Modélisation des entrées multiples dans les logiciels interactifs

Stéphane Chatty

ENAC
7 avenue Edouard Belin
31055 Toulouse Cedex, France
chatty@enac.fr

RESUME

Le développement pour surfaces interactives crée de nouvelles exigences sur les environnements de programmation : gérer des entrées en parallèle, la découverte des périphériques, l'équivalence entre périphériques, et les interactions combinées. Nous analysons ces sujets et décrivons les solutions que nous proposons dans le cadre de la modélisation des logiciels utilisée dans l'environnement IntuiKit, qui repose sur la représentation d'un logiciel interactif sous forme d'un arbre de composants, et sur la représentation du flux d'exécution comme une série de réponses à des abonnements à des événements.

INTRODUCTION

Après les jeux vidéo, les surfaces interactives seront probablement la première catégorie d'interfaces graphiques post-WIMP à toucher le grand public. Ou tout au moins ce sera la première catégorie d'interfaces qui ne reposent pas exclusivement sur le couple clavier-souris: elles restent suffisamment proches des interfaces WIMP pour qu'on souhaite utiliser les mêmes outils de développement et utiliser les mêmes composants interactifs sur surface interactive et sur écran d'ordinateur. Pour faire cela toutefois, il faut permettre aux programmeurs de gérer des configuration de périphériques de pointage plus complexes qu'une simple souris : comment permettre que plusieurs instances d'un même menu soient manipulés en parallèle par plusieurs utilisateurs, comment exiger que deux périphériques de pointage au moins soient disponibles ? En cela, les surfaces interactives constituent une première étape vers la gestion de l'informatique ubiquitaire où l'environnement matériel d'exécution d'un programme sera inconnu à l'avance, et où il faudra le gérer explicitement dans le programme.

Nous avons identifié plusieurs sujets à traiter dans un environnement de programmation pour réaliser des composants réutilisables entre surfaces interactives et ordinateurs de bureau :

- la gestion de plusieurs flux d'interaction en parallèle ;
- la gestion d'un nombre variable de pointeurs, y compris dans le mécanisme de sélection d'objets graphiques, le "picking" ;
- la gestion de périphériques composites, l'équivalence, la différence, la simulation et la composition entre périphériques ;
- la description d'interactions combinant plusieurs flux.

De plus, ces sujets doivent être traités en permettant aux programmeurs de conserver de bonnes propriétés d'architecture à leurs programmes, pour réutiliser des composants et les spécialiser, pour changer l'apparence visuelle de leur application, etc.

Dans cet article nous décrivons comment nous avons étendu l'environnement de programmation d'IHM à base de modèles IntuiKit pour traiter ces sujets. Nous montrons comment la modélisation des logiciels interactifs utilisée dans IntuiKit, qui repose sur la notion d'arbre de composants pour décrire la structure des programmes et des données et sur l'abonnement aux événements et les flots de données pour décrire le flux d'exécution, se prête naturellement à ces extensions.

Gestion des périphériques multiples

Dans les environnements de programmation d'IHM classiques, les périphériques d'interaction sont implicites. Il existe une souris et un clavier, et il n'est nul besoin d'y faire référence. On s'abonne donc aux mouvements et aux pressions de touches en sachant à quoi s'attendre. Mais s'il y a plusieurs pointeurs, comment choisir celui qu'on désire ? comment vérifier d'abord qu'il est présent ? et comment gérer le fait que le périphérique "souris" est un pointeur, et que le périphérique "DiamondTouch" représente de multiples pointeurs qui apparaissent et disparaissent ? Il faut pouvoir exprimer des abonnements plus précis que "les mouvements", désigner les périphériques, gérer leurs apparitions et disparitions, et gérer leur structure complexe.

Langage d'abonnement

La modélisation de l'abonnement aux événements dans IntuiKit offre une solution au premier problème. Un abonnement à un type d'événements est décrit par un couple (source, spécification). La source est un objet auquel est associé un langage qui permet de décrire un ensem-

ble d'événements possibles. Par exemple, une souris possède un langage qui permet de désigner les événements de mouvement et de pression de bouton sur cette souris. La spécification est une expression dans le langage de la source. Un programmeur peut ainsi s'abonner auprès d'un objet donné ; il peut aussi introduire de nouvelles sources, chacune avec son propre langage d'abonnement. Ce modèle, principalement prévu pour que les programmeurs gèrent l'émission d'événements par leurs composants, s'applique aux périphériques multiples pour autant qu'on sache les désigner dans les programmes.

