S. Maury, C. Mertz, and P. Truillet. Ivy: un bus logiciel au service du développement de prototypes de systèmes interactifs. In *Proc. of IHM'02*, pp. 223–226. ACM Press, 2002.

[3] S. Card, J. Mackinlay, and G. Robertson. A morphological analysis of the design space of input devices. *ACM Trans. on Office Information Systems*, 9(2):99–122, 1991.

[4] S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST*, pages 195–204. Addison-Wesley, Nov. 1994.

[5] S. Chatty. Programs = data + algorithms + architecture. Consequences for interactive software. In *Proceedings of the 2007 joint conference on Engineering Interactive Software*. Springer-Verlag, Mar. 2007.

[6] S. Chatty, S. Sire, J. Vinot, P. Lecoanet, C. Mertz, and A. Lemort. Revisiting visual interface programming: Creating GUI tools for designers and programmers. In *Proceedings of the ACM UIST*. Addison-Wesley, Oct. 2004.

[7] F. Coldefy and S. Louis-dit-Picard. Digitable: an interactive multiuser table for collocated and remote collaboration enabling remote gesture visualization. In *Proceedings of the 4th IEEE workshop on projector-camera systems*, 2007.

[8] R. Diaz-Marino, E. Tse, and S. Greenberg. The grouplab diamondtouch toolkit. In *Video Proceedings of the ACM CSCW 2004 conference*, 2004.

[9] P. Dragicevic and J.-D. Fekete. Support for input adaptability in the icon toolkit. In *Proceedings of the Sixth International Conference on Multimodal Interfaces (ICMI'04)*, pages 212–219. ACM Press, 2004.

[10] P. Ferguson. The X11 Input extension: Reference pages. *The X Resource*, 4(1):195–270, 1992.

[11] R. Hill. Supporting concurrency, communication and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Trans. on Graphics*, 5(2):179–210, 1986.

[12] R. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Trans. on Computer-Human Interaction*, 6(1):1–46, 1999.

[13] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A protocol for table-top tangible user interfaces. In *Proceedings of Gesture Workshop 2005*.

[14] B. A. Myers. A new model for handling input. *ACM Trans. on Office Information Systems*, pages 289–320, July 1990.

[15] L. Nigay and J. Coutaz. Multifeature systems: The CARE properties and their impact on software design intelligence and multimodality. In J. Lee, editor, *Multimedia Interfaces: Research and Applications*, chapter 9. AAAI Press, 1997.

[16] X. Ren and S. Moriya. Efficient strategies for selecting small targets on pen-based systems: an evaluation experiment for selection strategies and strategy classifications. In *Proceedings of the EHCI conference*, IFIP Transactions series, pages 19–37. Kluwer Academic Publishers, 1998.

[17] C. Shen, F. D. Vernier, C. Forlines, and M. Ringel. Diamondspin: an extensible toolkit for around-the-table interaction. In *Proceedings of the CHI'04 conference*, pages 167–174. ACM Press, 2004.

[18] M. Wright, A. Freed, and A. Momeni. OpenSound Control: State of the art 2003. In *Proceedings of the NIME-03 conference*, 2003.