Nommage des sources

La solution au second problème, celui de la désignation, est une extension simple du modèle d'arbre utilisé dans IntuiKit. Un programme y est réputé constitué d'un arbre de composants ; l'arbre représente autant la structuration logicielle (visibilité des variable et des sous-composants) que le cycle de vie des composants : quand un composant est instancié, tous ses sous-composants le sont aussi. L'arbre fournit un moyen simple pour nommer et rechercher des composants pour s'y abonner, à la manière des fichiers dans un système arborescent ou des éléments dans un arbre XML. Il est donc naturel, autant sur le plan syntaxique que sur le plan sémantique, d'étendre ce principe aux périphériques : on peut les considérer comme des composants situés *en dehors* de l'arbre de l'application, cette dernière étant vue comme une partie d'un arbre plus grand qui représente l'ordinateur, ses processus et ses ressources. On peut donc rechercher

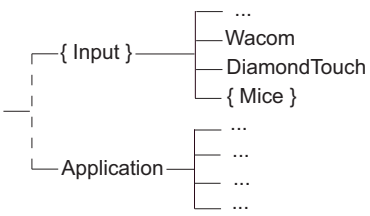


Figure 1 **:** Les périphériques dans l'arbre étendu

les sources d'événements dans cet arbre étendu pour s'y abonner. Pour cela, un système d'espaces de nommage permet d'ajouter l'accès à de nouvelles parties de l'arbre étendu, comme on ajoute de nouveaux pilotes à un ordinateur. Par exemple la ligne de XML qui suit

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

recherche une DiamondTouch branchée sur l'ordinateur et crée un abonnement dessus.

Dynamicité

Sur une table interactive, le nombre de points de contact varie. Il débute en général à zéro, puis peut monter jusqu'à dix ou plus. Chacun des points de contact est un pointeur, susceptible d'être utilisé comme une souris ou presque. Lorsqu'il apparaît, il faut lui associer un rendu visuel (curseur, déformation de l'image, ombre), et créer des abonnements sur ses actions. Pour permettre cela, nous généralisons le mécanisme de base déjà proposé dans IntuiKit pour décrire la gestion de données du noyau fonctionnel : l'apparition ou la disparition de parties de l'arbre, quelle qu'en soit la cause, est un événement auquel un programmeur peut associer une action. Puisque les périphériques sont considérés comme des parties de l'arbre étendu, leur apparition ou leur disparition provoque aussi un événement, qui permet d'effectuer les créations et associations désirées. La source correspondante est l'ensemble des périphériques d'interaction, accessible à travers le symbole global `'input'` :

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

Périphériques hiérarchiques

Tous les périphériques ne sont pas de même nature. Pour simuler une table DiamondTouch il faut au minimum quatre souris. Une tablette Wacom Intuos 2, qui gère jusqu'à deux stylos à la fois, distingue des milliers de stylos chacun de manière unique. Comment représenter cette organisation ? La solution que nous proposons est de dissoudre la notion de périphérique dans celle de composant, qui a l'avantage d'être récursive. Ainsi, on peut considérer qu'un périphérique contient des sous-périphériques. Cela permet tout d'abord de rendre compte des situations complexes : on peut s'abonner indifféremment à un périphérique, un sous-périphérique, ou même un composant fabriqué en assemblant des périphériques. Cela permet ensuite de rendre compte des situations où un capteur peut détecter un nombre indéfini d'objets : le capteur est considéré comme un ensemble de sous-composants dont l'apparition ou la disparition provoque un événement, comme dans le cas de l'ensemble des périphériques. Cela permet enfin de choisir, en utilisant le mécanisme d'encapsulation des composants, ce qu'un périphérique rend visible de sa composition interne.

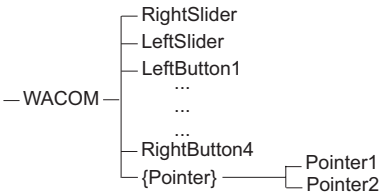


Figure 2 **:** La structure interne d'une tablette Wacom

Configuration des entrées

Lorsqu'on désire qu'un composant logiciel soit réutilisable, il faut fournir un moyen de communication entre le programmeur de ce composant et ceux qui le réutiliseront, pour en décrire les contraintes d'usage. Dans le cas qui nous intéresse, ce qui est le plus susceptible de changer dans le contexte est la structure des périphériques. Une DiamondTouch distingue entre quatre utilisateurs, mais fournit seulement un tableau des coordonnées horizontales des points de contact ainsi qu'un tableau des coordonnées verticales, sans corrélation. Une

Entertaible ou une table à base de FTIR fournissent une liste de points de contacts, mais pas d'identification de l'utilisateur.

La solution que nous proposons consiste à donner au programmeur d'applications le moyen d'inspecter les caractéristiques des périphériques dispoibles. Lorsqu'elles sont incompatibles avec les composants réutilisés dans l'application, le programmeur a la possibilité d'utiliser le système de flots de données et d'événements pour corriger cela en masquant le périphérique derrière un ensemble de filtres.

Inspection et encapsulation

Sur quelle base décider qu'un périphérique donné convient pour l'usage prévu dans une application ? La solution que nous proposons s'intègre dans un schéma plus général de vérification de compatibilité entre composants, et pourrait servir de base à un système de typage. Le mécanisme d'encapsulation de composants d'IntuiKit permet à un programmeur de composant de choisir quelles propriétés (c'est-à-dire quelles valeurs actives), quels événements, et quels sous-composants en seront visibles à l'extérieur. Tout programmeur qui réutilise ce composant peut accéder à ce qui est exporté pour y connecter des flots de données, s'abonner, ou explorer la structure du composant. Pour cela, il est possible de lister les propriétés, événements, et sous-composants visibles. Il est aussi possible d'accéder directement à l'un d'entre eux comme ci-dessous:

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

Cela permet aux programmeurs d'applications de vérifier que le composant qu'ils étudient est conforme à ce qu'ils attendent. Il est prévu qu'ultérieurement les composants décrivent les séquences d'événements qu'ils sont susceptibles d'émettre, pour rendre ces vérifications encore plus complètes, comme dans [1].

Par ailleurs, le mécanisme d'encapsulation d'IntuiKit prévoit la possibilité de publier un événement, propriété ou sous-composant sous un autre nom que le nom d'origine. Cela permet à un programmeur de prendre un périphérique, l'encapsuler dans un composant, et le faire passer pour un autre périphérique. Par exemple, il est simple avec ce schéma de simuler un écran tactile avec une tablette Wacom.

Sources filtres

L'encapsulation fonctionne quand l'équivalence entre périphériques peut s'obtenir simplement en cachant ou en renommant des parties. Mais la situation est souvent plus complexe : des calculs peuvent être nécessaires. Par exemple, pour simuler un écran tactile avec une souris il faut mémoriser une "position courante" et y ajouter les déplacements de la souris après leur avoir appliqué une fonction d'accélération, puis contenir la position courante dans un rectangle correspondant à la taille de l'écran. De la même manière, pour simuler un écran tactile à partir d'une DiamondTouch il faut calculer le barycentre des abscisses et celui des ordonnées des points de contact.

La solution proposée est d'introduire dans IntuiKit une généralisation du système d'événements et du système de flots de données : le *filtrage de sources*. Le principe est de permettre à un composant de s'abonner aux événements émis par un autre composant, de se poser en relais pour les abonnements, et d'appliquer des transformations aux événements avant de les ré-émettre. Cela permet tout d'abord au filtre de proposer son propre langage d'abonnement. Par exemple, on pourra masquer une DiamondTouch derrière un filtre qui offre les abonnements d'un écran tacile. Si l'on combine cette possibilité avec l'encapsulation, en faisant du filtre le composant parent du périphérique d'origine et en cachant ce dernier, l'opération est quasi-parfaite. Reste seulement à ajouter le calcul nécessaire pour modifier les événements; il suffit pour cela d'ajouter dans le nouveau composant une brique de calcul similaire à celles utilisées dans la gestion des flots de données.

Le filtrage est utilisé par exemple pour la reconnaissance de gestes : la source "gestes" est un filtre qui effectue des calculs de classification et se branche sur n'importe quelle source capable d'émettre des suites de positions. C'est aussi le mécanisme que l'on peut utiliser pour simuler le véritable multi-point sur une DiamondTouch, en embarquant un algorithme de calcul de suivi de points qui exploite les histogrammes fournis par la DiamondTouch.

Sources distantes

Les concepteurs d'IHMs sur table veulent parfois accéder aux périphériques à travers des communications entre processus. Cela peut être parce que le périphérique est réellement distant, par exemple dans le cas de deux tables utilisées comme support de communication [7]. Cela peut être parce qu'il n'y a pas de pilote pour le périphérique dans le système d'exploitation sur lequel s'exécute l'application. Ou cela peut être parce qu'une application n'a pas été construite avec le support pour l'adaptation et qu'il faut simuler le périphérique depuis l'extérieur. Dans ces cas, on peut utiliser la capacité d'IntuiKit à transporter événements et flots de données sur des bus logiciels tels qu'Ivy [2] ou Open Sound Control [18]. On construit alors un agent logiciel qui émet des événements et apparaît dans l'arbre IntuiKit comme un périphérique; on peut alors lui appliquer toutes les méthodes de travail décrites plus haut dans cet article.

Interaction parallèle

Le but des surfaces interactives est de permettre à plusieurs utilisateurs d'interagir en même temps, ou à un utilisateur d'utiliser plusieurs moyens de pointage. Non content de savoir quels sont les périphériques accessibles et quelle est leur nature, il faut aussi gérer l'interaction

en parallèle. Nous allons voir comment le modèle à événements permet de rendre compte de ces situations de manière naturelle.

La première exigence est que l'environnement de programmation fournisse un moyen d'ajouter de nouvelles sources d'événements asynchrones. C'est le cas de tous les environnements modernes, car ils doivent déjà fournir un tel mécanisme pour la communication entre processus.

Il faut ensuite permettre de fabriquer des interacteurs qui se manipulent en parallèle. Paradoxalement, c'est aussi une caractéristique courante : le modèle à événements induit une architecture naturellement parallèle, et pour peu que l'environnement gère l'animation cette architecture est en général déjà exploitée car l'animation se produit souvent en parallèle de l'interaction. Le seul risque est en fait que les interacteurs soient programmés de manière à stocker leur état de manière globale, ce qui empêcherait d'utiliser deux fois le même type d'interacteur. L'architecture d'arbre et le fait d'encourager la représentation des états de l'interaction sous forme de propriétés de composants, voire directement sous forme de composants appelés machines à états, évite en général ce problème.

Enfin, reste le problème d'exprimer des comportements qui combinent plusieurs flux d'interaction en provenance de plusieurs sources. C'est ici que le modèle de flux de contrôle qui repose sur un mécanisme unique d'abonnement aux événements prend toute son importance. Si tout transfert de contrôle est le résultat d'un abonnement et si tous les abonnements obéissent à un même schéma, alors on peut combiner à volonté les abonnements. C'est ainsi qu'une machine à états pourra avoir une transition étiquetée par "toucher par l'utilisateur 1" et une autre transition par "toucher par l'utilisateur 2", mettant ainsi en oeuvre un bouton qui ne s'efonce que si deux utilisateurs appuient dessus ensemble. C'est ainsi que les concepteurs d'applications sur table interactive qui utilisent IntuiKit peuvent concevoir les styles d'interaction en toute liberté.

Conclusion

Dans cet article nous avons décrit comment la représentation de logiciels interactifs sous forme d'un arbre de composants communiquant par événements et flots de données peut être étendue pour gérer les situations rencontrées dans la programmation de surfaces interactives. Ces extensions, qui préservent l'approche à base de modèles exécutables, permettent d'ajouter l'adaptabilité aux périphériques à l'ensemble des caractéristiques de cette approche qui facilitent la conception et le déploiement d'IHMs post-WIMP.

BIBLIOGRAPHIE

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 *Proc. of the ACM UIST*, pages 195–204. Addison-Wesley, Nov. 1994.
5. S. Chatty. Programs = data + algorithms + architecture. Consequences for interactive software. In *Proc. 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 *Proc. 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 *Proc. 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 Proc. of the ACM CSCW 2004 conference*, 2004.
9. P. Dragicevic and J.-D. Fekete. Support for input adaptability in the icon toolkit. In *Proc. 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 *Proc. 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 *Proc. 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 *Proc. 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 *Proc. of the NIME-03 conference*, 2003.